Wolfram *Mathematica*® Tutorial Collection

DYNAMIC INTERACTIVITY

For use with Wolfram *Mathematica*® 7.0 and later.

For the latest updates and corrections to this manual:
visit reference.wolfram.com

For information on additional copies of this documentation:
visit the Customer Service website at www.wolfram.com/services/customerservice
or email Customer Service at info@wolfram.com

Comments on this manual are welcomed at:
comments@wolfram.com

Content authored by:
Theodore Gray and Lou D'Andria

Printed in the United States of America.

15 14 13 12 11 10 9 8 7 6 5 4 3 2

Contents

Introduction to Dynamic

This tutorial describes the principles behind `Dynamic`, `DynamicModule` and related functions, and goes into detail about how they interact with each other and with the rest of *Mathematica*.

These functions are the foundation of a higher-level function `Manipulate` that provides a simple yet powerful way of creating a great many interactive examples, programs, and demonstrations, all in a very convenient, though relatively rigid, structure. If that structure solves the problem at hand, you need look no further than `Manipulate` and you do not need to read this tutorial. However, do continue with this tutorial if you want to build a wider range of structures, including complex user interfaces.

This is a hands-on tutorial. You are expected to evaluate all the input lines as you reach them and watch what happens. The accompanying text will not make sense without evaluating as you read.

The Fundamental Principle of Dynamic

Ordinary *Mathematica* sessions consist of a series of static inputs and outputs, which form a record of calculations done in the order in which they were entered.

Evaluate each of these four input cells one after the other.

```
In[1]:=  x = 5;

In[2]:=  x^2
Out[2]= 25

In[1]:=  x = 7;

In[4]:=  x^2
Out[4]= 49
```

The first output still shows the value from when x was 5, even though it is now 7. This is, of course, very useful, if you want to see a history of what you have been doing. However, you may often want a fundamentally different kind of output, one that is automatically updated to always reflect its current value. This new kind of output is provided by `Dynamic`.

Evaluate the following cell; note that the result will be 49 because the current value of x is 7.

```
In[5]:= Dynamic[x^2]
```

In fact it is generally the case that when you first evaluate an input that contains variables wrapped in `Dynamic`, you will get the same result as you would have without `Dynamic`. But if you subsequently change the value of the variables, the displayed output will change retroactively.

Evaluate the following cells one at a time, and note the change in the value displayed above.

```
In[6]:= x = 9;

In[7]:= x = 15;

In[8]:= x = 10;
```

The first two static outputs are still 25 and 49 respectively, but the single dynamic output now displays 100, the square of the last value of x. (This sentence will, of course, become incorrect as soon as the value of x is changed again.)

There are no restrictions on the kinds of values that can go into a dynamic output. Just because x was initially a number does not mean it cannot become a formula or even a graphic in subsequent evaluations. This might seem like a simple feature, but it is the basis for a very powerful set of interactive capabilities.

Each time the value of x is changed, the dynamic output above is updated automatically. (You might need to scroll back to see it.)

```
In[9]:= x = Integrate[1/(1 - y^3), y];

In[10]:= x = Plot[Sin[x], {x, 0, 2 Pi}];

In[1]:= x = 0;
```

`Dynamic[`*expr*`]`	an object that displays as the dynamically updated current value of *expr*

Basic dynamic expression.

Dynamic and Controls

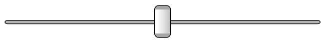
`Dynamic` is often used in connection with controls such as sliders and checkboxes. The full range of controls available in *Mathematica* is discussed in "Control Objects"; here sliders are used to illustrate how things work. The principles of using `Dynamic` with other controls is basically the same.

A slider is created by evaluating the `Slider` function, in which the first argument is the position and the optional second argument specifies the range and step size, with the default range from 0 to 1 and the default step size 0.

This is a slider in a centered position.

```
In[11]:= Slider[0.5]
```

Out[11]=



Click on the thumb and move it around. The thumb moves, but nothing else happens since the slider is not connected to anything.

This associates the position of the slider with the current value of the variable x. (This form is explained in more detail later.)

```
In[12]:= Slider[Dynamic[x]]
```

Out[13]=

This creates a new dynamic output of x since the last one has probably scrolled off your screen by now.

```
In[14]:= Dynamic[x]
Out[14]= 0.
```

Drag the last slider around. As the slider moves, the value of x changes and the dynamic output updates in real time.

The slider also responds to changes in the value of x.

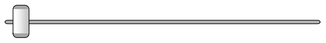
To see this, evaluate this line.

```
In[15]:= x = 0.8;
```

You should see the slider jump, and the dynamic output of x change, simultaneously.

This creates another x slider.

In[16]:=
```
Slider[Dynamic[x]]
```

Out[16]=


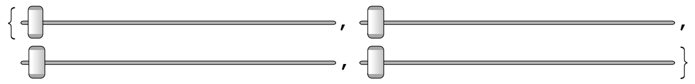
Notice that if you move *either* of the two sliders you now have, the other one moves in "lock sync." Both are connected, dynamically and bi-directionally, to the current value of x.

Dynamic and Other Functions

`Dynamic` and control constructs such as `Slider` are in many ways just like any other functions in *Mathematica*. They can occur anywhere in an output, in tables, and even inside typeset mathematical expressions. Wherever these functions occur, they carry with them the behavior of dynamically displaying or changing in real time the current value of the expression or variable they are linked to. `Dynamic` is a simple building block, but the rest of *Mathematica* turns it into a flexible tool for creating nimble, zippy, and often fun little interactive displays.

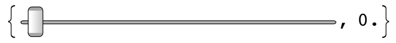
This makes a table of x sliders, which are updated in sync.

In[2]:=
```
Table[Slider[Dynamic[x]], {4}]
```

Out[2]=


You can combine a slider with a display of its current value in a single output.

In[3]:=
```
{Slider[Dynamic[x]], Dynamic[x]}
```

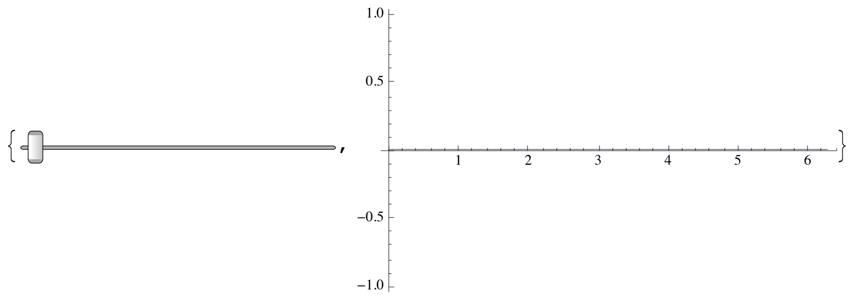
Out[3]=


```
, 0.
```

The great power of `Dynamic` lies in the fact that it can display any function of x just as easily.

```
In[20]:= {Slider[Dynamic[x]], Dynamic[Plot[Sin[10 y x], {y, 0, 2 Pi}]]}
```

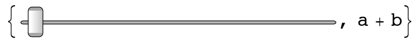
Out[20]=



Using integer-valued sliders, you can create dynamically updated algebraic expressions.

```
In[21]:= {Slider[Dynamic[x1], {1, 10, 1}], Dynamic[Expand[(a + b)^x1]]}
```

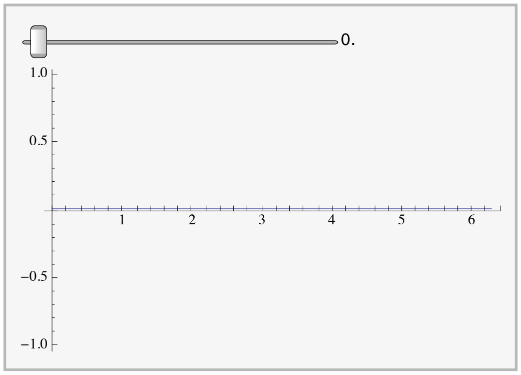
Out[21]=



You can use dynamic expressions with `Panel`, `Row`, `Column`, `Grid`, and other formatting constructs.

```
In[22]:= Panel[Column[{Row[{Slider[Dynamic[x]], Dynamic[x]}],
            Dynamic[Plot[Sin[10 y x], {y, 0, 2 Pi}]]}]]
```

Out[22]=



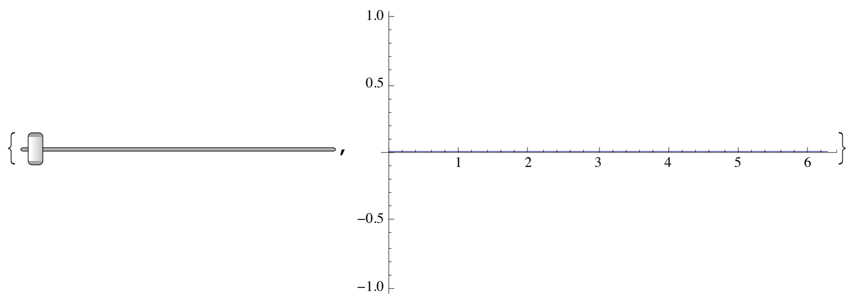
Notice that the last example resembles the output of `Manipulate`. This is no coincidence, because `Manipulate` in fact produces a combination of `Dynamic`, controls, and formatting constructs, not fundamentally different from what you can do yourself using these lower-level functions.

Localizing Variables in Dynamic Output

Here is another copy of a slider connected to a simple plot.

```
In[23]:= {Slider[Dynamic[x]], Dynamic[Plot[Sin[10 y x], {y, 0, 2 Pi}]]}
```

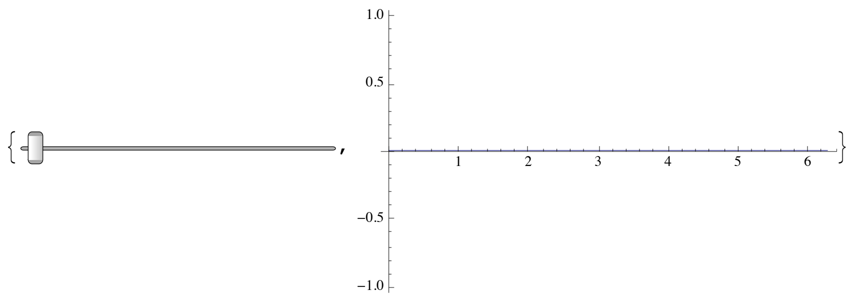
Out[23]=



This is a slider connected to another function.

```
In[24]:= {Slider[Dynamic[x]], Dynamic[Plot[Tan[10 y x], {y, 0, 2 Pi}]]}
```

Out[24]=



If you have both these outputs visible and drag either slider, you will notice that they are communicating with each other. Move the slider in one example, and the other example moves too. This is because you are using the global variable x in both examples. Although this can be very useful in some situations, most of the time you would probably be happier if these two sliders could be moved independently. The solution is a function called `DynamicModule`.

`DynamicModule[{`x`,`y`,…},`*expr*`]`	an object which maintains the same local instance of the symbols x, y, … in the course of all evaluations of `Dynamic` objects in *expr*
`DynamicModule[{`$x=x_0$`,`$y=y_0$`},`*expr*`]`	specifies initial values for x, y, …

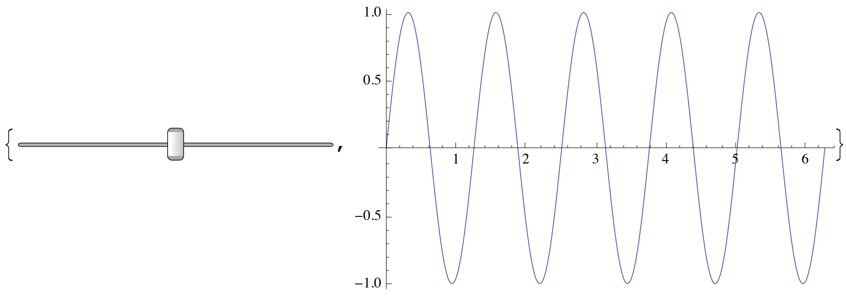
Localizing and initializing variables for `Dynamic` objects.

`DynamicModule` has arguments identical to `Module` and is similarly used to localize and initialize variables, but there are important differences in how they operate.

Here are the same two examples with "private" values of x.

```
In[25]:= DynamicModule[{x = .5},
          {Slider[Dynamic[x]], Dynamic[Plot[Sin[10 y x], {y, 0, 2 Pi}]]}]
```

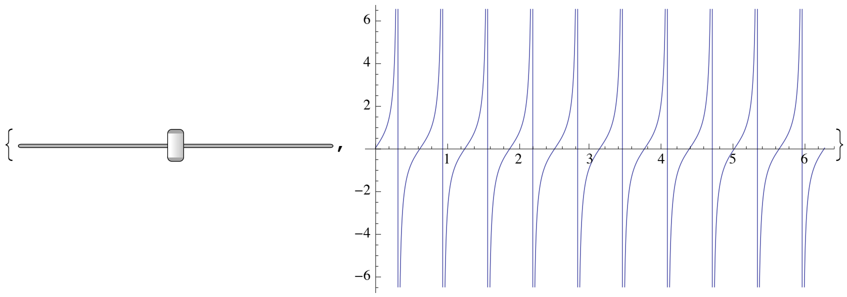
Out[25]=



Notice that these two examples now work independently of each other.

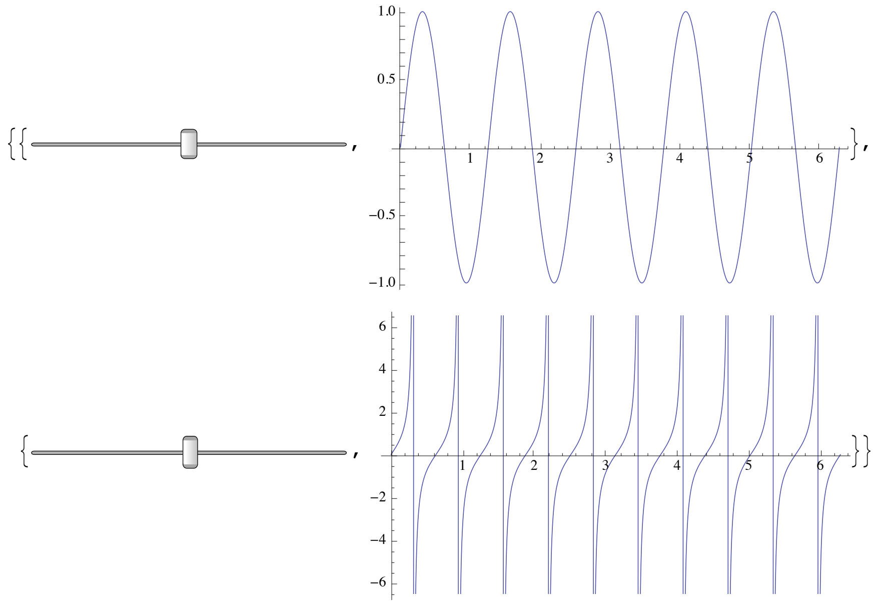
```
In[26]:= DynamicModule[{x = .5},
          {Slider[Dynamic[x]], Dynamic[Plot[Tan[10 y x], {y, 0, 2 Pi}]]}]
```

Out[26]=



Multiple `DynamicModules` can be placed in a single output, and they maintain separate values of the variables associated with their respective areas in the output.

```
In[27]:= {DynamicModule[{x = .5},
           {Slider[Dynamic[x]], Dynamic[Plot[Sin[10 y x], {y, 0, 2 Pi}]]}],
         DynamicModule[{x = .5}, {Slider[Dynamic[x]],
             Dynamic[Plot[Tan[10 y x], {y, 0, 2 Pi}]]}]}
```

Out[27]=


You might be tempted to use `Module` in place of `DynamicModule`, and in fact this would appear to work at first. However, it is not a good idea for several reasons, which are discussed in more detail in "Advanced Dynamic Functionality".

`DynamicModule` does its work in the front end, not in the kernel. It remains unchanged by evaluation, and when formatted as output, it creates an invisible object embedded in the output expression which handles the localization. As long as that space of output remains in existence (i.e., is not deleted), the invisible object representing the `DynamicModule` will maintain the values of the variables, allowing them to be used in subsequent evaluations of `Dynamic` expressions within the scope (area) of the `DynamicModule`.

If you save a notebook containing a `DynamicModule`, close that notebook, then later reopen it in a new *Mathematica* session, the values of all the local variables will still be preserved and the sliders inside the `DynamicModule` will be in the same positions. This will *not* be the case with sliders linked to global variables (like the earliest examples in this tutorial), nor with sliders linked to variables localized with `Module` instead of `DynamicModule`. Such variables store their values in the current *Mathematica* kernel session, and they are lost as soon as you quit *Mathematica*.

In addition to localizing variables to particular regions of output, `DynamicModule` provides options to automatically initialize function definitions when an expression containing a `DynamicModule` is opened, and to clean up values when the expression is closed or deleted. More details are found in `DynamicModule`.

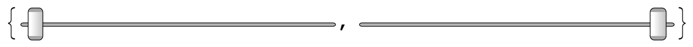
The Second Argument of Dynamic

Dynamic connections are by default bi-directional. Sliders connected to a variable move together because they both reflect and control the value of the same variable. When you drag a slider thumb, the system constructs and evaluates expressions of the form *expr* = *new*, where *expr* is the expression given in the first argument to `Dynamic` and *new* is the proposed new value determined by where you have dragged the slider thumb. If the assignment can be done, the new value is accepted. If the assignment fails, the slider will not move.

These two sliders move in opposite directions when you move the first one. However, trying to move the second slider gives an error because you cannot assign a new value to the expression `1 - x`.

```
In[1]:= DynamicModule[{x = 0}, {Slider[Dynamic[x]], Slider[Dynamic[1 - x]]}]
```

Out[1]=



You can keep an arbitrary expression in the first argument of `Dynamic`, but change the dynamically executed evaluation by using the optional second argument. This is a convenient way to specify "inverse functions" that update the values of variables in the first arguments. *Mathematica* does not attempt to deduce such inverse functions automatically from the first argument of `Dynamic`; you have to supply one yourself.

`Dynamic[`*expr*`,`*f*`]`	continually evaluates *f*[*val*, *expr*] during interactive changing or editing of *val*

Inverse functions.

This specifies how the value of *x* is to be updated and makes the second slider interactive. You can move either slider and the other slider responds by moving in the opposite direction.

```
In[30]:= DynamicModule[{x = 0}, {Slider[Dynamic[x]], Slider[Dynamic[1 - x, (x = 1 - #) &]]}]
```

Out[30]=

Now the dynamically executed expression in the second slider is the pure function `(x = 1 - #) &`, which is given the proposed new value in `#`. Note that the function is responsible for actually doing the assignment to whatever variable you want to change; you cannot just say `(1 - #) &` if you want to change `x`.

The ability to interpose your own arbitrary function between the mouse position and the state of *Mathematica* is very powerful, and you can use it for purposes beyond simple inverse functions. The function given in the second argument is effectively free do to anything it wants.

This defines "detents" that snap the slider to integer values if the thumb is within a certain tolerance of a round number.

```
In[31]:= DynamicModule[{x = 0},
          {Slider[Dynamic[x, If[Abs[# - Round[#]] < 0.1, x = Round[#], x = #] &], {0, 5}],
           Dynamic[x]}]
```

Out[31]=

This makes the variable take on rational numbers (integer fractions) instead of decimals.

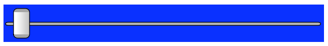
```
In[32]:= DynamicModule[{x = 0},
          {Slider[Dynamic[x, (x = Rationalize[#]) &], {0, 5}], Dynamic[x]}]
```

Out[32]=

For complete control over the tracking behavior, it is possible to specify separate functions that are called at the start, middle, and end of a mouse click on the slider thumb. If you are familiar with conventional user-interface programming, you will recognize these as separate, high-level event functions for the mouse-down, mouse-drag, and mouse-up events.

This changes the background color while the click-and-drag operation is underway.

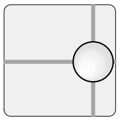
```
In[33]:= DynamicModule[{x = 0, bg = RGBColor[0, 0, 1]}, Style[Slider[Dynamic[x,
             {(bg = RGBColor[1, 0, 0]) &,
              (x = #) &,
              (bg = RGBColor[0, 0, 1]) &}]], Background → Dynamic[bg]]]
```

Out[33]=


The second argument of `Dynamic` also lets you restrict the movement of a slider and effectively implement geometric constraints.

You can only move the thumb of this `Slider2D` along a circle.

```
In[34]:= DynamicModule[{pt = {1, 0}},
          Slider2D[Dynamic[pt, (pt = # / Norm[#]) &], {-1, 1}, Exclusions → {0, 0}]]
```

Out[34]=


Where Should Dynamic Be Placed in an Expression?

The fundamental behavior of `Dynamic` is to build a copy of the input expression into the output cell. To be more specific, `Dynamic` has the attribute `HoldFirst` and remains unchanged by evaluation.

The result of evaluating `Dynamic`$[x+y]$ is `Dynamic`$[x+y]$, which you can see by examining the `InputForm` representation of the output.

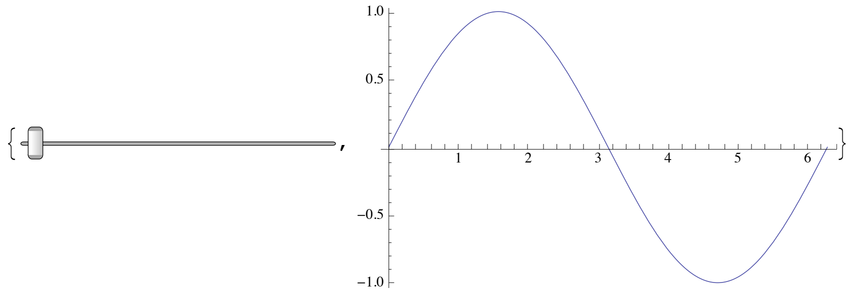
```
In[35]:= Dynamic[x + y] // InputForm

Out[35]//InputForm= Dynamic[x + y]
```

You do not see `Dynamic` in ordinary output because, when formatted for display in the front end, `Dynamic`$[x+y]$ is represented by an object that contains a copy of the unevaluated input ($x+y$), but displays as the evaluated value of that expression. The `Dynamic` wrapper is still present in the output, but it is invisible.

Because `Dynamic` does its work entirely in the front end, you cannot use it inside functions that need to access the value of an expression in order to do their work.

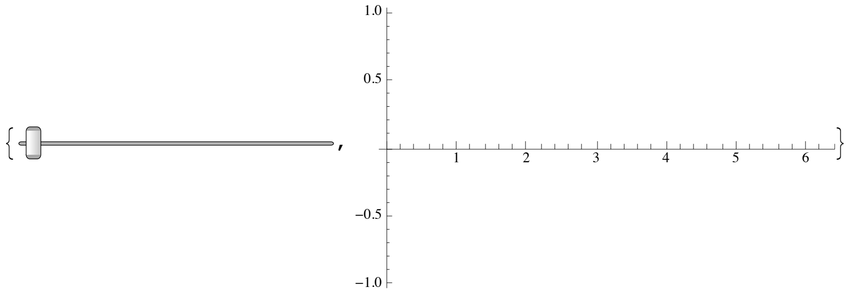
For example, this works.

```
In[36]:= DynamicModule[{x},
          {Slider[Dynamic[x], {1, 5}], Dynamic[Plot[Sin[x i], {i, 0, 2 Pi}]]}]
```

Out[36]=


But this does not.

```
In[37]:= DynamicModule[{x},
          {Slider[Dynamic[x], {1, 5}], Plot[Sin[Dynamic[x] i], {i, 0, 2 Pi}]}]
```

Out[37]=


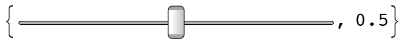
The `Plot` command needs to have specific numerical values for x to make a plot, but the `Dynamic[`x`]` inside the function being plotted does not *evaluate* into anything in the kernel. It remains inert as `Dynamic[`x`]`, preventing the `Plot` command from doing anything sensible.

Another way to look at it is that the expression inside a `Plot` command does not appear directly anywhere in the output. `Dynamic` is a formatting function that does its work in the front end, not in the kernel, so if it is used in a way where it will never be placed as output, it is probably a mistake.

When combining `Dynamic` with controls, it is particularly important to get the `Dynamic` in the right place.

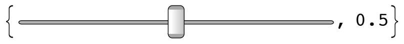
This example works as expected; move the slider and the value of x changes.

```
In[38]:= DynamicModule[{x = 0.5}, {Slider[Dynamic[x]], Dynamic[x]}]
```

Out[38]=


This example looks good at first, but if you move the slider, x does not change.

```
In[39]:= DynamicModule[{x = 0.5}, {Dynamic[Slider[x]], Dynamic[x]}]
```

Out[39]=


That is because when the `Dynamic` wrapped around `Slider[`x`]` evaluates its contents, the value of x is substituted, and the result is a slider whose first argument is a specific number, with no trace of the variable name left. The slider in this case is a *dynamic* display of a *static* slider.

What is needed is a *static* slider, which contains within it a *dynamic* reference to the value of the variable. In the case of controls, there is a simple rule for where to put the `Dynamic`. The first argument of any control function, such as `Slider`, `Checkbox`, or `PopupMenu`, will almost always be `Dynamic[`*var*`]`.

Beyond these cases where `Dynamic` will *not* work in a particular position, there is often a great deal of flexibility about where to place `Dynamic`. It is often used as the outermost function in an input expression, but this is by no means necessary, and in more sophisticated applications, `Dynamic` is usually used deeper in the expression and can even be nested.

This displays a table of ten copies of the value of x.

```
In[40]:=  Dynamic[Table[x, {i, 10}]]
Out[40]=  {0., 0., 0., 0., 0., 0., 0., 0., 0., 0.}
```

`Dynamic` is wrapped around the whole expression, so evaluation of the `Table` command is delayed until the output is displayed in the notebook. Any time the value of x is changed, the `Table` command will be reevaluated.

The output from this example looks exactly the same.

```
In[41]:=  Table[Dynamic[x], {i, 10}]
Out[41]=  {0., 0., 0., 0., 0., 0., 0., 0., 0., 0.}
```

But in this case the `Table` command is evaluated immediately, generating a list of ten separate `Dynamic` expressions, each of which evaluates x separately after the overall result has been placed in the notebook.

When x is changed, the first example sends a single request to the kernel to get the value of `Table[`x`, {`i`, 10}]`, while the second example sends ten separate requests to the kernel to get the value of x. It might seem that the first example is obviously more efficient, and in this case it is. However, you should also avoid the other extreme, wrapping too many things into a single `Dynamic`, which can also be inefficient.

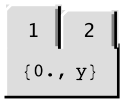
This initializes x and y to set up a new slider connected to the value of x.

```
In[6]:=  x = 0.5;

In[7]:=  y = Plot3D[Sin[n m], {n, 0, 4}, {m, 0, 4}];

In[8]:=  Slider[Dynamic[x]]

Out[8]=
```

This is a tab view with two groups of dynamic expressions, both showing the dynamic values of x (a simple number) and y (a 3D plot).

```
In[9]:= TabView[{{Dynamic[x], Dynamic[y]}, Dynamic[{x, y}]}]
```

Out[9]=



```
1 | 2
{0., y}
```

Drag the slider around, and note that the value of x in the first tab updates quite rapidly. On most computers it will be essentially instantaneous. However, updates are more sluggish in the second tab. Each individual `Dynamic` expression keeps track (quite carefully) of exactly when it might need to be reevaluated in order to remain up-to-date. In the second tab, the output is forcing the whole expression $\{x, y\}$, including the large, slow 3D plot, to be reevaluated every time the value of x changes. By using two separate `Dynamic` expressions in the first tab, you allow the value of x to be updated without needing to also reevaluate y, which has not actually changed. (You may want to delete the last output before proceeding, as it will slow down any examples containing the global x as long as it is visible on screen.)

It is hard to make blanket statements about where `Dynamic` should be placed in every case, but generally speaking if you are building a large, complex output where only small parts of it will change, the `Dynamic` should probably be wrapped just around those parts. On the other hand, if all or most of the output is going to change in response to a single variable changing its value, then it is probably best to wrap `Dynamic` around the whole thing.

Dynamic in Options

`Dynamic` can be used on the right-hand side of options, in those cases where the option value will be transmitted to the front end before being used. This is a somewhat subtle distinction related to the discussion in "Where Should Dynamic Be Placed in an Expression".

An option like `PlotPoints` in plotting commands cannot have `Dynamic` on the right-hand side, because the plotting command needs to know a specific numerical value before the plot can be generated. Remember that `Dynamic` has the effect of delaying evaluation until the expression reaches the front end, and in the case of `PlotPoints`, that is too late since the value is needed right away. On the other hand, options to functions that do their work in the front end can usually, and usefully, accept `Dynamic` in their option values.

For example, you can control the size of a block of text in two ways.

`Dynamic` can be wrapped around a whole `Style` expression.

```
In[10]:= h = 12;
```

```
In[11]:= {Slider[Dynamic[h], {6, 100}], Dynamic[Style["Some Text", FontSize → h]]}
```

Out[11]=

Or `Dynamic` can be only in the `FontSize` option value.

```
In[59]:= {Slider[Dynamic[h], {6, 100}], Style["Some Text", FontSize → Dynamic[h]]}
```

Out[59]=

There are two potential advantages to putting the `Dynamic` in the option value. First, suppose the dynamically regenerated expression is very large, for example if the block of text is the entire document, it is inefficient to retransmit it from the kernel to the front end every time the font size is changed, as is necessary if `Dynamic` encloses the whole expression.

Second, the output of a `Dynamic` expression is not editable (since it is liable to be regenerated at any moment), which makes the output of the first example non-editable. But the text in the second example can be edited freely since it is ordinary static output: only the option value is dynamic.

Dynamic option values can be also set in the Option Inspector. They are allowed at the cell, notebook, or global level, and in stylesheets. (Note, however, that if you set a dynamic option value in a position where the value will be inherited by many cells, for example in a stylesheet, there can be a significant impact on performance.)

You can set dynamic option values through `SetOptions`, as well.

```
In[51]:= x = 0;
```

```
In[51]:= SetOptions[EvaluationNotebook[], Background → Dynamic[Hue[x]]]
```

Having linked the background color of the notebook to the global variable x, it can now be controlled by a slider or by a program.

```
In[52]:= Slider[Dynamic[x]]
```

Out[52]=

Of course, it is good to be able to return to normal.

```
In[53]:= SetOptions[EvaluationNotebook[], Background → Inherited]
```

Dynamic and Infinite Loops

If you are not careful, you can easily throw `Dynamic` into an infinite loop.

This counts upwards as fast as possible for as long as it remains on screen.

```
In[54]:= DynamicModule[{x = 1}, Dynamic[x = x + 1]]
```

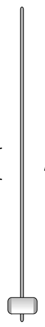
This is not a bug (but delete the above output if it is distracting you to have it there).

Because the output is updated and the screen redrawn after each cycle of an infinite loop, it is actually quite a useful thing to be able to do. Generally speaking, the system will remain responsive to typing, evaluation, and so on, even as the infinitely updating `Dynamic` zips along.

It is also useful to make such a self-triggering `Dynamic` that stops changing at some point.

This is a "droopy" slider, which always drops back to zero no matter what you drag it to.

```
In[55]:= DynamicModule[{x = 1}, {VerticalSlider[Dynamic[x]], Dynamic[x = Max[0, x - 0.01]]}]
```

```
Out[55]= {
```

```
, 0}
```

If you have a CPU monitor running, you will see that while the slider is dropping there is a small load on the CPU (for redrawing the screen, primarily), but once it reaches zero, the load drops to nothing. The dynamic tracking system has noticed that the value of x did not change: therefore, further updating is not necessary until someone changes the value of x again (e.g., when you click on the slider). "Advanced Dynamic Functionality" describes in more detail how the dynamic tracking system works.

A Good Trick to Know

Because it has the attribute `HoldFirst`, `Dynamic` does not evaluate its first argument. This is fundamental to the workings of `Dynamic`, but it can lead to a somewhat unexpected behavior.

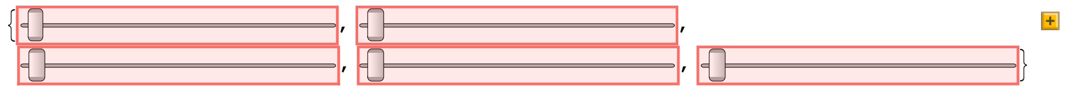
For example, suppose you have a list of numbers you wish to be able to modify by creating one slider to control each value.

This creates the list and a dynamic display of its current value.

```
In[1]:=  data = {.1, .5, .3, .9, .2};

In[2]:=  Dynamic[data]
Out[2]= data
```

This attempts to make a table of sliders, one for each element of the list, using *list*[[*i*]] to access the individual members.

```
In[3]:=  Table[Slider[Dynamic[data[[i]]]], {i, 5}]
```

Out[3]=


Surprisingly, this does not work! You can see an error indication around the sliders, they cannot be moved, and the dynamic output above never changes. You might even jump to the conclusion that part extraction syntax cannot be used in this way with controls. Nothing could be further from the truth.

The problem is that the variable *i* was given a temporary value by the `Table` command, but that value was never used, because `Dynamic` is `HoldFirst`.

Looking at the `InputForm` of the table of sliders reveals the problem.

```
In[59]:=  Table[Slider[Dynamic[data[[i]]]], {i, 5}] // InputForm
Out[59]//InputForm= {Slider[Dynamic[data[[i]]]], Slider[Dynamic[data[[i]]]],
                     Slider[Dynamic[data[[i]]]], Slider[Dynamic[data[[i]]]],
                     Slider[Dynamic[data[[i]]]]}
```

What is needed is to do a replacement of the variable *i* with its temporary value, even inside held expressions.

This can be done with `/.` or with the somewhat peculiar but convenient idiomatic form demonstrated here.

```
In[60]:=  Table[With[{i = i}, Slider[Dynamic[data[[i]]]]], {i, 5}]
```

This output shows that `Dynamic` does in fact work perfectly with part extraction syntax, a very useful property.

Slow Evaluations inside Dynamic

`Dynamic` wrapped around an expression that will take forever, or even more than just a few seconds, to finish evaluating is a bad thing.

If you evaluate this example, you will have to wait about 5 seconds before seeing the output `$Aborted`.

```
In[61]:= Dynamic[While[True]]
```

During the wait for the `Dynamic` output to evaluate, the front end is frozen, and no typing or other action is possible. Because updating of ordinary dynamic output locks up the front end, it is important to restrict the expressions you put inside `Dynamic` to things that will evaluate relatively quickly (preferably, within a second or so). Fortunately computers, and *Mathematica*, are fast, so a wide range of functions, including complex 2D and 3D plots, can easily be evaluated in a fraction of a second.

To avoid locking up the front end for good, dynamic evaluations are internally wrapped in `TimeConstrained`, with a timeout value of, by default, 5 seconds. (This can be changed with the `DynamicEvaluationTimeout` option.) In certain extreme cases, `TimeConstrained` can fail to abort the calculation, in which case the front end will, a few seconds later, put up a dialog box allowing you to terminate dynamic updating until the offending output has been deleted.

Fortunately there is an alternative if you need to have something slow in a `Dynamic`. The option `SynchronousUpdating → False` allows the dynamic to be evaluated in a way that does not lock up the front end. During evaluation of such an asynchronous `Dynamic` the front end continues operating as usual, but the main Shift+Return evaluation queue is occupied evaluating the `Dynamic`, so further Shift+Return evaluations will wait until the `Dynamic` finishes. (Normal synchronous `Dynamic` evaluations do not interfere with Shift+Return evaluations.)

Evaluate this example, and you will see a gray placeholder rectangle for about 10 seconds, after which the result will be displayed.

```
In[62]:= Dynamic[{DateList[], Pause[10]; DateList[]}, SynchronousUpdating → False]
Out[62]= {{2009, 1, 2, 15, 47, 41.977359}, {2009, 1, 2, 15, 47, 51.981876}}
```

Importantly, during that 10-second pause you are free to continue working on other things in the front end.

"Advanced Dynamic Functionality" gives more details about the differences between synchronous and asynchronous dynamic evaluations. In general, you should not plan to use asynchronous ones unless is it absolutely necessary. They do not update as quickly, and can interact in a very surprising, though not technically incorrect, way with controls and other synchronous evaluations.

Further Reading

The implementation details behind `Dynamic` and `DynamicModule` are worth understanding if you plan to use complex constructions, particularly those involving nested `Dynamic` expressions. This is discussed in "Advanced Dynamic Functionality".

Advanced Dynamic Functionality

"Introduction to Manipulate" and "Introduction to Dynamic" provide most of the information you need to use *Mathematica*'s interactive features accessible through the functions `Manipulate`, `Dynamic`, and `DynamicModule`. This tutorial gives further details on the workings of `Dynamic` and `DynamicModule` and describes advanced features and techniques for achieving maximum performance for complex interactive examples.

Many examples in this tutorial display a single output value and use `Pause` to simulate slow calculations. In real life, you will instead be doing useful computations and displaying sophisticated graphics or large tables of values.

Please note that this is a hands-on tutorial. You are expected to actually evaluate each of the input lines as you reach them in your reading, and watch what happens. The accompanying text will not make sense without evaluating as you read.

Module versus DynamicModule

`Module` and `DynamicModule` have similar syntax and in many respects behave similarly, at least at first glance. They are, however, fundamentally different in such areas as when their variables are localized, where the local values are stored, and in what universe the variables are unique.

`Module` works by replacing all occurrences of its local variables with new, uniquely named variables, constructed so that they do not conflict with any variables in the current session of the *Mathematica* kernel.

You can see the names of these localized variables by allowing them to "escape" the context of the module without having been assigned a value.

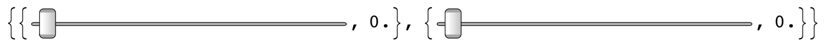
```
In[3]:= Module[{x}, x]

Out[3]= x$651
```

The local variables can be updated dynamically just like any other variables.

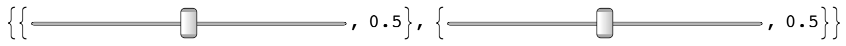
```
In[4]:= Module[{x}, Dynamic[x]]

Out[4]= x$653
```

That is why sliders inside Module seem to work just as well as sliders inside DynamicModule.

```
In[9]:= Table[Module[{x = .5}, {Slider[Dynamic[x]], Dynamic[x]}], {2}]
```

Out[9]=


```
In[10]:= Table[DynamicModule[{x = .5}, {Slider[Dynamic[x]], Dynamic[x]}], {2}]
```

Out[10]=


Both examples produce seemingly independent sliders that allow separate settings of separate copies of the variable x. The problem with sliders inside Module is that a different kernel session may coincidentally share the same localized variable names. So if this notebook is saved and then reopened sometime later, the sliders may "connect" to variables in some other Module that happen to have the same local variables at that time.

This will not happen with the sliders inside DynamicModule because DynamicModule waits to localize the variables until the object is displayed in the front end and generates local names that are unique to the current session of the front end. Localization happens when DynamicModule is first created as output and then repeats anew each time the file that contains DynamicModule is opened, so there can never be a name conflict among examples generated in different sessions.

Variables generated by Module are purely kernel session variables; when the kernel session ends, the values are irretrievably lost. DynamicModule, on the other hand, generates a structure in the output cell that is responsible for maintaining the values of the variables, allowing them to be saved in files. This is a somewhat subtle concept, best explained by way of two analogies. First, you can think of DynamicModule as a sort of persistent version of Module.

Consider this command.

```
In[5]:= Module[{x = 2, y, z},
         x = 4;
         y = x^2;
         x = 8;
         z = x^3;
        ]
```

The module in this example evaluates a series of expressions in order, and from one line to the next the values of all the local module variables are preserved (obviously). You can have as many lines as you like in the compound expression, but they all have to be there at the start; once the Module has finished execution, it evaporates along with all its local variables.

`DynamicModule`, on the other hand, creates an environment in which evaluations of expressions in `Dynamic` that appear within the body of the `DynamicModule` are like additional lines in the compound expression in the previous example. From one dynamic update to the next the values of all the variables are preserved, just as if the separate evaluations were separate lines in a compound expression, all within the local variable context created by `DynamicModule`.

This preservation of variable values extends not just to subsequent dynamic evaluations within the same session, but to all future sessions. Because all the local variable values are stored and preserved in the notebook file, if the notebook is opened in an entirely new session of *Mathematica*, the values will still be there, and dynamic updates will resume just where they left off. `DynamicModule` is like an indefinitely extendable `Module`.

Another way to think about the difference between `Module` and `DynamicModule` is that while `Module` localizes its variables for a certain duration of *time* (while the body of the module is being evaluated), `DynamicModule` localizes its variables for a certain area of *space* in the output.

As long as that space of the output remains in existence, the values of the variables defined for it will be preserved, allowing them to be used in subsequent evaluations of `Dynamic` expressions within the scope (area) of the `DynamicModule`. Saving the output into a file puts that bit of real estate into hibernation, waiting for the moment when the file is opened again. (In computer science terms, this is sometimes referred to as a freeze-dried or serialized object.)

The ability of `DynamicModule` to preserve state across sessions is also a way of extending the notion of editing in a file. Normally when you edit text or expressions in a file, save the file, and reopen it, you expect it to open the way you left it. Editing means changing the contents of a file.

Ordinary kernel variables do not have this property; if you make an assignment to x, then quit and restart *Mathematica*, x does not have that value anymore. There are several reasons for this, not least of which is the question of *where* the value of x should be saved.

`DynamicModule` answers this question by defining a specific location (the output cell) where values of specific variables (the local variables) should be preserved. Arbitrary editing operations, like moving a slider, typing in an input field, or dragging a dynamic graphics object, change the values of the local variables. And since these values are automatically preserved when the file is saved, the sliders, and other objects, open exactly where they were left. Thus `DynamicModule` lets you make any quantity editable in the same way that text and expressions can be edited and saved in notebook files.

Front End Ownership of DynamicModule Variable Values

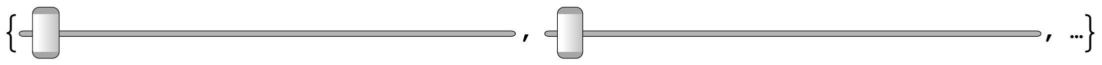
Ordinary variables in *Mathematica* are owned by the kernel. Their values reside in the kernel, and when you ask *Mathematica* to display the value in the front end, a transaction is initiated with the kernel to retrieve the value. The same is true of dynamic output that refers to the values of ordinary variables.

