

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}]
```



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]
```



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)]
```



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.

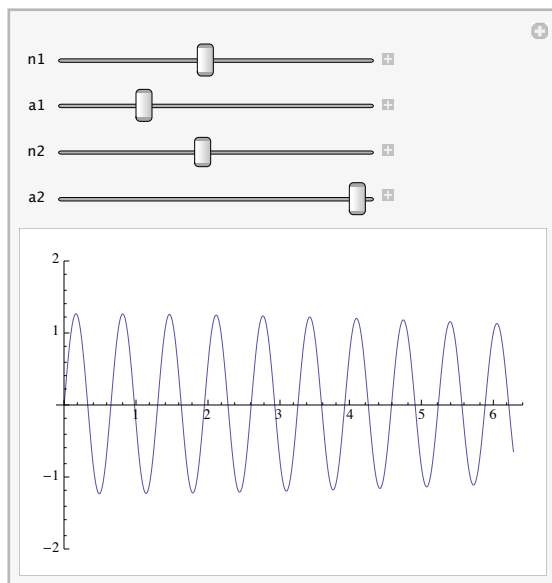
In order to take advantage of a USB controller in `Manipulate`, all you have to do is plug it in, and use the mouse to select (highlight) the cell bracket containing the `Manipulate` output you want to control. *Mathematica* automatically detects the controller, and `Manipulate` automatically links as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



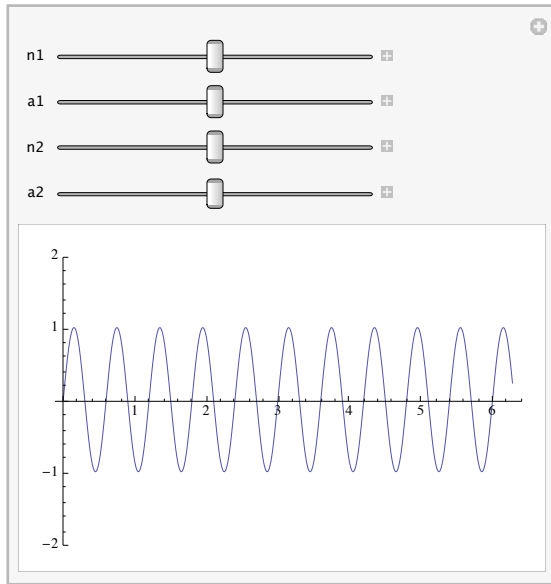
By default, `Manipulate` connects the x axis of the left gamepad to the first parameter, the y axis of the left joystick to the second parameter, the x axis of the right joystick to the third parameter, and the y axis of the right gamepad to the fourth parameter. You can verify this by moving each joystick 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],
{a1, 0, 1}, {n1, 1, 20}, {a2, 0, 1}, {n2, 1, 20}, 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.

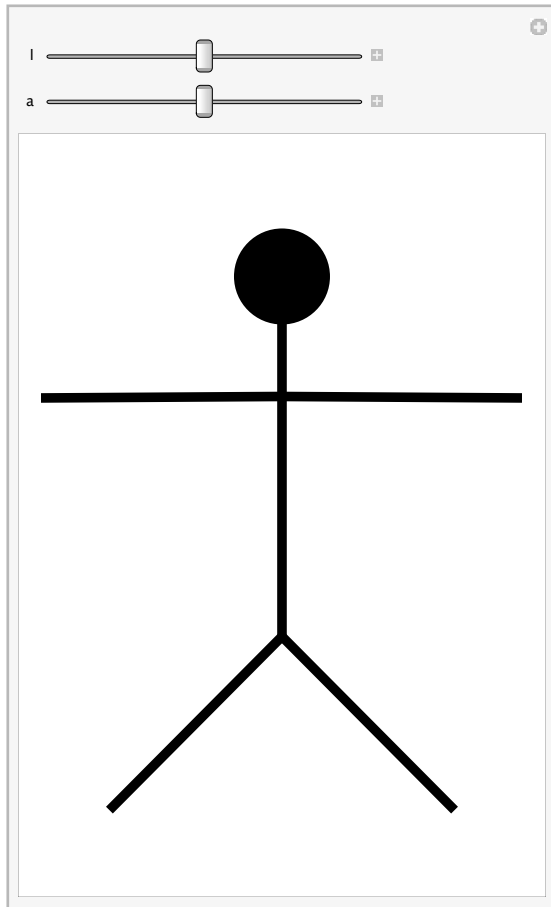
A small variation on velocity control can be had with the option `ControllerMethod -> "Cyclic"`. With this setting the linkage is velocity-based, but when you reach one end of the parameter's range of values, instead of stopping it cycles around to the opposite end.

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[1], Sin[1]}, {0, 0}, {Cos[1], Sin[1]}}],
  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}}, {1, -Pi / 2, 0},
  {a, -Pi / 2, Pi / 2}, ControllerMethod -> "Absolute"]

```

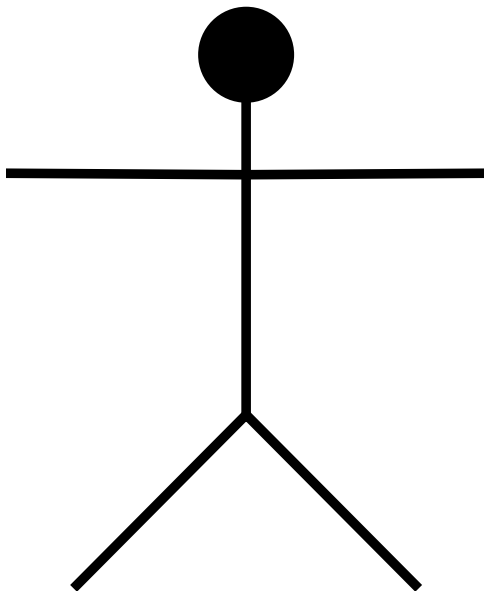


If any of these examples seem not to be working, chances are it is because you have forgotten to highlight the cell bracket containing them. This is a common mistake. As a convenience, and to avoid the need to select each output as soon as it is generated, if you wiggle any gamepad controller immediately 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.

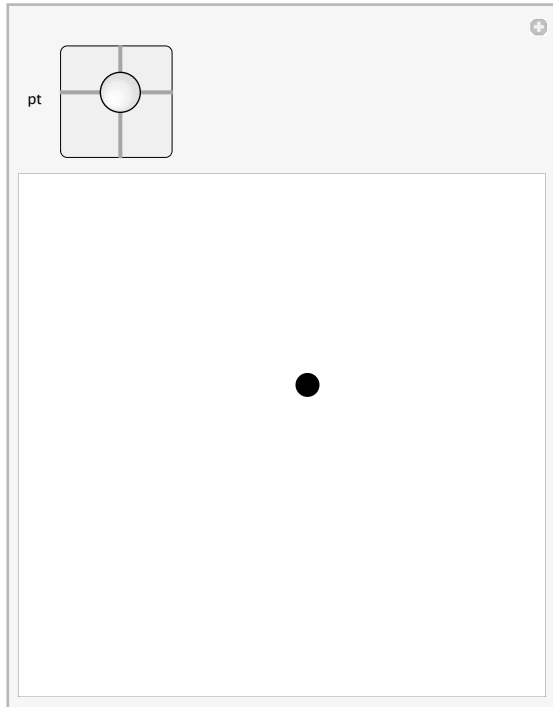
```
ControllerManipulate[Graphics[{Thickness[0.02],
  Line[{{-Cos[1], Sin[1]}, {0, 0}, {Cos[1], Sin[1]}]],
  Line[{{0, 0}, {0, 1.5}}],
  Line[{{-Cos[a], 1 + Sin[a]}, {0, 1}, {Cos[a], 1 + Sin[a]}]],
  Disk[{0, 1.5}, .2]],
PlotRange -> {{-1, 1}, {-1, 2}}],
{1, -Pi / 2, 0},
{a, -Pi / 2, Pi / 2},
ControllerMethod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

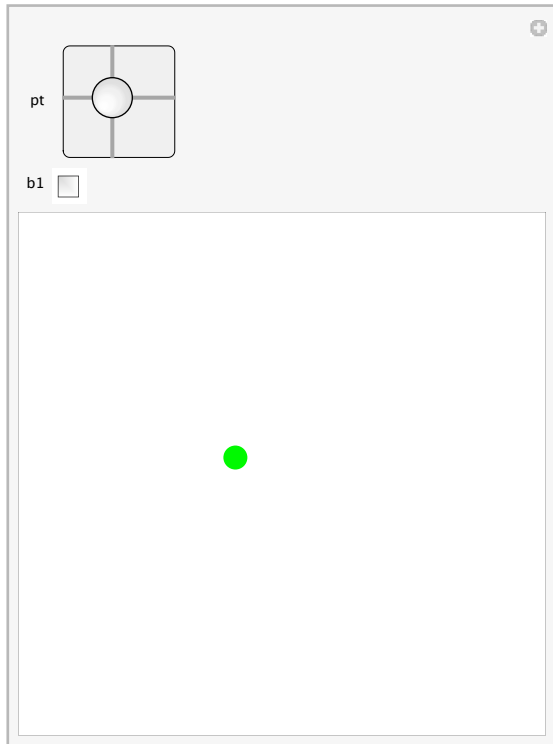
If the `Manipulate` contains `Slider2D` variables, whose values are $\{x, y\}$ pairs of numbers, they will automatically be linked to both directions of available joysticks. This example responds in both directions to the left-hand joystick on a gamepad.

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



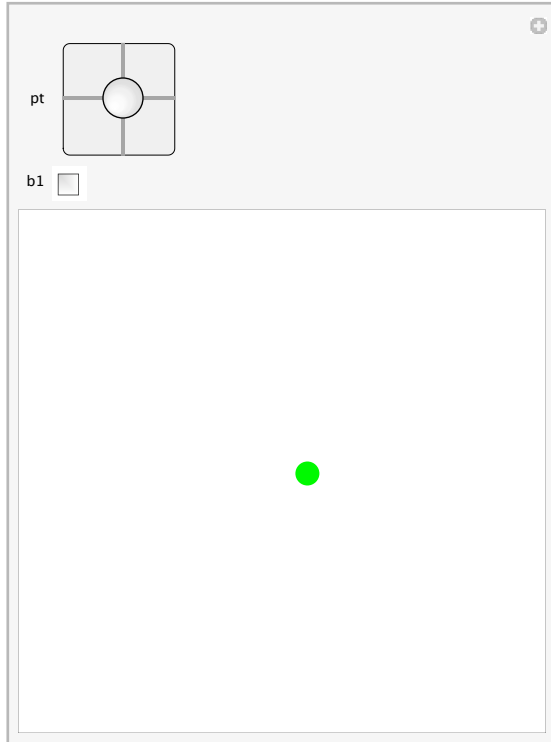
Push buttons on the controller are by default linked up to any Boolean (True/False) parameters specified in the `Manipulate`. Which button is which can be a bit hard to guess on a given controller (a concept which is explained elsewhere), but on the Logitech model the group of four buttons on the right 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"]
```

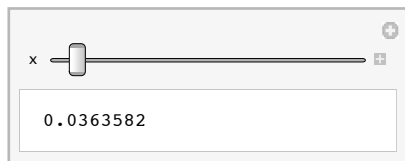


The exact rules by which `Manipulate` connects controller axes to parameters are somewhat complex, but basically they try to allocate the available analog and Boolean controls to the parameters in the `Manipulate` so as to make maximum use of the available joysticks, buttons, knobs, and other widgets 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}]
```



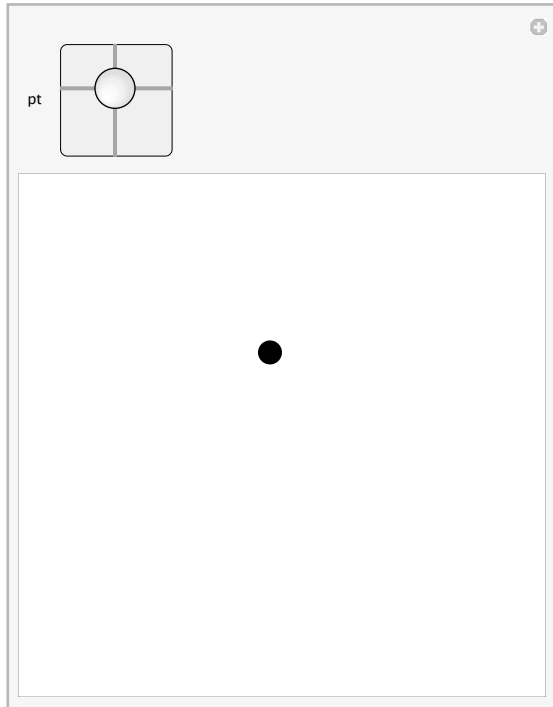
"x2" similarly refers to the x axis of the secondary, or right-hand joystick.

```
Manipulate[x, "x2" → {x, 0, 1}]
```



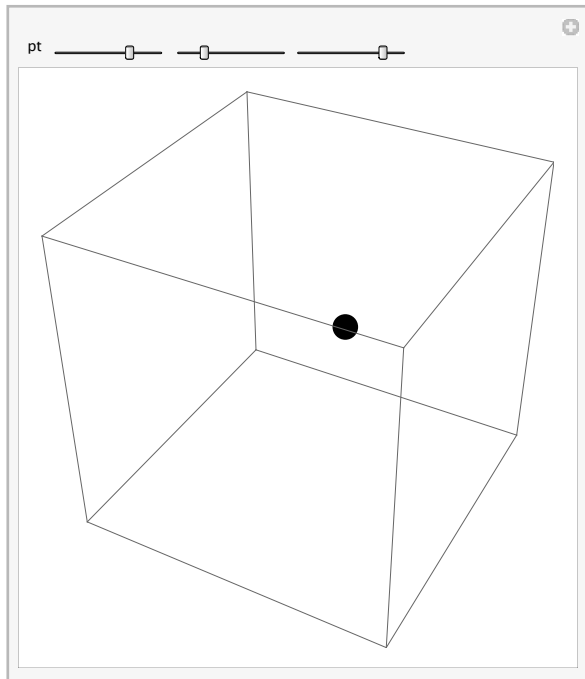
Axes can be combined into multidimensional parameters. For example, "XY" refers to both directions of the left or primary joystick, combined into a single $\{x, y\}$ value. Such a combined axis must be connected to a slider2D style of parameter, as in this example.

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



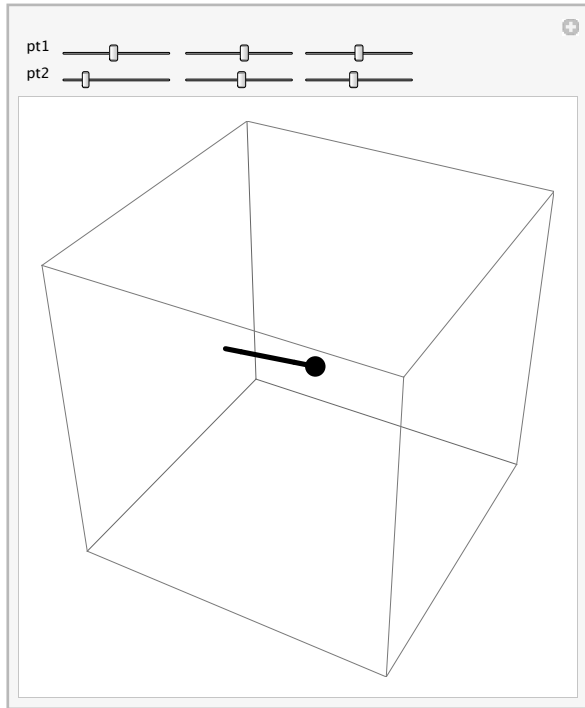
Three-axis variables are also supported as "XYZ". When using a three-axis joystick controller, the axes will correspond to the three degrees of freedom of the joystick. When using a dual-joystick gamepad, each joystick only has two degrees of freedom. In this case the "XYZ" axis is linked to the x and 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.

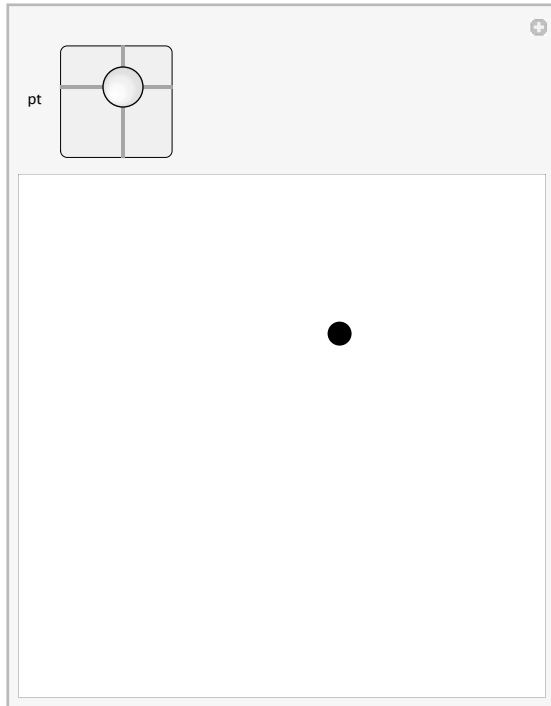
```
Manipulate[
  Graphics3D[{Thickness[0.01], PointSize[0.04], Point[pt1], Line[{pt1, pt1 + pt2}]},
    PlotRange -> 1], "XYZ" -> {pt1, {-1, -1, -1}, {1, 1, 1}},
  "XYZ2" -> {pt2, {-1, -1, -1}, {1, 1, 1}}, ControllerMethod -> "Absolute"]
```



(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*

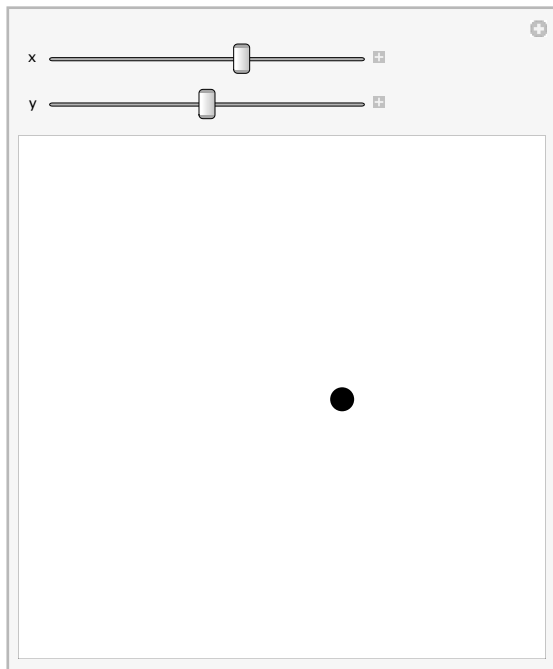
 ► **Controller Device 2: Apple IR**

 ► **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.

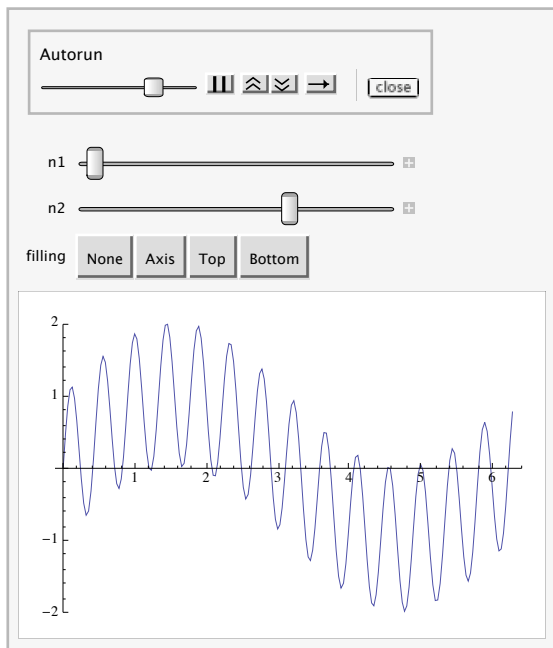
Autorun

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.

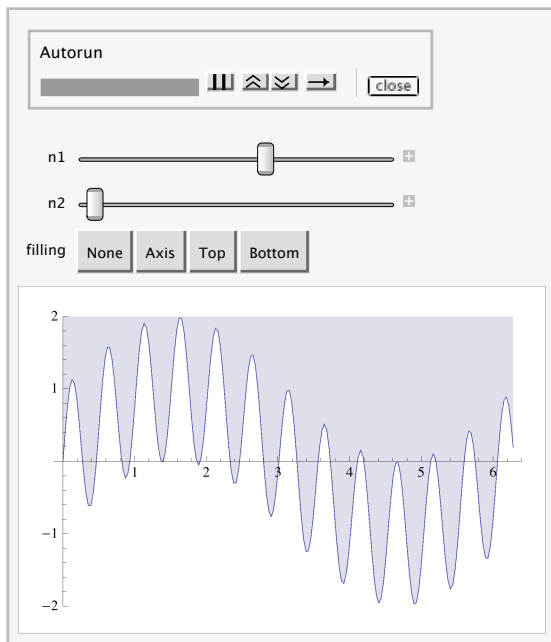
Click the  menu in the top right corner of a `Manipulate` output and select **Autorun** from the bottom of the menu. You will see an **Autorun** panel appear at the top of the `Manipulate`, containing animation controls and a `close` button. By default the animation runs each individual variable through 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}}
```