Consider this example.

```
In[6]:= x = 0;
```

```
In[6]:= Table[Slider[Dynamic[x]], {500}]
```

Out[7]=



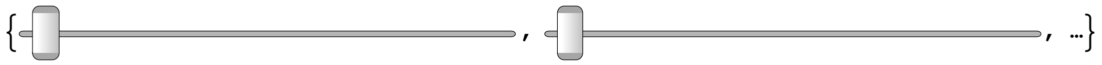
When one slider is moved, the other 499 move in sync with it. This requires 500 separate transactions with the kernel to retrieve the value of x. (The semantics of *Mathematica* are complex enough that there is no guarantee that evaluating x several times in a row will actually return the same value each time: it would not be possible for the front end to improve efficiency by somehow sharing a single value retrieved from the kernel with all the sliders.)

Variables declared with `DynamicModule`, on the other hand, are owned by the front end. Their values reside in the front end, and when the front end needs a value, it can be retrieved locally with very little overhead.

The following example thus runs noticeably faster.

```
In[8]:= DynamicModule[{x = 0}, Table[Slider[Dynamic[x]], {500}]]
```

Out[8]=



If a complex function is applied to such a variable, its value must of course be sent to the kernel. This happens transparently, with each side of the system being kept informed on a just-in-time basis of any changes to variable values.

Whether it is better to use a normal kernel variable or a `DynamicModule` variable in a given situation depends on a number of factors. The most important is the fact that values of all

`DynamicModule` variables are saved in the file when the notebook is saved. If you need a value preserved between sessions, it must be declared in a `DynamicModule`. On the other hand, a temporary variable holding a large table of numbers, for example, might be a poor choice for a `DynamicModule` variable as it could greatly increase the size of the file. It is quite reasonable to nest a `Module` inside a `DynamicModule` and vice versa, or to partition variables between the front end and kernel.

In many situations the limiting factor in performance is the time needed to retrieve information from the kernel: by making variables local to the front end, speed can sometimes be increased dramatically.

Automatic Updates of Dynamic Objects

The specification for dynamic output is simple: `Dynamic[`*expr*`]` should always display the value you would get if you evaluated *expr* now. If a variable value, or some other state of the system, changes, the dynamic output should be updated immediately. Of course, for efficiency, not every dynamic output should be reevaluated every time any variable changes. It is critical that dependencies be tracked so that dynamic outputs are evaluated only when necessary.

Consider these two expressions.

```
In[9]:=  Dynamic[a + b + c]
Out[9]= a + b + c
```

```
In[10]:=  Dynamic[If[a, b, c]]
Out[10]= If[a, b, c]
```

The first expression might change its value any time the value of *a*, *b*, or *c* changes, or if any patterns associated with *a*, *b*, or *c* are changed. The second expression depends on *a* and *b* (but not *c*) while *a* is `True` and on *a* and *c* (but not *b*) while *a* is `False`. If *a* is neither `True` nor `False`, then it depends only on *a* (because the `If` statement returns unevaluated).

Figuring out these dependencies a priori is impossible (there are theorems to this effect), so instead the system keeps track of which variables or other trackable entities are actually encountered during the process of evaluating a given expression. Data is then associated with those variable(s) identifying which dynamic expressions need to be notified if the given variable receives a new value.

An important design goal of the system is to allow monitoring of variable values by way of dynamic output referencing them, without imposing any more load than absolutely necessary on the system, especially if the value of the variable is being changed rapidly.

Consider this simple example.

```
In[11]:= Dynamic[x]
Out[11]= x

In[13]:= Do[x, {x, 1, 5 000 000}]
```

When the dynamic output is created, it is evaluated, and the symbol x is tagged with information identifying the output that needs to be updated if its value should be changed.

When the loop is started and x is first given a new value, the data associated with it is consulted, and the front end is notified that the dynamic output needs to be updated. The data associated with x is then deleted. Essentially the system forgets all about the dynamic output, and subsequent assignments in the loop incur absolutely no speed penalty because of the existence of a dynamic output monitoring the value of x.

Much later (on a computer time scale; only a fraction of a second on a human time scale) when the screen is next redrawn and the dynamic output containing the reference to x is reevaluated, the connection between the dynamic output and the variable x is noticed again, and the association is reestablished.

Meanwhile the loop has continued to run. The next time the assignment is done after the screen is updated, another notification will be sent to the front end, and the process repeats.

By default, dynamic outputs triggered by changes in variable values are updated no faster than twenty times per second (this rate can be changed with the `SystemOption` `"DynamicUpdateInterval"`). In the previous example you will typically see the value jump by tens or hundreds of thousands with each update (more the faster your computer is), and the overall speed of the computation is slowed down by only a percent or two, nearly zero if you have a multiprocessor system.

You might expect that having a dynamic output monitoring the value of a symbol that is being changed rapidly in a tight loop would slow that loop down significantly. But the overhead is in fact zero-order in the rate at which the variable is changed, and in practice is usually minimal.

Dynamic outputs are only updated when they are visible on screen. This optimization allows you to have an open-ended number of dynamic outputs, all changing constantly, without incurring an open-ended amount of processor load. Outputs that are scrolled off-screen, above or below the current document position, will be left unexamined until the next time they are scrolled on-screen, at which point they are updated before being displayed. (Thus the fact that they stopped updating is not normally apparent, unless they have side effects, which is discouraged in general.)

Dynamic output can depend on things other than variables, and in these cases tracking is also done carefully and selectively.

This gives a rapidly updated display of the current mouse position in screen coordinates.

```
In[14]:= Dynamic[MousePosition[]]
Out[14]= {1058, 553}
```

As long as the output is visible on screen, there will be a certain amount of CPU activity any time the mouse is moved, because this particular dynamic output is being redrawn immediately with every movement of the mouse. But if it is scrolled off-screen, the CPU usage will vanish.

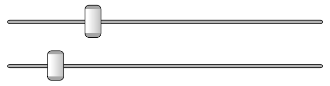
Refresh

Normally, dynamic output is updated whenever the system detects any reason to believe it might need to be (see "Automatic Updates of Dynamic Objects" for details about what this means). `Refresh` can be used to modify this behavior by specifying explicitly what should or should not trigger updates.

This updates when either slider is moved.

```
In[15]:= DynamicModule[{x, y}, Column[{
            Slider[Dynamic[x]],
            Slider[Dynamic[y]],
            Dynamic[{x, y}]}]]
```

Out[15]=



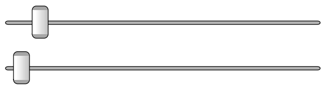
```
{0.245, 0.11}
```

`Refresh` with a `TrackedSymbols` option can be used to specify a list of those symbols that should be tracked, with all other reasons for updating being ignored.

This updates only when x changes, ignoring changes in y.

```
In[16]:= DynamicModule[{x, y}, Column[{
            Slider[Dynamic[x]],
            Slider[Dynamic[y]],
            Dynamic[Refresh[{x, y}, TrackedSymbols → {x}]]}]]
```

Out[16]=



```
{0.065, 0.}
```

When you move the second (y) slider, nothing happens, but when you move the first slider, the expression is updated to reflect the current value of both variables. You might say that after moving the second slider, the dynamic output is wrong, since it does not reflect the current state of the system. But that is essentially the whole reason for the existence of the `Refresh` command. It allows you to override the system's mandate to always update dynamic output any time it is potentially out of date.

The setting `TrackedSymbols -> Automatic` can be used to track only those symbols that occur explicitly (lexically) in the expression given in the first argument to `Refresh`. For example, if you use a function that depends on a global variable that does not occur lexically inside `Refresh`, changes to the value of the global variable will not cause updating, when normally they would.

`Refresh` can also be used to cause updates at regular time intervals. It is important to understand that this is *not* a feature that should be used lightly. It is fundamental to the design of `Dynamic` that it does not need to update on any fixed schedule, because it simply always updates immediately whenever doing so would be useful. But there are some situations where this either cannot, or just unfortunately does not, happen.

One potentially vexing case is `RandomReal`. Every time you evaluate `RandomReal[]`, you get a different answer, and you might think that `Dynamic[RandomReal[]]` should therefore constantly update itself as fast as possible. But this would normally not be useful, and would in fact have negative consequences for a number of algorithms that use randomness internally (e.g., a Monte Carlo integration inside `Dynamic` should probably not update constantly simply because it will, in fact, give a slightly different answer each time).

For this reason, `RandomReal[]` is not "ticklish," in the sense that it does not trigger updates. If you want to see new random numbers, you have to use `Refresh` to specify how frequently you want the output updated. Another example of non-ticklish functions are file system operations.

This gives you a new number every second.

```
In[17]:= Dynamic[Refresh[RandomReal[], UpdateInterval → 1]]
Out[17]= 0.722136
```

This is not updated automatically.

```
In[18]:= Dynamic[FileByteCount[ToFileName[
             {$TopDirectory, "SystemFiles", "FrontEnd", "Palettes"}, "BasicMathInput.nb"]]]
Out[18]= $Failed
```

In the unlikely event that the file containing the **BasicMathInput** palette changes size, this `Dynamic` will not be updated. If you want to monitor the size of a file, you need to use `Refresh` to specify a polling interval. (On sufficiently advanced operating systems it would theoretically be possible for *Mathematica* to efficiently receive notifications of file system activity, and future versions of *Mathematica* might in fact update such expressions automatically. As with other `Dynamic` expressions, automatic correctness is always the goal.)

Finally, several functions you might think would trigger dynamic updates in fact do not: for example, `DateList` and `AbsoluteTime`. As with `RandomReal`, it would cause more trouble than it is worth for these functions to automatically trigger updates, and `Refresh` can trivially be used to create clock-like objects. The function `Clock` is intended specifically as a time-based function that *is* ticklish.

This updates approximately every second.

```
In[19]:= Dynamic[Refresh[DateList[], UpdateInterval → 1]]
Out[19]= {2009, 1, 2, 17, 3, 24.196806}
```

This updates without an explicit `Refresh`.

```
In[26]:= Dynamic[Clock[{1, 10}]]
```

Nesting Refresh

In the Refresh section examples, `Refresh` is always the outermost function inside `Dynamic`. You might almost wonder why its options are not simply options to `Dynamic`. But in fact it is often important to place `Refresh` as deeply in the expression as possible, especially if it specifies a time-based updating interval.

Consider this example.

```
In[20]:= DynamicModule[{showclock = True}, {Checkbox[Dynamic[showclock]],
           Dynamic[If[showclock, Refresh[DateList[], UpdateInterval → 0.05], "No clock"]]}]

Out[20]= {☐, No clock}
```

When the checkbox is checked, `Refresh` is causing frequent updating of the clock, and CPU time is being consumed to keep things up-to-date. When the checkbox is unchecked, however, the `Refresh` expression is no longer reached by evaluation, the output remains static, and no CPU time is consumed. If `Refresh` were wrapped around the whole expression inside `Dynamic`, CPU time would be consumed constantly, even if the clock were not being displayed. The words "No clock" would be constantly refreshed, pointlessly. (This refreshing is not visible; there is no flicker of the screen, but CPU time is being consumed nevertheless.)

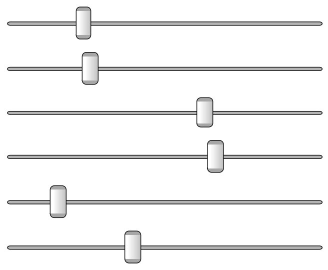
Nesting Dynamic

`Dynamic` expressions can be nested, and the system takes great care to update them only when necessary. Particularly when the contents of a `Dynamic` contain further interactive elements, it is important to keep track of what will stay static and what will update, when a given variable is changed.

Consider this example.

```
In[21]:= DynamicModule[{n = 5, data = Table[RandomReal[], {20}]},
          Column[{
            Slider[Dynamic[n], {1, 20, 1}],
            Dynamic[Column[Table[With[{i = i}, Slider[Dynamic[data[[i]]]]], {i, n}]]]}]]
```

Out[21]=



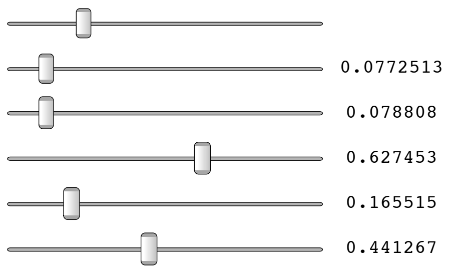
The position of the first slider determines the number of sliders underneath it, and each of those sliders in turn is connected to the value of one element of a list of data. Because the number of sliders is variable, and changes dynamically in response to the position of the first slider, the table that generates them needs to be inside `Dynamic`.

The example works, but now suppose you want to display the value of each number in the list next to its slider.

You might at first try this.

```
In[22]:= DynamicModule[{n = 5, data = Table[RandomReal[], {20}]},
          Column[{
            Slider[Dynamic[n], {1, 20, 1}],
            Dynamic[Grid[Table[With[{i = i},
                {Slider[Dynamic[data[[i]]]], data[[i]]}], {i, n}]
             ]]}]]
```

Out[22]=



```
0.0772513
0.078808
0.627453
0.165515
0.441267
```

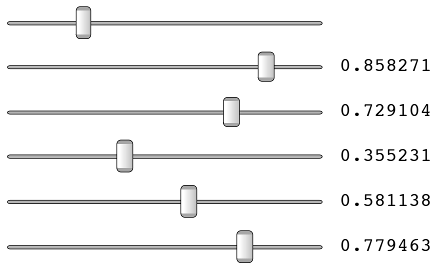
Now any time you click one of the lower sliders, it moves only one step, then stops. The problem is that the *data*[[*i*]] expressions in the second column of the grid are creating a dependency in the outer `Dynamic` on the values in *data*.

As soon as *data* changes, the contents of the outer `Dynamic`, including the slider you are trying to drag, are destroyed and replaced with a nearly identical copy (in which the displayed value of one of the *data*[[*i*]] has been changed). In other words, the act of dragging the slider destroys it, preventing any further activity.

The solution to this is to prevent the outer `Dynamic` from depending on the value of data, by making sure that all occurrences of data in the expression are wrapped in `Dynamic`.

```
In[23]:= DynamicModule[{n = 5, data = Table[RandomReal[], {20}]},
          Column[{
            Slider[Dynamic[n], {1, 20, 1}],
            Dynamic[Grid[Table[With[{i = i},
                {Slider[Dynamic[data[[i]]]], Dynamic[data[[i]]]}], {i, n}]
             ]]}]]
```

Out[23]=



```
0.858271
0.729104
0.355231
0.581138
0.779463
```

Now you can drag any of the sliders and see dynamically updated values. This works because the outer `Dynamic` now depends only on the value of n, the number of sliders, not on the value of $data$. (Technically this is because `Dynamic` is `HoldFirst`: when it is evaluated, the expression in its first argument is never touched by evaluation, and therefore no dependencies are registered.)

When building large, complex interfaces using multiple levels of nested `Dynamic` expressions, these are important issues to keep in mind. *Mathematica* works hard to do exactly the right thing even in the most complex cases. For example, the output of `Manipulate` consists of a highly complex set of interrelated and nested `Dynamic` expressions: if the dependency tracking system did not work correctly, `Manipulate` would not work right.

Synchronous versus Asynchronous Dynamic Evaluations

Mathematica consists of two separate processes, the front end and the kernel. These really are separate processes in the computer science sense of the word: two independent threads of execution with separate memory spaces that show up separately in a CPU task monitor.

The front end and kernel communicate with each other through several *MathLink* connections, known as the main link, the preemptive link, and the service link. The main and preemptive links are pathways by which the front end can send evaluation requests to the kernel, and the kernel can respond with results. The service link works in reverse, with the kernel sending requests to the front end.

The main link is used for Shift+Return evaluations. The front end maintains a queue of pending evaluation requests to send down this link. When you use Shift+Return on one or more input cells, they are all added to the queue, and then processed one by one. At any one time, the kernel is only aware of a single main link evaluation, the one it is currently working on (if any). In the meantime, the front end remains fully functional; you can type, open and save files, and so on. There is no arbitrary limit on how long a main link evaluation can reasonably take. People routinely do evaluations that take days to complete.

The preemptive link works the same way as the main link in the sense that the front end can send an evaluation to it and get an answer, but it is administered quite differently on both ends. On the front end side, the preemptive link is used to handle normal `Dynamic` updates. There is no queue; instead, the front end sends one evaluation at a time and waits for the result before continuing with its other work. It is thus important to limit preemptive link evaluations to a couple of seconds at most. During any preemptive link evaluation, the front end is completely locked up, and no typing or other actions are possible.

On the kernel side, evaluation requests coming from the preemptive link are given priority over evaluations from the main link, including the current running main link evaluation (if any). If an evaluation request comes from the preemptive link while the kernel is processing a main link evaluation, the main link evaluation is halted at a safe point (usually within microseconds). The preemptive link evaluation is then run to completion, after which the main link evaluation is restarted and allowed to continue as before. The net effect is similar to, though not the same as, a threading mechanism. Multiple fast preemptive link evaluations can be executed during a single long, slow main link evaluation, giving the impression that the kernel is working on more than one problem at a time.

Preemptive link evaluations can change the values of variables, including those being used by a main link evaluation running at the same time. There is no paradox here, and the interleaving is done in a way that is entirely safe, though it can result in some fairly peculiar behavior until you understand what is going on.

For example, evaluate this to get a slider.

```
In[24]:= Slider[Dynamic[x]]
```

Out[24]=

Then evaluate this command, and during the ten seconds it takes to finish, drag the slider around randomly.

```
In[25]:= Table[Pause[1]; x, {10}]
Out[25]= {0, 0.2, 0., 0., 0., 0.5, 0.675, 0., 0., 0.}
```

You will not see anything happening (other than the slider moving) but when the second evaluation finishes, you will see that it has recorded ten different values of x, representing the positions the slider happened to be at during the ten points at which x was evaluated to build the list.

`Dynamic` normally uses the preemptive link for its evaluations. Evaluation is synchronous, and the front end locks up until it is finished. This is unavoidable in some cases, but can be suboptimal in others. By setting the option `SynchronousUpdating -> False`, you can tell the front end to use the main link queue, rather than the preemptive link. The front end then displays a gray box placeholder until it receives the response from the kernel.

In this case, the default (synchronous) update is appropriate because the front end needs to know the result of evaluating the `Dynamic[x]` for drawing with the correct font size.

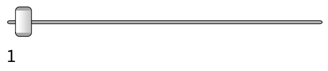
```
In[26]:= DynamicModule[{x = 12},
          {Slider[Dynamic[x], {10, 100}], Style["Hello", FontSize → Dynamic[x]]}]
```

Out[26]=

Here, the output cell is drawn before the second dynamic expression finishes. A gray box placeholder persists for one second until the result is known. Reevaluate the example to see the gray box again.

```
In[27]:= DynamicModule[{n = 1}, Column[{Slider[Dynamic[n], {1, 10}],
            Dynamic[Pause[n]; n, SynchronousUpdating → False]}]]
```

Out[27]=



Clicking the slider will update the display with a delay of between one and ten seconds. Notice that the cell bracket is outlined, just as if the cell were being Shift+Return evaluated. This is an indication that the evaluation is queued, and that you can continue with other work in the front end while the evaluation is progressing.

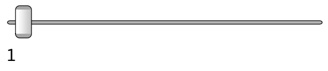
Asynchronous updating is useful for displaying full `Dynamic` subexpressions when it is possible to draw a screen around them and fill in their value later, in much the same way a web browser draws text around an image that is inserted later when it finishes downloading.

Why not always use asynchronous `Dynamic` expressions? There are several reasons. First, they are queued so that, by definition, they do not operate while another Shift+Return evaluation is underway. This is not the case for normal (synchronous) updates.

A synchronous `Dynamic` updates smoothly even if the `Pause` command above is running.

```
In[28]:= Pause[20]

In[29]:= DynamicModule[{n = 1}, Column[{Slider[Dynamic[n], {1, 10}], Dynamic[n]}]]
```

Out[29]=

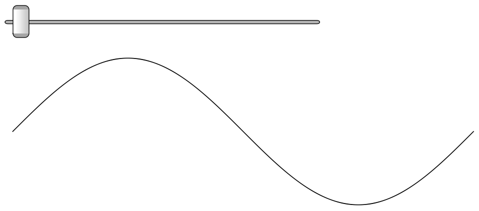


Also, many controls need to be synchronous in order to be responsive to mouse actions. Making them asynchronous may cause potentially strange interactions with other controls.

Here is a problematic example.

```
In[30]:= n = 1;
         Column[{Slider[Dynamic[n], {1, 10}],
           Dynamic[Graphics[Line[Table[{x, Sin[n x]}, {x, 0, 2 Pi, 0.0001}]]],
            SynchronousUpdating → False]}]
```

Out[31]=



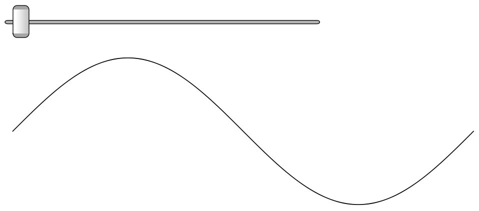
Move the slider around rapidly, and you will end up with a choppy, distorted sine wave, because the value of *n* changed during the evaluation of the `Table` command. This is the correct, expected behavior, but it is probably not what you wanted.

This problem does not occur if you use synchronous `Dynamic` expressions, generally does not happen with `DynamicModule` local variables, and can be avoided by storing the value of any potentially changing variables into a second variable before starting the asynchronous evaluations.

This fixes the problem.

```
In[32]:= n = 1;
         Column[{Slider[Dynamic[n], {1, 10}], Dynamic[
            Module[{n1 = n}, Graphics[Line[Table[{x, Sin[n1 x]}, {x, 0, 2 Pi, 0.0001}]]]],
            SynchronousUpdating → False]}]
```

Out[33]=



ControlActive and SynchronousUpdating→Automatic

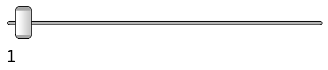
As a general rule, if you have a `Dynamic` that is meant to respond interactively to the movements of a slider or other continuous-action control, it should be able to evaluate in under a second, preferably well under. If the evaluation takes longer than that, you are not going to get satisfactory interactive performance, whether the `Dynamic` is updating synchronously or asynchronously.

But what if you have an example that simply cannot finish evaluating fast enough, yet you want to be able to make it respond to a slider? One option is to use asynchronous updating and simply accept that you will not get real-time interactive performance. If that is what you want to do, setting `ContinuousAction -> False` in the slider or other control is a good idea; that way you get only one update after the control is released, avoiding the starting up of potentially lengthy evaluations in the middle of a drag, before you have arrived at the value you want to stop at.

The cell bracket becomes outlined, indicating evaluation activity, only after you release the slider.

```
In[34]:= DynamicModule[{n = 1},
          Column[{Slider[Dynamic[n], {1, 10}, ContinuousAction → False],
            Dynamic[Pause[n]; n, SynchronousUpdating → False]}]]
```

Out[34]=



Another, much better solution is to provide a fast-to-compute preview of some sort during the interactive control dragging operation, then compute the full, slow output when the control is released. Several features exist specifically to support this.

The first is the function `ControlActive`, which returns its first argument if a control is currently being dragged, and its second argument if not. Unlike `Dynamic`, `ControlActive` is an ordinary function that evaluates in the kernel, returning one or the other of its arguments immediately. It can be embedded inside functions or option values.

The second feature is an option setting `SynchronousUpdating -> Automatic` for `Dynamic`, which makes the `Dynamic` synchronous when a control is being dragged, and asynchronous when the control is released. Together, these two features can be used to implement a fast, synchronously updated display to be used while a control is being dragged, along with a slower, asynchronously updated display when it is released.

The displayed text changes depending on whether or not the slider is being dragged.

```
In[35]:= DynamicModule[{n = 1}, Column[{Slider[Dynamic[n], {1, 10}],
            Dynamic[{n, ControlActive["Active", "Not Active"]}]}]]
```

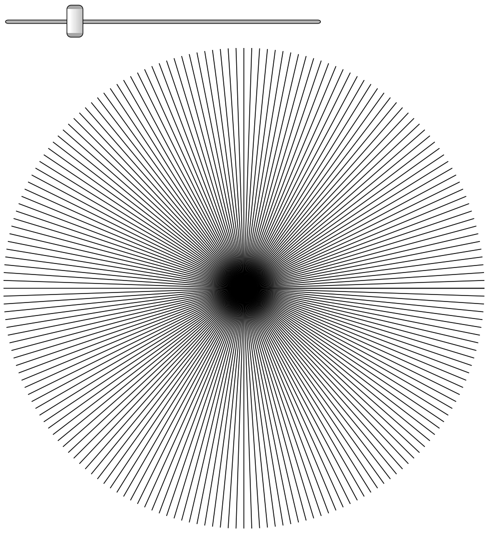
Out[35]=

```
{1, Not Active}
```

A simple number is displayed, synchronously, while the slider is being dragged, and when it is released, a graphic is generated asynchronously.

```
In[36]:= DynamicModule[{n = 3},
          Column[{Slider[Dynamic[n], {3, 1000, 1}], Dynamic[Graphics[ControlActive[Inset[n,
                {0, 0}], Line[Table[{{0, 0}, {Cos[t], Sin[t]}}, {t, 0., 2 Pi, 2 Pi / n}]]],
              ImageSize → 300, PlotRange → 1], SynchronousUpdating → Automatic]}]]
```

Out[36]=



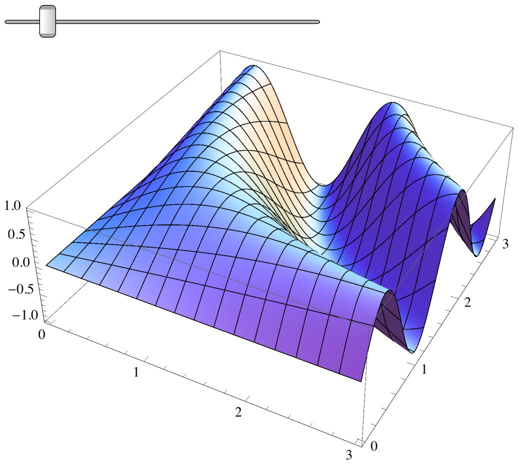
This example shows that the front end can remain responsive no matter how long the final display takes to compute and that the preview and the final display can be completely different.

Of course, in most cases, you will want a preview that is some kind of reduced, thinned out, skeletal, or other elided form of the final display. Then the crude form can be fast enough to give a smooth preview, and the computation of the final version, even if it takes awhile, does not block the front end. In fact, this behavior is so useful that it is the default in `Plot3D` and other plotting functions.

This displays a 3D plot with a very small number of plot points while the control is being dragged and then refines the image with a large number of plot points when the control is released.

```
In[37]:= DynamicModule[{n = 1},
          Column[{Slider[Dynamic[n], {1, 5}], Dynamic[Plot3D[Sin[n x y], {x, 0, 3},
              {y, 0, 3}, PlotPoints → ControlActive[10, 100], MaxRecursion → 0],
            SynchronousUpdating → Automatic]}]]
```

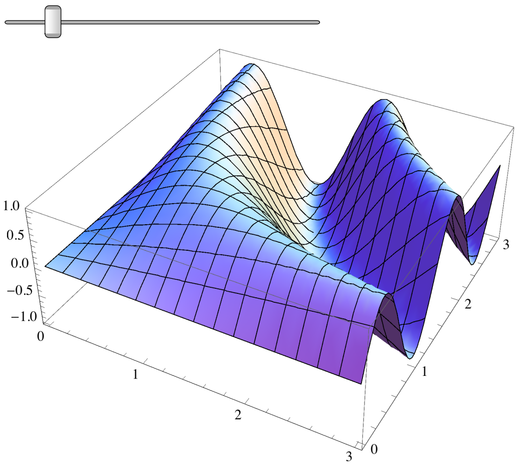
Out[37]=



By default, `Plot3D` produces a similar preview, though with a somewhat less extreme spread of quality.

```
In[38]:= DynamicModule[{n = 1}, Column[{Slider[Dynamic[n], {1, 5}], Dynamic[
            Plot3D[Sin[n x y], {x, 0, 3}, {y, 0, 3}], SynchronousUpdating → Automatic]}]]
```

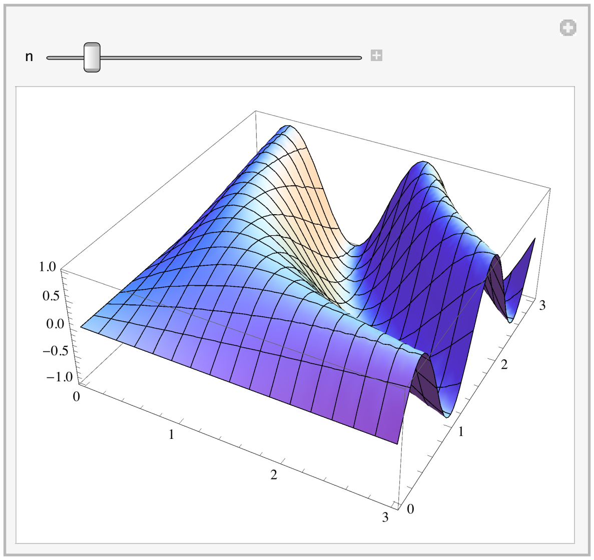
Out[38]=



In addition, `Manipulate` uses `SynchronousUpdating -> Automatic` in `Dynamic` by default so the example becomes as simple as it can be.

```
In[39]:= Manipulate[Plot3D[Sin[n x y], {x, 0, 3}, {y, 0, 3}], {n, 1, 5}]
```

Out[39]=



You may have noticed one subtlety. When the output of either of the above three examples is first placed in the notebook, you see a crudely drawn (control-active state) version, followed shortly thereafter by a refined (control-inactive) version. This is intentional: the system is providing a fast preview so you see something rather than just a gray rectangle. The first update is done synchronously, just as if a control were being dragged.

This preview-evaluation behavior is examined in more detail in the next section.

ImageSizeCache in Dynamic

`ImageSizeCache` is an option to `Dynamic` that specifies a rectangular size to be used in displaying a `Dynamic` whose value has not yet been computed. It is normally not specified in input, but is instead generated automatically by the front end and saved in files along with the `Dynamic` expression.

The interaction of `ControlActive`, `SynchronousUpdating`, and `ImageSizeCache` is subtle, complex, and very useful. The first two constructs are explained in ControlActive and SynchronousUpdating→Automatic. The remaining part is explained here.

Note first that `Dynamic` expressions with the default value of `SynchronousUpdating -> True` will never have a chance to use the value of their `ImageSizeCache` option, because they are always computed before being displayed, and, once computed, the actual image size will be used.

On the other hand, `Dynamic` expressions with `SynchronousUpdating → False` will be displayed as a gray rectangle while they are being computed for the first time. In that case, the size of the rectangle is determined by the value of the `ImageSizeCache` option. This allows the surrounding contents of the notebook to be drawn in the right place, so that when the `Dynamic` finishes updating, there is no unnecessary flicker and shifting around of the contents of the notebook. (Users of HTML will recognize this as the analog of the width and height parameters of the `img` tag.)

It is generally not necessary to specify the `ImageSizeCache` option explicitly, because the system will set it automatically as soon as the value of the `Dynamic` is computed successfully. (The computed result is measured, and the actual size copied into the `ImageSizeCache` option.) This automatically computed value is preserved if the `Dynamic` output is saved in a file.

Consider the following input.

```
In[40]:= Dynamic[Pause[3]; Style["Hello", 100], SynchronousUpdating → False]
```

```
Out[40]= Hello
```

When the input expression is evaluated, a small gray rectangle appears; because this `Dynamic` has never been evaluated, there is no cache of its proper image size, and a default small size is used.

Three seconds later, the result arrives, and the dynamic output is displayed. At this point an actual size is known, and is copied to the `ImageSizeCache` option. You can see the value by clicking anywhere in the output cell and choosing **Show Expression** from the **Cell** menu. (This shows you the underlying expression representing the cell, exactly as it would appear in the notebook file if you were to save this cell.) Note the presence of an `ImageSizeCache` option.

Now type a space in some innocuous place in the raw cell expression (to force a reparsing of the cell contents), and choose **Show Expression** again to reformat the cell. This time you will see a gray rectangle the size of the final output for three seconds, followed by the proper output. This is also what you would see if you opened a notebook containing previously saved, asynchronous dynamic output.

The behavior of the setting `SynchronousUpdating -> Automatic` is similar, but subtly different. As we saw in the examples in "ControlActive and SynchronousUpdating→Automatic", with the `Automatic` setting, a synchronous preview-evaluation is done when the output is first placed, to provide a (hopefully) rapid display of the contents of the `Dynamic` expression before the slower, asynchronous value is computed. Because the first evaluation is synchronous, no gray rectangle is ever displayed.

But this preview evaluation is done only if the `ImageSizeCache` option is not present. A `Dynamic` with `SynchronousUpdating -> Automatic` and an `ImageSizeCache` option specifying explicit dimensions will not do a synchronous preview evaluation, and will instead display a gray rectangle (of the correct size) pending the result of the first asynchronous evaluation.

This may seem like baffling behavior at first, until you consider the practical effect of it. Generally speaking, `Dynamic` expressions will always have an `ImageSizeCache` option (created automatically by the front end) except for the very first time they appear, when they are originally placed as output from an evaluation. Any time they are opened from a file they will have a known, cached size.

In `Manipulate`, which accounts for the vast majority of dynamic outputs, the default setting is `SynchronousUpdating -> Automatic` and the described behavior lets the output show up cleanly with a preview image in place when it is first generated. When a file containing dozens of `Manipulate` outputs is opened, you will get a useful behavior that is familiar from web browsers: the text displays immediately, and graphics (and other dynamic content) fill in later as fast as they are able. So you can scroll through a file rapidly, without any delay associated with precomputing potentially many preview images before the first page of the file can be displayed.

If the initial evaluations when the `Manipulate` output was first placed were not synchronous, there would be flicker and resizing/shifting of the surroundings, because the size would not be known. But when the `Manipulate` output is opened from a file, the size is known, and the final output can be placed smoothly without flicker.

One-Sided Updating of ControlActive

After evaluating in the kernel, `ControlActive` can trigger an update of the `Dynamic` containing it, but in a highly asymmetric fashion, only when it is going from the active to the inactive state. When making a transition in the other direction, from inactive to active, `ControlActive` does not trigger any update on its own.

The reason for this somewhat unusual behavior is that `ControlActive` is a completely global concept. It returns the active state if any control anywhere in *Mathematica* is currently being dragged—even controls that have nothing to do with a particular `Dynamic` that happen to contain a reference to `ControlActive`. If `ControlActive` caused updates on its own, then as soon as you clicked any control, all `Dynamic` expressions containing references to `ControlActive` (e.g., a default dynamic `Plot3D` output) would immediately update, which would be entirely pointless. Instead, only those outputs that have some other reason for updating will pick up the current value of `ControlActive`.

On the other hand, when the control is released, it is desirable to fix up any outputs that were drawn in control-active form, to give them their final polished appearance. Thus, when `ControlActive` is going into its inactive state, it needs to, on its own, issue updates to any `Dynamic` expression that may have been drawn in the active state.

Dragging the slider does not change the Active/Inactive display because `ControlActive` does not trigger updates on its own.

```
In[49]:= DynamicModule[{x},
          {Slider[Dynamic[x]], Dynamic[ControlActive["Active", "Inactive"]]}]
```

Out[49]=

```
{, Inactive}
```

This Active/Inactive display updates because x in the dynamic output changes.

```
In[41]:= DynamicModule[{x},
          {Slider[Dynamic[x]], Dynamic[{x, ControlActive["Active", "Inactive"]}]}]
```

Out[41]=

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

Watch carefully what happens when you click the slider. If you click and hold the mouse without moving it, the display will remain `Inactive`. But as soon as you move it, the display updates to `Active`. This is happening because x changed, causing the `Dynamic` as a whole to update, thus picking up the current state of `ControlActive`.

Now carefully release the mouse button without moving the mouse. Note that the display does revert to `Inactive` even though x has not changed.

DynamicModule Wormholes

The variables declared in a `DynamicModule` are localized to a particular rectangular area within one cell in a notebook. There are situations in which it is desirable to extend the scope of such a local variable to other cells or even other windows. For example, you might want to have a button in one cell that opens a dialog box that allows you to modify the value of a variable declared in the same scope as the button that opened the dialog.

This can be done with one of the more surreal constructs in *Mathematica*, a `DynamicModule` wormhole. `DynamicModule` accepts the option `DynamicModuleParent`, whose value is a `NotebookInterfaceObject` that refers to another `DynamicModule` anywhere in the front end. For purposes of variable localization, the `DynamicModule` with this option will be treated as if it resided inside the one referred to, regardless of where the two actually are (even if they are in separate windows).

The tricky part in setting up such a wormhole is getting the `NotebookInterfaceObject` necessary to refer to the parent `DynamicModule`. This reference can be created only after the `DynamicModule` has been created and placed as output, and it is valid only for the current session.

To make the process easier, and in fact avoid all reference to explicit `NotebookInterfaceObject`s, `DynamicModule` also accepts the option `InheritScope`, which automatically generates the correct value of the `DynamicModuleParent` option to make the new `DynamicModule` function as if it were inside the scope of the `DynamicModule` from which it was created. This is confusing, so an example is in order.

Evaluate this to create an output with a **+** button and a number.

```
In[42]:= DynamicModule[{x = 1}, {Button["+", ++x], Button["Make - Palette", CreatePalette[
              DynamicModule[{}, Button["-", --x], InheritScope → True]]], Dynamic[x]}]

Out[42]= {+, Make – Palette, 1}
```

Clicking the **+** button increments the value of a `DynamicModule` local variable, which is displayed at the end of the output. To decrement the number you have to click the **Make - Palette** button, which creates a new (very small) floating palette window containing a **-** button.

This **-** button is living in a wormhole created by the `InheritScope` option of the `DynamicModule` containing it. Clicking the button decrements the value of a local, private variable in the scope of a distant `DynamicModule` in another window.

`InheritScope` can be used only when the code creating the second `DynamicModule` is executed from inside a button or other dynamic object located within the first `DynamicModule`. By using `DynamicModuleParent` explicitly, it is possible to link up arbitrary existing `DynamicModule`s, but doing so is tricky, and beyond the scope of this document.

Introduction to Manipulate

The single command `Manipulate` lets you create an astonishing range of interactive applications with just a few lines of input. `Manipulate` is designed to be used by anyone who is comfortable using basic commands such as `Table` and `Plot`: it does not require learning any complicated new concepts, nor any understanding of user interface programming ideas.

The output you get from evaluating a `Manipulate` command is an interactive object containing one or more controls (sliders, etc.) that you can use to vary the value of one or more parameters. The output is very much like a small applet or widget: it is not just a static result, it is a running program you can interact with.

This tutorial is designed for people who are familiar with the basics of using the *Mathematica* language, including how to use functions, the various kinds of brackets and braces, and how to make simple plots. Some of the examples will use more advanced functions, but it is not necessary to understand exactly how these work in order to get the point of the example.

Despite the length of this tutorial, it is only half the story. "Advanced Manipulate Functionality" provides further information about some of the more sophisticated features of this rich command.

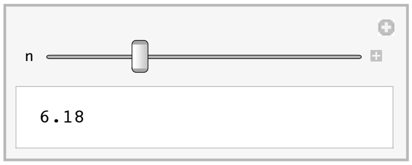
Manipulate Is as Easy as Table

At its most basic, the syntax of `Manipulate` is identical to that of the humble function `Table`. Consider this `Table` command, which produces a list of numbers from one to twenty.

```
Table[n, {n, 1, 20}]
{1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20}
```

Simply replace the word `Table` with the word `Manipulate`, and you get an interactive application that lets you explore values of n with a slider.

```
Manipulate[n, {n, 1, 20}]
```



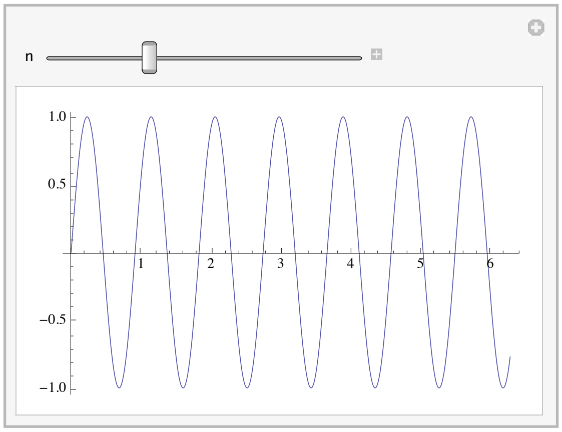
```
6.18
```

If you are reading this documentation inside *Mathematica*, you can click and drag the slider to see the displayed value change in real time (meaning that it changes while you are dragging the slider, not just when you release it). If you are reading a static form of the documentation, you will see the slider moved to an arbitrary position. (By default, it starts out on the left side, but in the following examples the slider has typically been moved away from its initial position.)

In both `Table` and `Manipulate`, the form {*variable*, *min*, *max*} is used to specify an "iterator", giving the name of the variable and the range over which to vary it.

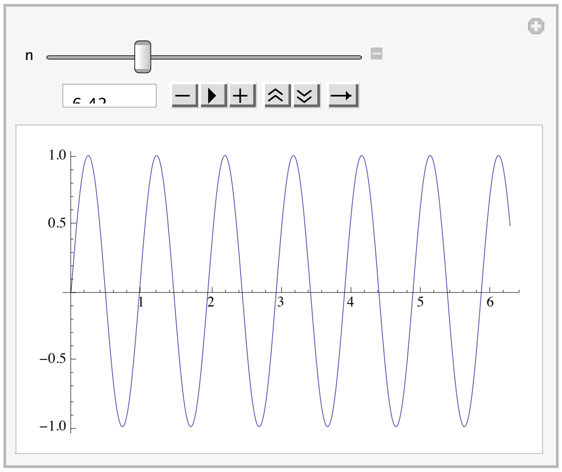
Of course the whole point of `Manipulate` (and `Table` for that matter) is that you can put any expression you like in the first argument, not just a simple variable name. Moving the slider in this very simple output already starts to give an idea of the power of `Manipulate`.

```
Manipulate[Plot[Sin[n x], {x, 0, 2 Pi}], {n, 1, 20}]
```



Again, if you are reading this in a static form you will have to trust that the graph changes in real time when the slider is moved.

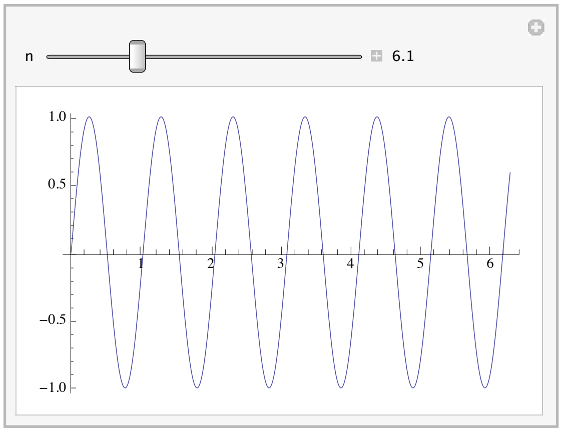
Note that the slider has an extra icon next to it which, when clicked, opens a small panel of additional controls. Here, the panel from the previous example is opened.



The panel allows you to see the numerical value of the variable, as well as set it in motion using the animation controls.

If you want to see the value of the variable without having to open the subpanel, you can add the option `Appearance -> "Labeled"` to the variable specification. (Note the number displayed to the right of the plus sign, which is updated in real time as the slider is moved.)

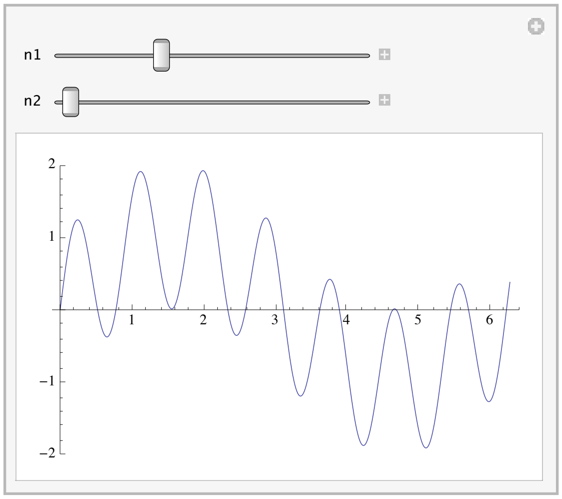
```
Manipulate[Plot[Sin[n x], {x, 0, 2 Pi}], {n, 1, 20, Appearance → "Labeled"}]
```



This is also the first hint that `Manipulate` goes far beyond the relative simplicity of `Table`, both in its output and in the flexibility and range of what can be specified in the list of variables.

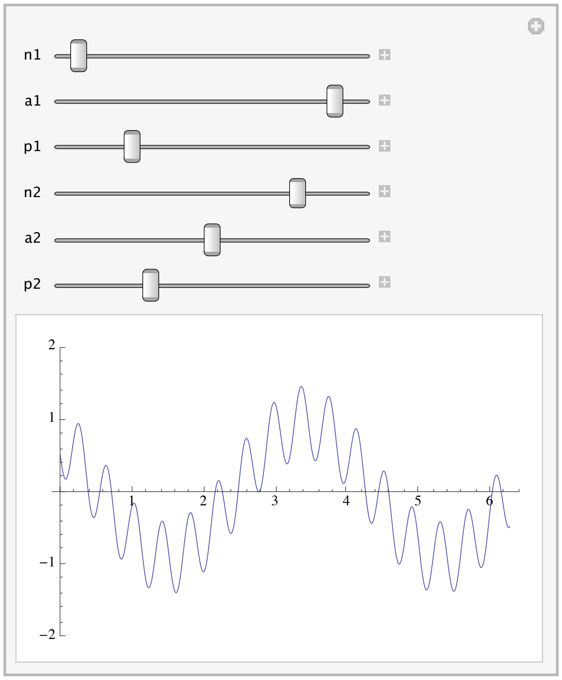
Just like `Table`, `Manipulate` allows you to give more than one variable range specification.

```
Manipulate[Plot[Sin[n1 x] + Sin[n2 x], {x, 0, 2 Pi}, PlotRange → 2],
 {n1, 1, 20}, {n2, 1, 20}]
```



You can have as many variables as you like, including so many that a similar `Table` command would try to enumerate an unreasonably large number of entries.

```
Manipulate[Plot[a1 Sin[n1 (x + p1)] + a2 Sin[n2 (x + p2)], {x, 0, 2 Pi}, PlotRange → 2],
 {n1, 1, 20}, {a1, 0, 1}, {p1, 0, 2 Pi}, {n2, 1, 20}, {a2, 0, 1}, {p2, 0, 2 Pi}]
```



You can open any or all of the subpanels to see numerical values, and you are free to animate many different variables at the same time if you like.

One way to think of `Manipulate` is as a way to interactively explore a large parameter space. You can move around that space at will, exploring interesting directions as they appear. As you will see in later sections, `Manipulate` has many features designed to make such exploration easier and more rewarding.

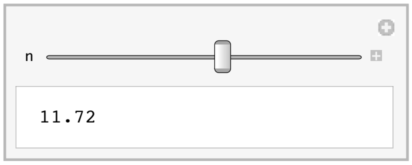
Symbolic Output and Step Sizes

The previous examples are graphical, and indeed the most common application for `Manipulate` is producing interactive graphics. But `Manipulate` is capable of making any *Mathematica* function interactive, not just graphical ones.

Often the first issue in examples involving symbolic, rather than graphical, output is that you want to deal with integers, rather than continuously variable real numbers. In `Table` the default step size is 1, so you naturally get integers, while in `Manipulate` the default is to allow continuous variation (which you could think of as a step size of zero). Compare these two examples, and note that `Manipulate` allows values in between those returned by `Table`.

```
Table[n, {n, 1, 20}]
{1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20}
```

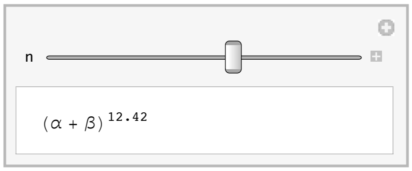
```
Manipulate[n, {n, 1, 20}]
```



```
11.72
```

Functions involving algebraic manipulations, for example, often do nothing interesting when given noninteger parameter values. This `Expand` function never expands anything.

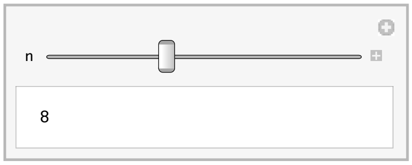
```
Manipulate[Expand[(α + β)^n], {n, 1, 20}]
```



$(\alpha + \beta)^{12.42}$

Fortunately it is trivial to add an explicit step size of 1 to the `Manipulate` command, yielding exactly the same set of possible values in `Manipulate` as is returned by `Table`.

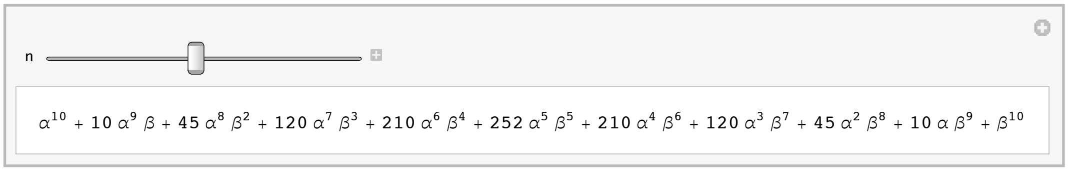
```
Manipulate[n, {n, 1, 20, 1}]
```



```
8
```

With an explicit step size, the `Expand` example is much more interesting.

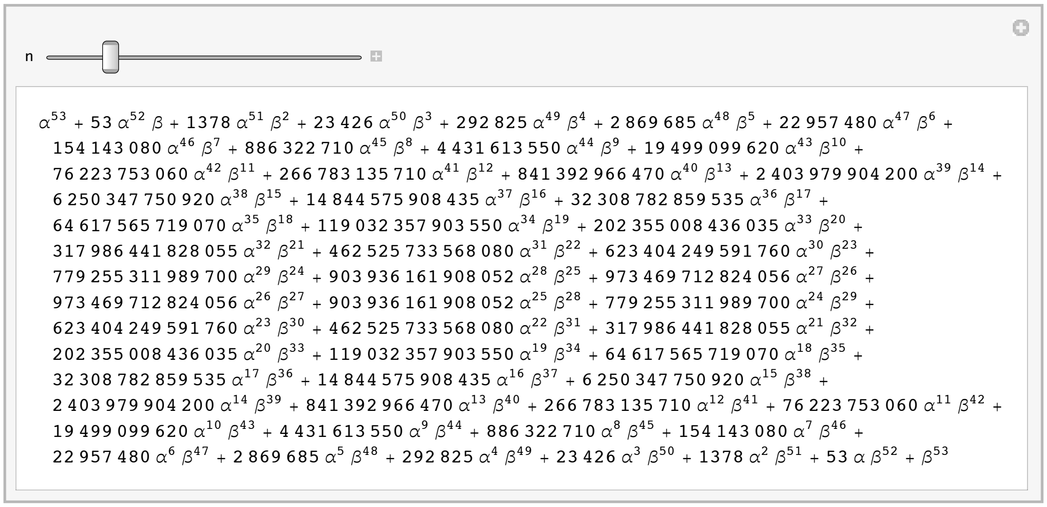
```
Manipulate[Expand[(α + β)^n], {n, 1, 20, 1}]
```



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

The fact that only one value is displayed at a time allows you to create examples that go far beyond what would be practical in a `Table` command. An important property of `Manipulate` output is that there is no fixed panel size or arbitrary limit as to how large the output panel can grow.

```
Manipulate[Expand[(α + β)^n], {n, 1, 300, 1}]
```

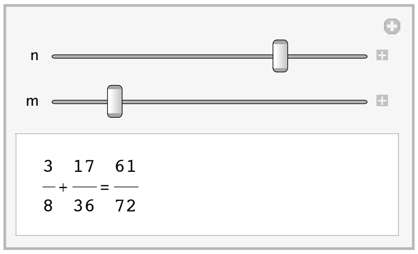


$$\begin{aligned}
&\alpha^{53} + 53\,\alpha^{52}\,\beta + 1378\,\alpha^{51}\,\beta^{2} + 23\,426\,\alpha^{50}\,\beta^{3} + 292\,825\,\alpha^{49}\,\beta^{4} + 2\,869\,685\,\alpha^{48}\,\beta^{5} + 22\,957\,480\,\alpha^{47}\,\beta^{6} +\\
&\quad 154\,143\,080\,\alpha^{46}\,\beta^{7} + 886\,322\,710\,\alpha^{45}\,\beta^{8} + 4\,431\,613\,550\,\alpha^{44}\,\beta^{9} + 19\,499\,099\,620\,\alpha^{43}\,\beta^{10} +\\
&\quad 76\,223\,753\,060\,\alpha^{42}\,\beta^{11} + 266\,783\,135\,710\,\alpha^{41}\,\beta^{12} + 841\,392\,966\,470\,\alpha^{40}\,\beta^{13} + 2\,403\,979\,904\,200\,\alpha^{39}\,\beta^{14} +\\
&\quad 6\,250\,347\,750\,920\,\alpha^{38}\,\beta^{15} + 14\,844\,575\,908\,435\,\alpha^{37}\,\beta^{16} + 32\,308\,782\,859\,535\,\alpha^{36}\,\beta^{17} +\\
&\quad 64\,617\,565\,719\,070\,\alpha^{35}\,\beta^{18} + 119\,032\,357\,903\,550\,\alpha^{34}\,\beta^{19} + 202\,355\,008\,436\,035\,\alpha^{33}\,\beta^{20} +\\
&\quad 317\,986\,441\,828\,055\,\alpha^{32}\,\beta^{21} + 462\,525\,733\,568\,080\,\alpha^{31}\,\beta^{22} + 623\,404\,249\,591\,760\,\alpha^{30}\,\beta^{23} +\\
&\quad 779\,255\,311\,989\,700\,\alpha^{29}\,\beta^{24} + 903\,936\,161\,908\,052\,\alpha^{28}\,\beta^{25} + 973\,469\,712\,824\,056\,\alpha^{27}\,\beta^{26} +\\
&\quad 973\,469\,712\,824\,056\,\alpha^{26}\,\beta^{27} + 903\,936\,161\,908\,052\,\alpha^{25}\,\beta^{28} + 779\,255\,311\,989\,700\,\alpha^{24}\,\beta^{29} +\\
&\quad 623\,404\,249\,591\,760\,\alpha^{23}\,\beta^{30} + 462\,525\,733\,568\,080\,\alpha^{22}\,\beta^{31} + 317\,986\,441\,828\,055\,\alpha^{21}\,\beta^{32} +\\
&\quad 202\,355\,008\,436\,035\,\alpha^{20}\,\beta^{33} + 119\,032\,357\,903\,550\,\alpha^{19}\,\beta^{34} + 64\,617\,565\,719\,070\,\alpha^{18}\,\beta^{35} +\\
&\quad 32\,308\,782\,859\,535\,\alpha^{17}\,\beta^{36} + 14\,844\,575\,908\,435\,\alpha^{16}\,\beta^{37} + 6\,250\,347\,750\,920\,\alpha^{15}\,\beta^{38} +\\
&\quad 2\,403\,979\,904\,200\,\alpha^{14}\,\beta^{39} + 841\,392\,966\,470\,\alpha^{13}\,\beta^{40} + 266\,783\,135\,710\,\alpha^{12}\,\beta^{41} + 76\,223\,753\,060\,\alpha^{11}\,\beta^{42} +\\
&\quad 19\,499\,099\,620\,\alpha^{10}\,\beta^{43} + 4\,431\,613\,550\,\alpha^{9}\,\beta^{44} + 886\,322\,710\,\alpha^{8}\,\beta^{45} + 154\,143\,080\,\alpha^{7}\,\beta^{46} +\\
&\quad 22\,957\,480\,\alpha^{6}\,\beta^{47} + 2\,869\,685\,\alpha^{5}\,\beta^{48} + 292\,825\,\alpha^{4}\,\beta^{49} + 23\,426\,\alpha^{3}\,\beta^{50} + 1378\,\alpha^{2}\,\beta^{51} + 53\,\alpha\,\beta^{52} + \beta^{53}
\end{aligned}$$

(In printed forms of this documentation, the slider is set fairly low to avoid wasting paper, but when moved all the way to the right, the output smoothly grows to cover many pages worth of vertical space.)

As with `Table`, if you use rational numbers for the minimum and step, you will get perfect rational numbers in the variable, not approximate real numbers. Here is an example that uses the formatting function `Row` to create a simple example of adding fractions.

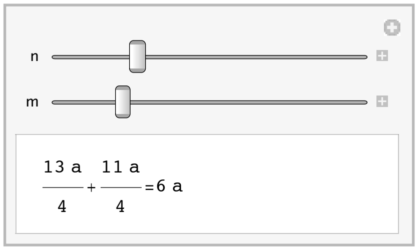
```
Manipulate[Row[{n, "+", m, "=", n + m}],
 {n, 1 / 2, 1 / 3, 1 / 144}, {m, 1 / 2, 1 / 3, 1 / 144}]
```



```
3   17   61
— + —— = ——
8   36   72
```

You can even use end points and step sizes that are symbolic expressions rather than just plain numbers.

```
Manipulate[Row[{n, "+", m, "=", n + m}], {n, a, 10 a, a / 12}, {m, a, 10 a, a / 12}]
```



```
13 a   11 a
———— + ———— =6 a
 4      4
```

Types of Controls

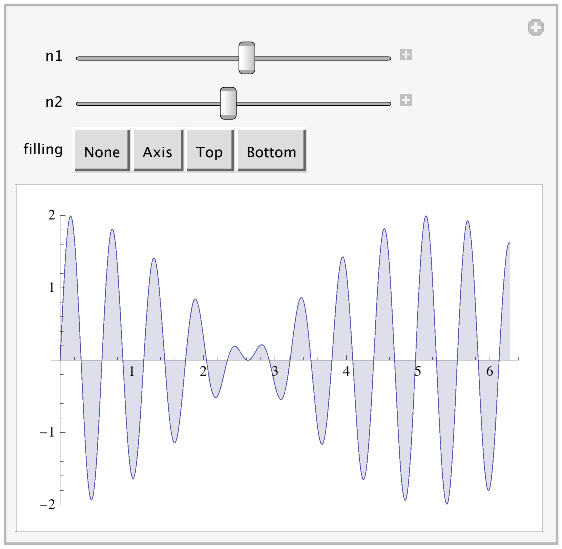
`Manipulate` supports a wide range of alternate ways of specifying variables, which generate different kinds of controls for those variables. This includes checkboxes, popup menus, and others in addition to sliders.

The principle is that for each variable, you ask for a particular set of possible values, and `Manipulate` automatically chooses an appropriate type of control to make those values conveniently available. For a typical numerical `Table`-like iterator, a slider is the most convenient interface.

You might, on the other hand, want to specify a discrete list of possible values (numeric or symbolic) rather than a range. This is done with an iterator of the form {*variable*, {*val1*, *val2*, …}}.

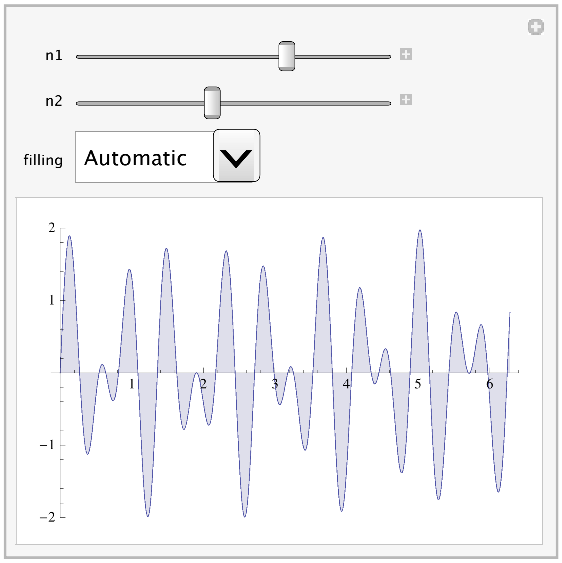
(Note the extra level of list compared to the range specification.) If you ask for a small number of separate values, you will get a row of buttons.

```
Manipulate[Plot[Sin[n1 x] + Sin[n2 x], {x, 0, 2 Pi}, Filling → filling, PlotRange → 2],
 {n1, 1, 20}, {n2, 1, 20}, {filling, {None, Axis, Top, Bottom}}]
```



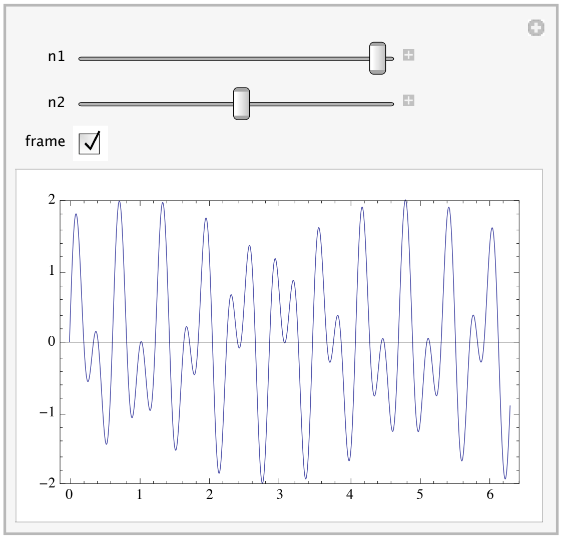
If you ask for a larger number of discrete values, `Manipulate` will switch to using a popup menu.

```
Manipulate[Plot[Sin[n1 x] + Sin[n2 x], {x, 0, 2 Pi}, Filling → filling, PlotRange → 2],
 {n1, 1, 20}, {n2, 1, 20},
 {filling, {None, Axis, Top, Bottom, Automatic, 1, 0.5, 0, -0.5, -1}}]
```



If you use the specific values `True` and `False`, you will get a checkbox.

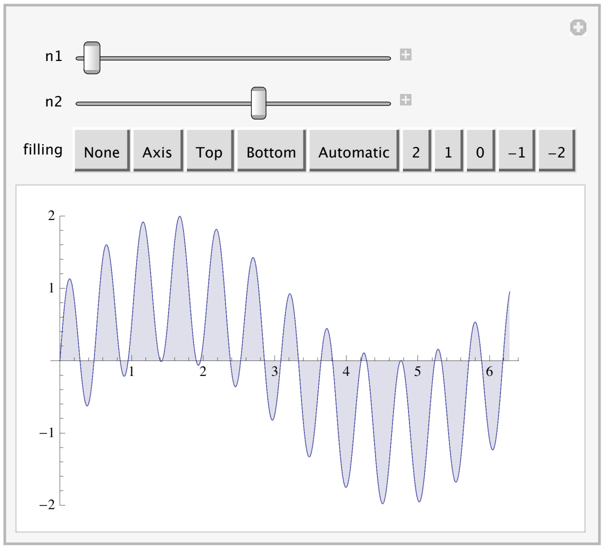
```
Manipulate[Plot[Sin[n1 x] + Sin[n2 x], {x, 0, 2 Pi}, Frame → frame, PlotRange → 2],
 {n1, 1, 20}, {n2, 1, 20}, {frame, {True, False}}]
```



These choices are of course somewhat arbitrary, but they are designed to be convenient, and you can always override the automatic choice of control type using a `ControlType` option inserted into the variable specification. (The full list of possible control types is given in the documentation for `Manipulate`.)

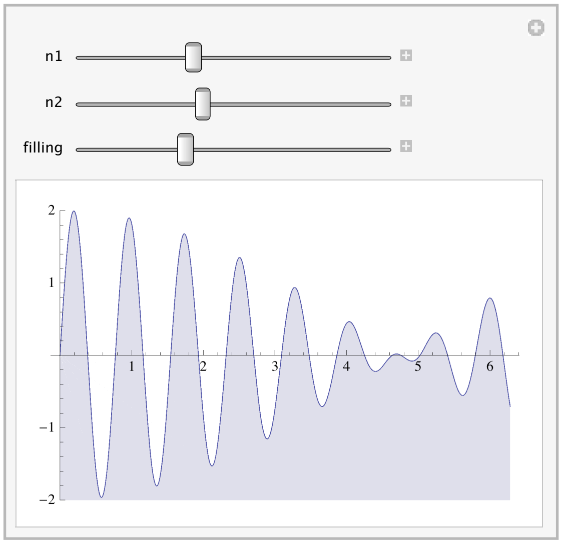
For example, you can ask for a row of buttons even if the automatic behavior would have chosen a popup menu, using the option `ControlType -> SetterBar`.

```
Manipulate[Plot[Sin[n1 x] + Sin[n2 x], {x, 0, 2 Pi}, Filling → filling, PlotRange → 2],
 {n1, 1, 20}, {n2, 1, 20}, {filling,
  {None, Axis, Top, Bottom, Automatic, 2, 1, 0, -1, -2}, ControlType → SetterBar}]
```



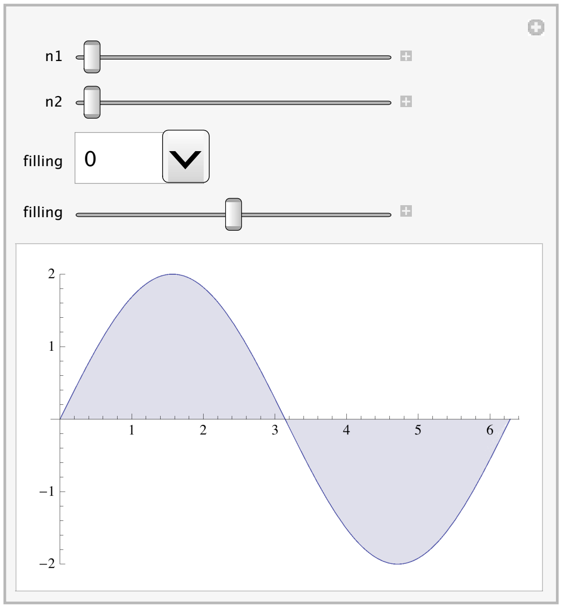
Sliders can be used to scan through discrete symbolic values, not just through numerical ranges (and this allows you to animate through them as well). The option `ControlType -> Manipulator` asks for the default control used by `Manipulate`, which is a slider plus an optional control panel with numerical value and animation controls (see the previous example). `ControlType -> Slider` asks for a plain slider.

```
Manipulate[Plot[Sin[n1 x] + Sin[n2 x], {x, 0, 2 Pi}, Filling → filling, PlotRange → 2],
 {n1, 1, 20}, {n2, 1, 20}, {filling, {None, Axis, Top, Bottom,
   Automatic, 1, 0.5, 0, -0.5, -1}, ControlType → Manipulator}]
```



It is even possible to use two different controls to adjust the value of the same variable. Here both a popup menu and a slider are connected to the value of the `filling` variable. If the slider is used to select a value that does not appear in the popup menu, the popup will appear blank, but remains functional. When a value is chosen from the popup menu, the slider is moved to the corresponding position. Both controls can thus be used interchangeably to adjust the same value, and each one follows along when the other is being used.

```
Manipulate[Plot[Sin[n1 x] + Sin[n2 x], {x, 0, 2 Pi}, Filling → filling, PlotRange → 2],
 {n1, 1, 20}, {n2, 1, 20},
 {filling, {None, 2, 1.5, 1, 0.5, 0, -0.5, -1, -1.5, -2}}, {filling, -2, 2}]
```

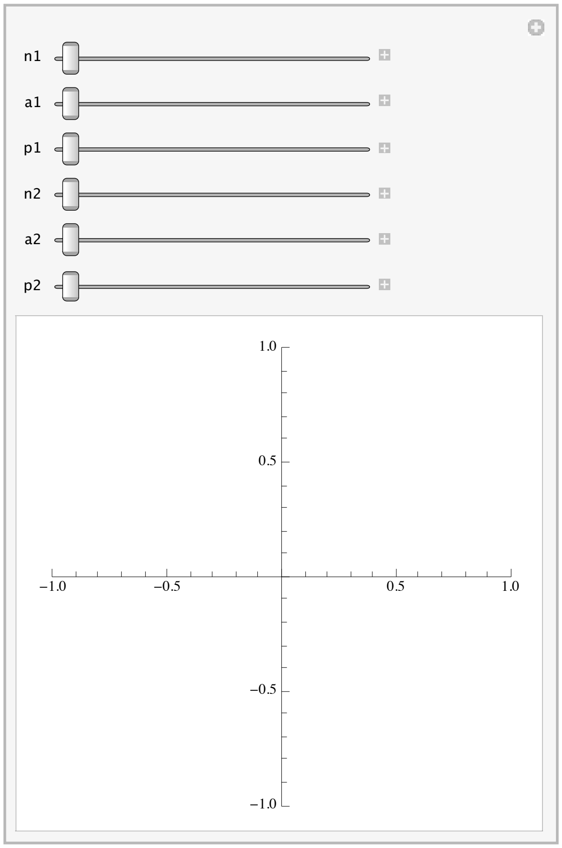


This is not an exhaustive list of the possible control types in `Manipulate`. See the `Manipulate` documentation for a more detailed listing. One of the most important control types, `Locator`, which allows you to place control points inside graphical output in a `Manipulate`, is discussed in "Locator", `Slider2D` is discussed in the "2D Sliders" section.

Initial Values and Labels

Here is a fun example for making Lissajous figures.

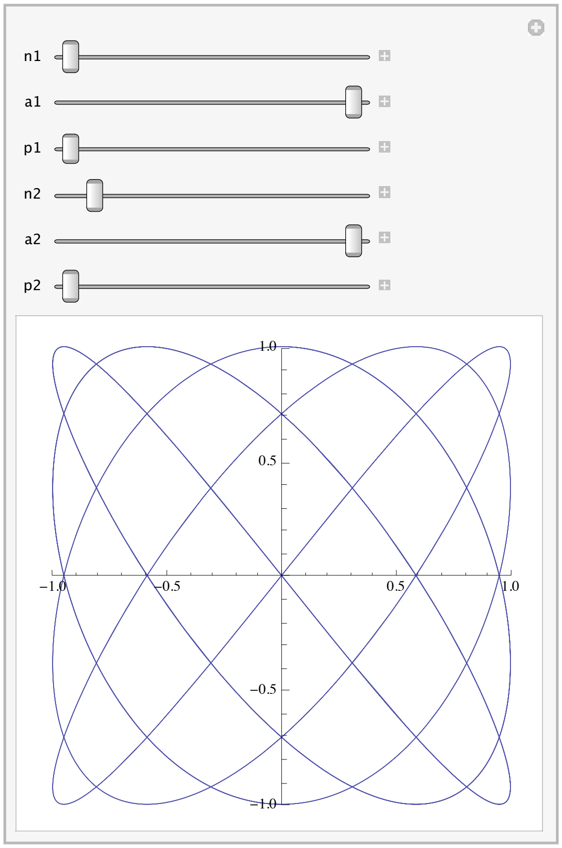
```
Manipulate[ParametricPlot[{a1 Sin[n1 (x + p1)], a2 Cos[n2 (x + p2)]},
   {x, 0, 20 Pi}, PlotRange → 1, PerformanceGoal → "Quality"],
  {n1, 1, 4}, {a1, 0, 1}, {p1, 0, 2 Pi},
  {n2, 1, 4}, {a2, 0, 1}, {p2, 0, 2 Pi}]
```



Unfortunately you see nothing at first: until you move the $a1$ and $a2$ (amplitude) variables away from their initial values of zero, there is nothing to see. It would be convenient to set their initial value to something other than the default left-most value. This is done by using a variable specification of the form {{*var*, *init*}, *min*, *max*}.

Here is the same example with both amplitudes set to 1 initially, and the default frequency values set to give a pleasing initial figure.

```
Manipulate[ParametricPlot[{a1 Sin[n1 (x + p1)], a2 Cos[n2 (x + p2)]},
   {x, 0, 20 Pi}, PlotRange → 1, PerformanceGoal → "Quality"],
  {n1, 1, 4}, {{a1, 1}, 0, 1}, {p1, 0, 2 Pi},
  {{n2, 5 / 4}, 1, 4}, {{a2, 1}, 0, 1}, {p2, 0, 2 Pi}]
```



It is fun to watch how one shape turns into another, and in this connection it is good to know about an unusual feature of sliders in *Mathematica*. If you hold down the Option key (Macintosh) or Alt key (Windows), the action of the slider will be slowed down by a factor of 20 relative to the movements of the mouse. In other words, when you drag the mouse left and right, the thumb will move only 1/20th as much as it normally would. If you move outside the area of the slider, the value will start moving slowly in that direction as long as the mouse remains clicked.

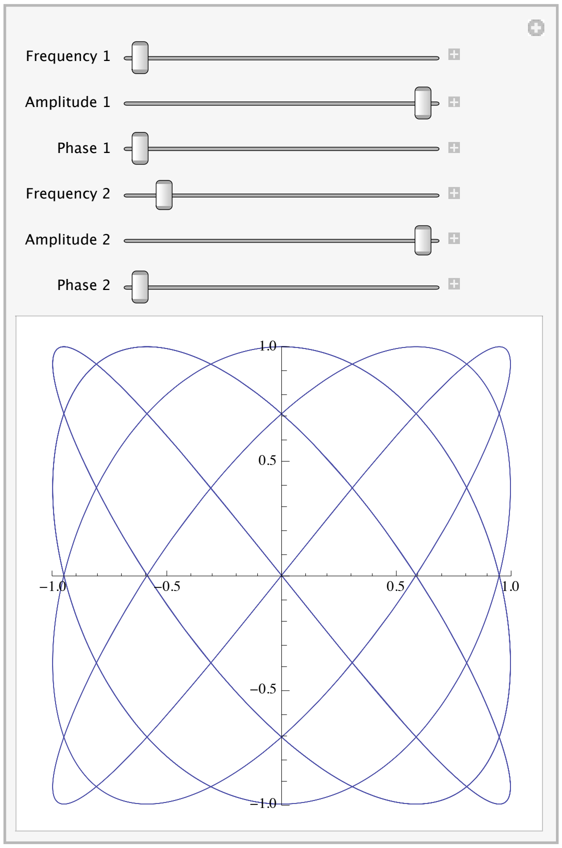
By holding down the Shift or Ctrl keys, or both, in addition to the Option/Alt key, you can slow the movement down by additional factors of 20 (one for each additional modifier key). With all three held down, it is possible to move the thumb by less that one part per million of its full range, which can be helpful in examples like this where beautiful patterns are hidden in very small ranges of parameter space.

(The option `PerformanceGoal -> "Quality"` is used in this example to ensure that `ParametricPlot` draws smooth curves even when a slider is being moved: the need for this option is explained in more detail in "Advanced Manipulate Functionality".)

By default `Manipulate` uses the names of the variables to label each control. But you may want to provide longer, more descriptive labels, which can be done by using variable specifications of the form {{*var*, *init*, *label*}, *min*, *max*}.

Here is the same example with labels.

```
Manipulate[ParametricPlot[{a1 Sin[n1 (x + p1)], a2 Cos[n2 (x + p2)]},
  {x, 0, 20 Pi}, PlotRange → 1, PerformanceGoal → "Quality"],
 {{n1, 1, "Frequency 1"}, 1, 4}, {{a1, 1, "Amplitude 1"}, 0, 1},
 {{p1, 0, "Phase 1"}, 0, 2 Pi}, {{n2, 5 / 4, "Frequency 2"}, 1, 4},
 {{a2, 1, "Amplitude 2"}, 0, 1}, {{p2, 0, "Phase 2"}, 0, 2 Pi}]
```

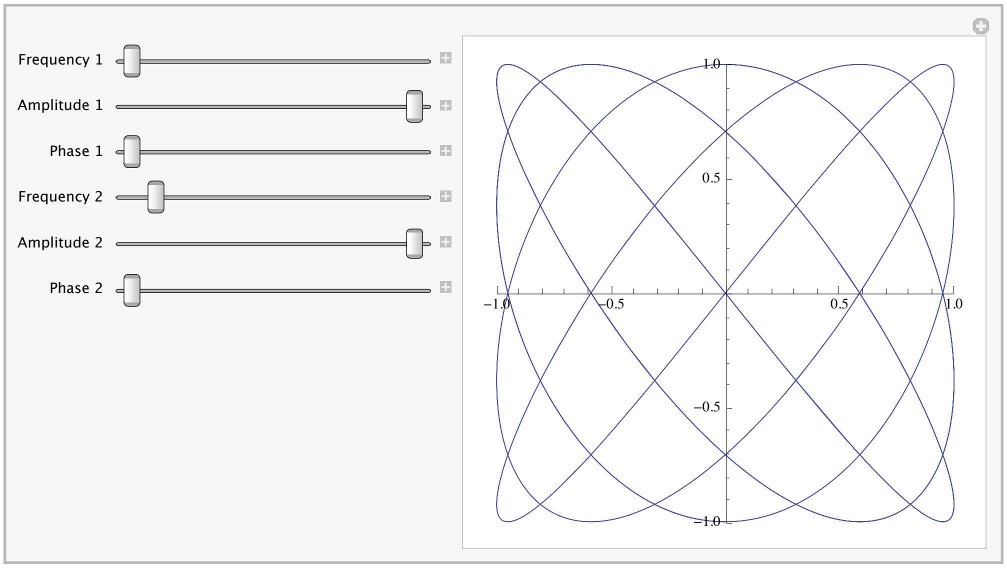


Beautifying the Control Area

`Manipulate` supports a number of features that allow you to rearrange, annotate, and generally pretty up the control area, to make it suit the needs of a particular example. (Advanced users should remember, however, that `Manipulate` is by no means the only way to create interactive interfaces in *Mathematica*, and if you cannot do what you want using `Manipulate`, you can easily start using functions such as `Dynamic` and `DynamicModule` directly to create free-form, open-ended user interfaces not tied to the particular conventions of `Manipulate`. These features are explained in detail in "Introduction to Dynamic" and "Advanced Dynamic Functionality".)

When you have a small number of controls, it is usually most convenient to have them above the content area of the `Manipulate` panel. But because screens are typically wider than they are tall, if you have a large number of controls, you may find it better to put them on the left side, using the `ControlPlacement` option.

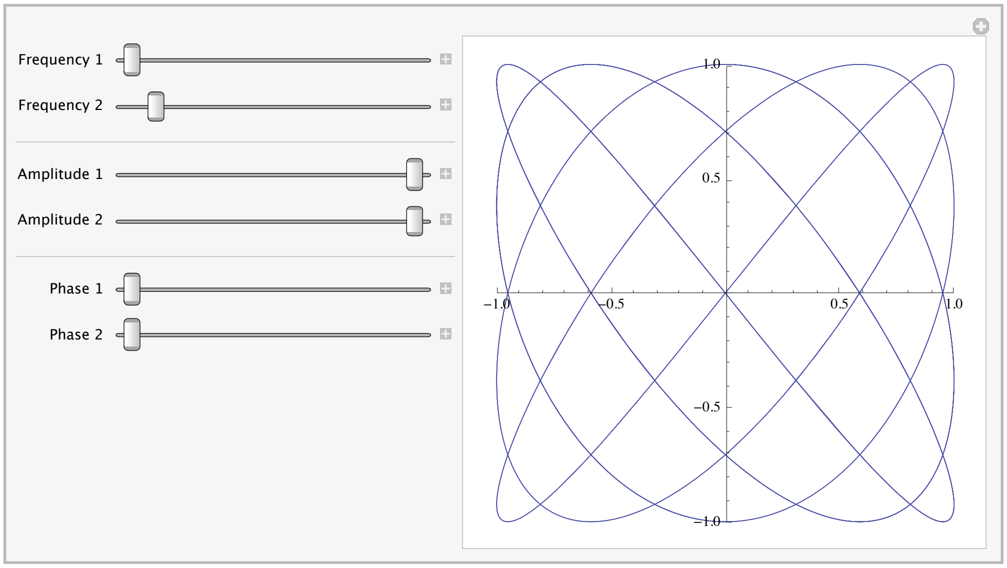
```
Manipulate[ParametricPlot[{a1 Sin[n1 (x + p1)], a2 Cos[n2 (x + p2)]}, {x, 0, 20 Pi},
  PlotRange → 1, PerformanceGoal → "Quality"], {{n1, 1, "Frequency 1"}, 1, 4},
 {{a1, 1, "Amplitude 1"}, 0, 1}, {{p1, 0, "Phase 1"}, 0, 2 Pi},
 {{n2, 5 / 4, "Frequency 2"}, 1, 4}, {{a2, 1, "Amplitude 2"}, 0, 1},
 {{p2, 0, "Phase 2"}, 0, 2 Pi}, ControlPlacement → Left]
```



When `ControlPlacement` is used at the level of the `Manipulate` as a whole, it sets the default position of all the controls. But the option can also be used inside individual variable specifications, allowing you to distribute controls to multiple sides of the output field.

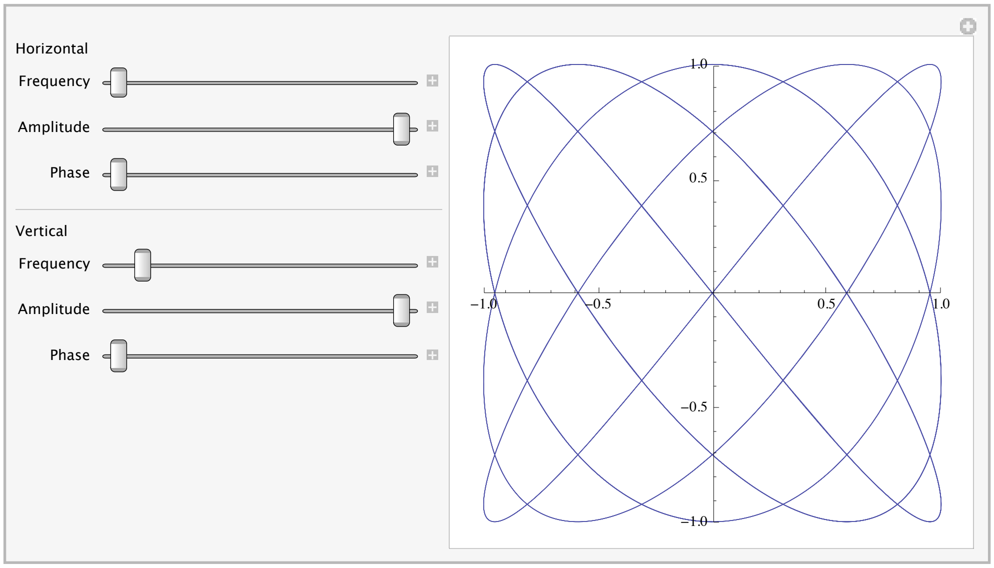
In the following example the controls naturally fall into two groups of three, or three groups of two. You can use the keyword `Delimiter` inserted in the sequence of variable specifications to indicate where you would like dividing lines put. Here two unlabeled delimiters break the controls up into three groups.

```
Manipulate[ParametricPlot[{a1 Sin[n1 (x + p1)], a2 Cos[n2 (x + p2)]},
   {x, 0, 20 Pi}, PlotRange → 1, PerformanceGoal → "Quality"],
  {{n1, 1, "Frequency 1"}, 1, 4}, {{n2, 5 / 4, "Frequency 2"}, 1, 4},
  Delimiter, {{a1, 1, "Amplitude 1"}, 0, 1}, {{a2, 1, "Amplitude 2"}, 0, 1},
  Delimiter, {{p1, 0, "Phase 1"}, 0, 2 Pi},
  {{p2, 0, "Phase 2"}, 0, 2 Pi}, ControlPlacement → Left]
```



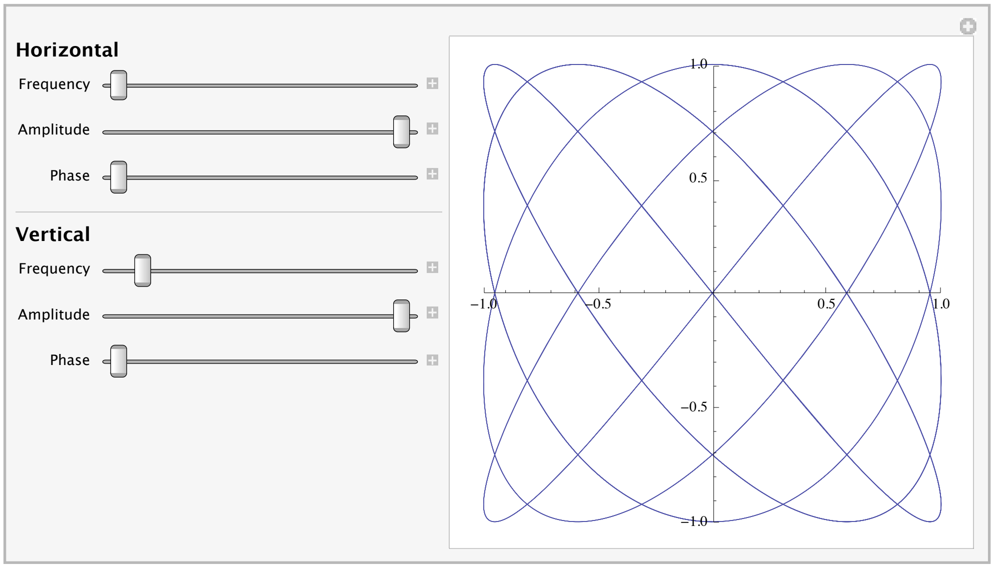
Alternately strings, or delimiters and strings, can be used to label the groups of controls.

```
Manipulate[ParametricPlot[{a1 Sin[n1 (x + p1)], a2 Cos[n2 (x + p2)]},
  {x, 0, 20 Pi}, PlotRange → 1, PerformanceGoal → "Quality"],
 "Horizontal", {{n1, 1, "Frequency"}, 1, 4},
 {{a1, 1, "Amplitude"}, 0, 1}, {{p1, 0, "Phase"}, 0, 2 Pi},
 Delimiter, "Vertical", {{n2, 5 / 4, "Frequency"}, 1, 4},
 {{a2, 1, "Amplitude"}, 0, 1}, {{p2, 0, "Phase"}, 0, 2 Pi}, ControlPlacement → Left]
```



Quite a variety of things can be interspersed with the controls, including styled text, arbitrary expressions, and even dynamic objects that update independently of the main output window. Here is a simple example of using `Style` to make the group headings more prominent.

```
Manipulate[ParametricPlot[{a1 Sin[n1 (x + p1)], a2 Cos[n2 (x + p2)]},
   {x, 0, 20 Pi}, PlotRange → 1, PerformanceGoal → "Quality"],
  Style["Horizontal", 12, Bold], {{n1, 1, "Frequency"}, 1, 4},
  {{a1, 1, "Amplitude"}, 0, 1}, {{p1, 0, "Phase"}, 0, 2 Pi},
  Delimiter, Style["Vertical", 12, Bold], {{n2, 5 / 4, "Frequency"}, 1, 4},
  {{a2, 1, "Amplitude"}, 0, 1}, {{p2, 0, "Phase"}, 0, 2 Pi}, ControlPlacement → Left]
```



Examples of more complex arrangements and dynamic labels are shown in "Advanced Manipulate Functionality".