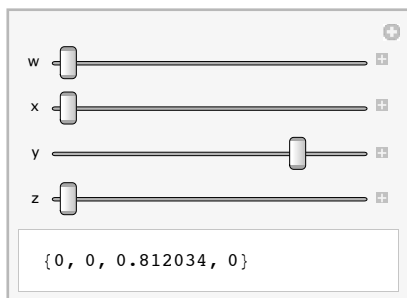
The default behavior of **Autorun** simulates something you could do yourself with the mouse, moving one control at a time. If you add the option `AutorunSequencing -> All` to the `Manipulate` input, the **Autorun** command in the resulting output will instead move all the controls simultaneously, as 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]
```



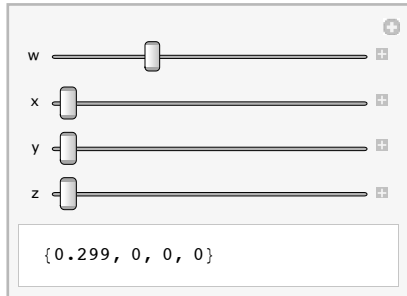
You can also use `AutorunSequencing` to exclude certain controls from the **Autorun** animation, or change the order in which the controls are animated. In the following example, the third control is animated first, then the first control, then the fourth, and the second control is left at its default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

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



`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}}]
```



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.

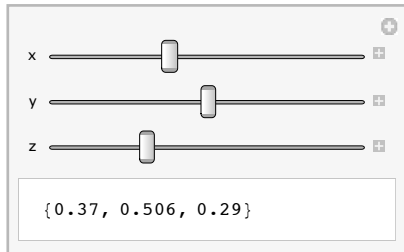
If `AutorunSequencing` does not give you enough control over the animation sequence, you can use the **Bookmarks** feature described in the next section to define a list of "way points"—combinations of parameter values—and then create an animation that smoothly interpolates through those defined points.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



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 \oplus 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 \oplus 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 \oplus 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.

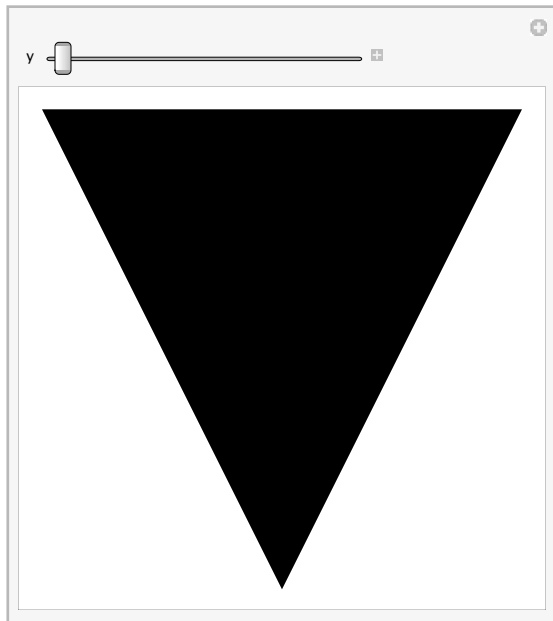
This tutorial also, in places, assumes a familiarity with the lower-level dynamic mechanism covered in "Introduction to Dynamic" and "Advanced Dynamic Functionality".

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 accompanying 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.

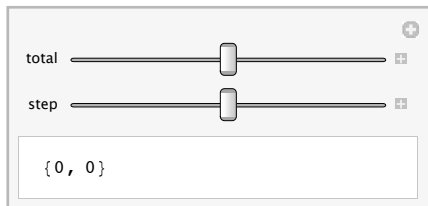
```
Manipulate[
  y = Max[-1, y - 0.05]; Graphics[Polygon[{{-1, 1}, {0, y}, {1, 1}}], PlotRange -> 1],
  {{y, 1}, -1, 1}]
```



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}]
```



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}]

```

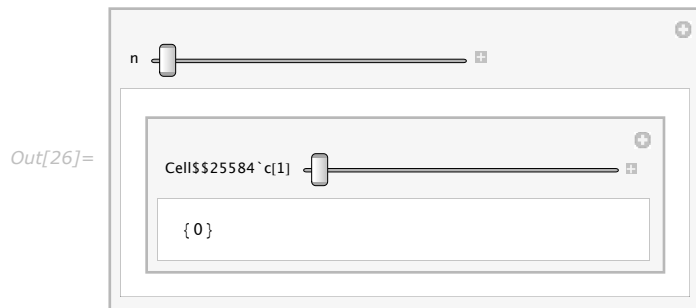
This example updates the content area only when `n` changes its value as a result of moving the slider.

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}]
```

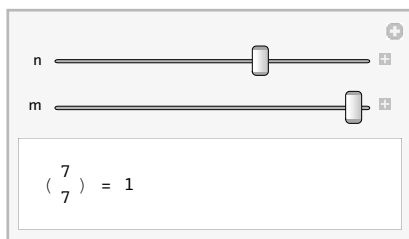


While nesting `Manipulate` many levels deep is possible, and will work, it is probably not the most useful feature in the world. But by nesting once, you have in effect created a parameterized user interface construction interface. The outer `Manipulate` allows you to control parameters that determine 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 $m \leq n$, 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.

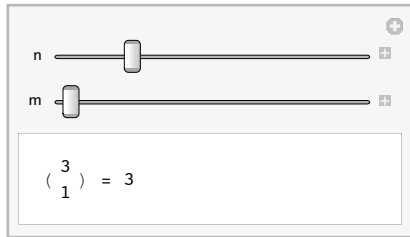
```
Manipulate[Row[{"(", Column[{n, m}, Center], ") = ", Binomial[n, m]}],
  {n, 1, 10, 1}, {m, 1, n, 1}]
```



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.

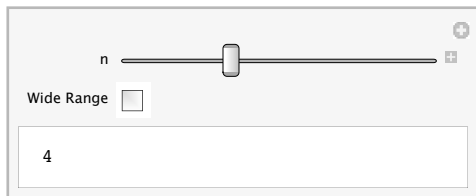
```
Manipulate[If[m > n, m = n];
  Row[{"(", Column[{n, m}, Center], ") = ", Binomial[n, m]}],
  {n, 1, 10, 1}, {m, 1, n, 1}]
```



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}}]
```



Dealing with Slow Evaluations

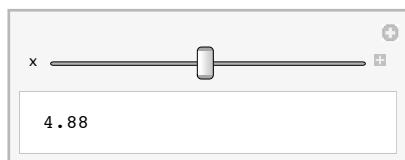
`Manipulate` does not precompute all the possible output values you could reach by moving its sliders: that would be completely impractical for all but the most trivial cases. That means it has to calculate, format, and display the current value in real time as each slider is being dragged. Obviously 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 computers 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.)