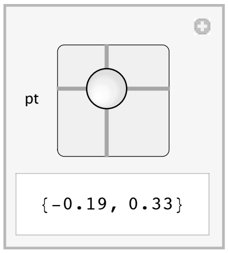
2D Sliders

A clever feature of *Mathematica* is support for two-dimensional sliders, which allow you to use both directions of mouse movement to control two values simultaneously. (Ordinary one-dimensional sliders in a sense waste one of the two degrees of freedom a mouse is capable of.)

To get a 2D slider, use pairs of numbers for both the *min* and *max*, as in $\{var, \{x_{min}, y_{min}\}, \{x_{max}, y_{max}\}\}$

The value of the variable will also be an $\{x, y\}$ pair. In this trivial example, just look at the value of the variable to get a feel for how the control works.

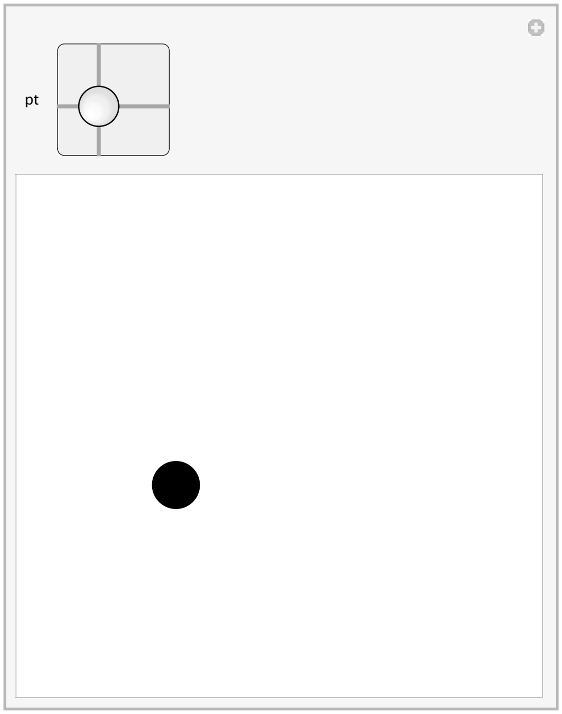
```
Manipulate[pt, {pt, {-1, -1}, {1, 1}}]
```



```
{-0.19, 0.33}
```

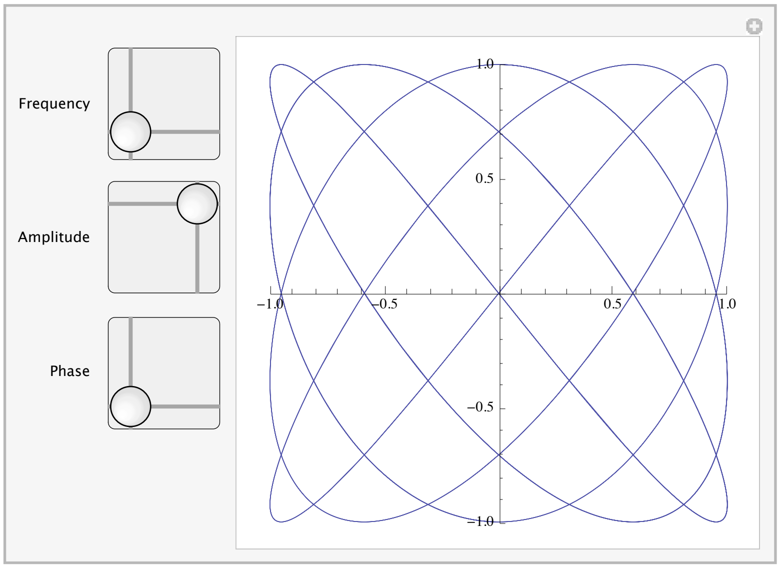
The following example shows more graphically how the value of a 2D slider corresponds to a coordinate point.

```
Manipulate[Graphics[{PointSize[0.1], Point[pt]}, PlotRange → 1],
 {pt, {-1, -1}, {1, 1}}]
```



To do something more interesting, you can recast the Lissajous figure from the previous section with three 2D sliders instead of six 1D sliders. You are controlling the same six parameters, but now you can do it two at a time.

```
Manipulate[
 ParametricPlot[{a[[1]] Sin[n[[1]] (x + p[[1]])], a[[2]] Cos[n[[2]] (x + p[[2]])]},
  {x, 0, 20 Pi}, PlotRange → 1, PerformanceGoal → "Quality"],
 {{n, {1, 5 / 4}, "Frequency"}, {1, 1}, {4, 4}},
 {{a, {1, 1}, "Amplitude"}, {0, 0}, {1, 1}},
 {{p, {0, 0}, "Phase"}, {0, 0}, {2 Pi, 2 Pi}}, ControlPlacement → Left]
```

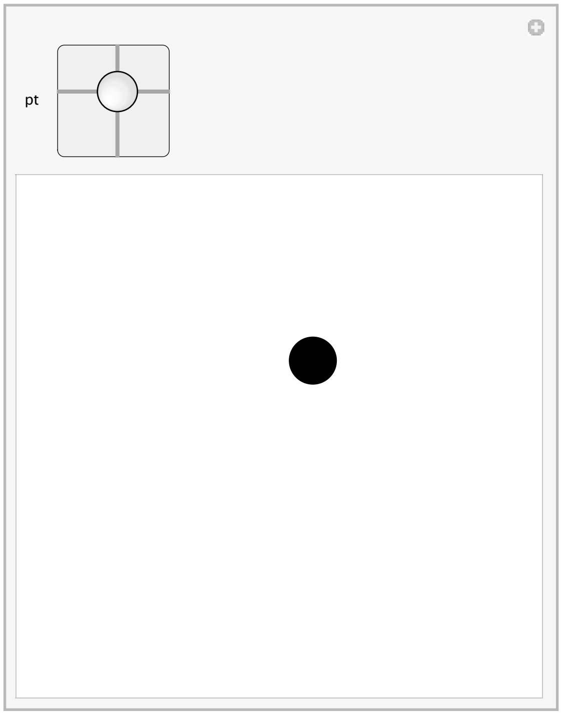


This creates an example that is compact and fun. Note that fine control using the Option, Shift, and Ctrl keys to slow down the motion of sliders (as explained in "Initial Values and Labels") works for 2D sliders as well as 1D sliders.

Graphics beyond Plotting

So far high-level plotting functions have mostly been used, but it is equally interesting to use *Mathematica*'s low level graphics language inside `Manipulate`. The following example, repeated from the previous section, is a trivial example of using the low-level graphics language.

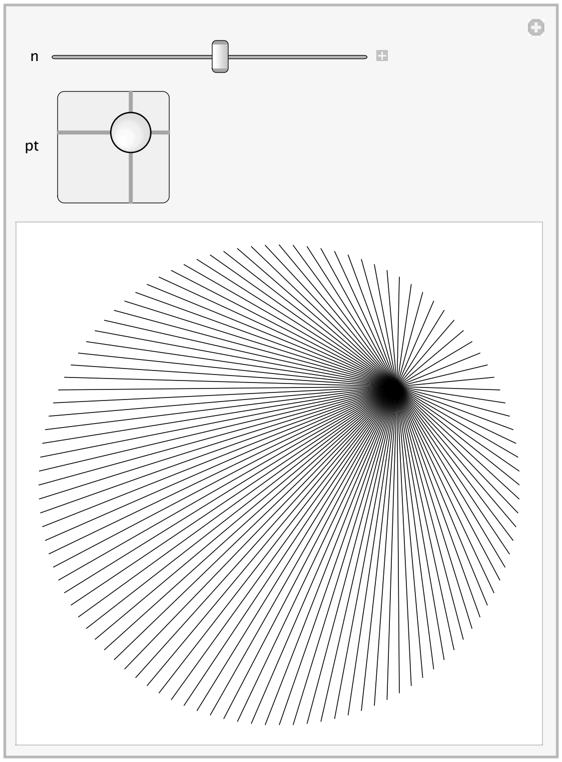
```
Manipulate[Graphics[{PointSize[0.1], Point[pt]}, PlotRange → 1],
 {pt, {-1, -1}, {1, 1}}]
```



This example also makes the important point that anytime you use `Graphics` inside `Manipulate`, you probably want to set an explicit `PlotRange` option. (`PlotRange -> 1` means 1 in all directions from the origin, and is equivalent to `PlotRange -> {{-1, 1}, {-1, 1}}`.) If you omit the `PlotRange` option *Mathematica*'s automatic plot range determination will cause the dot to appear not to move at all, because the plot range is always exactly centered around it.

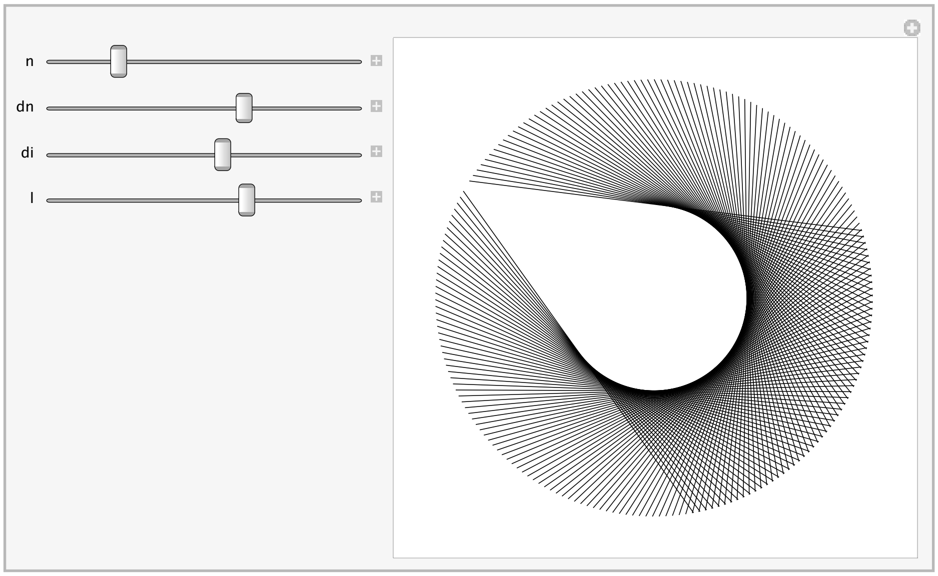
Simple (or complicated) *Mathematica* programming can add arbitrary graphical elements to the output. For example, here we have lines to the center point instead of a dot, with a second linear slider determining the number of lines.

```
Manipulate[
 Graphics[{Line[Table[{{Cos[t], Sin[t]}, pt}, {t, 2. Pi / n, 2. Pi, 2. Pi / n}]]},
  PlotRange → 1], {{n, 30}, 1, 200, 1}, {pt, {-1, -1}, {1, 1}}]
```



Here is a fun little string-figure example also based on creating a table of lines.

```
Manipulate[
 Graphics[Line[Table[{{Sin[n + i], Cos[n + i]}, {Sin[n + dn + i], Cos[n + dn + i]}},
    {i, 0, di, di / l}]], PlotRange → 1.1], {n, 0, 2 π},
 {dn, 0, 2 π}, {di, 1, 2 π}, {l, 1, 200}, ControlPlacement → Left]
```



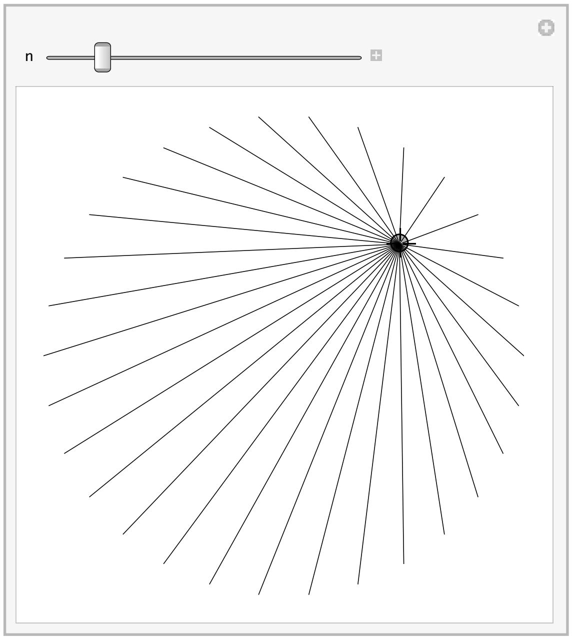
Because *Mathematica* is a sophisticated programming language, it is possible to use `Manipulate` to explore parameterized programs or algorithms interactively. The *Mathematica* graphics language is explained in "The Structure of Graphics", and many more examples like this can be found in The Wolfram Demonstrations Project.

Locator

For creating interactive graphics examples, one of the most important features of `Manipulate` is the ability to place a control point, called a `Locator`, inside graphics that appear in the output area.

Consider the previous example with lines going to a center point. While using a 2D slider is a fine way to control the center point, you might prefer to be able to simply click and drag the center point itself. This can be done by adding `Locator` to the control specification for the *pt* variable. In this case it is not necessary to specify a *min* and *max* range, because it can be taken automatically from the graphic. (It is, however, necessary to specify an initial value.)

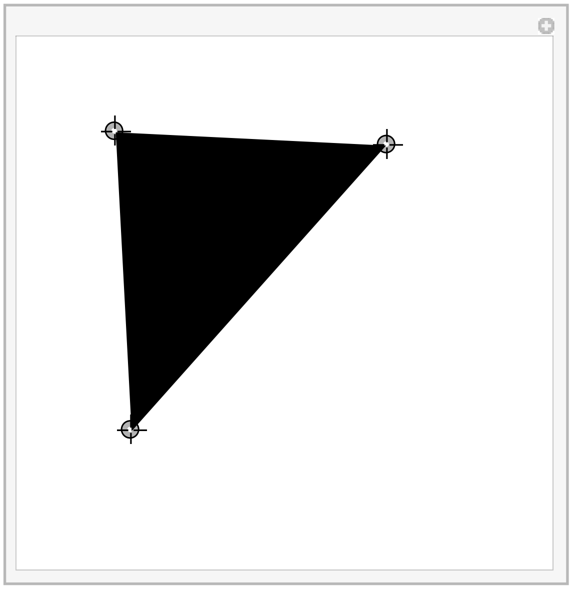
```
Manipulate[
 Graphics[{Line[Table[{{Cos[t], Sin[t]}, pt}, {t, 2. Pi / n, 2. Pi, 2. Pi / n}]]},
  PlotRange → 1], {{n, 30}, 1, 200, 1}, {{pt, {0, 0}}, Locator}]
```



Now you can click anywhere in the graphic and the center point of the lines will follow the mouse as long as you keep the mouse button down. (It is not necessary to click exactly on the center; it will jump to wherever you click, anywhere in the graphic.)

You can have multiple `Locator` controls by listing them individually, and it is perfectly fine to have a `Manipulate` with no controls outside the content area, so you can create purely graphical examples.

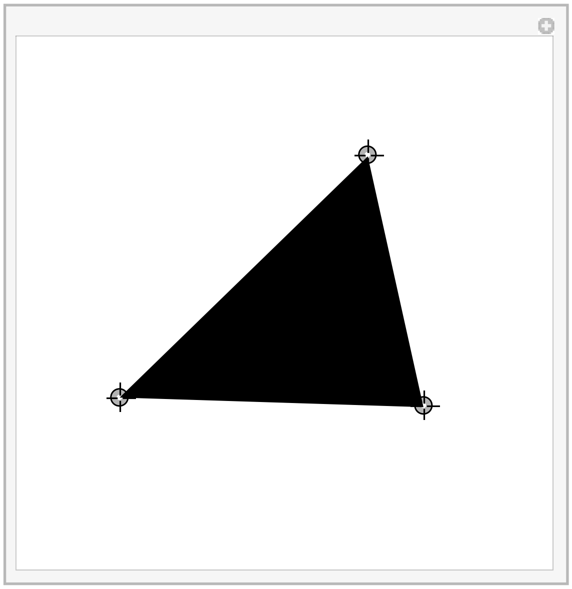
```
Manipulate[Graphics[Polygon[{pt1, pt2, pt3}], PlotRange → 1],
 {{pt1, {0, 0}}, Locator}, {{pt2, {0, 1}}, Locator}, {{pt3, {1, 0}}, Locator}]
```



When there are multiple locators, you can still click anywhere in the graphic, and the nearest `Locator` will jump to where you click and start tracking the mouse.

Instead of using multiple separate variables, each of which corresponds to a single $\{x, y\}$ point, you can use a single variable whose value is a list of points.

```
Manipulate[Graphics[Polygon[pts], PlotRange → 1],
 {{pts, {{0, 0}, {1, 0}, {0, 1}}}, Locator}]
```

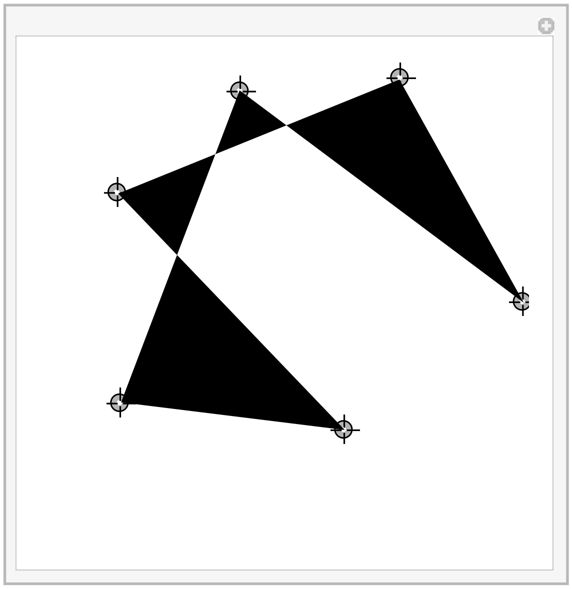


Again, if you click anywhere in the graphic, not on a particular `Locator`, the nearest one will jump to the mouse and start tracking it.

Due to internal limitations, it is not possible to combine individual `Locator` variables with a variable that is a list of multiple `Locator` variables: you can have only one multipoint `Locator` variable in a `Manipulate`. However, in exchange, it is possible to add the option `LocatorAutoCreate -> True` to that one `Locator` multivariable specification, and thereby allow you to create and destroy `Locator` points interactively (changing the length of the list of points stored in the variable).

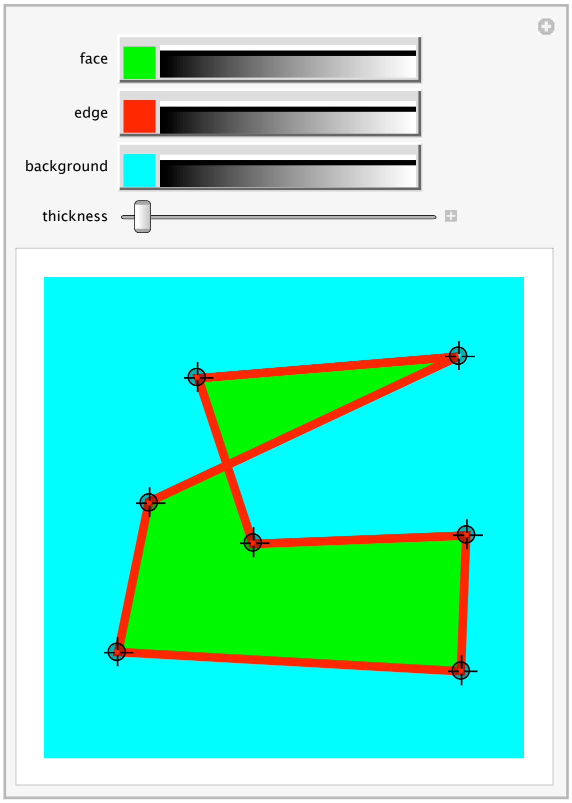
In the following example, hold down the Cmd key (Macintosh) or Alt key (Windows) and click anywhere that is not an existing `Locator` to create a new one at that location. Cmd/Alt click an existing `Locator` to destroy it. When you add or remove a `Locator`, you are changing the length of the list of points stored in the *pts* variable, thus changing the number of vertices in the displayed polygon.

```
Manipulate[Graphics[Polygon[pts], PlotRange → 1],
 {{pts, {{0, 0}, {.5, 0}, {0, .5}}}, Locator, LocatorAutoCreate → True}]
```



You can of course combine `Locator` controls with normal `Manipulate` variables. For example, you can use some sliders and color choosers to control the appearance of the polygon.

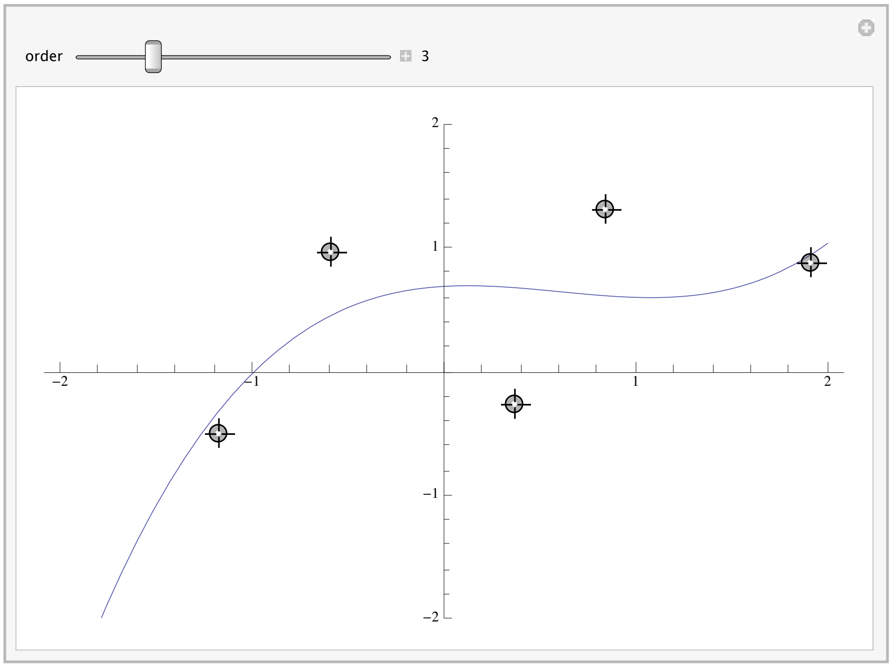
```
Manipulate[
 Graphics[{FaceForm[face], EdgeForm[{edge, Thickness[thickness]}], Polygon[pts]},
  PlotRange → 1, Background → background],
 {face, Green},
 {edge, Red},
 {background, Cyan},
 {{thickness, 0.02}, 0, 0.1},
 {{pts, {{0, 0}, {.5, 0}, {0, .5}}}, Locator, LocatorAutoCreate → True}]
```



While a case can be made that the previous examples are frivolous, they are meant to demonstrate the generality of the system: it provides a framework inside of which anything is possible. And the following example shows that even just a couple of lines of code can do something quite remarkable: create an interactive polynomial curve-fitting environment.

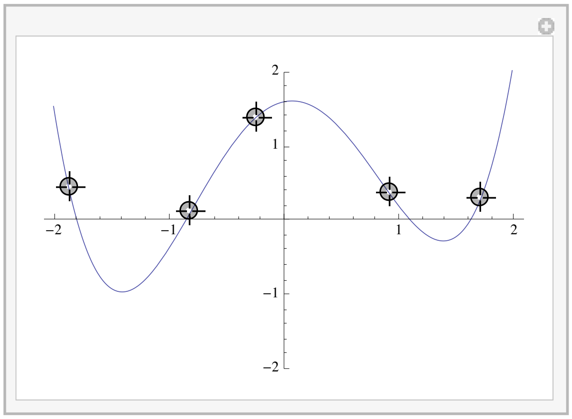
The locator thumbs represent data points that are being fit by least squares with a polynomial whose order is determined by the "order" slider. Five points are provided initially, but you can add new ones by Cmd/Alt clicking any blank area of the graphic, or remove one by Cmd/Alt clicking it.

```
Manipulate[Module[{x}, Plot[Fit[points, Table[x^i, {i, 0, order}], x],
    {x, -2, 2}, PlotRange → 2, ImageSize → 500, Evaluated -> True]],
  {{order, 3}, 1, 10, 1, Appearance → "Labeled"},
  {{points, RandomReal[{-2, 2}, {5, 2}]}, Locator, LocatorAutoCreate → True}]
```



The fact that an example of this sophistication can be constructed using such a small volume of code is really quite remarkable. And if you want to really impress someone with the compactness of *Mathematica* code, the following example shows how to do it using only two lines, with some loss of generality. Practice a bit and you can type this from scratch in 30 seconds or less.

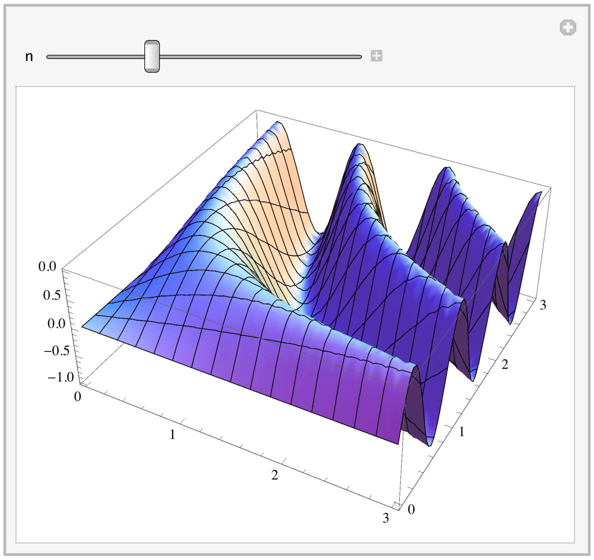
```
Manipulate[Plot[InterpolatingPolynomial[points, x], {x, -2, 2}, PlotRange → 2],
  {{points, RandomReal[{-2, 2}, {5, 2}]}, Locator}]
```



3D Graphics

`Manipulate` can be used to explore 3D graphics just as easily as 2D, though performance issues become more of a concern. Consider this simple example.

```
Manipulate[Plot3D[Sin[n x y], {x, 0, 3}, {y, 0, 3}], {n, 1, 5}]
```



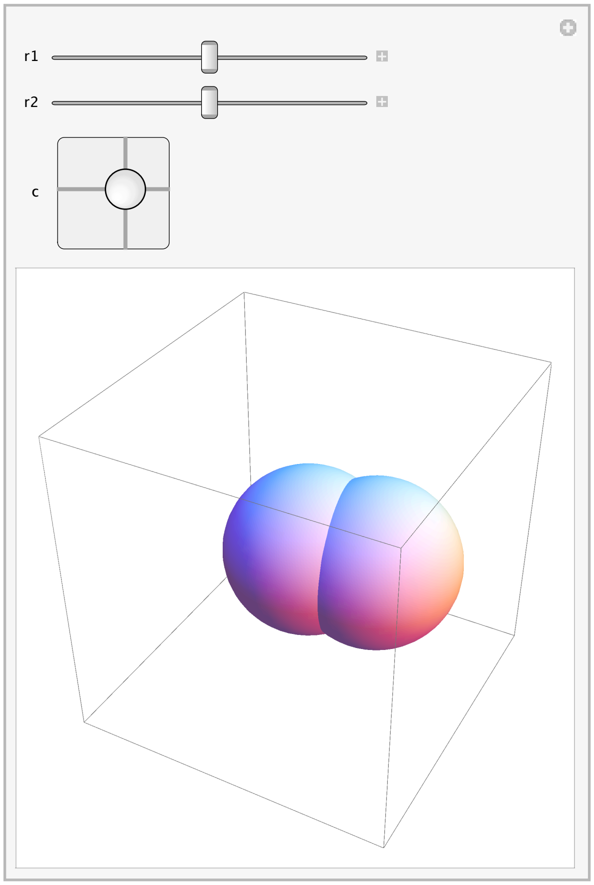
For large values of n the function oscillates rapidly, and in order to produce a smooth picture, the default adaptive sampling algorithm in `Plot3D` produces a fairly large number of polygons, with correspondingly long computation and rendering times.

Fortunately, `Plot3D` and other built-in plotting functions automatically adjust their internal algorithms and settings when used inside `Manipulate` in order to deliver increased speed while a control is being dragged, sometimes at the expense of rendering quality. As soon as the mouse button is released, a high-quality version of the plot is generated asynchronously (meaning other operations in the front end can continue while the plot is being generated). Asynchronous evaluations are discussed in further detail in "Synchronous Versus Asynchronous Dynamic Evaluations" in "Advanced Dynamic Functionality".

The net result is that while you drag the slider, a fast, but somewhat crude, rendering of the plot is created in real time, and when you release the control, a smooth rendering shows up a moment later. (This happens because `Plot3D`, and most other plotting functions, refer to the function `ControlActive` in the default settings of the various options that control rendering quality and speed. See "Dealing with Slow Evaluations" in "Advanced Manipulate Functionality" for more about using `ControlActive` within `Manipulate`.)

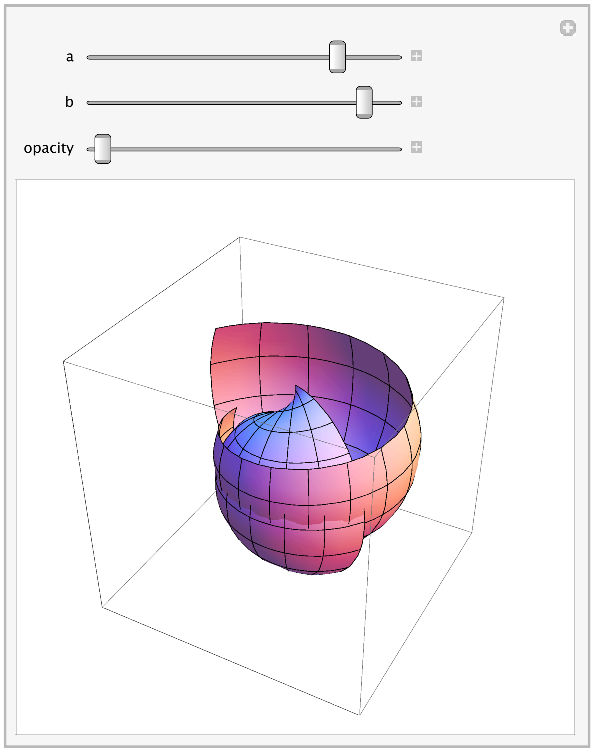
As in 2D, you can use the low-level graphics language just as easily as higher-level plotting commands. In this example you can see how *Mathematica* handles spheres that intersect with each other and with the bounding box.

```
Manipulate[Graphics3D[{Sphere[{0, 0, 0}, r1], Sphere[{c[[1]], c[[2]], 0}, r2]},
  PlotRange → 2], {{r1, 1}, 0, 2}, {{r2, 1}, 0, 2}, {c, {-2, -2}, {2, 2}}]
```



This example shows how opacity (which is to say, transparency) can be used to see inside nested 3D structures.

```
Manipulate[SphericalPlot3D[θ + ϕ, {θ, 0, a π}, {ϕ, 0, b π}, SphericalRegion → True,
  PlotRange → 10, Ticks → None, BaseStyle → Opacity[opacity]],
 {a, 0.1, 2}, {b, 0.1, 2}, {opacity, 1, 0}]
```



(Note that adding transparency to a 3D graphic can slow down rendering significantly.)

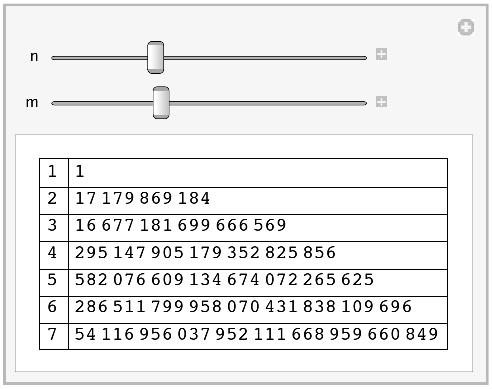
You can rotate a 3D graphic inside a `Manipulate` output by clicking and dragging it in the ordinary way. In most cases if you subsequently move one of the `Manipulate` controls, the graphic will stay rotated to the position you moved it to manually, unless the graphics expression in the `Manipulate` contains an explicit `ViewPoint` option, or wraps the graphical output in additional formatting constructs.

All Types of Output Are Supported

`Manipulate` is designed to work with the full range of possible types of output you can get with *Mathematica*, and it does not stop with graphical and algebraic output. Any kind of output supported by *Mathematica* can be used inside `Manipulate`. Here are some examples which may be less than obvious.

Formatting constructs such as `Grid`, `Column`, `Panel`, etc. can be used to produce nicely formatted outputs. (See "Grids, Rows, and Columns" for more information about formatting constructs.)

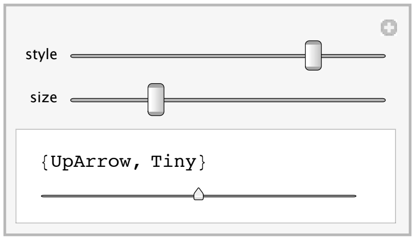
```
Manipulate[Grid[Table[{i, i^m}, {i, 1, n}], Alignment → Left, Frame → All],
 {n, 1, 20, 1}, {m, 1, 100, 1}]
```



1	1
2	17 179 869 184
3	16 677 181 699 666 569
4	295 147 905 179 352 825 856
5	582 076 609 134 674 072 265 625
6	286 511 799 958 070 431 838 109 696
7	54 116 956 037 952 111 668 959 660 849

You can even wrap `Manipulate` around functions that generate user interface elements like sliders and tab views. (See "Control Objects" and "Viewers and Annotation" for more information about user interface elements.) In this example we use two sliders to control the appearance of a third slider.

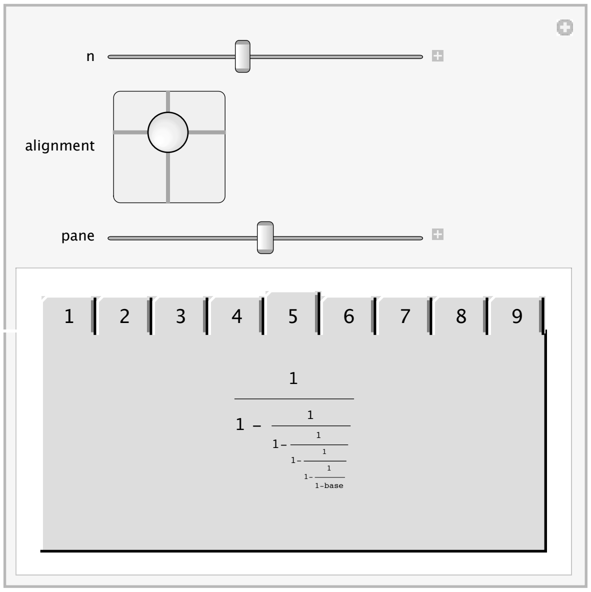
```
Manipulate[Column[{{style, size}, Slider[0.5, Appearance → {style, size}]}],
 {style, {"Automatic", "Vertical", "LeftArrow",
   "RightArrow", "UpArrow", "DownArrow"}, ControlType → Slider},
 {size, {"Automatic", "Tiny", "Small", "Medium", "Large"}, ControlType → Slider}]
```



```
{UpArrow, Tiny}
```

In this more complicated example the structure of a `TabView` is controlled by a `Manipulate`. `Dynamic[`*pane*`]` allows the current pane of the `TabView` to be selected either by using the slider created by `Manipulate`, or by clicking the `TabView` in the output area. The output is fully active.

```
Manipulate[TabView[Table[Nest[1/(1 - #) &, base, i], {i, 1, n}],
  Dynamic[pane], Alignment → alignment],
 {n, 1, 20, 1}, {alignment, {-1, -1}, {1, 1}}, {pane, 1, n, 1}]
```



$$\frac{1}{1-\frac{1}{1-\frac{1}{1-\frac{1}{1-\frac{1}{1-\text{base}}}}}}$$

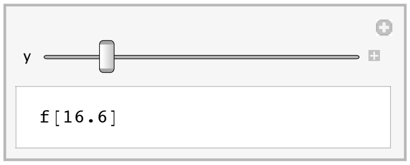
This example may be somewhat alarming, but is meant only to illustrate that `Manipulate` is a fully general function, not limited to exploring any fixed domain of graphical or algebraic examples. There is literally nothing you can see in a cell in a *Mathematica* notebook that you cannot interactively explore using `Manipulate` (subject only, of course, to the speed of your computer).

Saving Definitions of Functions Used inside Manipulate

Suppose you define a function, then use it in the first argument of a `Manipulate`.

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

```
Manipulate[f[y], {y, 0, 100}]
```



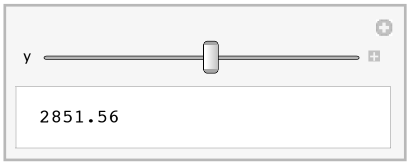
```
f[16.6]
```

This example will work well, until you try saving it in a file and then reopening it in a fresh session of *Mathematica*. Then the function `f` will not be defined until you manually evaluate the cell containing its definition. (In fact, if you are reading this documentation inside *Mathematica* you will see `f` appearing in the output area of the `Manipulate` at first, for exactly this reason.)

`Manipulate` supports the option `SaveDefinitions -> True`, which causes it to automatically build into the `Manipulate` output a copy of all the definitions of functions referred to in the `Manipulate` input (and recursively any functions they refer to). These definitions are then reestablished in any new *Mathematica* sessions the `Manipulate` output is opened in, before contents of the `Manipulate` are evaluated for the first time.

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

```
Manipulate[g[y], {y, 0, 100}, SaveDefinitions → True]
```



```
2851.56
```

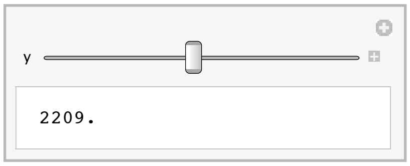
Thus if you are reading this inside *Mathematica*, the second example should correctly display a number even when first opened.

You can use `SaveDefinitions` to store function definitions or datasets, but be warned that if you refer to a large volume of data, it will of course be present in the file containing the saved `Manipulate` output, potentially creating a very large file.

An alternative in such a case is to use the `Initialization` option to load a package of data from a file or other source, rather than building it into the `Manipulate` output. The `Initialization` option can be given any arbitrary block of *Mathematica* code to be evaluated before the contents of the `Manipulate` are first evaluated in any fresh session of *Mathematica*. The right-hand side of the `Initialization` option will be evaluated only once per session.

For example, you can achieve the same result as earlier using the `Initialization` option instead of `SaveDefinitions`.

```
Manipulate[h[y], {y, 0, 100}, Initialization :→ (h[x_] := x^2)]
```



```
2209.
```

You can think of `SaveDefinitions` as a convenient automatic way of setting an `Initialization` option with all the definitions you need to run the example. (`SaveDefinitions` does not actually interfere with the use of the `Initialization` option: you can use both if you like.)

Gamepads and Joysticks

When interacting with a `Manipulate` output using a mouse, you are limited to moving only one control at a time. However, there are many USB controller devices available which overcome this limitation by placing a button or joystick under each finger, thus greatly increasing the number of controls you can move simultaneously.

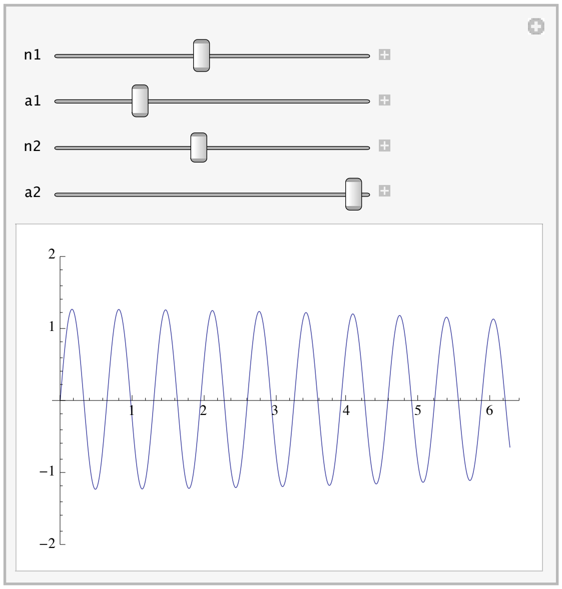
In order to take advantage of a USB controller in `Manipulate`, all you have to do is plug it in, and use the mouse to select (highlight) the cell bracket containing the `Manipulate` output you want to control. *Mathematica* automatically detects the controller, and `Manipulate` automatically links as many parameters as possible with the available joysticks and buttons.

While *Mathematica* will work, or attempt to work, with any USB controller device (gamepad, joystick, simulated airplane throttle control—even data acquisition devices that use the USB controller interface standard), some definitely work better than others for controlling `Manipulate` outputs. Generally speaking, dual-joystick gamepads, such as commonly used with video games, provide a good set of controls, typically four analog axes and a large number of buttons.

For the remainder of this section we will assume you are using a Logitech Dual Action gamepad. (This inexpensive controller is widely available and has better mechanical and electrical performance than many other units, even significantly more expensive ones.) If you are using a single joystick or another brand of gamepad there may be some differences in which controller parts map to which `Manipulate` parameters.