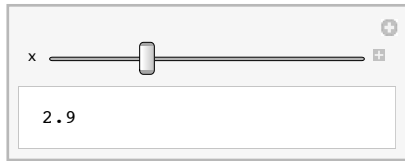
```
Manipulate[Pause[1]; x, {x, 0, 10}]
```



```
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]
```



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.

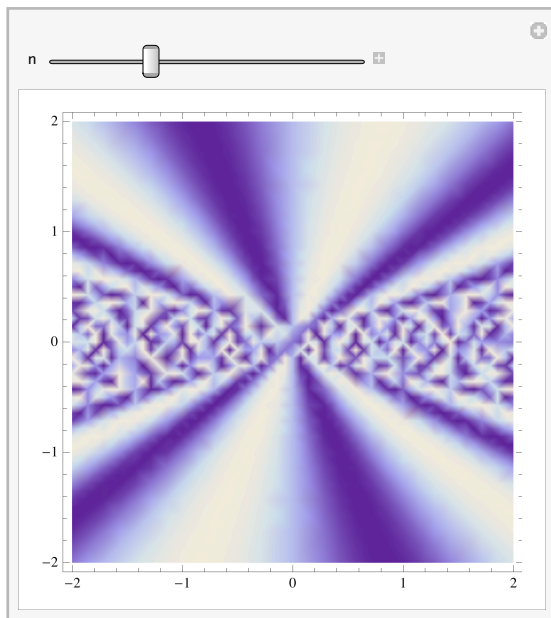

```
Manipulate[ControlActive[x, Pause[1]; Framed[x]], {x, 0, 10}]
```



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.

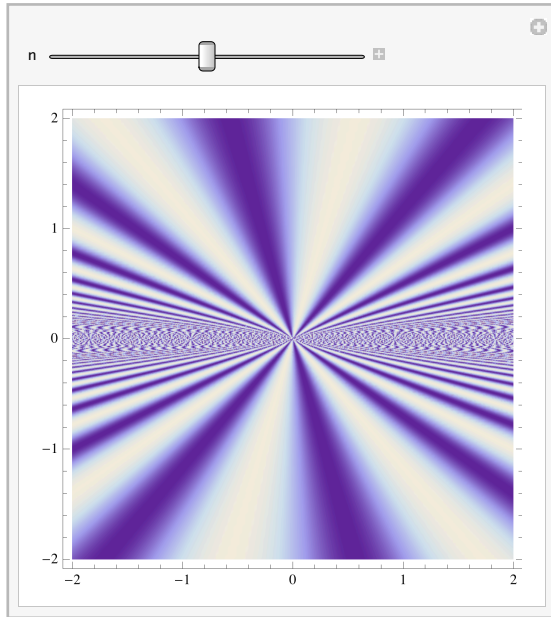
Here is a slightly more realistic example of where `ControlActive` can be useful. This example shows how the default behavior of `DensityPlot` is to use fewer sample points while the slider is being dragged.

```
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.

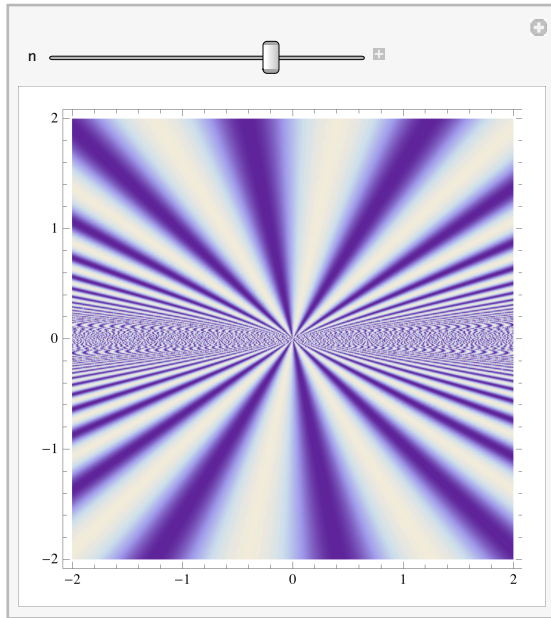
```
Manipulate[
  DensityPlot[Sin[n x / y], {x, -2, 2}, {y, -2, 2}, PlotPoints -> 150], {n, 1, 10}]
```



(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

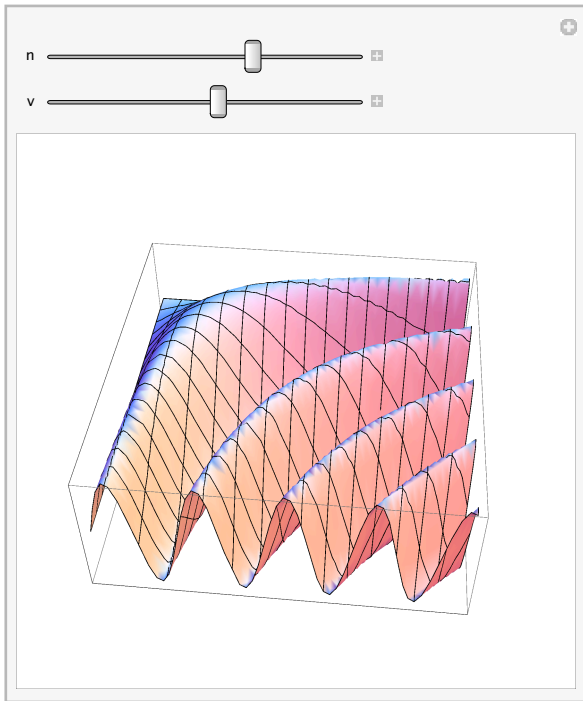
You might want to read "Introduction to Dynamic" before finishing this section, as we refer to the use of explicit `Dynamic` expressions, which are not explained in this tutorial.

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.

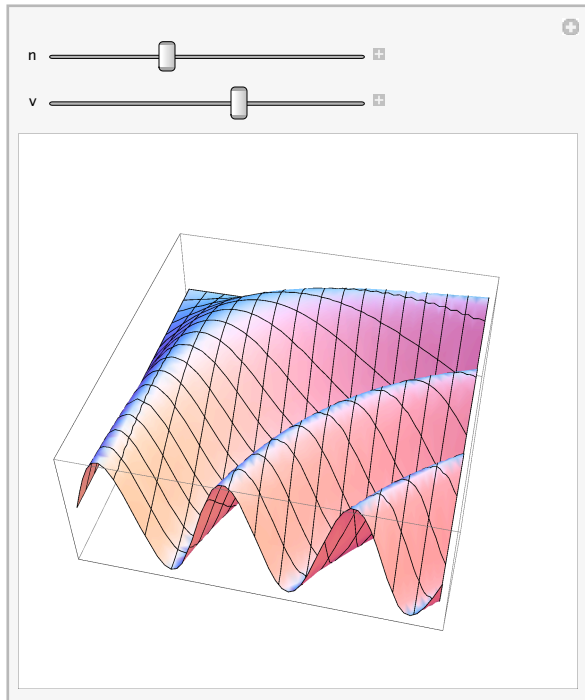
```
Manipulate[Plot3D[Sin[n x y], {x, 0, 3}, {y, 0, 3}, ViewPoint -> {2, v, 2},  
SphericalRegion -> True, Ticks -> None], {n, 1, 4}, {v, -2, 2}]
```



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 viewpoint 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.

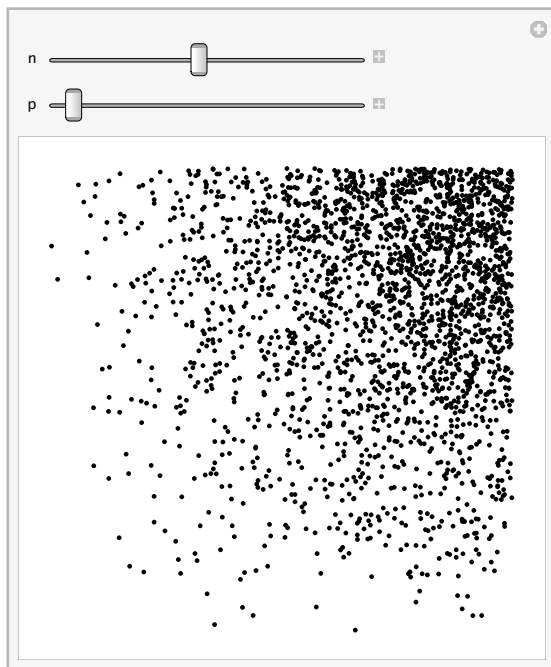
To explain exactly why this works requires an understanding of the internals of the `Dynamic` mechanism explained in "Introduction to Dynamic" and "Advanced Dynamic Functionality". In short, `Manipulate` always wraps `Dynamic` around the expression given in its first argument, and normally any changes 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[datap]], AspectRatio → 1},
  {n, 100, 5000, 1}, {p, 0.1, 10}, SynchronousUpdating → False]
```