With a gamepad plugged in, select the cell bracket of the cell containing the following output. Initially nothing will happen, because the gamepad's joysticks are in their neutral, undeflected position. But if you move them, you will see one or more of the parameters start to change. The rate at which the parameter changes is proportional to the degree of deflection of the joystick.

```
Manipulate[Plot[a1 Sin[n1 x] + a2 Sin[n2 x], {x, 0, 2 Pi}, PlotRange → 2],
 {n1, 1, 20}, {a1, 0, 1}, {n2, 1, 20}, {a2, 0, 1}]
```



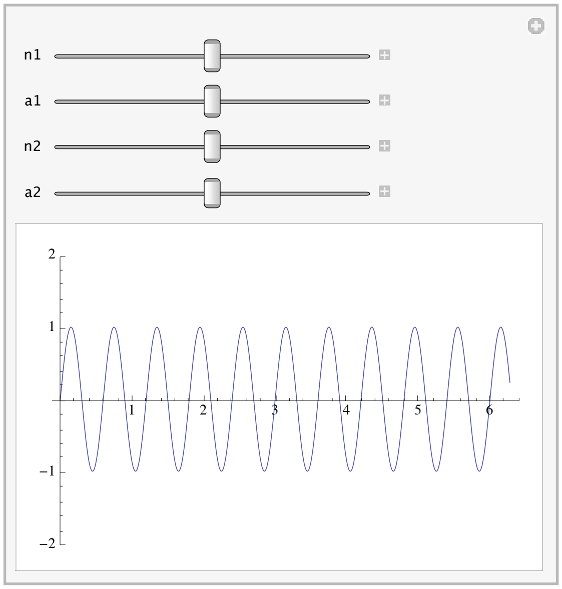
By default, `Manipulate` connects the x axis of the left gamepad to the first parameter, the y axis of the left joystick to the second parameter, the x axis of the right joystick to the third parameter, and the y axis of the right gamepad to the fourth parameter. You can verify this by moving each joystick in a given direction and watching which parameter changes. (If you are using something other than a Logitech Dual Action gamepad you may see a different mapping: each manufacturer does things a bit differently, and while *Mathematica* has tables that attempt to normalize many commonly available controllers, new ones are always being introduced.)

By default, the mapping is "velocity-based", which is to say that the rate at which the parameter changes is controlled by the position of the joystick. The joystick position is thus not directly connected to the value of the variable.

You might instead want the value of the variable to be determined by the absolute position of the joystick, and there are two ways to achieve this. On many gamepads, including the recommended Logitech model, the joysticks are also buttons: if you press down on a joystick it clicks like a button, and when the joystick is controlling a `Manipulate`, this causes the corresponding parameter(s) to become directly linked to the position of the joystick. You can use this direct mode to rapidly jump to any position, then release the joystick to stop the parameter value there.

If your gamepad does not have buttons in the joysticks, or you just want the linkage to always be direct, you can use the option `ControllerMethod -> "Absolute"`.

```
Manipulate[Plot[a1 Sin[n1 x] + a2 Sin[n2 x], {x, 0, 2 Pi}, PlotRange → 2],
 {n1, 1, 20}, {a1, 0, 1}, {n2, 1, 20}, {a2, 0, 1}, ControllerMethod → "Absolute"]
```

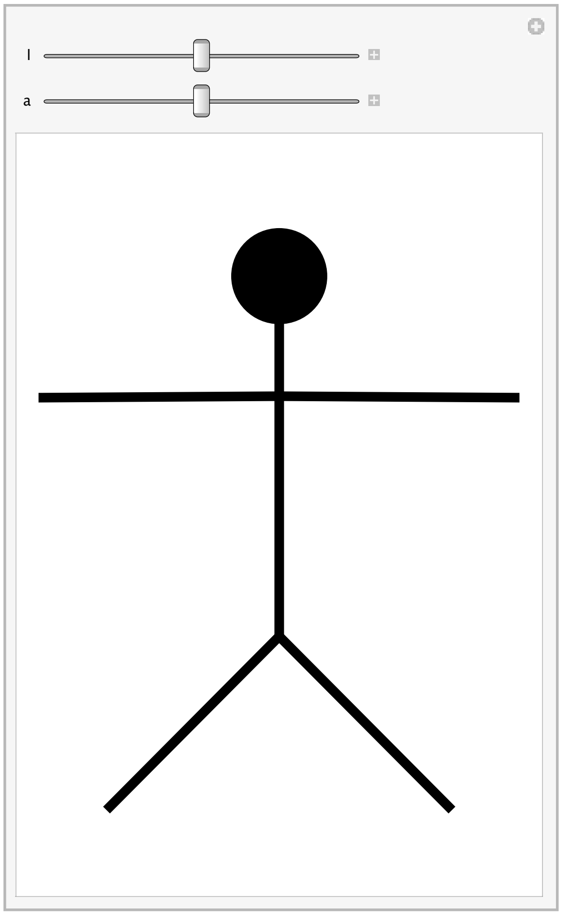


Note that there are disadvantages to direct linkage, most notably that as soon as you highlight the cell bracket (with a gamepad connected), all of the parameter values immediately jump to their middle positions, which is of course what they must do if the joysticks are in their neutral positions. While you can still use the mouse to set values, they will be overridden by the gamepad as soon as it is touched.

A small variation on velocity control can be had with the option `ControllerMethod -> "Cyclic"`. With this setting the linkage is velocity-based, but when you reach one end of the parameter's range of values, instead of stopping it cycles around to the opposite end.

Whether direct or velocity-based linking is best depends on the example. The previous example is generally more satisfactory with velocity linking, while the following example is definitely better with direct linking.

```
Manipulate[Graphics[{Thickness[0.02],
   Line[{{-Cos[l], Sin[l]}, {0, 0}, {Cos[l], Sin[l]}}],
   Line[{{0, 0}, {0, 1.5}}],
   Line[{{-Cos[a], 1 + Sin[a]}, {0, 1}, {Cos[a], 1 + Sin[a]}}],
   Disk[{0, 1.5}, .2]},
  PlotRange → {{-1, 1}, {-1, 2}}], {l, -Pi / 2, 0},
 {a, -Pi / 2, Pi / 2}, ControllerMethod → "Absolute"]
```

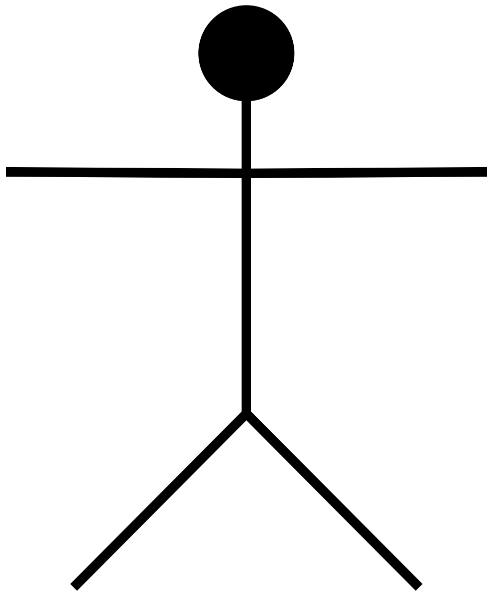


If any of these examples seem not to be working, chances are it is because you have forgotten to highlight the cell bracket containing them. This is a common mistake. As a convenience, and to avoid the need to select each output as soon as it is generated, if you wiggle any gamepad controller immediately after generating an output that references controllers, *Mathematica* will automatically select the output cell for you. But this only happens immediately after the output is generated, after that it is up to you to choose, by selecting it, which `Manipulate` output you want the controller connected to.

If you want a given `Manipulate` to always respond to the controller whether it is selected or not, you can add the option `ControllerLinking -> All`, but this feature should be used with caution. If you have multiple such outputs on screen, they will all attempt to move simultaneously, which is rarely helpful. The option is best used in situations where you are creating a fixed-format output window, rather than when creating examples meant to be used in a scrolling document such as this one.

In examples like the previous one, which do not make much sense unless you have a gamepad available, it is often pointless to display the sliders associated with the parameters, or the rest of the framework of `Manipulate`. The function `ControllerManipulate` is basically identical to `Manipulate` in all its features and syntax, except that it does not display any frame or sliders.

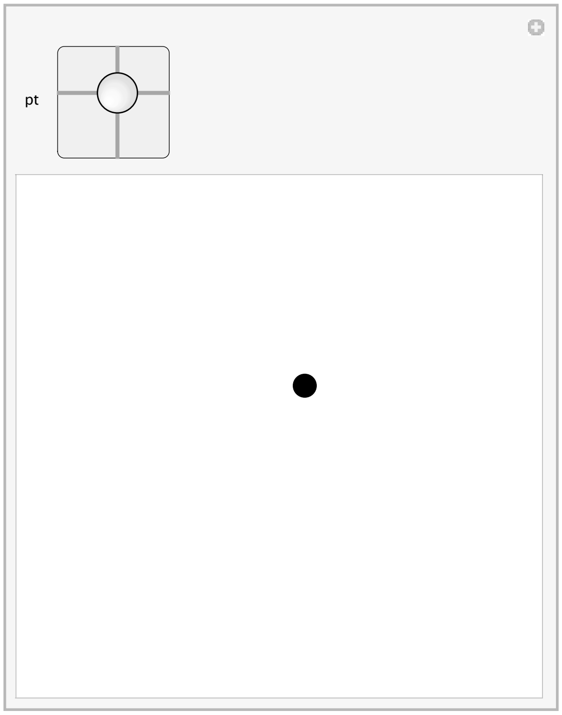
```
ControllerManipulate[Graphics[{Thickness[0.02],
   Line[{{-Cos[l], Sin[l]}, {0, 0}, {Cos[l], Sin[l]}}],
   Line[{{0, 0}, {0, 1.5}}],
   Line[{{-Cos[a], 1 + Sin[a]}, {0, 1}, {Cos[a], 1 + Sin[a]}}],
   Disk[{0, 1.5}, .2]},
  PlotRange → {{-1, 1}, {-1, 2}}],
 {l, -Pi / 2, 0},
 {a, -Pi / 2, Pi / 2},
 ControllerMethod → "Absolute"]
```



`Manipulate` will continue to be used in subsequent examples because it is helpful to be able to see the controls to understand how they are being affected by the gamepad, but many of these examples would look and work just as well with `ControllerManipulate`.

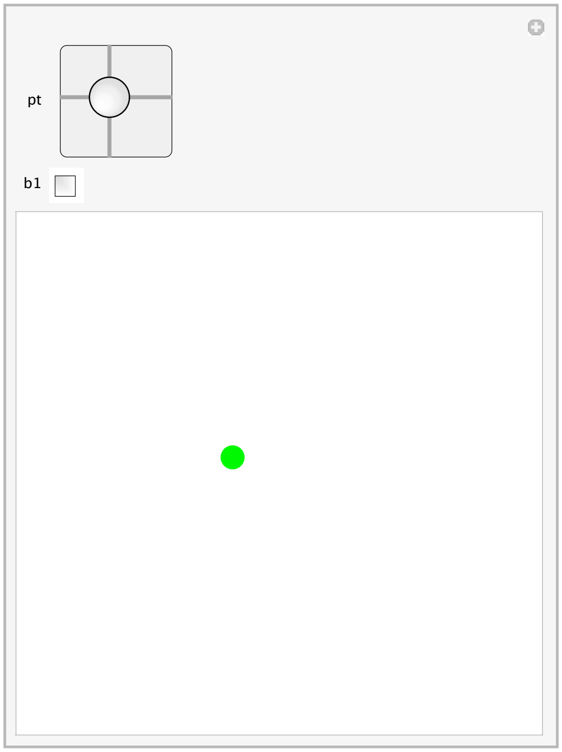
If the `Manipulate` contains `Slider2D` variables, whose values are $\{x, y\}$ pairs of numbers, they will automatically be linked to both directions of available joysticks. This example responds in both directions to the left-hand joystick on a gamepad.

```
Manipulate[Graphics[{PointSize[0.05], Point[pt]}, PlotRange → 1],
 {pt, {-1, -1}, {1, 1}}]
```



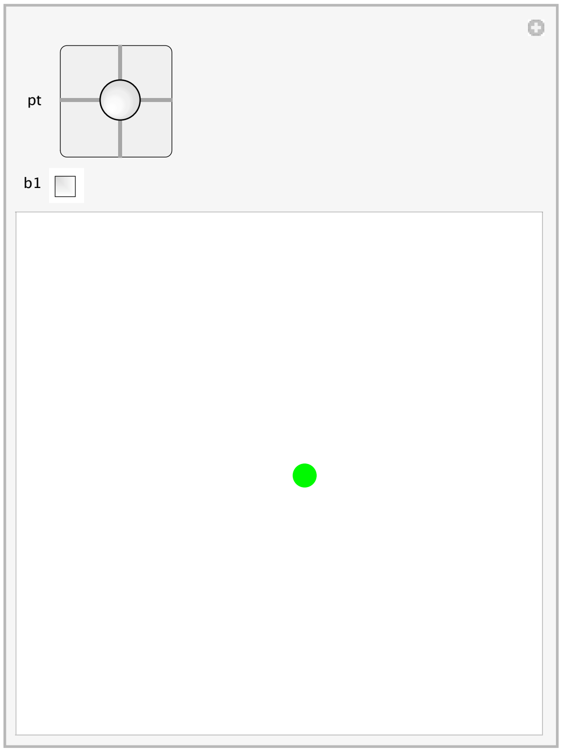
Push buttons on the controller are by default linked up to any Boolean (`True`/`False`) parameters specified in the `Manipulate`. Which button is which can be a bit hard to guess on a given controller (a concept which is explained elsewhere), but on the Logitech model the group of four buttons on the right side are labeled 1 through 4, and are used by `Manipulate` in that order. In this example clicking the "1" button toggles the setting of **b1**, changing the color of the point.

```
Manipulate[Graphics[{If[b1, Red, Green], PointSize[0.05], Point[pt]},
  PlotRange → 1], {pt, {-1, -1}, {1, 1}}, {b1, {True, False}}]
```



This toggling behavior (flipping the value of the parameter once each time the button is pressed) is the equivalent of velocity-based linking. If you use the `ControllerMethod -> "Absolute"` option (see previous examples) the parameter will be linked directly, which is to say its value will be `False` all the time except while the button is actually being held down.

```
Manipulate[
 Graphics[{If[b1, Red, Green], PointSize[0.05], Point[pt]}, PlotRange → 1],
 {pt, {-1, -1}, {1, 1}}, {b1, {True, False}}, ControllerMethod → "Absolute"]
```

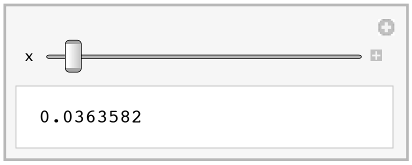


The exact rules by which `Manipulate` connects controller axes to parameters are somewhat complex, but basically they try to allocate the available analog and Boolean controls to the parameters in the `Manipulate` so as to make maximum use of the available joysticks, buttons, knobs, and other widgets on the controller. (Often the quickest and easiest way to figure out what has been linked to what is simply to wiggle the various knobs and see what happens.)

If you find that the default linking is not to your liking, it can be overridden by explicitly stating which controller axis should be connected to which parameter. The controller axes are named according to a logical system, but for most purposes it is enough to remember a few basic names: `"X"`, `"Y"`, `"XY"`, `"X1"`, `"X2"`, `"B1"`, `"B2"`, etc.

`"X"`, or its synonym `"X1"`, refers to the x axis of the primary, or the left-hand joystick on the gamepad. To specify that a parameter should be linked to this axis, use the following form.

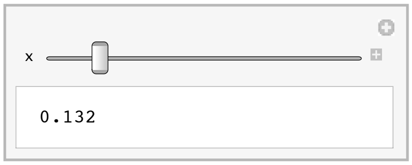
```
Manipulate[x, "X" → {x, 0, 1}]
```



```
0.0363582
```

`"X2"` similarly refers to the x axis of the secondary, or right-hand joystick.

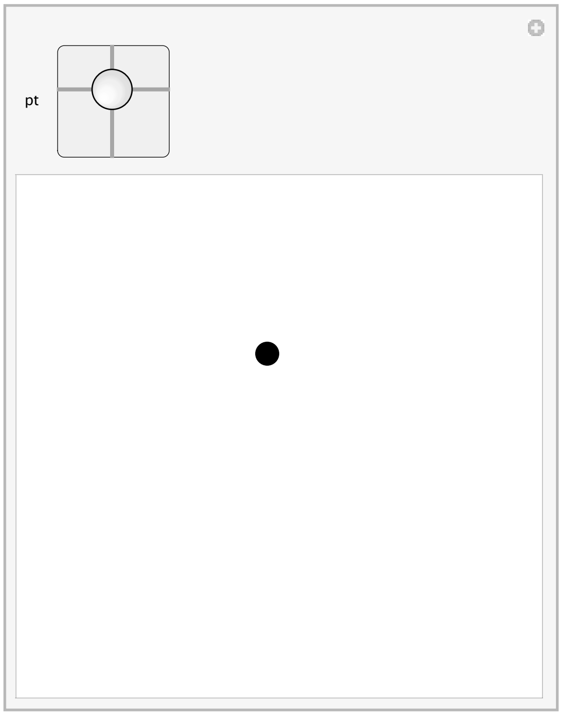
```
Manipulate[x, "X2" → {x, 0, 1}]
```



```
0.132
```

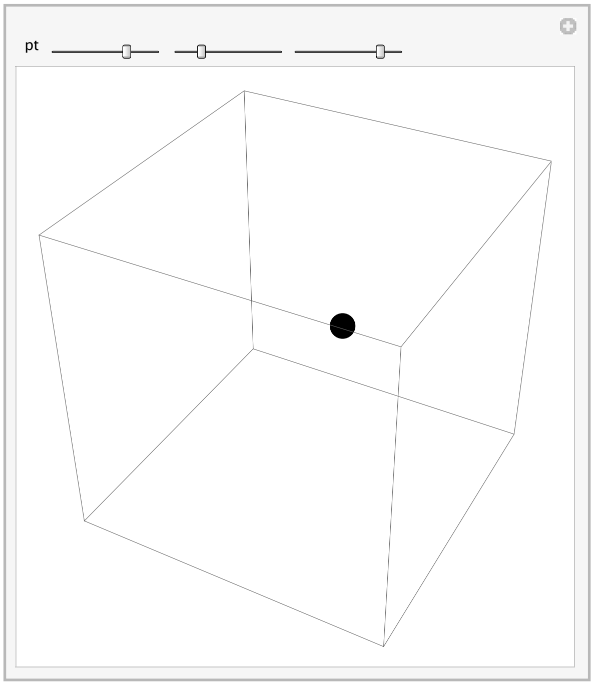
Axes can be combined into multidimensional parameters. For example, "XY" refers to both directions of the left or primary joystick, combined into a single {*x*, *y*} value. Such a combined axis must be connected to a Slider2D style of parameter, as in this example.

```
Manipulate[Graphics[{PointSize[0.05], Point[pt]}, PlotRange → 1],
 "XY" → {pt, {-1, -1}, {1, 1}}]
```



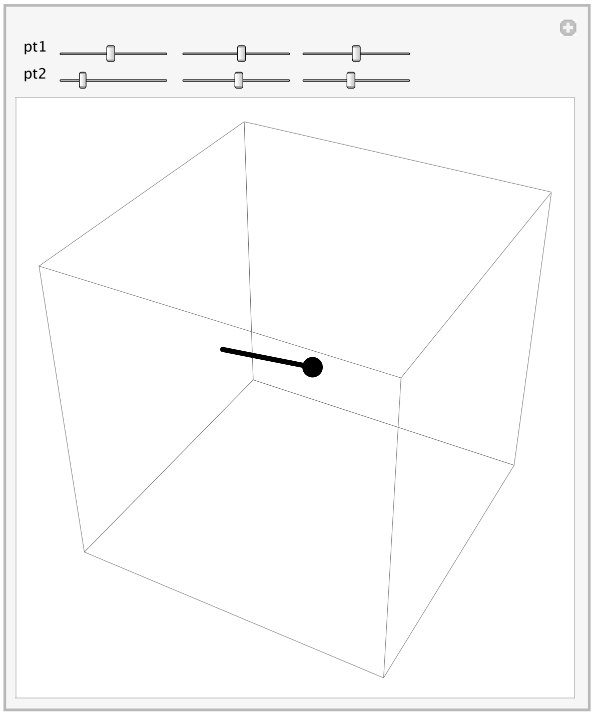
Three-axis variables are also supported as "XYZ". When using a three-axis joystick controller, the axes will correspond to the three degrees of freedom of the joystick. When using a dual-joystick gamepad, each joystick only has two degrees of freedom. In this case the "XYZ" axis is linked to the *x* and *y* directions of the left joystick plus the *x* direction of the right joystick. Whether this makes sense depends on the example: examples written specifically to take advantage of a three-degrees-of-freedom joystick may not work well with any other kind of controller.

```
Manipulate[Graphics3D[{PointSize[0.05], Point[pt]}, PlotRange → 1],
 "XYZ" → {pt, {-1, -1, -1}, {1, 1, 1}}]
```



Some controllers actually provide 6 analog degrees of freedom, which can be referred to as `"XYZ"` and `"XYZ2"`. For example, if you have a 3Dconnexion SpaceNavigator control, the following example will let you explore its three spatial and three angular degrees of freedom. If you do not have one, the example will be unsatisfactory.

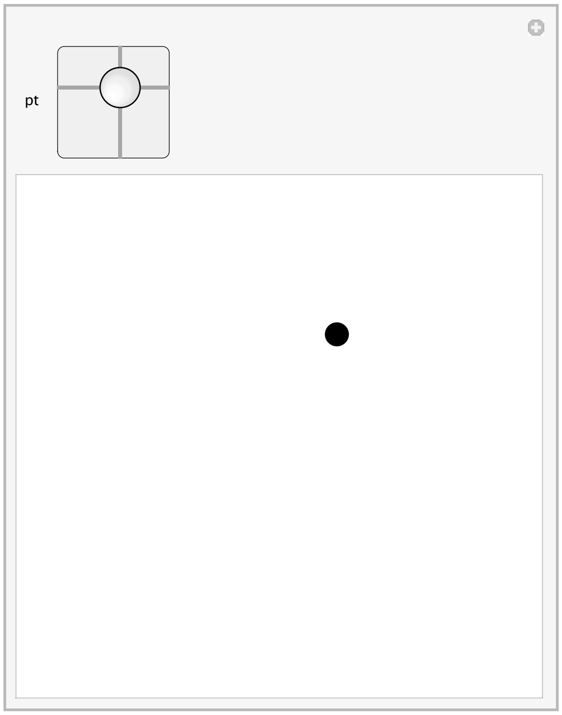
```
Manipulate[
 Graphics3D[{Thickness[0.01], PointSize[0.04], Point[pt1], Line[{pt1, pt1 + pt2}]},
  PlotRange → 1], "XYZ" → {pt1, {-1, -1, -1}, {1, 1, 1}},
 "XYZ2" → {pt2, {-1, -1, -1}, {1, 1, 1}}, ControllerMethod → "Absolute"]
```



(Note that 3D variables in `Manipulate` are available whether you are using a controller or not, but are not generally useful other than in connection with a joystick or gamepad.)

A typical gamepad has two x-y controllers, named `"XY"` and `"XY2"`. But what is less obvious is that there are also several more pseudo-analog axes available, generated by considering groups of four buttons as four directions: up, down, left, and right. For example, the "hat", a directional pad on the left side of a Logitech Dual Action gamepad, can be referenced as `"XY3"`.

```
Manipulate[Graphics[{PointSize[0.05], Point[pt]}, PlotRange → 1],
 "XY3" → {pt, {-1, -1}, {1, 1}}]
```

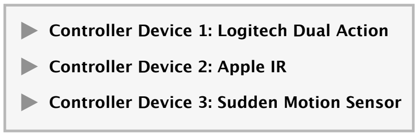


Two additional axes (`"XY4"`) are defined by the four buttons on the right side of the gamepad, and the four buttons on the front face (`"XY5"`). Needless to say this is highly specific to the Logitech brand of controller, but others typically have similar groups of buttons.

These pseudo-analog axes act just like real analog ones, except that in velocity-linked mode they always progress at the same speed, and in absolute mode they are always pegged at the full-left, center, or full-right positions.

When attempting to hook up specific axes it is often confusing trying to figure out which one is which on a given controller. The function `ControllerInformation[]` can be used to figure this out interactively. With your gamepad or joystick plugged in, evaluate this input (you have to evaluate it in your session of *Mathematica* with your controller plugged in to get current information).

```
ControllerInformation[]
```

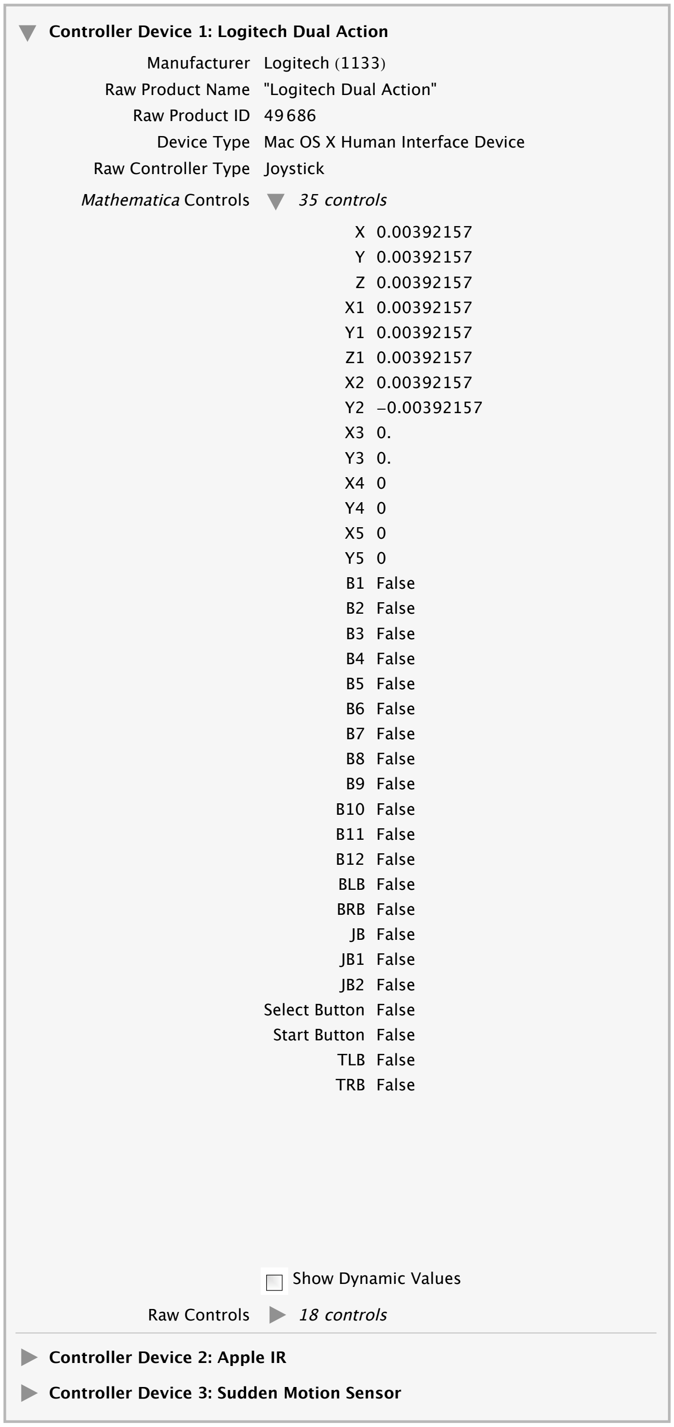


```
▶ Controller Device 1: Logitech Dual Action
▶ Controller Device 2: Apple IR
▶ Controller Device 3: Sudden Motion Sensor
```

Depending on what type of computer you are using you may get several built-in controls. For example, Macintosh laptops typically contain a position sensor that reads out the orientation of the computer at all times. This information is available and can be used with `Manipulate`, but is not used by default (otherwise all `Manipulate` functions that run on such laptops would constantly move around as you tilted the computer, which some might consider a nuisance).

Locate the controller you want to examine and click the disclosure triangle next to its name to open a panel of information, then open the ***Mathematica* Controls** subsection to see a list of all the available axis names.

```
ControllerInformation[]
```

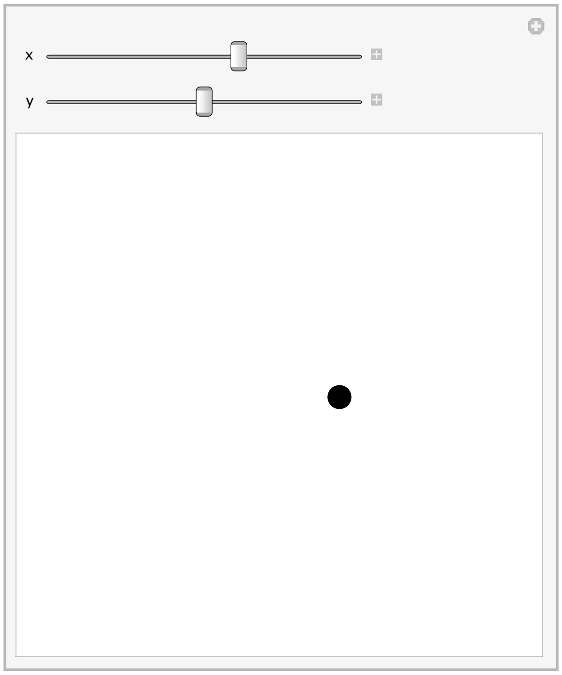


```
▼ Controller Device 1: Logitech Dual Action
           Manufacturer  Logitech (1133)
       Raw Product Name  "Logitech Dual Action"
         Raw Product ID  49686
            Device Type  Mac OS X Human Interface Device
    Raw Controller Type  Joystick
   Mathematica Controls  ▼ 35 controls
                                    X  0.00392157
                                    Y  0.00392157
                                    Z  0.00392157
                                   X1  0.00392157
                                   Y1  0.00392157
                                   Z1  0.00392157
                                   X2  0.00392157
                                   Y2  −0.00392157
                                   X3  0.
                                   Y3  0.
                                   X4  0
                                   Y4  0
                                   X5  0
                                   Y5  0
                                   B1  False
                                   B2  False
                                   B3  False
                                   B4  False
                                   B5  False
                                   B6  False
                                   B7  False
                                   B8  False
                                   B9  False
                                  B10  False
                                  B11  False
                                  B12  False
                                  BLB  False
                                  BRB  False
                                   JB  False
                                  JB1  False
                                  JB2  False
                        Select Button  False
                         Start Button  False
                                  TLB  False
                                  TRB  False

                          ☐ Show Dynamic Values
           Raw Controls  ▶ 18 controls
▶ Controller Device 2: Apple IR
▶ Controller Device 3: Sudden Motion Sensor
```

If **Show Dynamic Values** is checked, the values displayed in the panel will update in real time as you wiggle the controller or push its buttons, allowing you to easily determine which button corresponds to which named axis. (Do not forget the quotes around the axis names when using them in `Manipulate`.)

The option `ControllerMethod` can only be used at the level of the whole `Manipulate` to change the linking from velocity-based to absolute. If you want to make some axes absolute and some velocity-based, add `"Absolute"` to the name of any axes you want to have linked absolutely, as in this example, which has a velocity-based x direction and an absolute y direction.

```
Manipulate[Graphics[{PointSize[0.05], Point[{x, y}]}, PlotRange → 1],
 "X" → {x, -1, 1}, "YAbsolute" → {y, -1, 1}]
```



The opposite of `"Absolute"` in this notation is `"Relative"`, as in `"XRelative"`, etc.

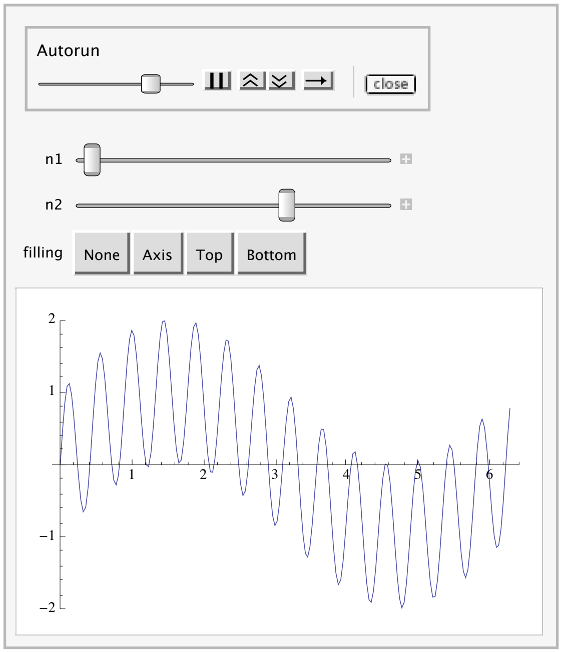
Autorun

In many ways `Manipulate` is a big improvement over simple linear animations. Rather than running through a fixed sequence, `Manipulate` lets you move back and forth at will. But what if you do not want to have to move a slider by hand? One option is to use the ⊞ icon next to each slider to open a panel with animation controls. A `Manipulate` with one variable being animated is virtually equivalent to `Animate`.

But if you have multiple variables and want to see the effect of changing all of them, it is inconvenient to use the individual animation controls. The **Autorun** feature of `Manipulate` solves this problem by providing a single animation control that runs all the variables through their ranges of values.

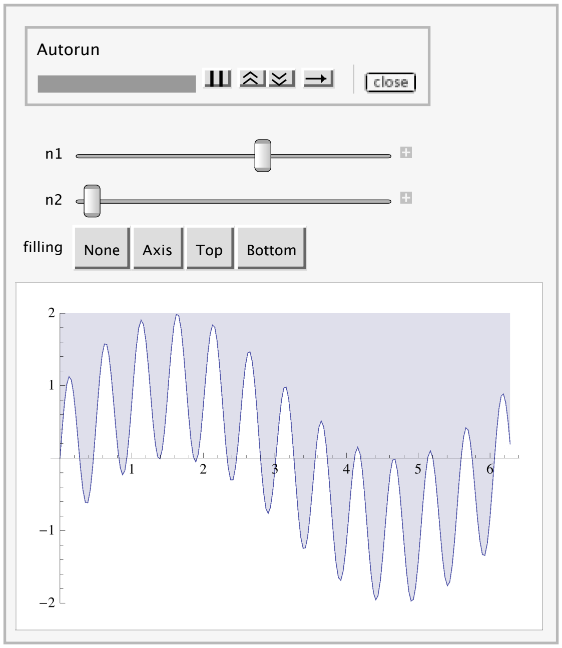
Click the ⊕ menu in the top right corner of a `Manipulate` output and select **Autorun** from the bottom of the menu. You will see an **Autorun** panel appear at the top of the `Manipulate`, containing animation controls and a close button. By default the animation runs each individual variable through its range of values, leaving the others at their default values. As with any animation control, you can change the speed and direction, or click the slider to move through the animation manually. The **Autorun** animation slider acts as a sort of master control driving all the other controls in a defined order.

```
Manipulate[Plot[Sin[n1 x] + Sin[n2 x], {x, 0, 2 Pi}, Filling → filling, PlotRange → 2],
 {n1, 1, 20}, {n2, 1, 20}, {filling, {None, Axis, Top, Bottom}}]
```



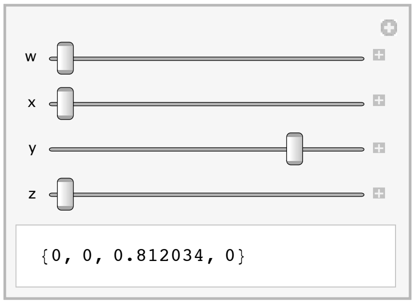
The default behavior of **Autorun** simulates something you could do yourself with the mouse, moving one control at a time. If you add the option `AutorunSequencing -> All` to the `Manipulate` input, the **Autorun** command in the resulting output will instead move all the controls simultaneously, as you can see in this example. This feature works better for some examples than for others.

```
Manipulate[Plot[Sin[n1 x] + Sin[n2 x], {x, 0, 2 Pi}, Filling → filling, PlotRange → 2],
 {n1, 1, 20}, {n2, 1, 20}, {filling, {None, Axis, Top, Bottom}},
 AutorunSequencing → All]
```



You can also use `AutorunSequencing` to exclude certain controls from the **Autorun** animation, or change the order in which the controls are animated. In the following example, the third control is animated first, then the first control, then the fourth, and the second control is left at its default value.

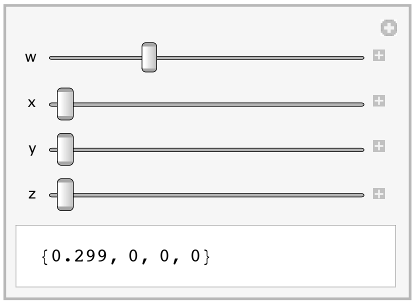
```
Manipulate[{w, x, y, z}, {w, 0, 1}, {x, 0, 1},
 {y, 0, 1}, {z, 0, 1}, AutorunSequencing → {3, 1, 4}]
```



```
{0, 0, 0.812034, 0}
```

AutorunSequencing allows you to specify the duration reserved for the animation of a particular control. This setting reserves two seconds for the first control, two seconds for the third control, and ten seconds for the fourth control. The second control is skipped as before.

```
Manipulate[{w, x, y, z}, {w, 0, 1}, {x, 0, 1}, {y, 0, 1},
 {z, 0, 1}, AutorunSequencing → {{1, 2}, {3, 2}, {4, 10}}]
```



```
{0.299, 0, 0, 0}
```

One reason to care about the details of AutorunSequencing is that it is possible to use the Export command to automatically generate an animation video (in, for example, QuickTime or Flash format). By default Export will generate an animation by running the Manipulate through one **Autorun** cycle.

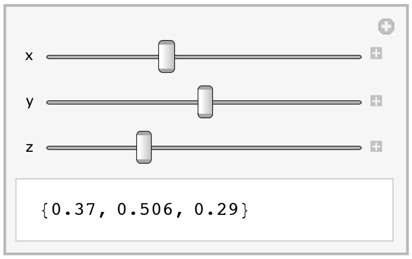
If AutorunSequencing does not give you enough control over the animation sequence, you can use the **Bookmarks** feature described in the next section to define a list of "way points"—combinations of parameter values—and then create an animation that smoothly interpolates through those defined points. This allows complete control over the exact path of the animation.

Bookmarking Combinations of Parameter Values

Manipulate functions, particularly when they have many controls, can be used to find a needle in a haystack: a particular combination of multiple parameter values that yields a particularly interesting result. When you have found a set of values like that, you might want to save it for future reference. Manipulate provides several features for doing this through the ⊕ menu in the top-right corner of the Manipulate output.

To get a single value out into a form you can use as a static input, use the **Paste Snapshot** command from the menu. The result will be inserted as a new cell below the `Manipulate` output.

```
Manipulate[{x, y, z}, {x, 0, 1}, {y, 0, 1}, {z, 0, 1}]
```



```
{0.37, 0.506, 0.29}
```

Here is the result of using **Paste Snapshot** with this example.

```
DynamicModule[{x = 0.37, y = 0.506, z = 0.29}, {x, y, z}]
```

The three current values have been copied into the variable definition block of a `DynamicModule`, and the first argument has been copied into the body. (`DynamicModule` is used because in cases where the body contains explicit uses of `Dynamic` this will result in more correct functioning. Depending on what you want to do with the result, you are of course free to replace `DynamicModule` with `Module`, `With`, or `Block` without needing to make any other changes to the expression. Or you can copy/paste the block of assignments into other code you are building, etc. The differences between `Module` and `DynamicModule` are discussed in further detail in "Advanced Dynamic Functionality".)

If, instead of immediately extracting that location, you just want to remember it so that it is easy to visit in the future, select **Add To Bookmarks** from the menu. That will bring up a panel that will let you name the bookmark and add it to list of bookmarks which are known to this `Manipulate` by clicking the add button, or cancel the addition by clicking the close button.

After adding a bookmark, the name you have specified for it will appear in the menu. Selecting its name from that menu will cause all the parameters to snap back to the values they had when that bookmark was added. Also note that every `Manipulate` remembers the initial settings of all its controls, and you can snap back to those values by choosing **Initial Settings** in this menu.

Once you start placing bookmarks, there are two other items in the menu which become relevant: **Paste Bookmarks** and **Animate Bookmarks**.

Bookmarks are lists of locations in a given parameter space, and you can extract the raw data in that list by choosing the **Paste Bookmarks** item. Every element of the resulting list is of the form `bookmarkName :> parameterValues`. This list is syntactically appropriate for reinserting into `Manipulate` input as the setting for the `Bookmarks` option. (This allows you to, for example, modify the bookmarks by manual editing, or run a program on them, before restoring them as active bookmarks in a new `Manipulate` output.)

The **Animate Bookmarks** menu command works much like the **Autorun** command described in the previous section, except that instead of animating each parameter through its range of values, it creates an animation that interpolates through the points specified by the bookmarks.

The interpolation which occurs when animating bookmarks is done internally via the `Interpolation` command. `Manipulate` even accepts the `InterpolationOrder` option to adjust how the animation proceeds from one point to the next. The default value of `Automatic` performs quadratic interpolation if there are enough bookmarks, and linear interpolation otherwise.

When a `Manipulate` output containing explicit bookmarks is exported to a video animation format using `Export`, the resulting video will be one cycle through the sequence generated by **Animate Bookmarks**. (If no bookmarks are present, the result is one cycle of **Autorun**.)

Advanced Manipulate Functionality

This tutorial covers advanced features of the `Manipulate` command. It assumes that you have read "Introduction to Manipulate" and thus have a good idea what the command is for and how it works overall.

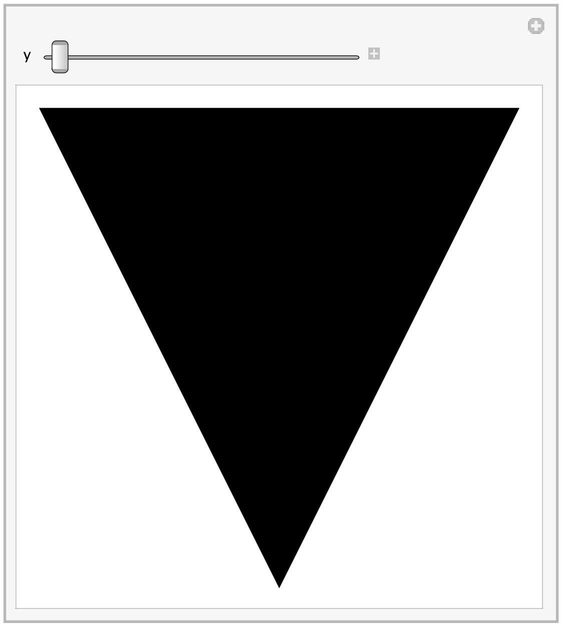
This tutorial also, in places, assumes a familiarity with the lower-level dynamic mechanism covered in "Introduction to Dynamic" and "Advanced Dynamic Functionality".

Please note that this is a hands-on tutorial. You are expected to actually evaluate each input line as you reach it in your reading, and watch what happens. The accom panying text will not make sense without evaluating as you read.

Controlling Automatic Reevaluation

Some `Manipulate` examples "spin," continually reevaluating their contents even when no sliders are being moved. Sometimes this is in fact exactly what you intend. For example, here is a droopy triangle, which always sags down in the middle. You can drag it back up using the slider, but as soon as you stop moving, it starts falling down again.

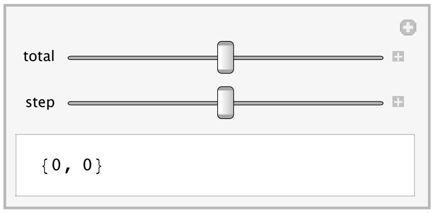
```
Manipulate[
 y = Max[-1, y - 0.05]; Graphics[Polygon[{{-1, 1}, {0, y}, {1, 1}}], PlotRange → 1],
 {{y, 1}, -1, 1}]
```



This happens because the variable `y` is being changed by the code in the first argument, and the system, correctly and helpfully, notices that since the value of `y` has changed, the contents need to be evaluated and displayed again, which in turn causes the value of `y` to change again, and so on, until, in this case, we reach a stable point at `y= -1`. After that the value of `y` no longer changes, and the contents are no longer continually redrawn, until you touch the slider again. (If you have a CPU activity monitor on your system you can verify that while the triangle is drooping, *Mathematica* is using CPU time, but once it reaches the bottom, *Mathematica*'s CPU usage stops.)

Of course it is possible to construct examples that do not stop. Here we declare two variables, both initialized to zero, and include code in the body of the `Manipulate` to continuously update the value of one of them based on the value of the other.

```
Manipulate[total = total + step; {step, total},
 {{total, 0}, -1000, 1000, 1}, {{step, 0}, -10, 10, 1}]
```



```
{0, 0}
```

Any time the step size is moved away from zero, the content area will continually update, and a CPU monitor will indicate that *Mathematica* is using CPU time. This will go on for as long as you let it. (Fortunately it does not totally consume the CPU, and other activities in the front end are not hindered by this activity; you can keep editing, evaluating, etc., while it is running. You may think of it sort of like an animated image or applet running in a web browser.)

In some cases, however, the continual reevaluation is pointless and undesirable. Consider this somewhat contrived example: any time it is present on screen (i.e., in an open window and not scrolled offscreen to the point where no part of it is visible), it will constantly reevaluate itself, consuming CPU time even though nothing is changing.

```
Manipulate[
 temp = n;
 temp = temp^3;
 Graphics[{Thickness[0.01], Line[{{0, 0}, {n, temp}}]}, PlotRange → 1],
 {n, -1, 1}]
```

This happens because the variable `temp` has its value changed during the evaluation, even if the value of `n` has not changed (i.e. it is reset to two different values each time through). The spinning is pointless because the value of `temp` is set before it is ever used.

Another way to get inadvertent and pointless spinning is to make a function definition or other complex assignment in the body of the `Manipulate`, as in this example.

```
Manipulate[
 f[x_] := x^3;
 Graphics[{Thickness[0.01], Line[{{0, 0}, {n, f[n]}}]}, PlotRange → 1],
 {n, -1, 1}]
```

In both these cases, the problem can be solved by making the offending variables be local variables inside a `Module`. (This is good programming practice in any case, quite aside from any desire to avoid pointless updating.)

```
Manipulate[Module[{temp},
  temp = n;
  temp = temp^3;
  Graphics[{Thickness[0.01], Line[{{0, 0}, {n, temp}}]}, PlotRange → 1]],
 {n, -1, 1}]
```

```
Manipulate[Module[{f},
  f[x_] := x^3;
  Graphics[{Thickness[0.01], Line[{{0, 0}, {n, f[n]}}]}, PlotRange → 1]],
 {n, -1, 1}]
```

Nothing you do to local `Module` variables will cause retriggering, because it is part of the definition of `Module` that values do not survive from one invocation to the next (therefore the result will not be any different the next time around just because of anything done to the value of a local variable during the current cycle).

Another solution is to use the `TrackedSymbols` option to `Manipulate` to control which variables are allowed to cause updating behavior. The default value, `Full`, means that any symbols that appear explicitly (lexically) in the first argument will be tracked. (This means, among other things, that temporary variables and other such problems inside the definitions of functions you use in your `Manipulate` example will not cause infinite reevaluation problems, because they do not occur explicitly in the first argument, only indirectly through functions you call.)

Taking the second example, if for some reason you do not want `f` to be a local `Module` variable, and you cannot move its definition outside the `Manipulate` (there are sometimes good reasons for both those conditions, in more complicated cases), you can use `TrackedSymbols` to disable updating triggered by `f`:

```
Manipulate[
 f[x_] := x^3;
 Graphics[{Thickness[0.01], Line[{{0, 0}, {n, f[n]}}]}, PlotRange → 1],
 {n, -1, 1}, TrackedSymbols :> {n}]
```

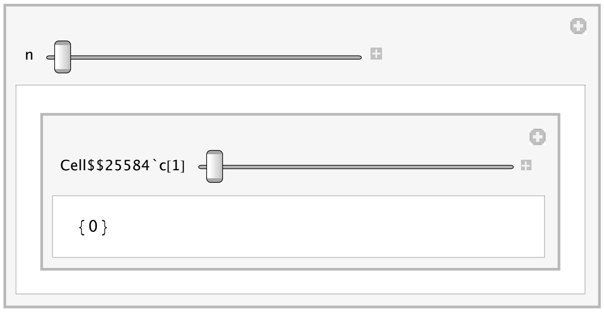
This example updates the content area only when `n` changes its value as a result of moving the slider.

The subject of when exactly a given dynamic expression will be updated is complex, and is addressed in "Introduction to Dynamic" and "Advanced Dynamic Functionality". In reading those, keep in mind that `Manipulate` simply wraps its first argument in `Dynamic` and passes the value of its `TrackedSymbols` option to a `Refresh` inside that. Everything to do with updating is handled by that `Dynamic` and `Refresh`.

Nesting Manipulate

You can put one `Manipulate` inside another. For example, here we use a slider in an outer `Manipulate` to control the number of sliders in the inner `Manipulate`.

```
In[26]:= Manipulate[With[{value = Table[c[i], {i, 1, n}],
            controls = Sequence @@ Table[{c[i], 0, 1}, {i, 1, n}]},
           Manipulate[value, controls]], {n, 1, 10, 1}]
```

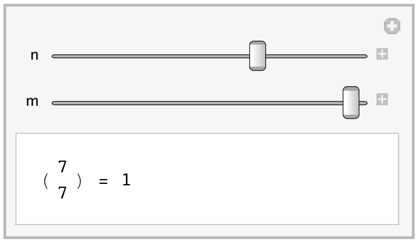
Out[26]=


While nesting Manipulate many levels deep is possible, and will work, it is probably not the most useful feature in the world. But by nesting once, you have in effect created a parameterized user interface construction interface. The outer Manipulate allows you to control parameters that determine the user interface presented by the inner Manipulate. With some slightly more complex programming than in the previous example, remarkable things can be done.

Interdependent Controls

It is possible to make the range of one slider in a Manipulate depend on the position of another slider. For example, the function Binomial[n, m] makes sense only when $m <= n$, so you might want to make an m-slider whose range is from 1 to the current value of n. You can do this simply by using n in the variable specification for m, like this.

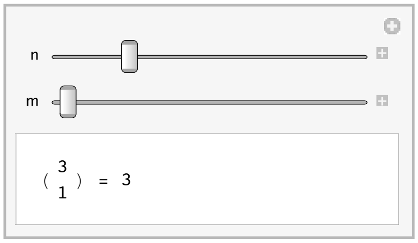
```
Manipulate[Row[{"(", Column[{n, m}, Center], ") = ", Binomial[n, m]}],
 {n, 1, 10, 1}, {m, 1, n, 1}]
```



Note that if you first move both sliders part way towards the right, then move the n-slider left, the m-slider will automatically move to the right, because its maximum is getting smaller. If you move n far enough to the left, to the point where it becomes smaller than the current value of m, the m-slider will display a red "out of range" indicator, because m is now larger than its maximum allowed value.

You might wonder why *m* is not automatically reset to the current maximum, when the maximum is set lower than its current value. The reason is that sometimes it is preferable to leave the value alone, and if you want to have it reset automatically, it is easy to do manually. For example, you can add an `If` statement to the code in the first argument.

```
Manipulate[If[m > n, m = n];
 Row[{"(", Column[{n, m}, Center], ") = ", Binomial[n, m]}],
 {n, 1, 10, 1}, {m, 1, n, 1}]
```

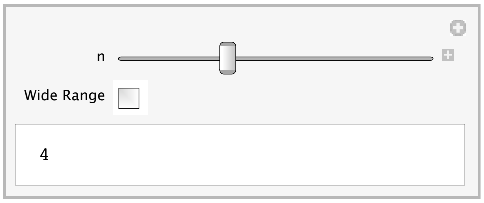


```
( 3 ) = 3
  1
```

Generally speaking, you can use `Manipulate` variables in the definition of other variables without restriction, though it is certainly possible in this way to create peculiar interactions that are more confusing than helpful.

This example shows another variation, using a check box to control the range of a slider: something like this can be useful in cases where you want to provide fine and coarse ranges, for example.

```
Manipulate[n,
 {n, 1, If[wide, 100, 10], 1}, {{wide, False, "Wide Range"}, {False, True}}]
```



```
4
```

Dealing with Slow Evaluations

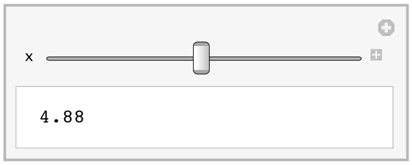
`Manipulate` does not precompute all the possible output values you could reach by moving its sliders: that would be completely impractical for all but the most trivial cases. That means it has to calculate, format, and display the current value in real time as each slider is being dragged. Obviously no matter how fast your computer, there is a limit to how much computation can be done in a finite amount of time, and if the expression you use in the first argument to `Manipulate` takes more than about a second to evaluate, you will not have a very satisfactory experience using the `Manipulate`.

Many very interesting and powerful computations can be done in under a second, and as comput ers get faster the range will only increase (people are not getting any faster, so the amount of time available before the example seems too sluggish should remain unchanged for quite a while). But some computations just cannot be done that fast, and some alternative is necessary if you want to use them within `Manipulate`. Fortunately there are several good ways of dealing with slow evaluations.

For purposes of this section we are going to use `Pause` to simulate a slow evaluation. The main reason for this is that any actual computation would run at such widely differing speeds on different users' computers that it would be hard to illustrate the point with any one example. So where you see a `Pause` command, please imagine that something terribly complex and interesting is being done, resulting in a fantastically detailed and enlightening output.

To get a feel for the problem, try dragging the following slider. While this example is not unacceptable, it is on the borderline of something not worth playing with. If the delay is increased to several seconds, it becomes quite pointless. (And beyond 5 seconds you will start seeing `$Aborted` instead of the number, because the system is protecting itself from unreasonably long evaluations, which block other activity in the front end in this situation.)

```
Manipulate[Pause[1]; x, {x, 0, 10}]
```

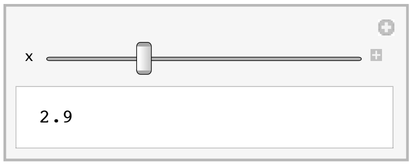


```
4.88
```

```
Manipulate[Pause[1]; x, {x, 0, 10}]
```

The simplest improvement is to add the option `ContinuousAction -> False` to the `Manipulate`.

```
Manipulate[Pause[1]; x, {x, 0, 10}, ContinuousAction → False]
```



```
2.9
```

In this example the slider moves smoothly and instantaneously as it is dragged, but the value in the output area does not attempt to track in real time. Instead it updates only when the slider is released.

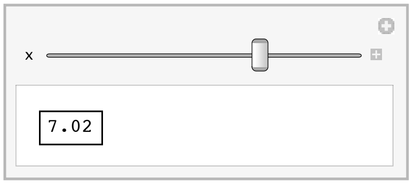
A more subtle difference is that when the value updates in this example, it does so without blocking other activity in the front end. You can see this by the fact that the cell bracket becomes outlined for a second each time the slider is released, and you can continue typing or doing other work in the front end during that second. There is no 5-second limit to such non-blocking evaluations, so by using the `ContinuousAction -> False` option, arbitrarily long evaluations can be used. (Though something that takes a minute is still probably better done as a normal Shift+Return evaluation than inside `Manipulate`.)

A more sophisticated alternative is to use the `ControlActive` function to present an alternative, simpler and faster display while the slider is being dragged, and do the long computation only when it is released.

`ControlActive` takes two arguments: the first is returned if the expression is evaluated while a control (e.g. a slider) is currently being dragged with the mouse, and the second if no control is currently active. (See the documentation for `ControlActive` for some fine print about exactly when which argument is returned.)

In this example we use just `x` as the preview to be displayed while the slider is being dragged, and `x` with a box around it, plus a second of delay, as the final display to be presented when the slider is released. Note that we have removed the `ContinuousAction -> False` option from the example above.

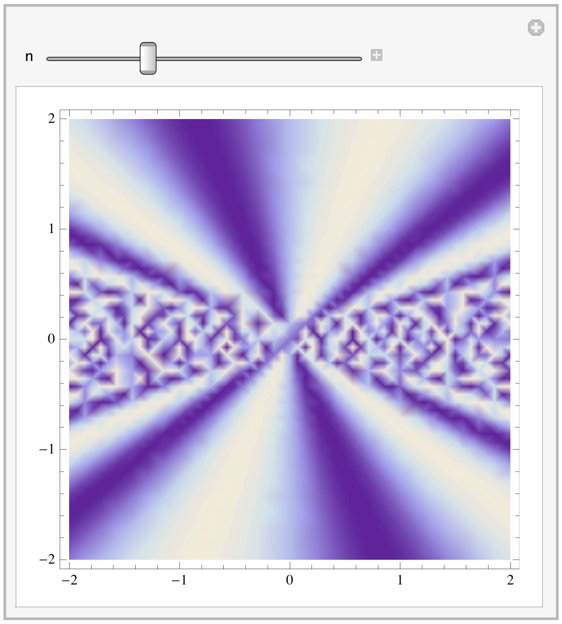
```
Manipulate[ControlActive[x, Pause[1]; Framed[x]], {x, 0, 10}]
```



Note that the cell bracket is outlined, indicating a nonblocking evaluation, only when the slider is released. While the slider is being dragged, evaluations are done in a blocking way for maximum interactive performance.

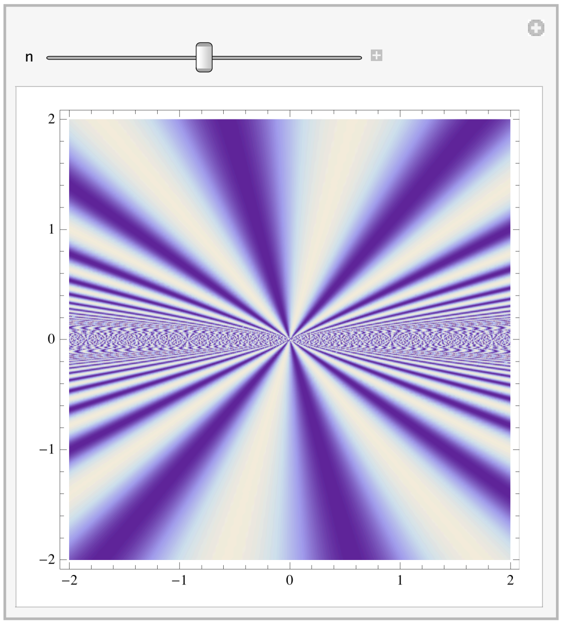
Here is a slightly more realistic example of where `ControlActive` can be useful. This example shows how the default behavior of `DensityPlot` is to use fewer sample points while the slider is being dragged.

```
Manipulate[DensityPlot[Sin[n x / y], {x, -2, 2}, {y, -2, 2}], {n, 1, 10}]
```



But even the higher number used after the slider is released is not enough to produce a satisfactory plot. If we just set a fixed, larger number of plot points, the result is pretty, but interactive performance is not good enough.

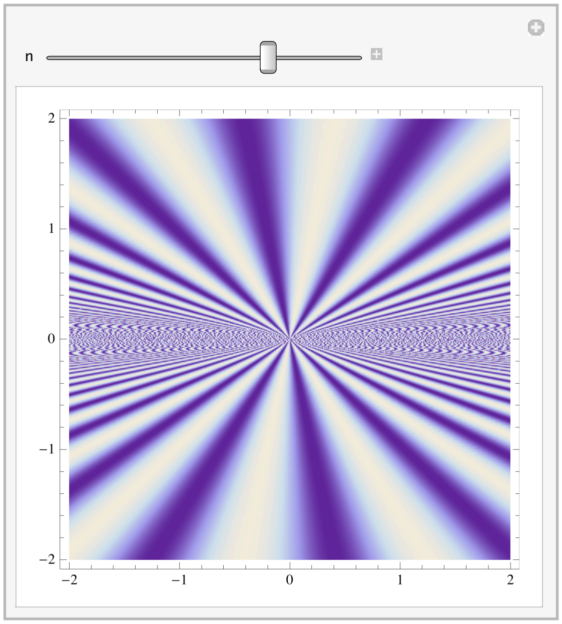
```
Manipulate[
 DensityPlot[Sin[n x / y], {x, -2, 2}, {y, -2, 2}, PlotPoints → 150], {n, 1, 10}]
```



(There is still a difference between the active and inactive forms, because by default several different options, not just `PlotPoints`, depend on `ControlActive`.)

The optimal combination of speed and quality can be achieved by using `ControlActive` explicitly in the value of the `PlotPoints` option.

```
Manipulate[DensityPlot[Sin[n x / y], {x, -2, 2},
  {y, -2, 2}, PlotPoints → ControlActive[30, 150]], {n, 1, 10}]
```



The result is an example that displays a crude, but virtually instantaneous, preview of the graphic, then spends many seconds constructing a high-resolution version when the slider is released.

The next section explains a more complex solution that works in cases where some changes to the parameters require a slow evaluation while others could update the display much more rapidly.

Using Dynamic inside Manipulate

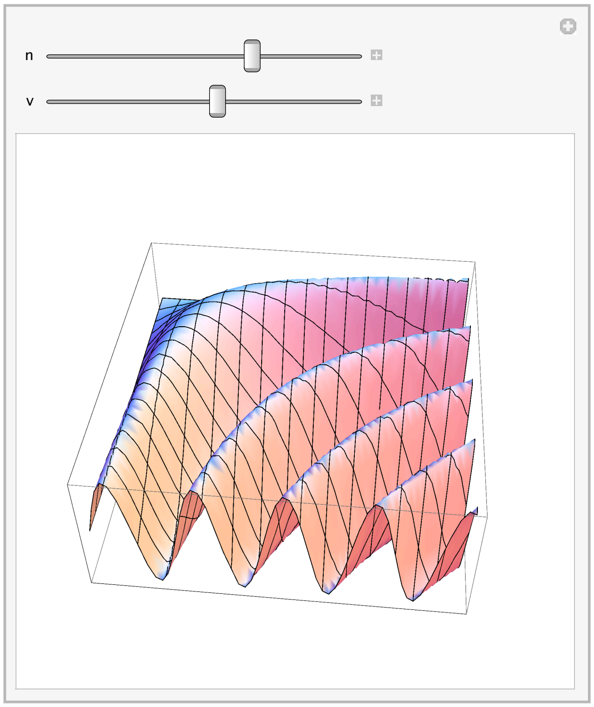
You might want to read "Introduction to Dynamic" before finishing this section, as we refer to the use of explicit `Dynamic` expressions, which are not explained in this tutorial.

When you move the sliders (or other controls) in a `Manipulate`, the expression given in the first argument is reevaluated from scratch for each new parameter value. "Dealing with Slow

Evaluations" discusses a number of general things that can be done if evaluation of this first argument is too slow to allow smooth interactive performance of the `Manipulate`. But in some cases it is possible to separate the evaluation into slower and faster parts, and thereby achieve much better performance.

Consider this example where one slider controls the contents of a 3D plot, while the other controls its viewpoint.

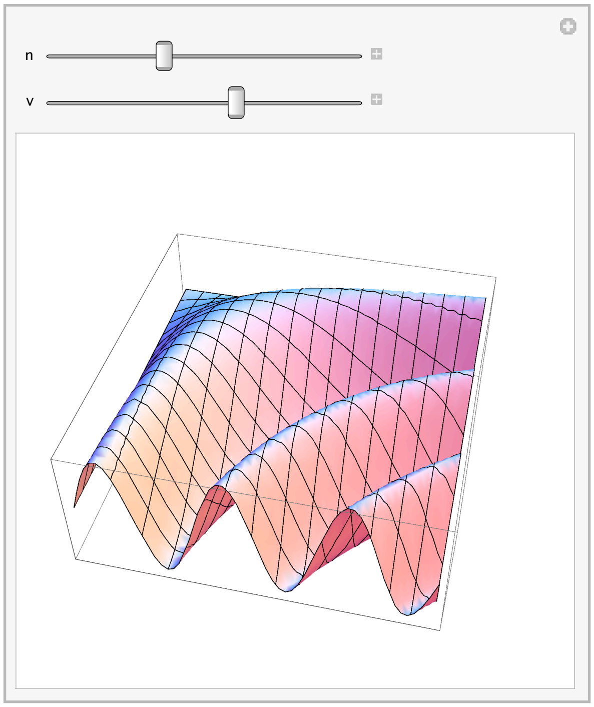
```
Manipulate[Plot3D[Sin[n x y], {x, 0, 3}, {y, 0, 3}, ViewPoint → {2, v, 2},
  SphericalRegion → True, Ticks → None], {n, 1, 4}, {v, -2, 2}]
```



When the n-slider is moved, it is obviously necessary to recompute the 3D plot, because it actually changes shape. The plot becomes jagged while the slider is being dragged, then improves shortly after you release it, which is correct and as expected. When you move the v-slider, on the other hand, there is no point in recomputing the function, because only the view-point has changed. But `Manipulate` has no way of knowing this (and in more complex cases it is genuinely impossible for this kind of distinction to be made in any automatic way), so the whole plot is regenerated from scratch each time v is changed.

To improve this example, we can tell `Manipulate` that the `ViewPoint` option should be updated separately from the rest of the output, which we can do by wrapping `Dynamic` around the right-hand side of the option.

```
Manipulate[Plot3D[Sin[n x y], {x, 0, 3}, {y, 0, 3}, ViewPoint → Dynamic[{2, v, 2}],
  SphericalRegion → True, Ticks → None], {n, 1, 4}, {v, -2, 2}]
```



Notice that now when the *v*-slider is moved, the plot does not revert to the jagged appearance, and actually rotates faster than before. This is because the plot is no longer being regenerated with each movement.

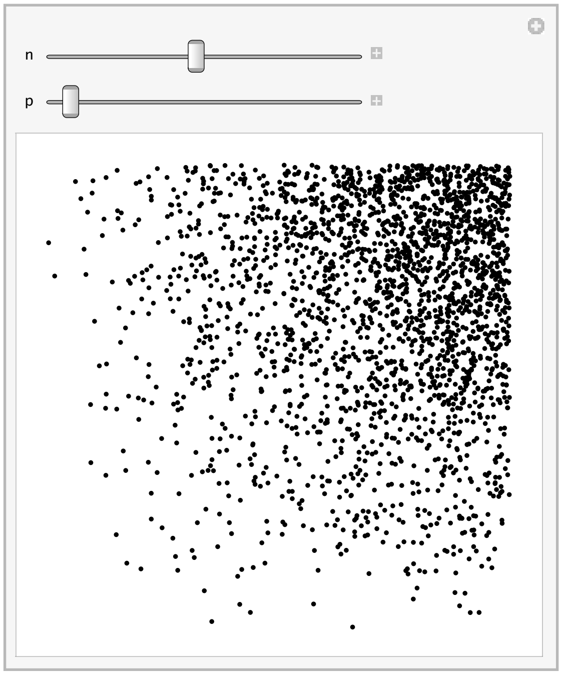
To explain exactly why this works requires an understanding of the internals of the `Dynamic` mechanism explained in "Introduction to Dynamic" and "Advanced Dynamic Functionality". In short, `Manipulate` always wraps `Dynamic` around the expression given in its first argument, and normally any changes to variables used in the first argument will trigger updates of that `Dynamic`. But when a variable occurs only inside an explicit `Dynamic` nested inside the one implicitly created by `Manipulate`, an update of the outer `Dynamic` will not be triggered, only an update of the inner `Dynamic` in which it resides.

Explaining the full range of what is possible by using `Dynamic` explicitly inside `Manipulate` is beyond the scope of this document, but another common case worth looking at involves a situation where the slow part of some computation involves only some of the input variables.

In the next example we construct a large table of numbers (using `RandomReal` in this case, but in a real-world example it might be a much more complicated, slower computation, or even one involving reading external data from the network). After constructing the data, we display it using a fairly simple, fast function (illustrated here by just raising the coordinate values to a power).

Note that when the n-slider is moved, the number of points changes, and they jump around because a new random set is generated each time. But when the p-slider is moved, updates are smoother, and the points do not jump around. This is because the inner `Dynamic` wrapped around the use of p prevents the first argument as a whole from being reevaluated. Thus no new random points are generated only the presentation of the existing ones is updated.

```
Manipulate[
 data = RandomReal[{0, 1}, {n, 2}];
 Graphics[{Point[Dynamic[data^p]]}, AspectRatio → 1],
 {n, 100, 5000, 1}, {p, 0.1, 10}, SynchronousUpdating → False]
```



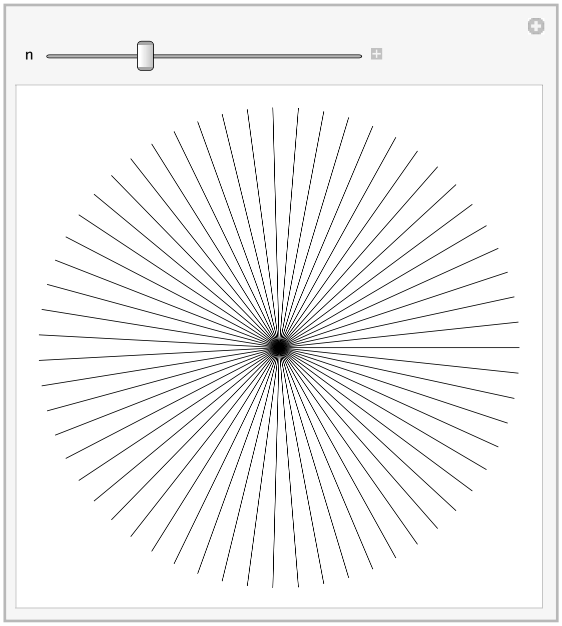
The option `SynchronousUpdating -> False` is used to cause the outer `Dynamic` (the one implicitly created by `Manipulate`) to update asynchronously (visible by the fact that the cell bracket becomes outlined when the *n*-slider is moved). Asynchronous updating does not give quite as smooth updating, but if the evaluation takes a long time, it does not block other activity in the front end.

The inner `Dynamic` uses the default synchronous updating, so when the *p*-slider is moved, updating is smooth and rapid.

It is thus practical, using the technique illustrated here, to make an example that takes many seconds, even minutes, to respond when one slider is changed, yet preserve rapid interactive performance when other controls, which do not require the long computation to be repeated, are changed.

You can also use `Dynamic` inside `Manipulate` to make the output dynamically respond to things other than the values of the `Manipulate`'s control variables. For example, here is an example taken from an earlier section, except that we have made it respond dynamically to the current mouse position.

```
Manipulate[
 Graphics[{Dynamic[With[{pt = MousePosition[{"Graphics", Graphics}, {0, 0}]},
     Line[Table[{{Cos[t], Sin[t]}, pt}, {t, 2. Pi / n, 2. Pi, 2. Pi / n}]]]]},
  PlotRange → 1], {{n, 30}, 1, 200, 1}]
```



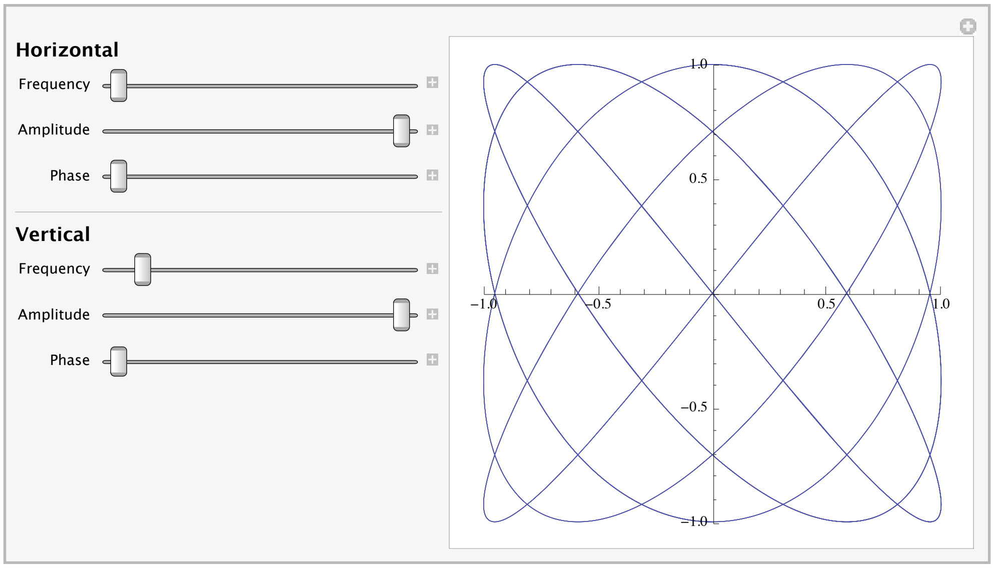
Any time the mouse is over the area of the plot, the center of the lines will follow it (without a click). Consult the documentation for `MousePosition` for further detail.

It is important to remember also that `Manipulate` is not the only way of creating interactive user interfaces in *Mathematica*. `Manipulate` is intended to be a simple, yet powerful, tool for defining user interfaces at a very high level. But when you reach the limits of what it is capable of doing, either in terms of control layout, updating behavior, or interaction with external systems, it is always possible (and often not terribly difficult) to drop to a lower level of interface programming using functions such as `Dynamic` and `EventHandler`.

Dynamic Objects in the Control Area

We saw in "Introduction to Manipulate" that it is possible to add a variety of elements to the control area of a `Manipulate`, for example titles and delimiters, as in this example.

```
Manipulate[ParametricPlot[{a1 Sin[n1 (x + p1)], a2 Cos[n2 (x + p2)]},
   {x, 0, 20 Pi}, PlotRange → 1, PerformanceGoal → "Quality"],
  Style["Horizontal", 12, Bold], {{n1, 1, "Frequency"}, 1, 4},
  {{a1, 1, "Amplitude"}, 0, 1}, {{p1, 0, "Phase"}, 0, 2 Pi},
  Delimiter, Style["Vertical", 12, Bold], {{n2, 5 / 4, "Frequency"}, 1, 4},
  {{a2, 1, "Amplitude"}, 0, 1}, {{p2, 0, "Phase"}, 0, 2 Pi}, ControlPlacement → Left]
```

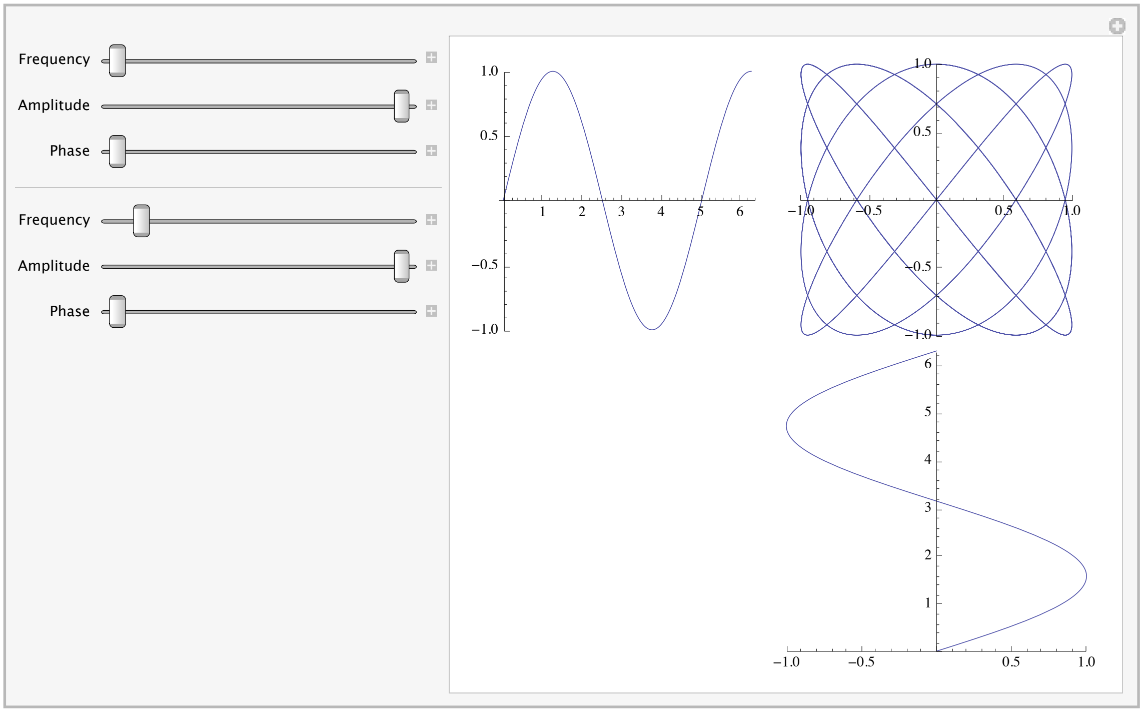


There is in fact virtually no limit to what can be put into the controls area, including arbitrary formatting constructs and dynamic objects, even controls that are not part of the `Manipulate`'s control specifications. Anything placed in the variables sequence that is either a string or has head `Dynamic`, `Style`, or `ExpressionCell` will automatically be interpreted as an annotation to be inserted in the controls area.

You might want to read "Introduction to Dynamic" before finishing this section, as we refer to the use of explicit `Dynamic` expressions, which are not explained in this tutorial.

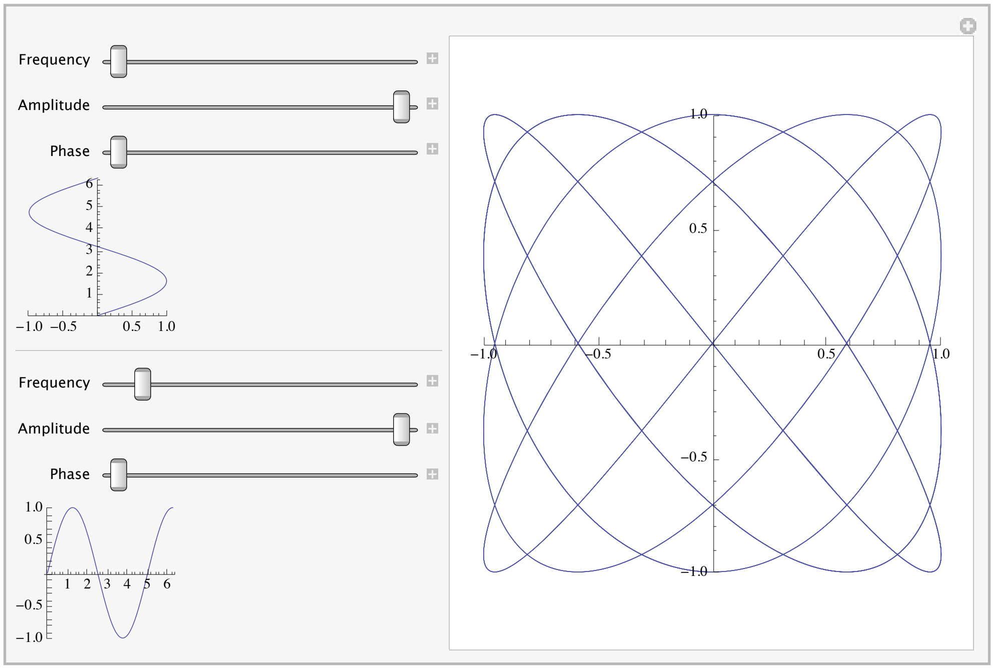
Suppose you want to show plots of the individual x and y sine functions that combine to form the Lissajous figure. You could do this by putting all three functions into the output area, using `Grid` to lay them out.

```
Manipulate[
 Grid[{
   {Plot[a2 Sin[n2 (x + p2)], {x, 0, 2 Pi}, AspectRatio → 1, PlotRange → 1],
    ParametricPlot[{a1 Sin[n1 (x + p1)], a2 Cos[n2 (x + p2)]},
     {x, 0, 20 Pi}, PlotRange → 1, PerformanceGoal → "Quality"]},
   {Null, ParametricPlot[{a1 Sin[n1 (x + p1)], x}, {x, 0, 2 Pi},
     AspectRatio → 1, PlotRange → {{-1, 1}, {0, 2 Pi}}]}
  }],
 {{n1, 1, "Frequency"}, 1, 4},
 {{a1, 1, "Amplitude"}, 0, 1},
 {{p1, 0, "Phase"}, 0, 2 Pi},
 Delimiter,
 {{n2, 5 / 4, "Frequency"}, 1, 4},
 {{a2, 1, "Amplitude"}, 0, 1},
 {{p2, 0, "Phase"}, 0, 2 Pi}, ControlPlacement → Left]
```



There is in fact a lot to be said for this presentation. But suppose you instead want to leave the main output area as it is, with a large, prominent presentation of the Lissajous figure itself, and show the individual sine functions only much smaller, in association with the controls for each direction. You can do this by placing a dynamic plot object into the controls area, as follows.

```
Manipulate[
 ParametricPlot[{a1 Sin[n1 (x + p1)], a2 Cos[n2 (x + p2)]},
  {x, 0, 20 Pi}, PlotRange → 1, PerformanceGoal → "Quality"],
 {{n1, 1, "Frequency"}, 1, 4},
 {{a1, 1, "Amplitude"}, 0, 1},
 {{p1, 0, "Phase"}, 0, 2 Pi},
 Dynamic[ParametricPlot[{a1 Sin[n1 (x + p1)], x}, {x, 0, 2 Pi},
   ImageSize → 100, AspectRatio → 1, PlotRange → {{-1, 1}, {0, 2 Pi}}]],
 Delimiter,
 {{n2, 5 / 4, "Frequency"}, 1, 4},
 {{a2, 1, "Amplitude"}, 0, 1},
 {{p2, 0, "Phase"}, 0, 2 Pi},
 Dynamic[Plot[a2 Sin[n2 (x + p2)], {x, 0, 2 Pi}, ImageSize → 100,
   AspectRatio → 1, PlotRange → 1]], ControlPlacement → Left]
```



Just as the headings in the example at the start of this section were mixed in with the controls simply by listing them in the variable specifications sequence, here we have placed dynamically updated plots in the variable specification sequence. `Dynamic` is used explicitly in these subplots so that they will update when the controls are moved. (The main output area does not need an explicit `Dynamic` because `Manipulate` automatically wraps `Dynamic` around its first argument.)

It is worth briefly discussing here why it is that an example like this can just work. The reason is that the output of `Manipulate` is not a special, fixed object that just connects a set of controls with a single output area. Instead, the output of `Manipulate` is built up using the same formatting, layout, user interface, and dynamic interactivity features that can be accessed at a lower level using the techniques discussed in "Introduction to Dynamic" (which in fact includes an example of how to build up a simple version of `Manipulate` by hand). In some ways the relationship between `Manipulate` and the lower level interactive features is like the relationship between `Plot` and `Graphics`. The result of evaluating a high-level `Plot` command is a low-level `Graphics` object, and if `Plot` is not able to generate the specific graphic you want, you are always free to use `Graphics` directly. You can use the `Prolog` and `Epilog` options to add arbitrary graphical elements to a `Plot`. Furthermore there is no graphical output you can get using `Plot` that you cannot get using `Graphics`. `Plot` has no special access to any features in *Mathematica* unavailable at the lower level.

Likewise, `Manipulate` does not have any special access to features unavailable with lower level functions: there is nothing you can do with `Manipulate` that you cannot do with `Dynamic`, it is just a higher-level, more convenient function for building a certain style of interface.

So when you use dynamic objects in the control labels, as in the example above, you're just adding a couple more `Dynamic` objects to the already fairly complex set of `Panel` objects, `Grid` objects, `Dynamic` objects, and `DynamicModule` objects that constitutes the output of a `Manipulate` command. There is nothing really different there, just more of it, so it should come as no surprise that the new dynamic elements interoperate smoothly with the others.

Although it is not always sensible to do so, it is possible to build completely arbitrary interactive dynamic user interfaces entirely in the controls area of a `Manipulate`.

Custom Control Appearances

You might want to read "Introduction to Dynamic" before reading this section, as we refer to the use of explicit `Dynamic` expressions, which are not explained in this tutorial.