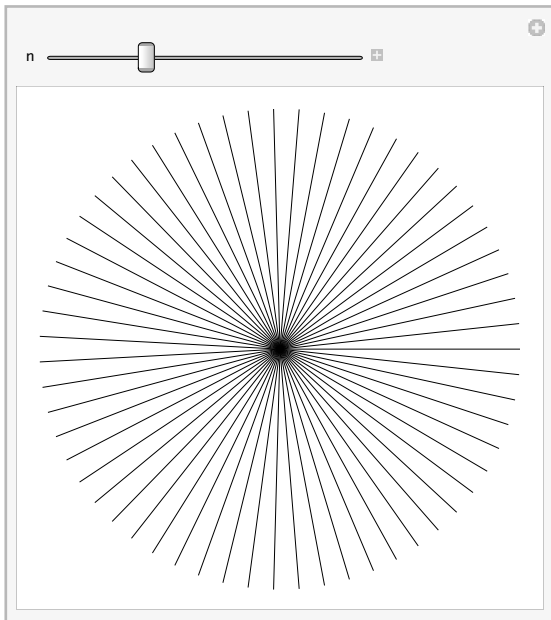
The option `SynchronousUpdating -> False` is used to cause the outer `Dynamic` (the one implicitly created by `Manipulate`) to update asynchronously (visible by the fact that the cell bracket becomes outlined when the n -slider is moved). Asynchronous updating does not give quite as smooth updating, 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 / n}]]],
  PlotRange -> 1], {{n, 30}, 1, 200, 1}]
```



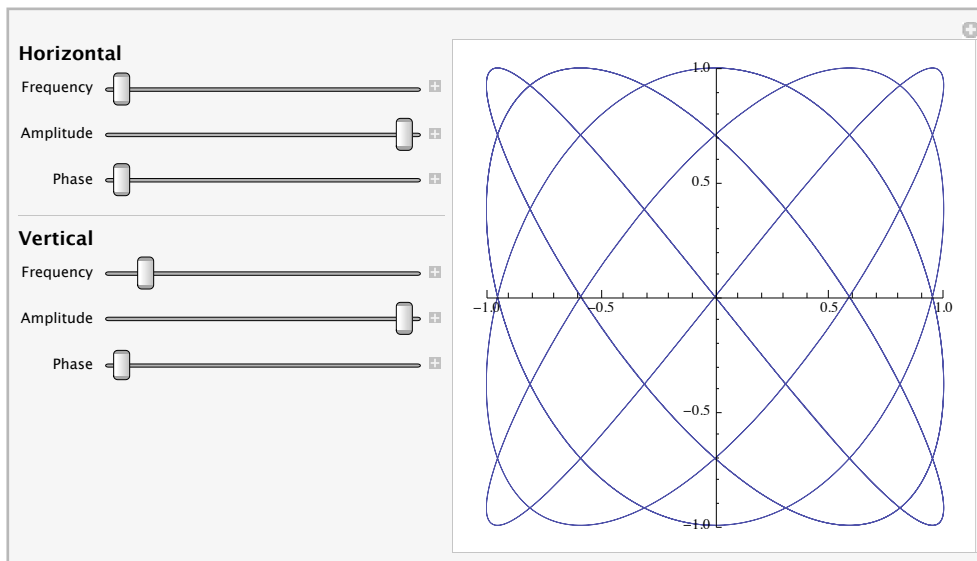
Any time the mouse is over the area of the plot, the center of the lines will follow it (without a click). Consult the documentation for `MousePosition` for further detail.

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]
```



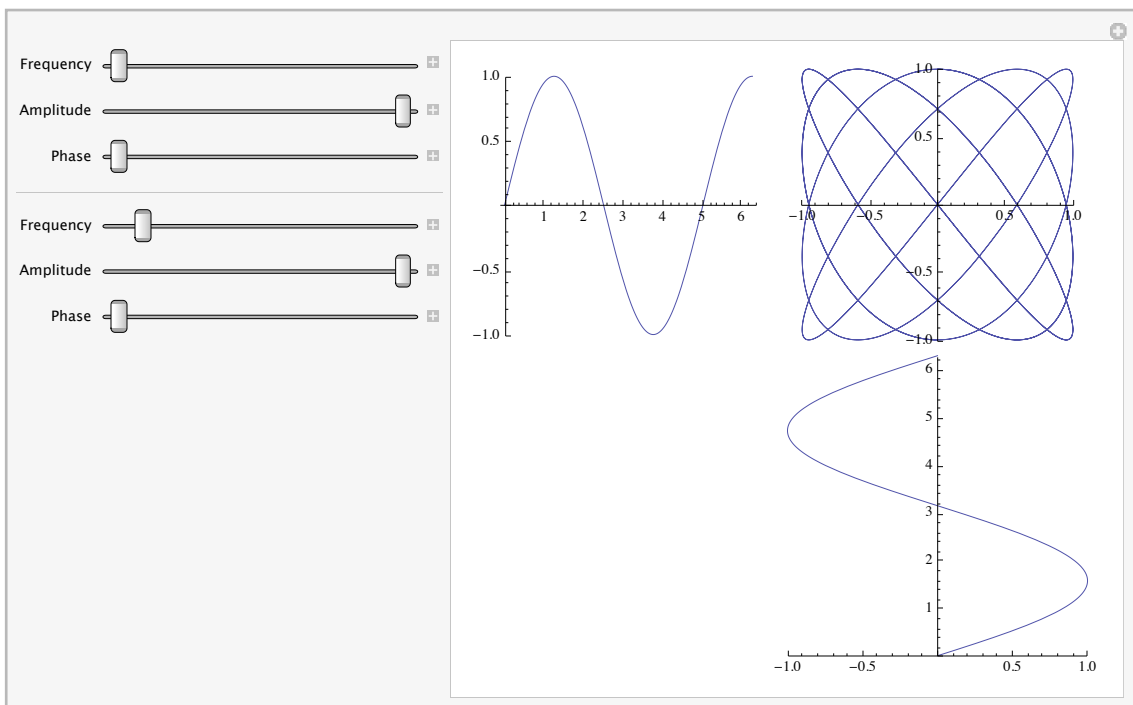
There is in fact virtually no limit to what can be put into the controls area, including arbitrary formatting constructs and dynamic objects, even controls that are not part of the `Manipulate`'s

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.

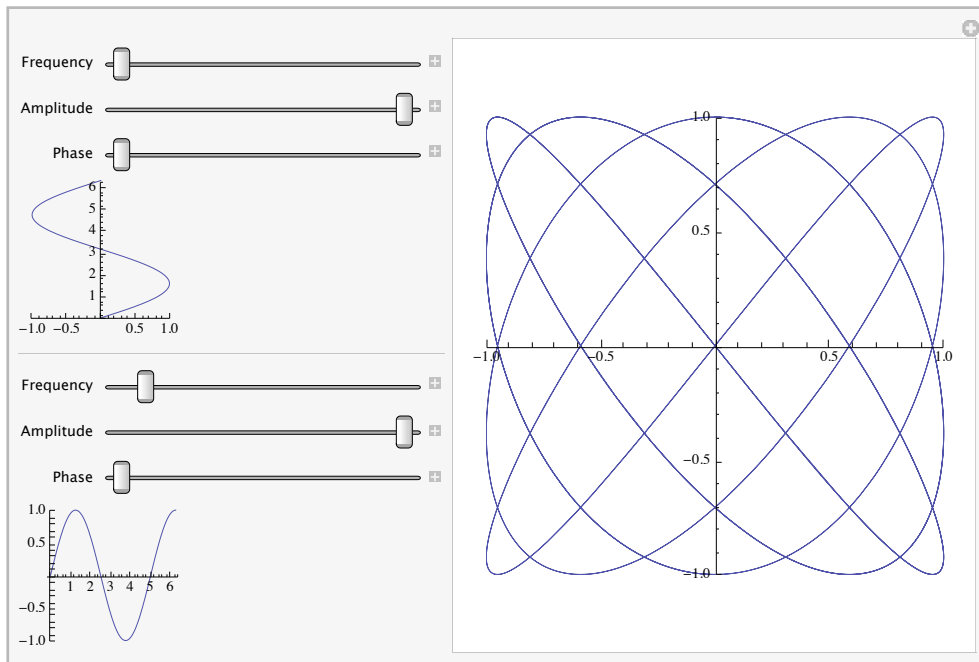
Suppose you want to show plots of the individual x and y sine functions that combine to form the Lissajous figure. You could do this by putting all three functions into the output area, using `Grid` to lay them out.

```
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

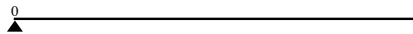
You might want to read "Introduction to Dynamic" before reading this section, as we refer to the use of explicit `Dynamic` expressions, which are not explained in this tutorial.

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];
```

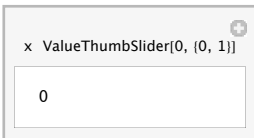
Here is an example of what this new control looks like: click anywhere to move the thumb around, just like with a normal slider.

```
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`.

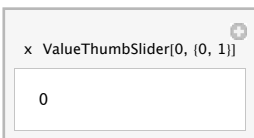
```
In[4]:= Manipulate[x, {x, 0, 1, ValueThumbSlider[##] &}]
```

```
Out[4]= 
```

(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.

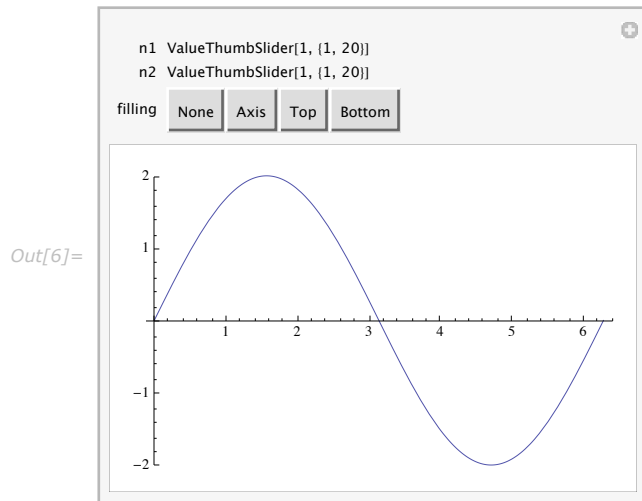
```
In[5]:= Manipulate[x, {x, ValueThumbSlider[#, {0, 1}] &}]
```

```
Out[5]= 
```

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}}]
```

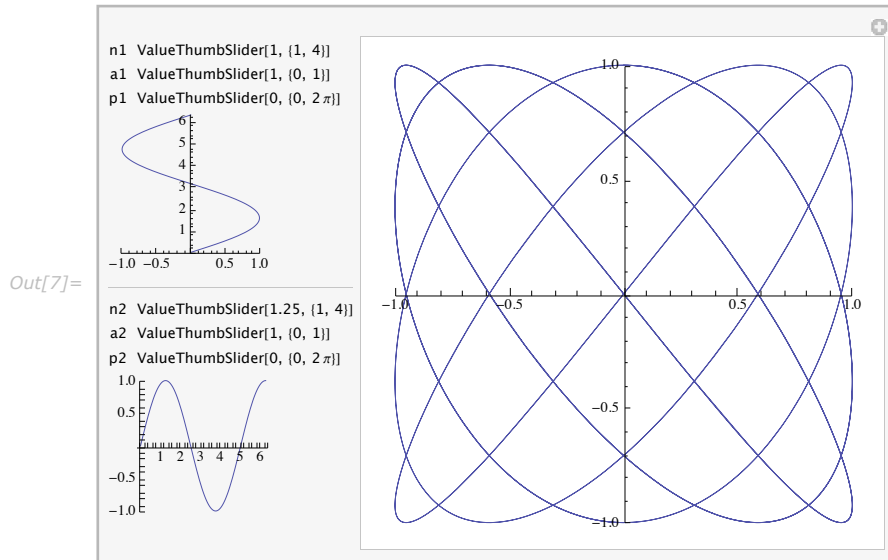


It is also possible to combine custom controls with other dynamic elements in the controls area (discussed in the previous section).

```

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]

```



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, 40320, 362880, 3628800}`

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.

`In[2]:= Table[$\frac{n+3}{n!}$, {n, 1, 10}]`

`Out[2]= {4, $\frac{5}{2}$, 1, $\frac{7}{24}$, $\frac{1}{15}$, $\frac{1}{80}$, $\frac{1}{504}$, $\frac{11}{40320}$, $\frac{1}{30240}$, $\frac{13}{3628800}$ }`

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.

`In[3]:= Table[$\frac{n + \text{Slider}}$, {n, 1, 10}]`

`Out[3]= {5, 3, $\frac{7}{6}$, $\frac{1}{3}$, $\frac{3}{40}$, $\frac{1}{72}$, $\frac{11}{5040}$, $\frac{1}{3360}$, $\frac{13}{362880}$, $\frac{1}{259200}$ }`

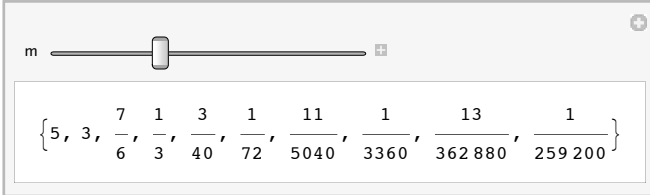
The "value" of this slider when it is given as input is determined by its position, in this case an integer between 1 and 10. It can be used anywhere in an input line that a textual number or variable name could be used.

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[Table[ $\frac{n+m}{n!}$ , {n, 1, 10}], {m, 1, 10, 1}]
```

Out[4]=






$\left\{5, 3, \frac{7}{6}, \frac{1}{3}, \frac{3}{40}, \frac{1}{72}, \frac{11}{5040}, \frac{1}{3360}, \frac{13}{362880}, \frac{1}{259200}\right\}$

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.

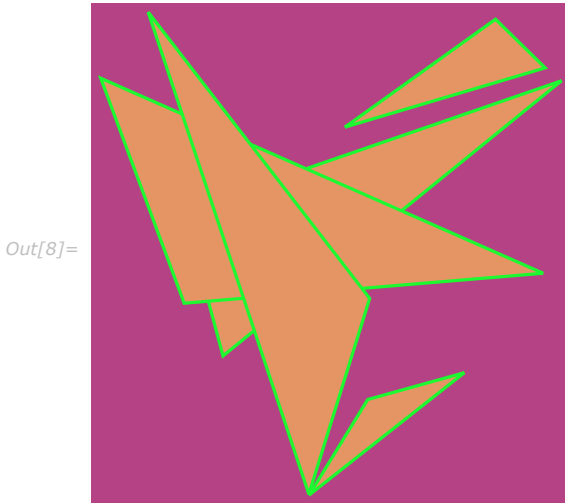
```
In[5]:= edgeColor = ;
```