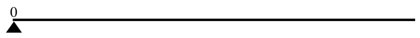
Suppose you want to use a type of control that is not supported by `Manipulate`, for example one you have built yourself using graphics and dynamics. Here is a block of code that defines a custom style of slider, one that shows its value at the thumb position. Do not worry about understanding the details of how this code works, though it is not overly complicated beyond the details of drawing the desired elements in the right places.

```
In[1]:= ValueThumbSlider[v_] := ValueThumbSlider[v, {0, 1}];
        ValueThumbSlider[Dynamic[var_], {min_, max_}, options___] :=
          LocatorPane[Dynamic[If[! NumberQ[var], var = min]; {var, 0}, (var = First[#]) &],
           Graphics[{AbsoluteThickness[1.5], Line[{{min, 0}, {max, 0}}],
             Dynamic[{Text[var, {var, 0}, {0, -1}], Polygon[{Offset[{0, -1}, {var, 0}],
                 Offset[{-5, -8}, {var, 0}], Offset[{5, -8}, {var, 0}]}]}]}],
            ImageSize → {300, 30}, PlotRange → {{min, max} + 0.1 {-1, 1} (max - min), {-1, 1}},
            AspectRatio → 1 / 10],
           {{min, 0}, {max, 0}}, Appearance → None];
```

Here is an example of what this new control looks like: click anywhere to move the thumb around, just like with a normal slider.

```
In[3]:= ValueThumbSlider[Dynamic[xx], {0, 10}]
```

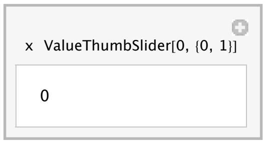
Out[3]=



To use a custom control in `Manipulate`, you include a pure function that is used to generate the control object as part of the variable specification. As long as your function conforms to the convention used by all the built-in control functions, with the variable (inside `Dynamic`) as the first argument and the range as the second argument, you can simply use the function name and the appropriate arguments will be passed to it automatically by `Manipulate`. Here we see our custom control used in a simple `Manipulate`.

```
In[4]:= Manipulate[x, {x, 0, 1, ValueThumbSlider[##] &}]
```

Out[4]=



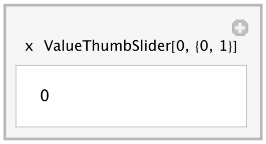
```
x  ValueThumbSlider[0, {0, 1}]

0
```

(The notation ## means that all the arguments will be passed in to the function, not just the first one.)

Note that if you supply the necessary information in the pure function, you do not have to specify the min and max as part of the variable specification.

```
In[5]:= Manipulate[x, {x, ValueThumbSlider[#, {0, 1}] &}]
```

Out[5]=



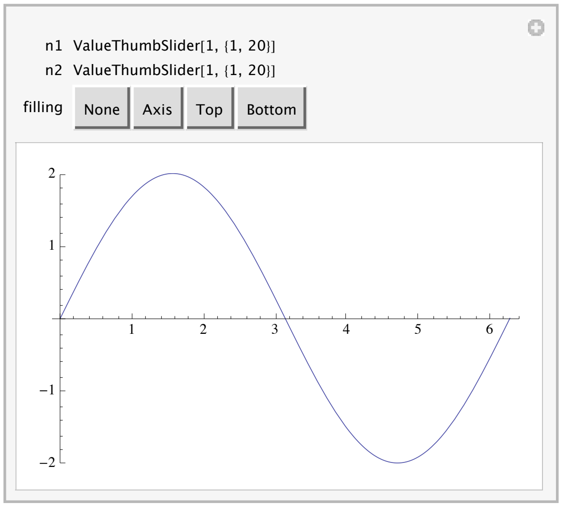
```
x  ValueThumbSlider[0, {0, 1}]

0
```

However, if you do that, then `Manipulate` is not aware of the range you chose to use in the slider, which means that the very nice Autorun feature (as described in the documentation for `Manipulate`) cannot work. So generally it is a good idea to include the range in the variable specification, and let the control function inherit it.

Naturally it is possible to combine standard and custom controls freely; here we use two of our new sliders together with a `SetterBar` supplied automatically by `Manipulate`.

```
In[6]:= Manipulate[Plot[Sin[n1 x] + Sin[n2 x], {x, 0, 2 Pi}, Filling → filling, PlotRange → 2],
         {n1, 1, 20, ValueThumbSlider[##] &},
         {n2, 1, 20, ValueThumbSlider[##] &},
         {filling, {None, Axis, Top, Bottom}}]
```

Out[6]=

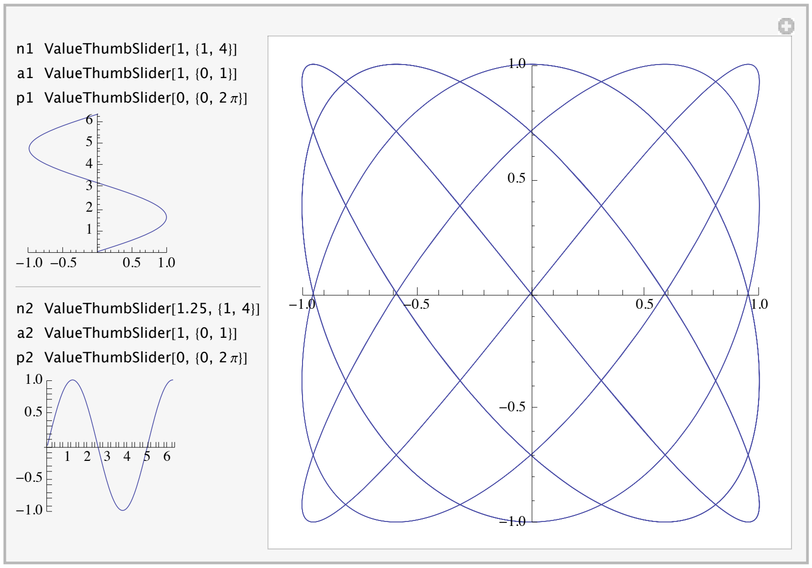


```
n1 ValueThumbSlider[1, {1, 20}]
n2 ValueThumbSlider[1, {1, 20}]
```

It is also possible to combine custom controls with other dynamic elements in the controls area (discussed in the previous section).

```
In[7]:= Manipulate[
          ParametricPlot[{a1 Sin[n1 (x + p1)], a2 Cos[n2 (x + p2)]},
            {x, 0, 20 Pi}, PlotRange → 1, PerformanceGoal → "Quality"],
          {{n1, 1}, 1, 4, ValueThumbSlider[##] &},
          {{a1, 1}, 0, 1, ValueThumbSlider[##] &},
          {{p1, 0}, 0, 2 Pi, ValueThumbSlider[##] &},
          Dynamic[ParametricPlot[{a1 Sin[n1 (x + p1)], x}, {x, 0, 2 Pi},
             ImageSize → 100, AspectRatio → 1, PlotRange → {{-1, 1}, {0, 2 Pi}}]],
          Delimiter,
          {{n2, 5 / 4.}, 1, 4, ValueThumbSlider[##] &},
          {{a2, 1}, 0, 1, ValueThumbSlider[##] &},
          {{p2, 0}, 0, 2 Pi, ValueThumbSlider[##] &},
          Dynamic[Plot[a2 Sin[n2 (x + p2)], {x, 0, 2 Pi}, ImageSize → 100,
             AspectRatio → 1, PlotRange → 1]], ControlPlacement → Left]
```

Out[7]=



```
n1  ValueThumbSlider[1, {1, 4}]
a1  ValueThumbSlider[1, {0, 1}]
p1  ValueThumbSlider[0, {0, 2 π}]

n2  ValueThumbSlider[1.25, {1, 4}]
a2  ValueThumbSlider[1, {0, 1}]
p2  ValueThumbSlider[0, {0, 2 π}]
```

This example should give some idea of how far `Manipulate` can be pushed to create complex interfaces. However, it is important to remember that `Manipulate` is not the only way to create interfaces in *Mathematica*. "Introduction to Dynamic" provides further information and examples showing how to create free-form interfaces not restricted to the model provided by `Manipulate`.

One of the nice things about building an interface like this inside `Manipulate` is that it lets you use **Autorun** (click the plus icon in the top right corner of the panel and choose **Autorun**) to put the example through its paces, varying each variable according to a sensible interpolating pattern.

On the other hand, `Manipulate` restricts you to a certain set of layouts and behaviors which, while very flexible and expandable, are still fixed compared to what is possible using the lower level features described in "Introduction to Dynamic".

Generalized Input

The fundamental paradigm of most computer languages, including *Mathematica*, is that input is given and processed into output. Historically, such input has consisted of strings of letters and numbers obeying a certain syntax.

Evaluate this input line to generate a table of output.

```
In[1]:= Table[n!, {n, 1, 10}]
Out[1]= {1, 2, 6, 24, 120, 720, 5040, 40 320, 362 880, 3 628 800}
```

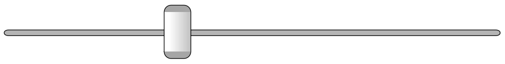
Starting in Version 3, *Mathematica* has supported the use of two-dimensional typeset mathematical notations as input, freely mixed with textual input.

This also generates a table of output.

```
                 n + 3
In[2]:= Table[ ——————, {n, 1, 10}]
                  n!
               5      7    1    1    1      11        1         13
Out[2]= {4, —, 1, ——, ——, ——, ———, —————, —————, ———————}
               2     24   15   80   504   40 320   30 240   3 628 800
```

Starting with Version 6, a wide range of nontextual objects can be used as input just as easily, and can be mixed with text or typeset notations.

Evaluate the following input, then move the slider and evaluate it again to see a new result.



```
                 n + [slider]
In[3]:= Table[ ————————————————, {n, 1, 10}]
                      n!
                7   1    3    1    11     1       13        1
Out[3]= {5, 3, —, —, ——, ——, ————, ————, ———————, ———————}
                6   3   40   72  5040  3360  362 880  259 200
```

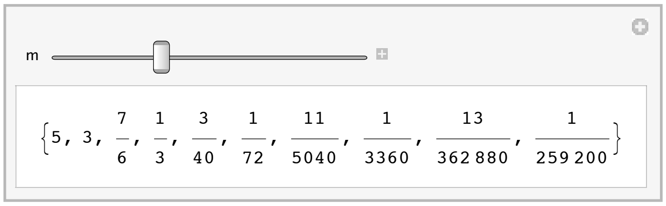
The "value" of this slider when it is given as input is determined by its position, in this case an integer between 1 and 10. It can be used anywhere in an input line that a textual number or variable name could be used.

How to create such controls is discussed in the next section, but it is worth noting first that in many cases there are better alternatives to this kind of input.

Casting this example in the form of a `Manipulate` allows you to see the effect of moving the slider in real time.

In[4]:= **Manipulate**$\left[\textbf{Table}\left[\frac{\textbf{n + m}}{\textbf{n !}}, \{\textbf{n, 1, 10}\}\right], \{\textbf{m, 1, 10, 1}\}\right]$

Out[4]=



$\left\{5,\ 3,\ \frac{7}{6},\ \frac{1}{3},\ \frac{3}{40},\ \frac{1}{72},\ \frac{11}{5040},\ \frac{1}{3360},\ \frac{13}{362\,880},\ \frac{1}{259\,200}\right\}$

But there are situations where using a control inside a traditional Shift+Return evaluated input works better. Some cases are: if the evaluation is very slow, if you want complete flexibility in editing the rest of the input line around the control(s), or if the point of the code is to make definitions that will be used later, and the controls are being used as a convenient way to specify initial values.

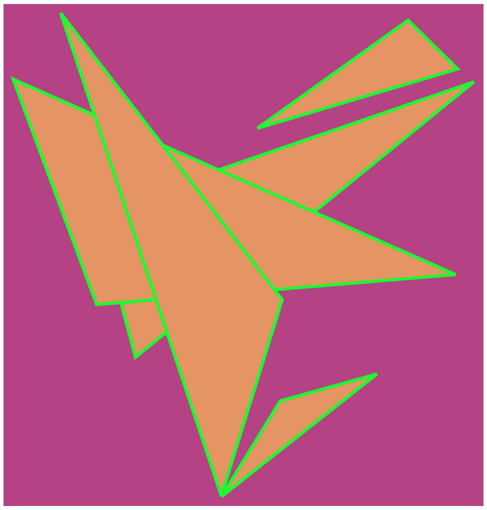
For example, you might want to set up a color palette using `ColorSetter` to initialize named colors that will be used in subsequent code.

```
In[5]:= edgeColor = 🟩;

In[6]:= fillColor = 🟧;

In[7]:= backgroundColor = Blend[{🟧, 🟪}];
```

Click any color swatch to get a dialog allowing you to interactively choose a new color. These values can then be used in subsequent programming just as if they had been initialized with more traditional textually specified values.

```
In[8]:= Graphics[{
          fillColor, EdgeForm[{AbsoluteThickness[2], edgeColor}],
          Polygon[RandomReal[{0, 1}, {5, 3, 2}]]}, Background → backgroundColor]
```

Out[8]=



These color swatches provide an informative, more easily edited representation of the colors than would an expression consisting of numerical color values.

How to Create Inline Controls

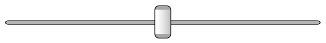
The most flexible and powerful way to create anything in *Mathematica* is to evaluate a function that returns it.

These examples use `Slider`, but the same principles apply to any controls. Control Objects lists all the available control objects.

You can create a slider by evaluating `Slider[]`.

```
In[9]:= Slider[]
```

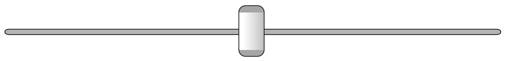
Out[9]=

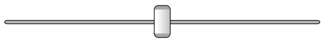


The resulting slider object can be copied and pasted into a subsequent input cell just as if it were any other kind of input. (Or you can just click in the output cell and start typing, which will cause it to be converted automatically into an input cell.)

Controls created this way are inert to evaluation.

For example, type `2+`, then paste the previous slider after the + to create this input line, and then evaluate it.

In[10]:= **2 +**


Out[10]= 2 +


When evaluated, the slider remains a slider, which is not wanted in this case (though it is very useful in other situations). What is needed instead is a slider that, when evaluated as input, becomes the value it is set to, rather than a slider object.

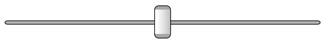
`DynamicSetting[`e`]`	an object that displays as e, but is interpreted as the dynamically updated current setting of e upon evaluation

Object that evaluates into its current setting.

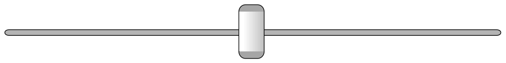
When `DynamicSetting` is wrapped around a slider and evaluated, the new slider looks identical to the original one, but has the hidden property of evaluating into its current setting.

This displays the new slider.

In[11]:= **`DynamicSetting[Slider[]]`**

Out[11]=


If the new slider is copied and pasted into an input line, the act of evaluation transforms the slider into its current value (by default 0.5 if it has not been moved with the mouse).

In[12]:= **2 +**


Out[12]= `2.5`

The examples in the previous section were created using `DynamicSetting` in this way.

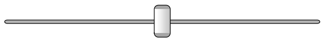
While copying and pasting can be used very effectively to build up input lines containing controls, it is often most convenient to use **Evaluate in Place** Ctrl+Shift+Enter (Command+Return on Mac) to transform control expressions in place, especially once you are familiar with the commands that create controls.

Ctrl+Shift+Enter	evaluate a selection "in place", replacing the selection with the output

Evaluating in place.

For example, enter the following input line.

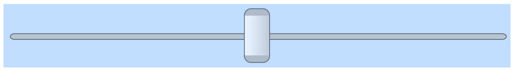
```
In[13]:= 2 + DynamicSetting[Slider[]]
```

```
Out[13]= 2 +
```


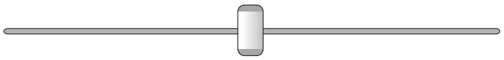
Then, highlight the entire control expression. (Triple-clicking the word `DynamicSetting` is an especially quick way to do this.)

```
In[14]:= 2 + DynamicSetting[Slider[]]
```

Type Ctrl+Shift+Enter (Command+Return on Mac), and the control expression will be transformed in place into an actual control. (Note that Ctrl+Shift+Enter is not the normal Shift+Enter used for evaluating input.)

```
In[14]:= 2 +
```


Evaluating the input cell with Shift+Return will then give the desired result.

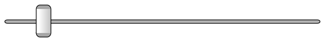
```
In[15]:= 2 +
```


```
Out[15]= 2.5
```

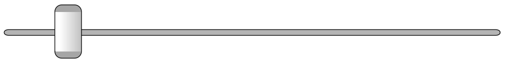
All the arguments of `Slider` can be used to change the initial value, range, and step size.

Start with this input expression.

```
In[16]:= Expand[(1 + x)^DynamicSetting[Slider[5,{1,50,1}]]]
```

```
Out[16]= (1 + x)
```


Then evaluate in place (Ctrl+Shift+Enter) to transform the text command into a slider.

```
In[17]:= Expand[(1 + x)          ]
```


Out[17]= $1 + 5\,x + 10\,x^2 + 10\,x^3 + 5\,x^4 + x^5$

Of course, this works with other kinds of controls as well.

```
In[18]:=  Expand[(1 + x)^DynamicSetting[PopupMenu[5,Table[i,{i,50}]]]]
```

```
Out[18]=  (1 + x)^[5 ▾]
```

This is the result after evaluating in place.

```
In[19]:=  Expand[(1 + x)^[5 ▾]]
```

Out[19]= $1 + 5\,x + 10\,x^2 + 10\,x^3 + 5\,x^4 + x^5$

Note that the control expressions do not contain a dynamic reference to a variable as they normally would (see "Introduction to Dynamic"). Controls used in input expressions as described here are static, inert objects, much like a textual command. They are not linked to each other, and nothing happens when you move one, except that it moves. Basically they are simply recording their current state, for use when you evaluate the input line.

Complex Templates in Input Expressions

It is possible to use whole panels containing multiple controls in input expressions. Constructing such panels is more complex than simply wrapping `DynamicSetting` around a single control, because you have to specify how the various control values should be assembled into the value returned when the template is evaluated.

The function `Interpretation` is used to assemble a self-contained input template object, which may contain many internal parts that interact with each other through dynamic variables. The arguments are `Interpretation[`*variables*`, `*body*`, `*returnvalue*`]`.

The first argument gives a list of local variables with optional initializers in the same format as `Module` or `DynamicModule`. (In fact, `Interpretation` creates a `DynamicModule` in the output. See "Introduction to Dynamic".)

The second argument is typeset to become the displayed form of the interpretation object. Typically it contains formatting constructs and controls with dynamic references to the variables defined in the first argument.

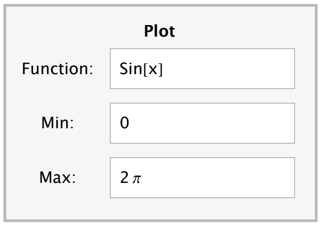
The third argument is the expression that will be used as the value of the interpretation object when it is given as input. Typically this is an expression involving the variables listed in the first argument.

`Interpretation[`*e*`,`*expr*`]`	an object which displays as *e*, but is interpreted as the unevaluated form of *expr* if supplied as input
`Interpretation[{`$x=x_0$`,`$y=y_0$`,…},`*e*`,`*expr*`]`	allows local variables in *e* and *expr*

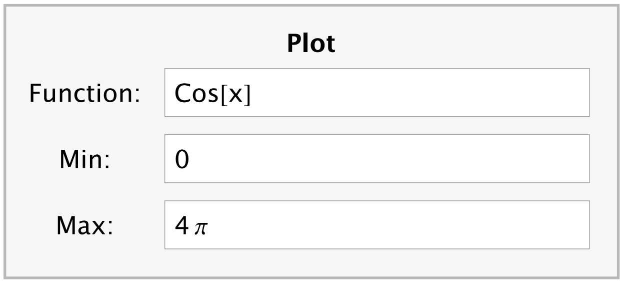
Object that displays as one expression and evaluates as another.

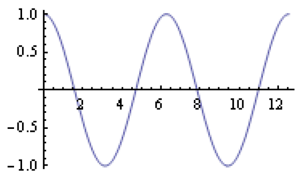
Evaluating the following input cell creates an output cell containing a template for the `Plot` command.

In[20]:=
```
Interpretation[{f = Sin[x], min = 0, max = 2 Pi},
 Panel[Grid[{
    {Style["Plot", Bold], SpanFromLeft},
    {"Function:", InputField[Dynamic[f]]},
    {"Min:", InputField[Dynamic[min]]}, {"Max:", InputField[Dynamic[max]]}}]],
 Plot[f, {x, min, max}]]
```

Out[20]=

Plot
Function: Sin[x]
Min: 0
Max: 2 π

This template can be copied and pasted into an input cell, and the values edited as you like. Shift+Return evaluation of the input cell generates a plot.

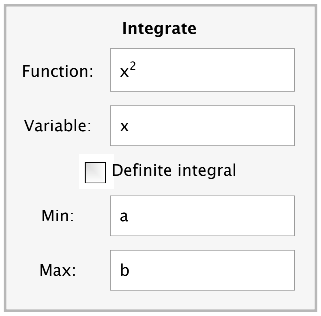
In[21]:=

Plot
Function: Cos[x]
Min: 0
Max: 4 π

Out[21]=


In the following more sophisticated example, the variable *definite* is used to communicate between the controls in the template, dimming the *min* and *max* value fields when indefinite integration is selected.

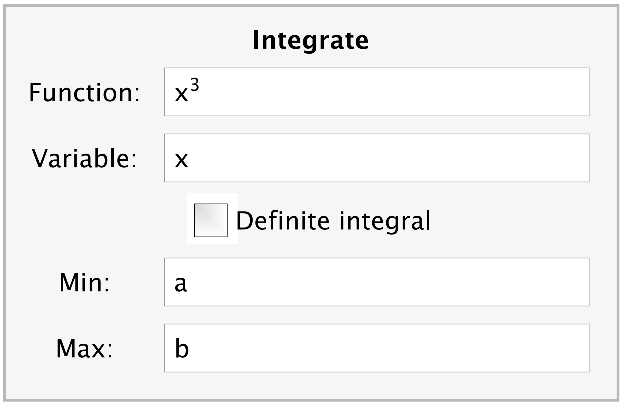
```
In[1]:= Interpretation[{f = x^2, var = x, definite = False, min = a, max = b}, Panel[Grid[{
            {Style["Integrate", Bold], SpanFromLeft},
            {"Function:",
             InputField[Dynamic[f], FieldSize → {{20, Infinity}, {1, Infinity}}]},
            {"Variable:", InputField[Dynamic[var]]},
            {Row[{Checkbox[Dynamic[definite]], "Definite integral"}], SpanFromLeft},
            {"Min:", InputField[Dynamic[min], Enabled → Dynamic[definite]]},
            {"Max:", InputField[Dynamic[max], Enabled → Dynamic[definite]]}}]],
         If[definite, Integrate[f, {var, min, max}], Integrate[f, var]]]
```

Out[1]=



This copy of the previous template gives the integral upon evaluation.

In[23]:=

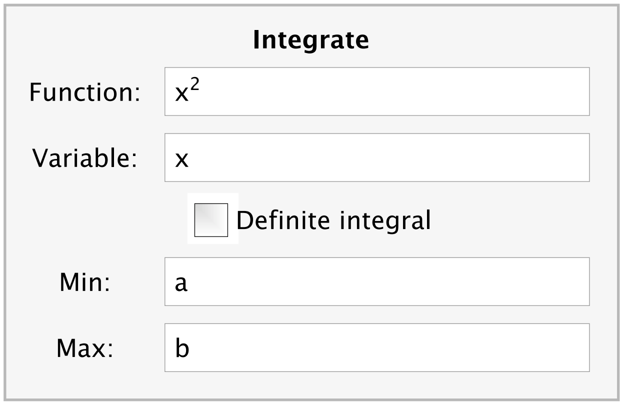


Out[23]= $\frac{x^4}{4}$

As with the single controls in earlier sections, these input templates can be copied and pasted into new input cells, and they can be freely intermixed with textual input.

To test the result of integration, wrap the template with `D` to take the derivative and verify that the result is the same as the starting point.

In[24]:= D[



Function: x^2
Variable: x
Min: a
Max: b

, x]

Out[24]= x^2

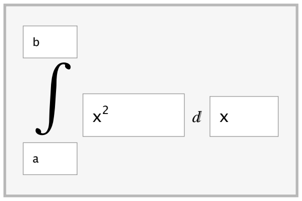
These examples are fairly generic: they look like dialog boxes in a lot of programs. But there are some important differences. For example, note the x^2 in the input field. Input fields in *Mathematica* may look like those in other programs, but the contents can be any arbitrary typeset mathematics, or even graphics or other controls. (See the next section to learn how to write templates that can be nested inside each other.)

Mathematica templates (and dialog boxes) are also not restricted to using a regular grid of text fields.

Here is a simple definite integration template laid out typographically.

```
In[25]:= Interpretation[{f = x^2, var = x, min = a, max = b}, Panel[Row[{
            Underoverscript[Style["∫", 36],
             InputField[min, FieldSize → Tiny], InputField[max, FieldSize → Tiny]],
            InputField[f, FieldSize → {{10, Infinity}, {1, Infinity}}],
            " ⅆ ", InputField[var, FieldSize → Tiny]}]],
          Integrate[f, {var, min, max}]]
```

Out[25]=



$\int_a^b x^2 \, d\, x$

Note that you do not need a template to evaluate integrals; they can be entered as plain typeset math formulas using keyboard shortcuts (as described in "Entering Two-Dimensional Expressions") or the **Basic Input** palette.

In[26]:= $\int_a^b x^2 \, dx$

Out[26]= $-\frac{a^3}{3} + \frac{b^3}{3}$

Whether it is useful to make a template like this or not depends on many things, but the important point is that in *Mathematica* the full range of formatting constructs, including text, typeset math, and graphics, is available both inside and around input fields and templates.

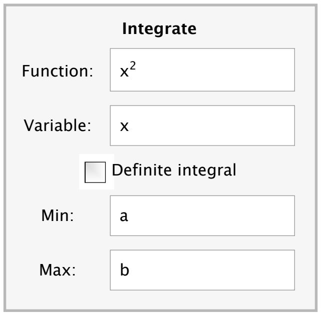
Advanced Topic: Dealing with Premature Evaluation in Templates

Templates defined like those in the previous section do not work as you might hope if the variables given in initializers already have other values assigned to them (for example, if the variable x has a value in the previous section), or if template structures are pasted into the input fields. To deal with evaluation issues correctly, it is necessary to use `InputField` objects that store their values in the form of unparsed box structures rather than expressions. (Box structures are like strings in the sense that they represent any possible displayable structure, whether it is a legal *Mathematica* input expression or not.)

This defines a template.

```
In[27]:= Interpretation[{
           f = MakeBoxes[x^2],
           var = MakeBoxes[x],
           definite = False,
           min = MakeBoxes[a],
           max = MakeBoxes[b]},
          Panel[Grid[{
             {Style["Integrate", Bold], SpanFromLeft},
             {"Function:",
              InputField[Dynamic[f], Boxes, FieldSize → {{20, Infinity}, {1, Infinity}}]},
             {"Variable:", InputField[Dynamic[var], Boxes]},
             {Row[{Checkbox[Dynamic[definite]], "Definite integral"}], SpanFromLeft},
             {"Min:", InputField[Dynamic[min], Boxes, Enabled → Dynamic[definite]]},
             {"Max:", InputField[Dynamic[max], Boxes, Enabled → Dynamic[definite]]}}]],
          With[{f = ToExpression[f], var = ToExpression[var],
            min = ToExpression[min], max = ToExpression[max]},
           If[definite, Integrate[f, {var, min, max}], Integrate[f, var]]]]
```

Out[27]=


Integrate

Function: x^2

Variable: x

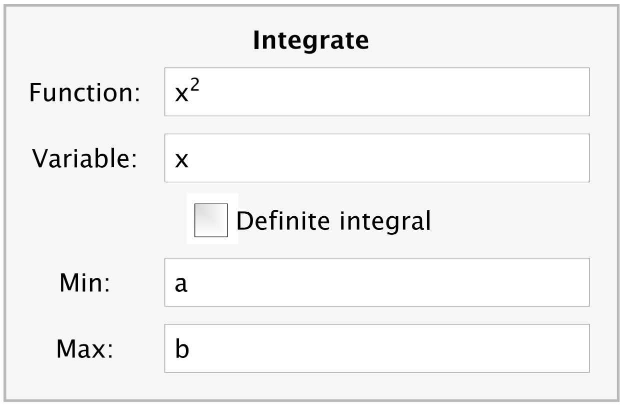
☐ Definite integral

Min: a

Max: b

This copy of the previous template gives the integral upon evaluation.

In[28]:=


Integrate

Function: x^2

Variable: x

☐ Definite integral

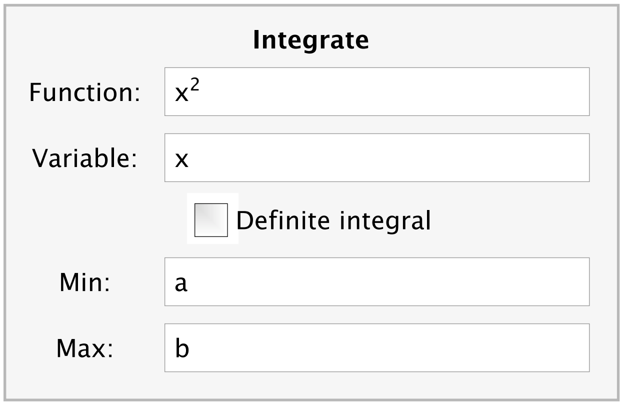
Min: a

Max: b

Out[28]= $\frac{x^3}{3}$

This template will work properly even under what might be considered abuse. For example, you can nest it repeatedly to integrate a function several times.

In[29]:=


Integrate

Function: x^2

Variable: x

☐ Definite integral

Min: a

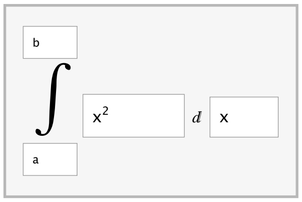
Max: b

Out[29]= $\frac{x^3}{3}$

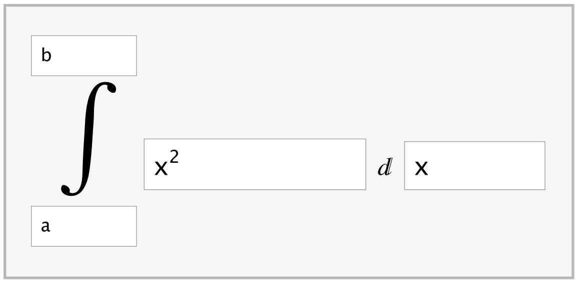
Note how the `InputField` grows automatically to accommodate larger contents. (This behavior is controlled by the `FieldSize` option.)

The typographic template can also be made robust to evaluation.

```
In[30]:= Interpretation[{
           f = MakeBoxes[x^2],
           var = MakeBoxes[x],
           definite = False,
           min = MakeBoxes[a],
           max = MakeBoxes[b]}, Panel[Row[{
             Underoverscript[Style["∫", 36],
              InputField[min, Boxes, FieldSize → Tiny],
              InputField[max, Boxes, FieldSize → Tiny]],
             InputField[f, Boxes, FieldSize → {{10, Infinity}, {1, Infinity}}],
             " ⅆ ", InputField[var, Boxes, FieldSize → Tiny]}]],
          With[{f = ToExpression[f], var = ToExpression[var], min = ToExpression[min],
            max = ToExpression[max]}, Integrate[f, {var, min, max}]]]
```

Out[30]=


And it can be nested, though this kind of thing can easily get out of hand, so it is probably more fun than useful.

In[31]:=


Out[31]= $-\frac{a^3}{3}+\frac{b^3}{3}$

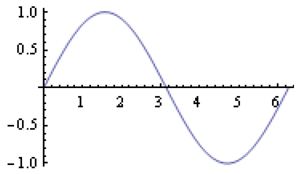
Graphics as Input

Graphic objects, including the output of `Graphics`, `Graphics3D`, plotting commands, and graphics imported from external image files, can all be used as input and freely mixed with textual input. There are no arbitrary limitations in the mixing of graphics, controls, typeset mathematics, and text.

Evaluate a simple plot command.

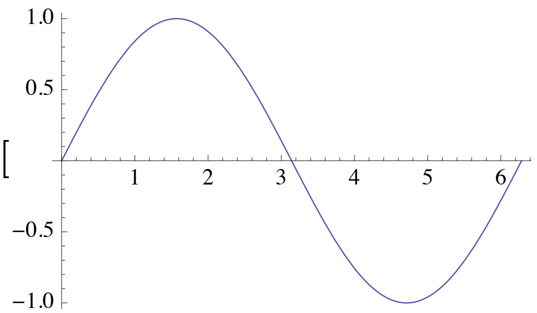
```
In[32]:= Plot[Sin[x], {x, 0, 2 Pi}]
```

Out[32]=

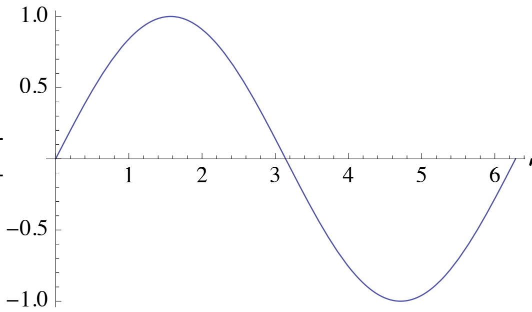


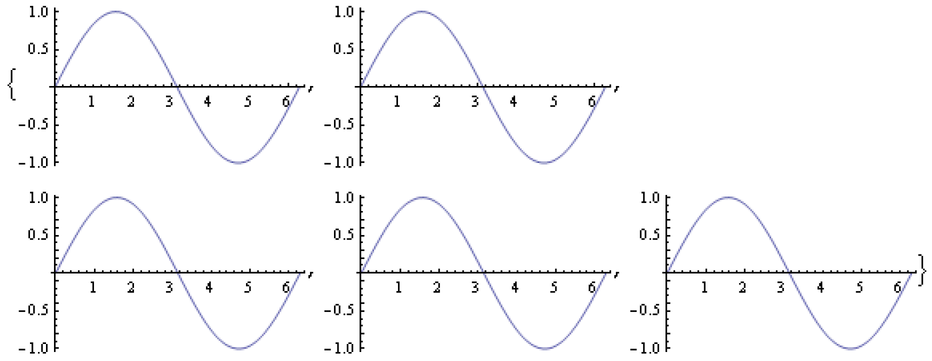
Then click to place the insertion point just to the left of the plot and type "`Table[`".

```
In[33]:= Table[
```



Complete the command by clicking and typing to the right of the plot, then evaluating.

In[2]:= **Table[**  **, {5}]**

Out[2]= 

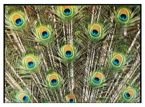
Notice how the plot appears in several different sizes depending on its context. There are three standard automatic plot sizes, the largest if the plot is by itself in an output, smaller if used in a list or table output, and smallest if in a textual input line. This is mainly a convenience to make graphical input less bulky. You are always free to resize the graphic by clicking it and using the resize handles, or by including an explicit `ImageSize` option.

You can import a bitmap image from a file.

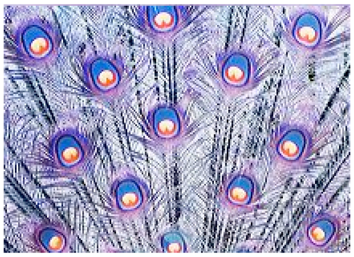
```
In[29]:= Import["ExampleData/peacock.tif"]
```

Out[29]=

Then copy/paste this into an input cell to do simple image processing on it.

In[30]:=


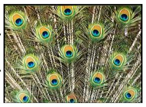
```
/. HoldPattern[Image[coords_, rest___]] :→ Image[255 - coords, rest]
```

Out[30]=


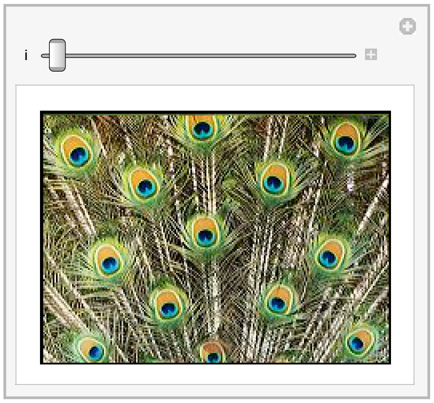
The ability to use graphics as input allows for remarkably rich input, as in this simple `Manipulate` example.

In[32]:=

```
Manipulate[
```



```
/. HoldPattern[Image[coords_, rest___]] :→
   Image[Map[Mod[#, 256] &, coords + i, {3}], rest], {i, 0, 255, 1}]
```

Out[32]=


Views

Mathematica supports a variety of objects that can be used to organize and display information in output. Known collectively as views, these objects range from the simple `OpenerView` to the complex and versatile `TabView`.

All have in common that they take a first argument containing a list of expressions to be displayed as separate panes in the view, and an optional second argument to determine which one should be displayed at the moment. All provide a user interface allowing you to change which pane is displayed: they are intended as interactive data-viewers.

The individual views are described first, then options and techniques common to all or most of them.

OpenerView

`OpenerView` allows you to open and close a pane containing an arbitrary expression. `OpenerView` is always given a list of two elements: the first element becomes the title (always visible) and the second becomes the contents that are revealed by clicking the disclosure triangle. In this example, click the triangle to reveal the word "Contents".

```
In[1]:= OpenerView[{"Title", "Contents"}]
Out[1]= ▶ Title
```

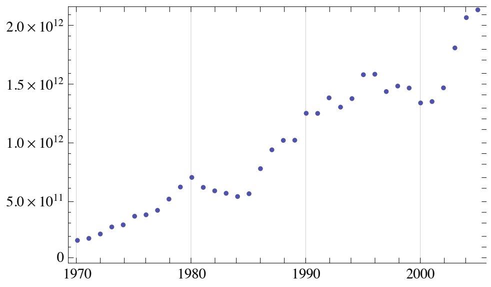
This control can be used to create objects that mimic the way disclosure triangles are used in other applications, for example, in the Finder (Macintosh) or Explorer (Windows). Typically the second element is bigger than the first, as in this example.

```
In[2]:= OpenerView[{"Plot", Plot[Sin[x], {x, 0, 2 Pi}]}]
Out[2]= ▶ Plot
```

A column or grid of more than one `OpenerView` objects lets you browse a large amount of data in a compact format.

```
In[3]:= Column[
         Map[OpenerView[{#, DateListPlot[CountryData[#, {{"GDP"}, {1970, 2005}}]]}] &,
          CountryData["GroupOf8"]]]
```

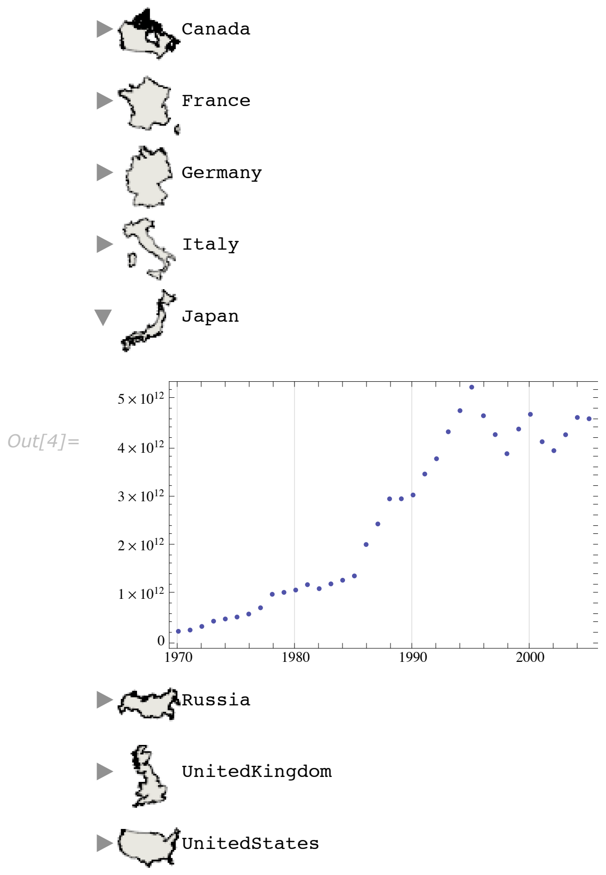
```
        ▶ Canada
        ▼ France
Out[3]=
```



```
        ▶ Germany
        ▶ Italy
        ▶ Japan
        ▶ Russia
        ▶ UnitedKingdom
        ▶ UnitedStates
```

The title is not limited to being a plain string: any arbitrary typeset expression or graphic can be used. Here, for example, is an outline of the country with its name as the title line.

```
In[4]:= Column[Map[
          OpenerView[{Row[{Rasterize[Show[CountryData[#, "Shape"], ImageSize → {40, 40}],
                 Background → None], #}], DateListPlot[
               CountryData[#, {{"GDP"}, {1970, 2005}}]]}] &, CountryData["GroupOf8"]]]
```



```
▶ Canada
▶ France
▶ Germany
▶ Italy
▼ Japan
▶ Russia
▶ UnitedKingdom
▶ UnitedStates
```

One advantage of a column like this over a `TabView`, for example, is that you can have two or more panes open at once, while other views typically let you see only one pane at a time.

Like other Views, `OpenerView` can be nested arbitrarily deeply. This example turns any expression into a nested tree of openers in which the closed state is the head of the expression and the open state is a column of openers for each argument.

```
In[5]:= OpenerTree[expr_] := OpenerView[
            {OpenerTree[Head[expr]], Column[Map[OpenerTree, Apply[List, expr]]]}];
        OpenerTree[expr_] := expr /; (Length[expr] === 0)
```

Here is a simple application shown with all the openers in the open state.

```
In[7]:=  OpenerTree[(a + b) (c + d)]

Out[7]=  ▼ Times
           ▼ Plus
             a
             b
           ▼ Plus
             c
             d
```

Here is a more deeply nested application with just a few opened.

```
In[8]:=  OpenerTree[Integrate[1 / (1 - x^9), x]]

Out[8]=  ▼ Times
           1/18
           ▼ Plus
             ▶ Times
             ▶ Times
             ▶ Times
             ▼ Log
               ▼ Plus
                 1
                 x
                 ▼ Power
                   x
                   2
             ▶ Times
             ▶ Times
             ▶ Times
             ▶ Times
             ▶ Times
```

For more information and a detailed listing of options, see `OpenerView`.

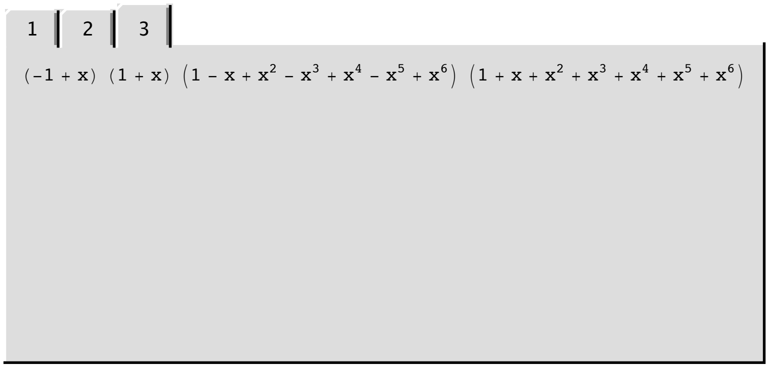
TabView

`TabView` is a rich object capable of creating surprisingly interesting user interfaces. Given a list of expressions, it returns a panel with a row of tabs that allow you to look at the expressions one at a time.