```
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]
```



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[]`.

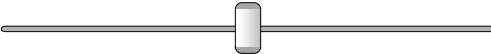
```
In[9]:= Slider[]
```




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.

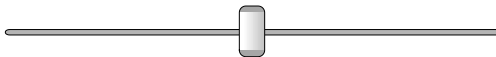
While copying and pasting can be used very effectively to build up input lines containing controls, it is often most convenient to use **Evaluate in Place** `Ctrl+Shift+Enter` (`Command+Return` on Mac) to transform control expressions in place, especially once you are familiar with the commands that 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 x2 + 10 x3 + 5 x4 + x5`

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)

This is the result after evaluating in place.

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

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

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.

<code>Interpretation[e, expr]</code>	an object which displays as e , but is interpreted as the unevaluated form of $expr$ if supplied as input
<code>Interpretation[{x=x₀, y=y₀, ...}, e, expr]</code>	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:

Min:

Max:

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.

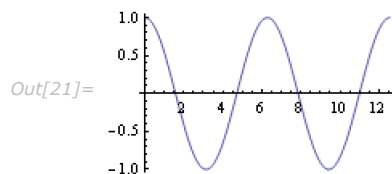
In[21]:=

Plot

Function:

Min:

Max:



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 = x2, 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]=

Integrate

Function:

Variable:

Definite integral

Min:

Max:

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

In[23]=

Integrate

Function:

Variable:

Definite integral

Min:

Max:

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.

Integrate

Function:

Variable:

Definite integral

Min:

Max:

`In[24]:= D[`

`, x]`

`Out[24]= x2`

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 = x2, 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]=`

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.

$$\text{In}[26]:= \int_a^b x^2 dx$$

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

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:

Variable:

Definite integral

Min:

Max:

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

In[28]:=

Integrate

Function:

Variable:

Definite integral

Min:

Max:

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:

Variable:

Definite integral

Min:

Max:

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]=

The image shows a typographic integral template. It features a large integral symbol \int with a small 'd' and a variable 'x' to its right. The limits of integration are 'a' at the bottom and 'b' at the top. The integrand is 'x^2'. Each of these elements is contained within a separate input field box.

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]:=

The image shows a nested typographic integral template. It features a large integral symbol \int with a small 'd' and a variable 'x' to its right. The limits of integration are 'a' at the bottom and 'b' at the top. The integrand is 'x^2'. Each of these elements is contained within a separate input field box.

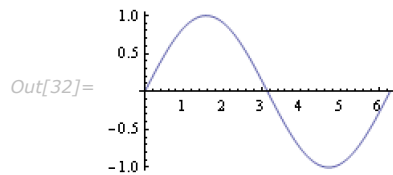
Out[31]= $-\frac{a^3}{3} + \frac{b^3}{3}$

Graphics as Input

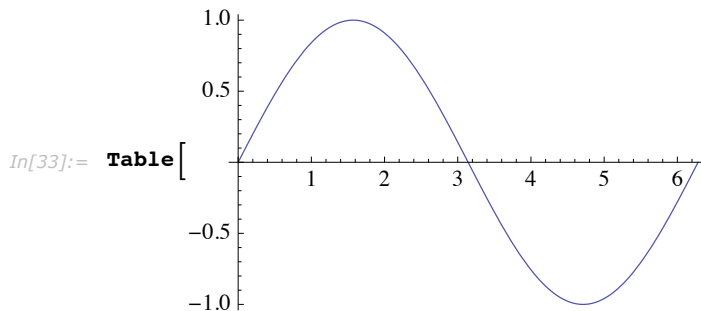
Graphic objects, including the output of `Graphics`, `Graphics3D`, plotting commands, and graphics imported from external image files, can all be used as input and freely mixed with textual input. There are no arbitrary limitations in the mixing of graphics, controls, typeset mathematics, and text.

Evaluate a simple plot command.

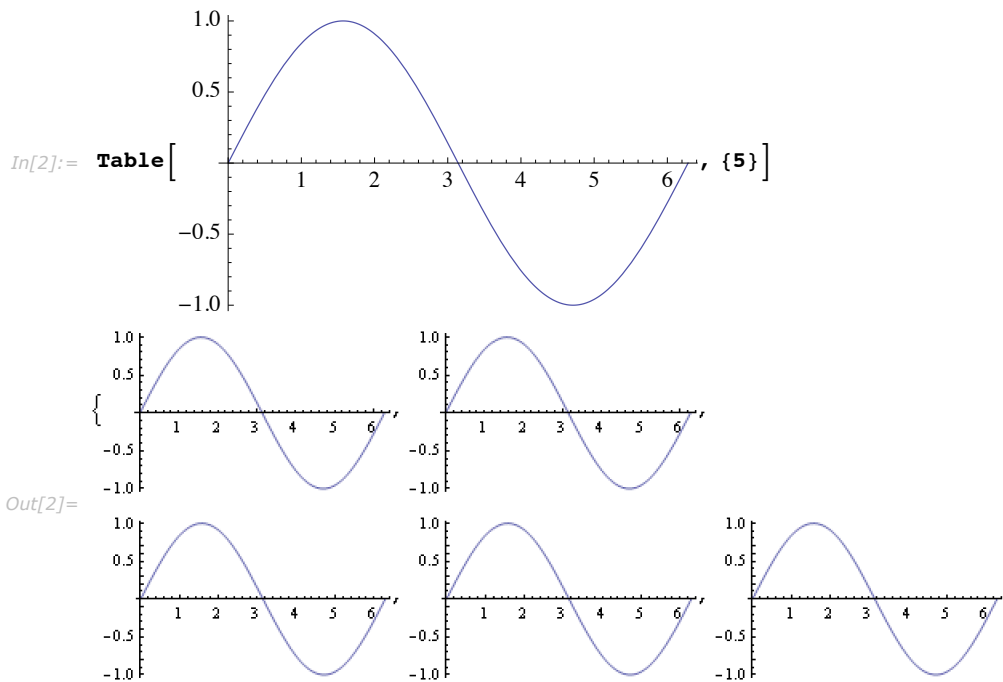
```
In[32]:= Plot[Sin[x], {x, 0, 2 Pi}]
```



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



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

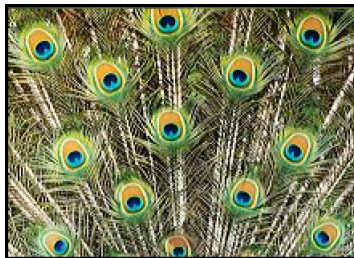


Notice how the plot appears in several different sizes depending on its context. There are three standard automatic plot sizes, the largest if the plot is by itself in an output, smaller if used in a list or table output, and smallest if in a textual input line. This is mainly a convenience to make graphical 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___**]] \Rightarrow **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___**]] \Rightarrow
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.

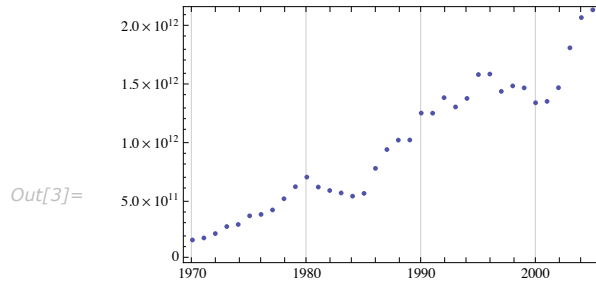
```
In[2]:= OpenerView[{"Plot", Plot[Sin[x], {x, 0, 2 Pi}]}]
Out[2]= ▶ Plot
```

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



▶ Germany

▶ Italy

▶ Japan

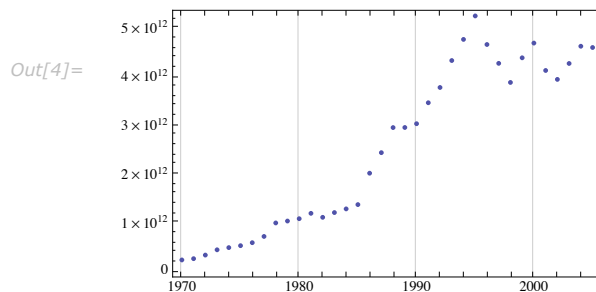
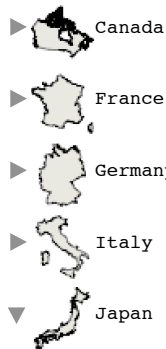
▶ Russia

▶ 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"]]]
```



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
         $\frac{1}{18}$ 
        ▼ Plus
          ▶ Times
          ▶ Times
          ▶ Times
          ▼ Log
            ▼ Plus
              1
              x
              ▼ Power
                x
                2
          ▶ Times
          ▶ Times
          ▶ Times
          ▶ Times
          ▶ Times
```

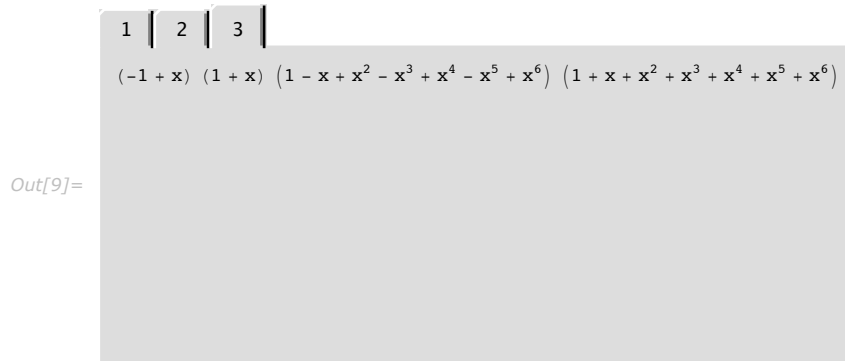
For more information and a detailed listing of options, see `OpenerView`.

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.

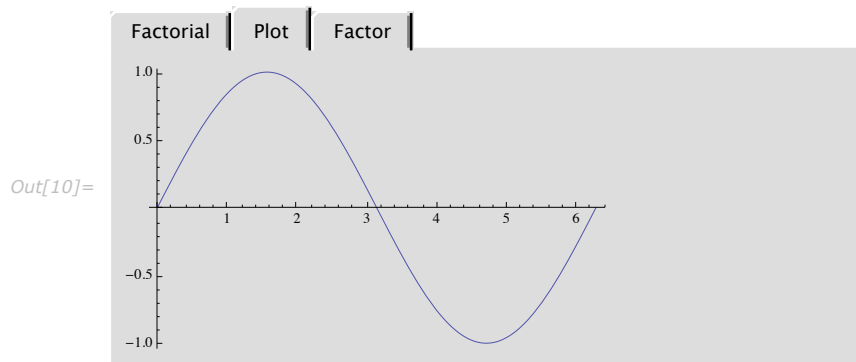
By default, the tabs are numbered sequentially. In the output below, click the tabs to flip between panes.

```
In[9]:= TabView[{40!, Plot[Sin[x], {x, 0, 2 Pi}], Factor[x^14 - 1]}
```



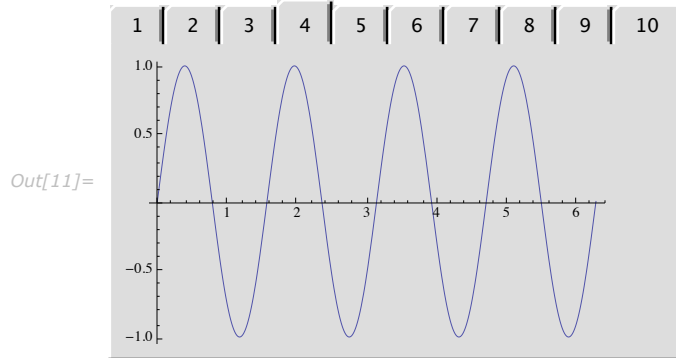
More descriptive tab labels can be added using the form *label* -> *contents*.

```
In[10]:= TabView[{"Factorial" -> 40!,  
"Plot" -> Plot[Sin[x], {x, 0, 2 Pi}], "Factor" -> Factor[x^14 - 1]}
```



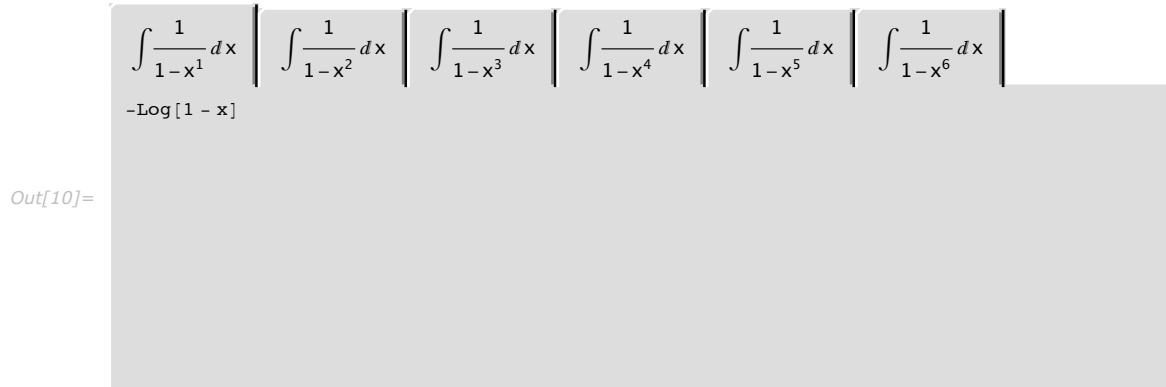
The contents can of course be programmatically generated. Here a `Table` command is used to generate ten different plots.