By default, the tabs are numbered sequentially. In the output below, click the tabs to flip between panes.

```
In[9]:=  TabView[{40 !, Plot[Sin[x], {x, 0, 2 Pi}], Factor[x^14 - 1]}]
```

Out[9]=

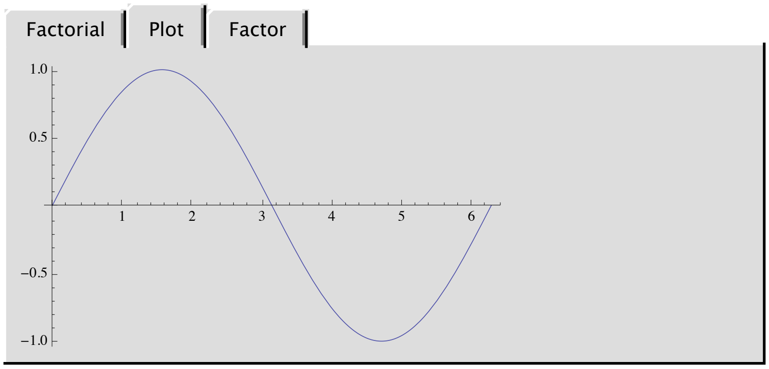


$(-1+x)\,(1+x)\,\left(1-x+x^2-x^3+x^4-x^5+x^6\right)\,\left(1+x+x^2+x^3+x^4+x^5+x^6\right)$

More descriptive tab labels can be added using the form *label* –> *contents*.

```
In[10]:= TabView[{"Factorial" → 40 !,
           "Plot" → Plot[Sin[x], {x, 0, 2 Pi}], "Factor" → Factor[x^14 - 1]}]
```

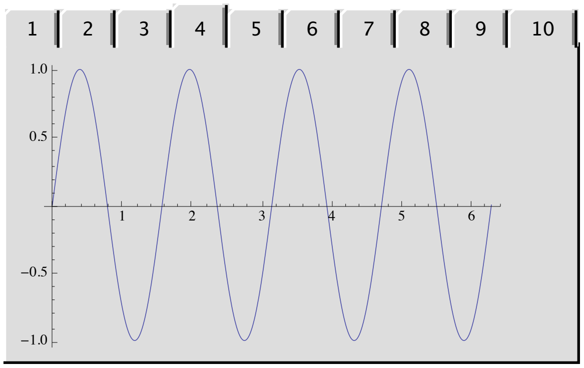
Out[10]=



The contents can of course be programmatically generated. Here a `Table` command is used to generate ten different plots.

```
In[11]:= TabView[Table[Plot[Sin[n x], {x, 0, 2 Pi}], {n, 10}]]
```

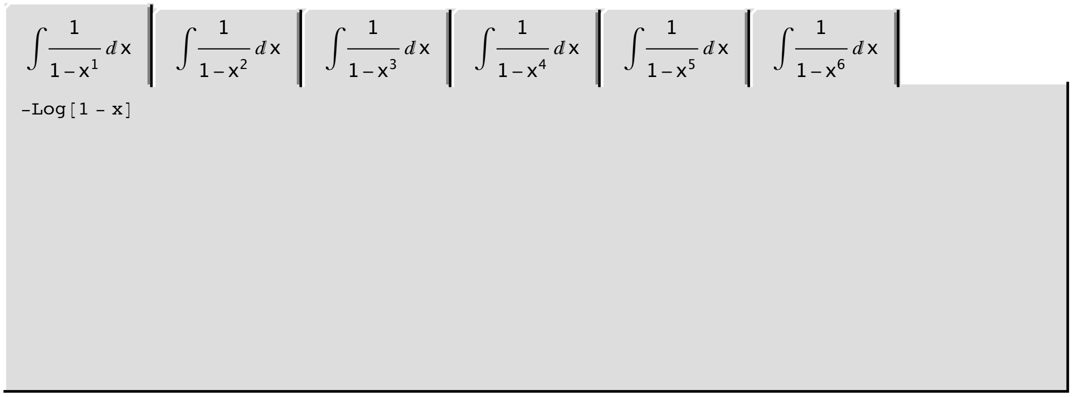
Out[11]=



The tab labels are not restricted to simple strings. Here typeset mathematical expressions are used as tab labels.

```
In[10]:= TabView[
          (HoldForm[Integrate[1 / (1 - x^#), x]] → Integrate[1 / (1 - x^#), x]) & /@ Range[6]]
```

Out[10]=

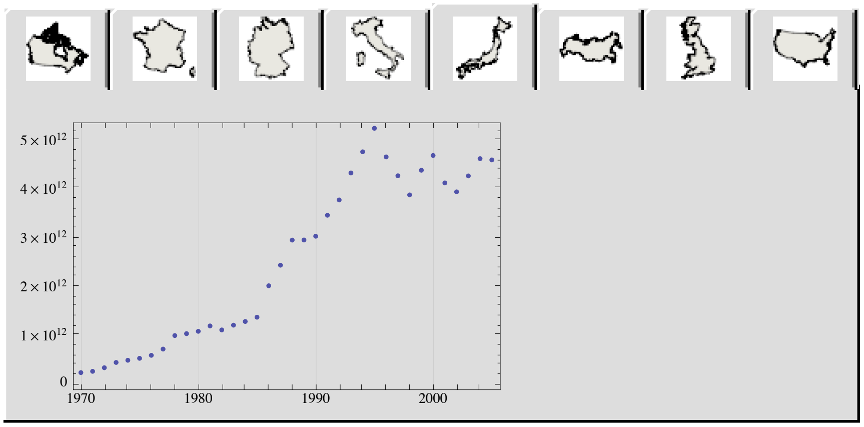


$\int \frac{1}{1-x^1} dx$
$\int \frac{1}{1-x^2} dx$
$\int \frac{1}{1-x^3} dx$
$\int \frac{1}{1-x^4} dx$
$\int \frac{1}{1-x^5} dx$
$\int \frac{1}{1-x^6} dx$
-Log[1 - x]

This example uses the shapes of countries as tab labels, with a plot of each country's GDP in its pane.

```
In[13]:= TabView[Rasterize[Show[CountryData[#, "Shape"], ImageSize → {40, 40}],
              Background → None] → DateListPlot[
              CountryData[#, {{"GDP"}, {1970, 2005}}]] & /@ CountryData["GroupOf8"]]
```

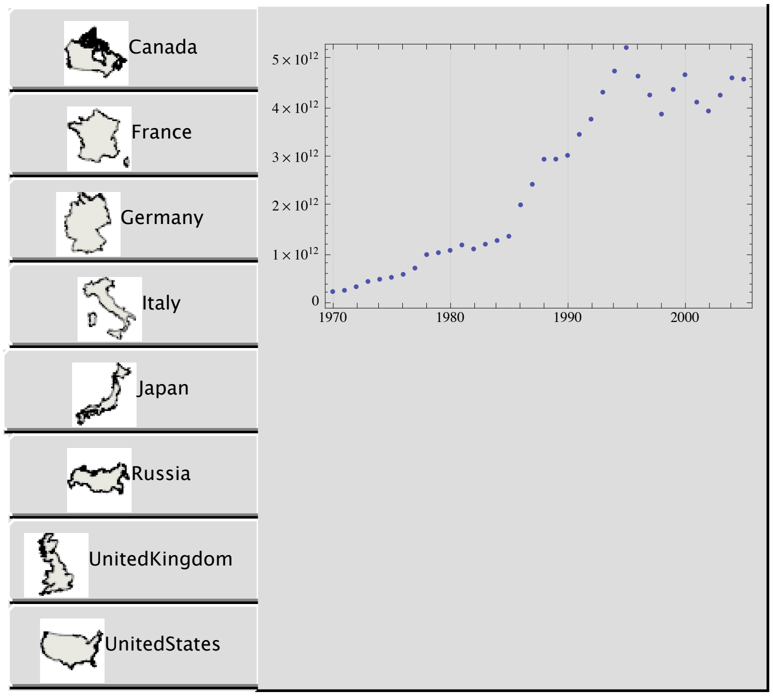
Out[13]=



When arranged across the top, tab labels need to be kept reasonably short. The `ControlPlacement` option can be used to move the tabs to any side of the panel.

```
In[14]:= TabView[
          Row[{Rasterize[Show[CountryData[#, "Shape"], ImageSize → {40, 40}], Background →
                 None], #}] → DateListPlot[CountryData[#, {{"GDP"}, {1970, 2005}}]] & /@
           CountryData["GroupOf8"], ControlPlacement → Left]
```

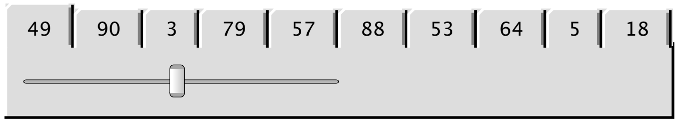
Out[14]=



The fact that tab labels can be absolutely anything, including typeset expressions, graphics, and dynamic output, makes `TabView` considerably more flexible than you might at first think. Here, for example, is a `TabView` where each pane includes a slider that allows you to adjust the label of the tab for that pane. (`Dynamic` and `DynamicModule` are explained in "Introduction to Dynamic".)

```
In[15]:= DynamicModule[{values = RandomInteger[{0, 100}, {10}]}, TabView[Table[With[{i = i},
             Dynamic[values[[i]]] → Slider[Dynamic[values[[i]]], {1, 100, 1}]], {i, 10}]]]
```

Out[15]=

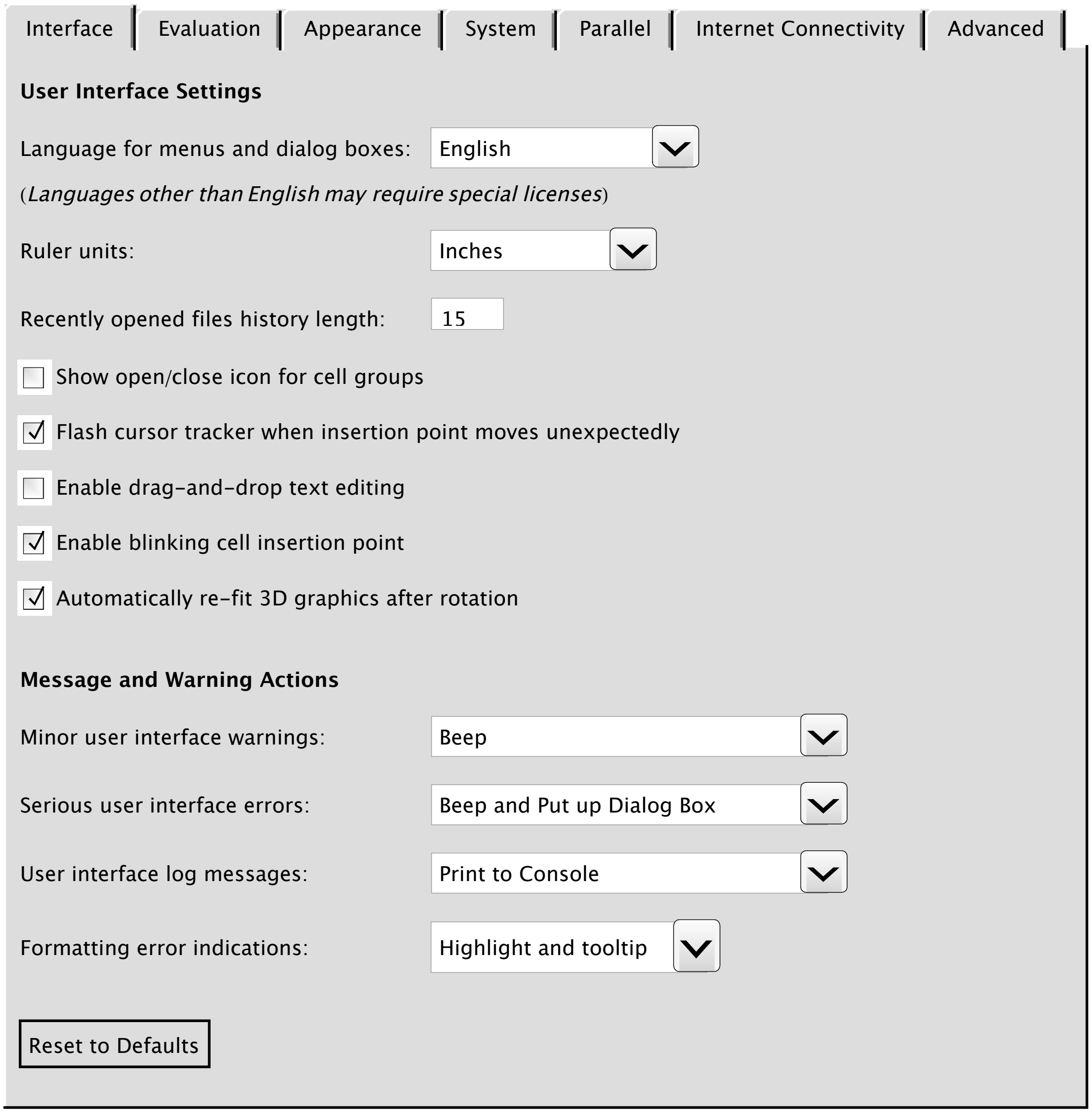


The "Controlling the Currently Displayed Pane" section contains further examples of dynamic tab labels.

`TabView` objects can be nested to arbitrary depth, allowing very large amounts of content to be presented in compact form. Here, for example, is a copy of the *Mathematica* **Preferences** dialog box, which is implemented as a set of nested `TabView` objects. The fact that a complex dialog box like this can be copied and pasted into a document without loss of functionality is an example of the power of *Mathematica*'s symbolic dynamic interface technology.

Note that this is a fully functional copy, so if you change anything here it will in fact immediately change your preference settings.

Out[29]=



User Interface Settings

Language for menus and dialog boxes: English

(*Languages other than English may require special licenses*)

Ruler units: Inches

Recently opened files history length: 15

☐ Show open/close icon for cell groups

☑ Flash cursor tracker when insertion point moves unexpectedly

☐ Enable drag-and-drop text editing

☑ Enable blinking cell insertion point

☑ Automatically re-fit 3D graphics after rotation

Message and Warning Actions

Minor user interface warnings: Beep

Serious user interface errors: Beep and Put up Dialog Box

User interface log messages: Print to Console

Formatting error indications: Highlight and tooltip

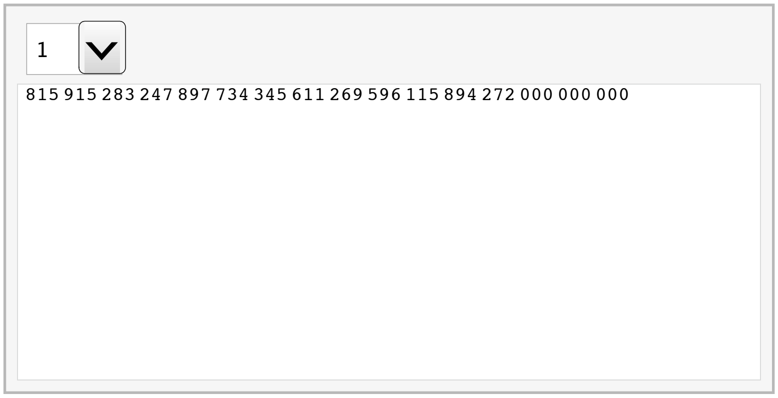
For more information and a detailed listing of options, see `TabView`.

MenuView

`MenuView` is much like `TabView`, except that it uses a popup menu rather than a row of tabs to select which pane is displayed. These two examples are identical to the first two examples given in the "TabView" section; we have simply substituted the word `MenuView` for `TabView`.

```
In[16]:= MenuView[{40 !, Plot[Sin[x], {x, 0, 2 Pi}], Factor[x^14 - 1]}]
```

Out[16]=

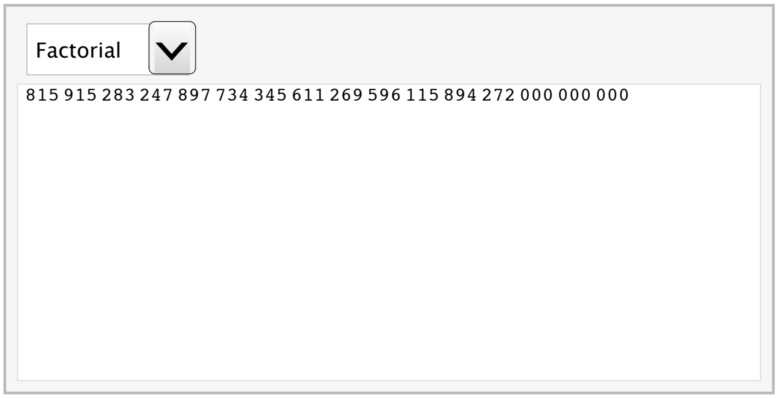


```
815 915 283 247 897 734 345 611 269 596 115 894 272 000 000 000
```

`MenuView` supports the same *label -> value* syntax as `TabView`, allowing you to specify more descriptive labels.

```
In[17]:= MenuView[{"Factorial" → 40 !,
           "Plot" → Plot[Sin[x], {x, 0, 2 Pi}], "Factor" → Factor[x^14 - 1]}]
```

Out[17]=

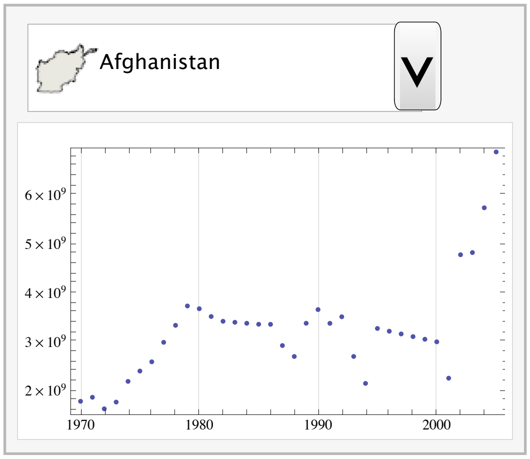


```
815 915 283 247 897 734 345 611 269 596 115 894 272 000 000 000
```

In the case of `TabView`, all the labels are displayed simultaneously, which means there is a fairly small practical limit to the number of panes you can have. `MenuView` displays only one label at a time, allowing you to use many more. For example, in the "TabView" section above there is a nice example with graphical tabs for the G8 countries. With `MenuView` the same thing can be done for as many as 237 countries.

```
In[18]:= MenuView[
          Row[{Rasterize[Show[CountryData[#, "Shape"], ImageSize → {40, 40}], Background →
                 None], #}] →
             Dynamic[DateListPlot[CountryData[#, {{"GDP"}, {1970, 2005}}]]] & /@
           CountryData[], ImageSize → Automatic]
```

Out[18]=



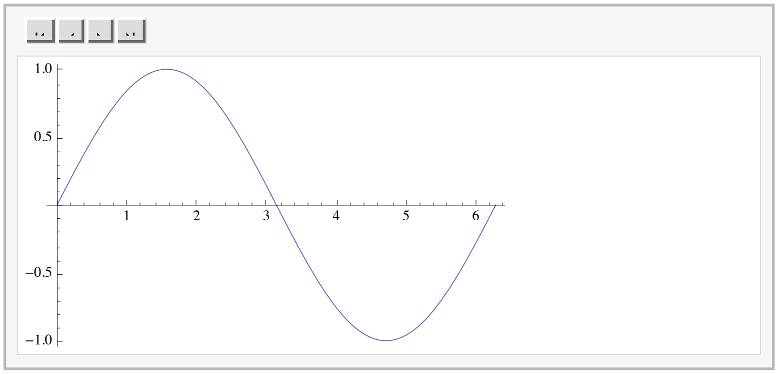
For more information and a detailed listing of options, see `MenuView`.

SlideView

`SlideView` is basically much like `TabView` or `MenuView`, except with a set of first/previous/next-/last buttons for navigating the panes.

```
In[19]:= SlideView[{40 !, Plot[Sin[x], {x, 0, 2 Pi}], Factor[x^14 - 1]}]
```

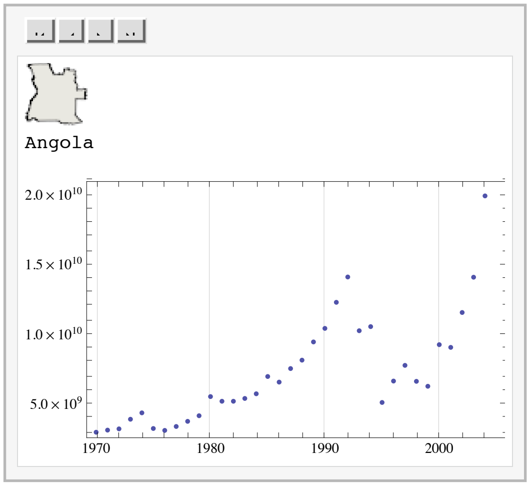
Out[19]=



The number of panes can be arbitrarily large, but navigation is limited to stepping through them like a slide show.

```
In[20]:= SlideView[
          Dynamic[Column[{Rasterize[Show[CountryData[#, "Shape"], ImageSize → {40, 40}],
                Background → None], #,
              DateListPlot[CountryData[#, {{"GDP"}, {1970, 2005}}]]}]] & /@
           CountryData[], ImageSize → Automatic]
```

Out[20]=



For more information and a detailed listing of options, see `SlideView`.

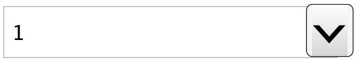
PopupView

`PopupView` might seem at first similar to `MenuView`, but they are actually quite different. `MenuView` and `TabView` both, in effect, have two items representing each pane: a label and the actual contents of the pane. `PopupView`, on the other hand, has only one item per pane: the main contents of the pane.

Given a list of expressions, `PopupView` displays them as a popup menu.

```
In[28]:= PopupView[Table[i!, {i, 20}]]
```

Out[28]=



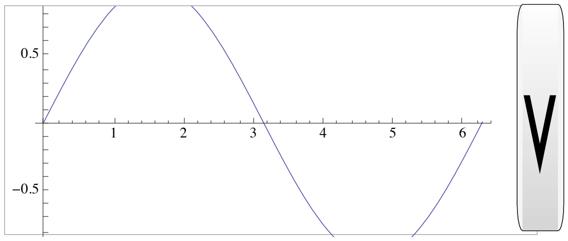
```
1
```

Readers familiar with the `PopupMenu` control may wonder how this is different, since both appear to do basically the same thing. The difference is largely one of intent: `PopupMenu` is intended as a control that affects something when an item is selected; it has a required first argument that holds a variable that tracks the currently selected item. `PopupView`, on the other hand, is intended simply as a way of displaying information, without necessarily having any effect when a different pane is selected.

As will other controls and views in *Mathematica*, `PopupView` fully supports arbitrary typeset or graphical content.

```
In[29]:= PopupView[{24!, Plot[Sin[x], {x, 0, 2 Pi}], Factor[x^6 - 1]}]
```

Out[29]=



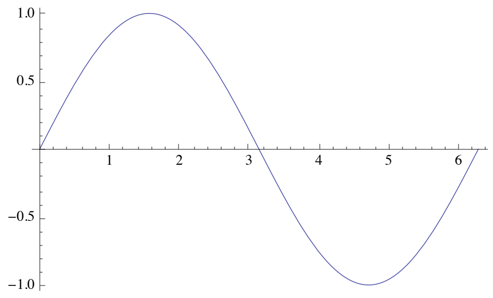
For more information and a detailed listing of options, see `PopupView`.

FlipView

`FlipView` is unusual in that it provides no visible user interface around the contents of its panes. It does, however, provide a mechanism for changing the pane being displayed. Clicking anywhere in the contents of the current pane flips the display to the next one.

```
In[30]:= FlipView[{40!, Plot[Sin[x], {x, 0, 2 Pi}], Factor[x^14 - 1]}]
```

Out[30]=



`FlipView` is also unusual in that instead of having a fixed overall size large enough to hold the largest pane, it is always exactly as large as the currently displayed pane. This is, in fact, simply a difference in the default value of the `ImageSize` option for `FlipView` versus the other views, as explained in the "Controlling whether a View Changes Size" section.

Controlling the Currently Displayed Pane

All View objects support an optional second argument that specifies which pane is currently visible. Given a literal value, this argument determines the initially displayed pane when the object is first created. Given a `Dynamic` variable, it can be used to externally influence, or to track, the currently displayed pane.

The set of values in the second argument that corresponds to the displayed panes depends on the view.

`OpenerView` normally starts in the closed state.

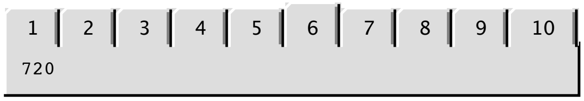
```
In[31]:= OpenerView[{"Title", "Contents"}]
Out[31]= ▶ Title
```

Its two panes are referred to with `True` (open) and `False` (closed), so this example will start in the open state.

```
In[32]:= OpenerView[{"Title", "Contents"}, True]

Out[32]= ▼ Title
           Contents
```

`TabView`'s panes are by default referred to by index number

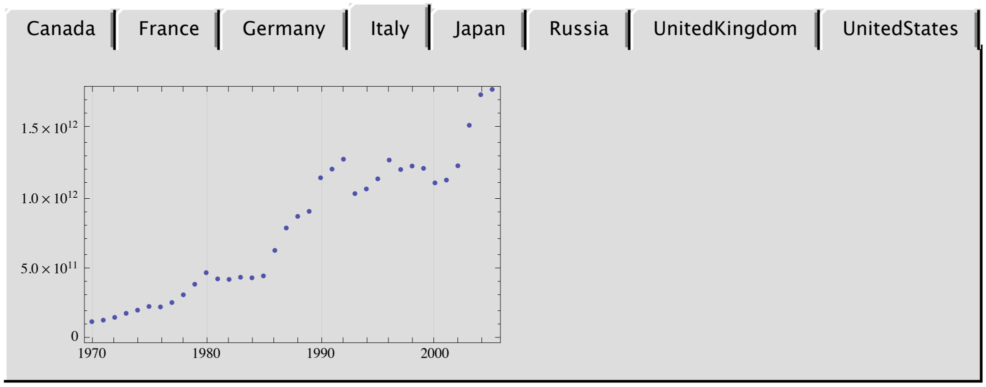
```
In[33]:= TabView[Table[i !, {i, 10}], 6]
```



```
Out[33]=  1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10
          720
```

```
In[34]:= TabView[Map[# → DateListPlot[CountryData[#, {{"GDP"}, {1970, 2005}}]] &,
            CountryData["GroupOf8"]], 4]
```

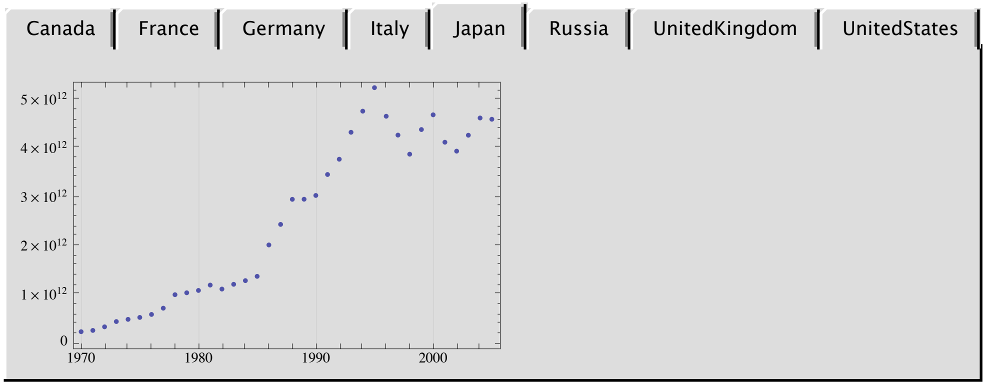
Out[34]=



Sometimes it is desirable to give symbolic identifiers to the panes in place of index numbers, allowing you to refer to them by name. This can be done using the form {*id*, *label* -> *contents*}. For example, here "Japan" is used rather than "6" in the second argument.

```
In[35]:= TabView[Map[{#, # → DateListPlot[CountryData[#, {{"GDP"}, {1970, 2005}}]]} &,
           CountryData["GroupOf8"]], "Japan"]
```

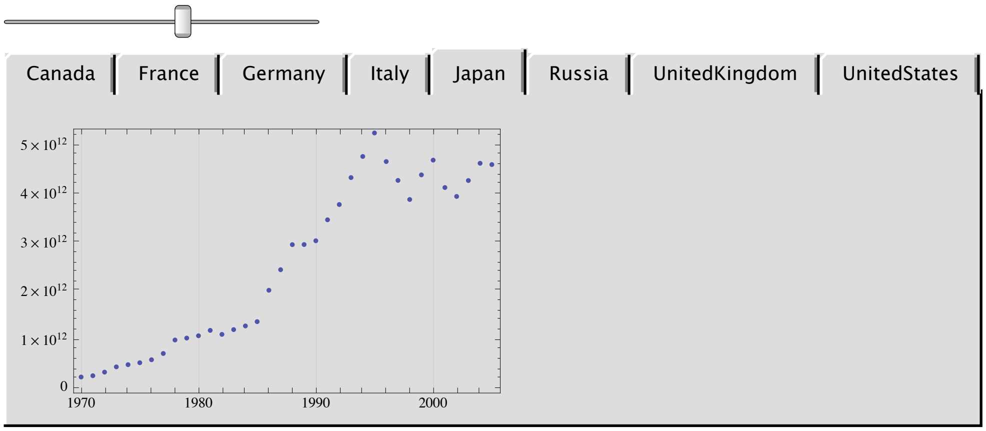
Out[35]=



Using a `Dynamic` variable in the second argument allows you to control the currently displayed pane from a separate control. (`Dynamic` and `DynamicModule` are explained in "Introduction to Dynamic".) For example, here a slider is added that allows you to flip through the tabs. Note that the linkage is automatically bidirectional: if you click one of the tabs, the slider moves to the corresponding position.

```
In[38]:= DynamicModule[{index = 1},
          Column[{Slider[Dynamic[index], {1, 8, 1}],
            TabView[Map[# → DateListPlot[CountryData[#, {{"GDP"}, {1970, 2005}}]] &,
              CountryData["GroupOf8"]], Dynamic[index]]}]]
```

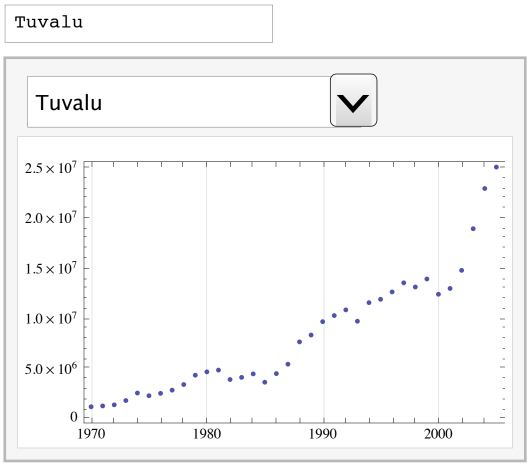
Out[38]=



In the previous example, having an index number to refer to the panes is good, as it makes linkage to a numerical `Slider` easy. If, on the other hand, you want to have a text field where you can enter the country name, having named panes is more convenient. This example provides a text field where you can enter a country name directly.

```
In[39]:= DynamicModule[{country = "Canada"},
          Column[{InputField[Dynamic[country], String], MenuView[
             Map[{#, # → Dynamic[DateListPlot[CountryData[#, {{"GDP"}, {1970, 2005}}]]]} &,
              CountryData[]], Dynamic[country], ImageSize → Automatic]}]]
```

Out[39]=


```
Tuvalu
```

Controlling whether a View Changes Size

Views always display one of several alternate panes. In determining the overall size of the View, there are two obvious alternatives: make the View big enough to hold the currently displayed pane, or make it big enough so that it never has to change size when switching between panes (i.e., as big as the biggest one in each dimension).

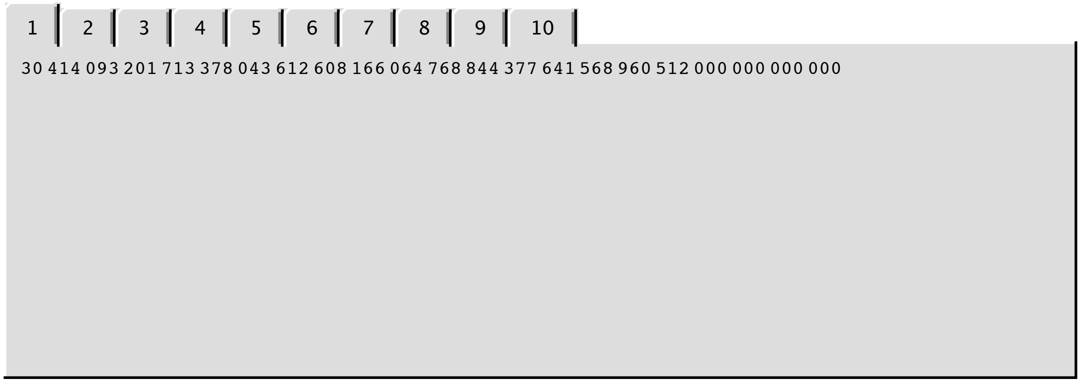
By default all Views, other than `OpenerView` and `FlipView`, are made large enough to never change size. (`OpenerView` in particular would make little sense if it did not get bigger when opened.)

The behavior of any View can be changed using the `ImageSize` option. `ImageSize -> All` means make the View as large as the largest pane, while `ImageSize -> Automatic` means make the View only as large as the currently displayed frame, potentially changing size any time the View is switched to a new pane. (You can also specify a fixed numerical `ImageSize`, in which case the View will attempt to fit its contents into the specified overall size.)

Compare these two examples (`ImageSize -> All` is the default; it is included only for clarity). The first one is always big, but stays the same size. The second one is only as big as it needs to be, and thus changes size.

```
In[40]:= TabView[Table[(50 i)!, {i, 10}], ImageSize → All]
```

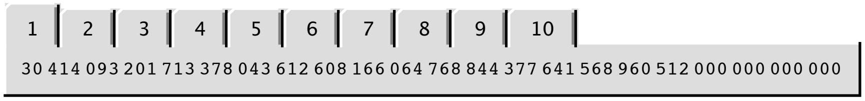
Out[40]=



```
1 2 3 4 5 6 7 8 9 10
30 414 093 201 713 378 043 612 608 166 064 768 844 377 641 568 960 512 000 000 000 000
```

```
In[41]:= TabView[Table[(50 i)!, {i, 10}], ImageSize → Automatic]
```

Out[41]=



```
1 2 3 4 5 6 7 8 9 10
30 414 093 201 713 378 043 612 608 166 064 768 844 377 641 568 960 512 000 000 000 000
```

The `ImageSize` option works this way for all View objects.

Which behavior is best depends on the situation. In general, tab views and similar controls used in applications other than *Mathematica* rarely change size, so if you are trying to make something that looks and acts like a traditional hard-coded application, `ImageSize -> All` is best. On the other hand, using `ImageSize -> Automatic` allows you to take advantage of the fact that, in *Mathematica*, dialog boxes and controls are not fixed objects. A great deal of freedom and flexibility is possible precisely because these objects *can* change size.

Dynamic Content in Views

This section assumes that you are familiar with the `Dynamic` mechanism (see "Introduction to Dynamic").

All the View objects fully support `Dynamic` content in any positions where it makes sense. Consider this `MenuView` example.

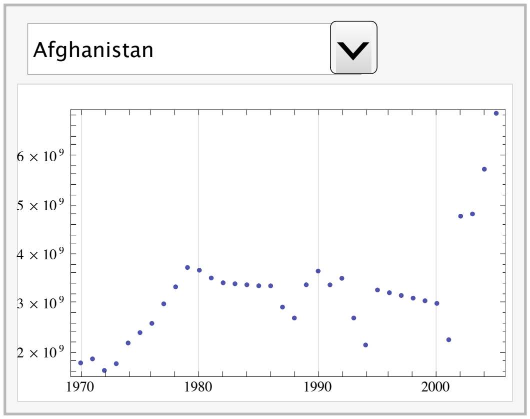
```
MenuView[Map[# → DateListPlot[CountryData[#, {{"GDP"}, {1970, 2005}}]] &,
  CountryData[]], ImageSize → Automatic]
```

In this form, the example computes in advance all 237 GDP plots, generating errors for some countries where data is missing, and doing far more computation than necessary, since it is unlikely anyone will try to look at every single country. The code takes a long time to evaluate and wastes a lot of memory.

By simply wrapping `Dynamic` around the contents of each page, the input evaluates almost instantly and produces an output that occupies very little memory.

```
In[42]:= MenuView[Map[# → Dynamic[DateListPlot[CountryData[#, {{"GDP"}, {1970, 2005}}]]] &,
           CountryData[]], ImageSize → Automatic]
```

Out[42]=



The trade-off is that each new country selected computes the GDP plot on the fly. Fortunately this generally happens so fast as to be unnoticeable. Errors for a particular country are displayed only if that country is selected.

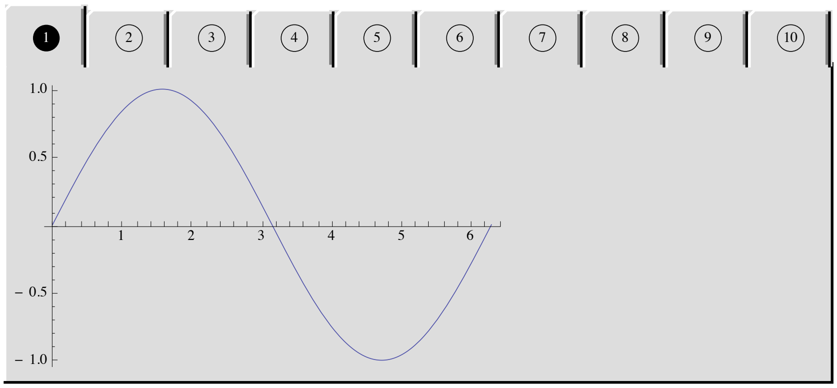
Note that with the setting `ImageSize-> All`(the default for all views except `OpenerView` and `FlipView`), every pane is formatted once when the object is first created, in order to determine the overall size of the view object. You can avoid this by setting the `ImageSize` option to `Automatic`or to a fixed numerical size.

(The astute reader will notice a subtlety here. With the setting `ImageSize-> All`and dynamic content in currently invisible panes, it would theoretically be necessary to continually update the values of all the hidden dynamics, since the size of the View as a whole should depend on the size of the largest pane, even if it is not currently being displayed. An intentional decision was made *not* to do such updating of hidden panes. As a result, a View with `ImageSize-> All` can in fact change size when a new pane is selected, if that pane contains dynamic content that has changed size since the last time it was displayed. The alternative would be for the View to change size mysteriously when activity in a hidden pane caused that hidden pane to change size. This would be peculiar and of little conceivable use.)

In the case of `TabView`, dynamic tab labels can be used to implement a variety of special behaviors. In this example, the currently selected tab is highlighted in a custom-defined way, by making the labels dynamically dependent on the variable that tracks the currently selected pane.

```
In[43]:= DynamicModule [{j}, TabView[Table[With[{i = i}, Graphics[Dynamic[
                If[i === j,
                 { Disk[], White, Inset[i]},
                 { Circle[], Inset[i]}]], ImageSize → 25, PlotRangePadding →.5]→
             Plot[Sin[i x], {x, 0, 2 Pi}]], {i, 10}], Dynamic[j]]]
```

Out[43]=



An important property of Views is that currently hidden panes are not updated. Consider this example.

```
In[44]:= OpenerView[
          {"Mouse Position", Column[{"Mouse Position", Dynamic[MousePosition[]]}] } ]
Out[44]= ▶ Mouse Position
```

When the output is in the open state, the current position of the mouse pointer is displayed and continuously updated, consuming a certain amount of CPU time. However, when the output is in the closed state, the mouse position is no longer tracked and no CPU time is used. (This is of course of more concern if the contents are something more compute-intensive than simply displaying the mouse position.)

This property allows you to do things like build large, complex `TabView`s in which expensive computations are done in each pane of the view, without incurring the cost of keeping all the panes updated all the time.

Views versus Controls

There are two classes of functions in *Mathematica* that represent relatively low-level user interface objects: Views and Controls. This tutorial describes the Views class of functions, but there is considerable overlap with Controls in some cases.

Views are designed to present multiple panes of data and provide a user interface for switching among them, so the logical first argument is the list of expressions representing the contents of the panes.

Controls are primarily designed to influence the value of a variable through a `Dynamic` connection, so the first argument of all control functions is the variable representing the value of the control.

What is potentially confusing is that views also allow you to control the value of a variable, just like controls do. In at least one case, `PopupView` versus `PopupMenu`, the functions are essentially identical with the arguments reversed.

```
In[45]:= PopupView[{1, 2, 3, 4}, Dynamic[popPosition]]
```

Out[45]=

```
In[46]:= PopupMenu[Dynamic[popPosition], {1, 2, 3, 4}]
```

Out[46]=

Why have both? In the case of `PopupView` and `PopupMenu` it is simply for consistency with the other View and Control functions, though there is the convenience that the second argument of `PopupView` is optional (since very often you do not need to provide any external control of the currently displayed pane). In the case of `PopupMenu`, the only purpose in creating the control is for it to control a variable, so the first argument is of course not optional.

Other than the set described in the next section, views do not correspond quite so directly with any control objects. It is, however, useful to keep in mind that views can, through their second argument, be used essentially as control objects: they can control and be controlled by the value of a variable, that is simply not their only purpose.

FlipView versus PaneSelector versus Toggler

There are three objects that appear (and in fact are) very similar but not identical: `FlipView`, `PaneSelector`, and `Toggler`. Each of these objects takes a list of expressions and displays one of them at a time. They differ in the details of their argument order and click behavior. (But mainly they differ in their intended use more so than in their actual behavior.)

`FlipView` and `PaneSelector` take identical arguments: a list of expressions and a number that specifies which pane should be displayed. The difference is that clicking anywhere in a `FlipView` flips to the next pane, while clicking in a `PaneSelector` allows you to edit the contents of the currently displayed pane (and there is no user interface to flip to any other pane).

```
In[47]:= FlipView[Table[{i, i !}, {i, 10}], 6]

Out[47]= {6, 720}

In[48]:= PaneSelector[Table[{i, i !}, {i, 10}], 6]
Out[48]= {6, 720}
```

`Toggler` behaves exactly like `FlipView` in that it flips between panes when clicked, but the arguments are in the opposite order, with the index number first (see previous section for why this actually makes sense). `Toggler` also uses `ImageSize -> All` by default, while `PaneSelector` uses `ImageSize -> Automatic`.

```
In[49]:= Toggler[6, Table[{i, i !}, {i, 10}]]

Out[49]= {1, 1}
```

For more information and a detailed listing of options, see `FlipView`, `PaneSelector`, and `Toggler`.