```
In[11]:= TableView[Table[Plot[Sin[n x], {x, 0, 2 Pi}], {n, 10}]]
```



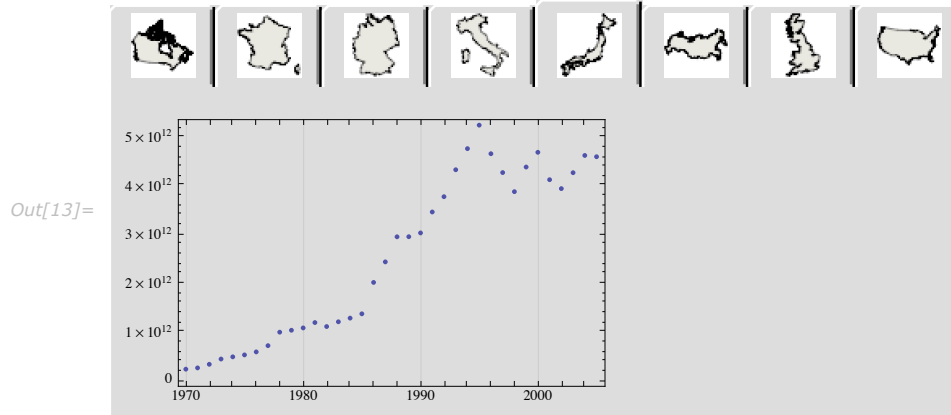
The tab labels are not restricted to simple strings. Here typeset mathematical expressions are used as tab labels.

```
In[10]:= TableView[
  (HoldForm[Integrate[1 / (1 - x^#), x]] → Integrate[1 / (1 - x^#), x]) & /@ Range[6]]
```



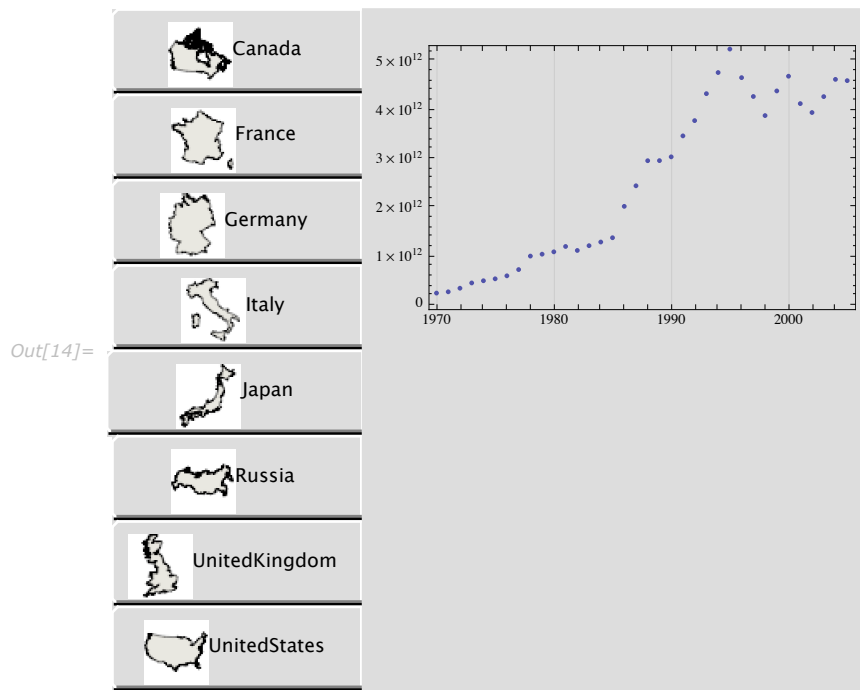
This example uses the shapes of countries as tab labels, with a plot of each country's GDP in its pane.

```
In[13]:= TabView[Rasterize[Show[CountryData[#, "Shape"], ImageSize -> {40, 40}],
  Background -> None] -> DateListPlot[
  CountryData[#, {"GDP"}, {1970, 2005}]] & /@ CountryData["GroupOf8"]]
```



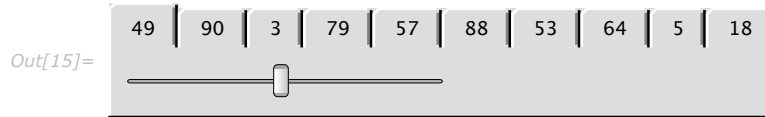
When arranged across the top, tab labels need to be kept reasonably short. The `ControlPlacement` option can be used to move the tabs to any side of the panel.

```
In[14]:= TabView[
  Row[{Rasterize[Show[CountryData[#, "Shape"], ImageSize -> {40, 40}], Background ->
    None], #] -> DateListPlot[CountryData[#, {"GDP"}, {1970, 2005}]] & /@
  CountryData["GroupOf8"], ControlPlacement -> Left]
```



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}]]]
```



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.

Interface | Evaluation | Appearance | System | Parallel | Internet Connectivity | Advanced

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

Out[29]=

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]=

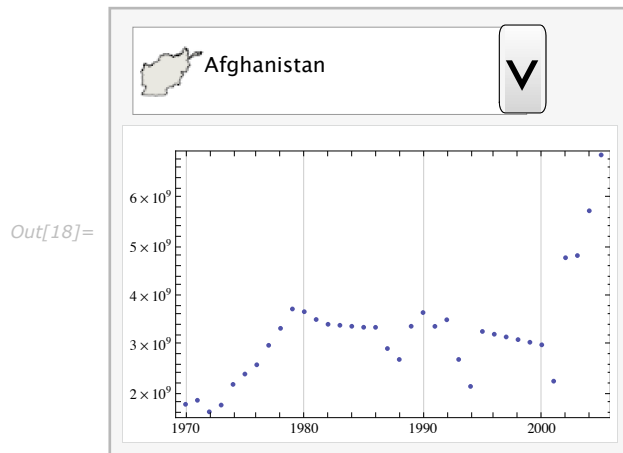
MenuView supports the same *label* → *value* syntax as TabView, allowing you to specify more descriptive labels.

```
In[17]:= MenuView[{"Factorial" → 40!,  
"Plot" → Plot[Sin[x], {x, 0, 2 Pi}], "Factor" → Factor[x^14 - 1]}
```

Out[17]=

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]
```



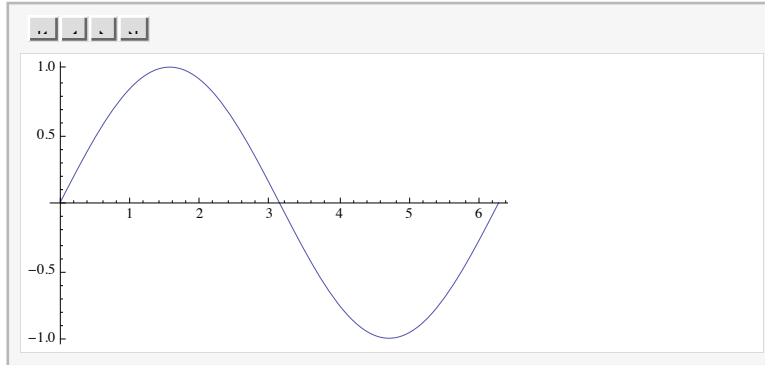
For more information and a detailed listing of options, see `MenuView`.

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]}
```

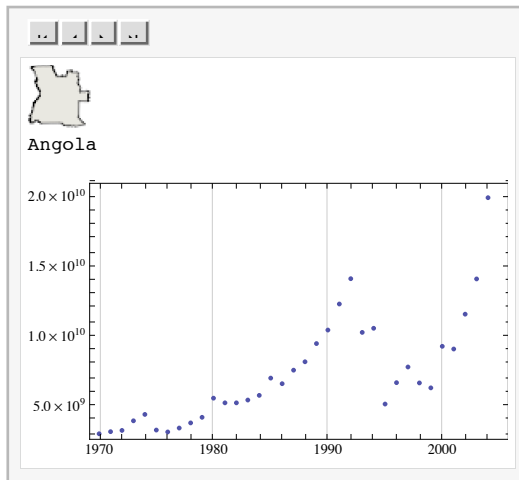
Out[19]=



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.

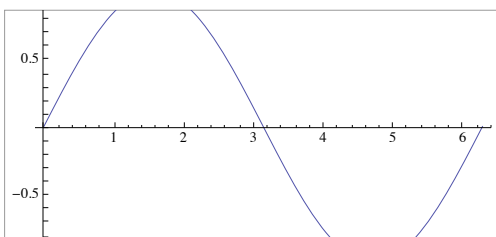

```
In[28]:= PopupView[Table[i!, {i, 20}]]
```

```
Out[28]=  
```

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]]
```

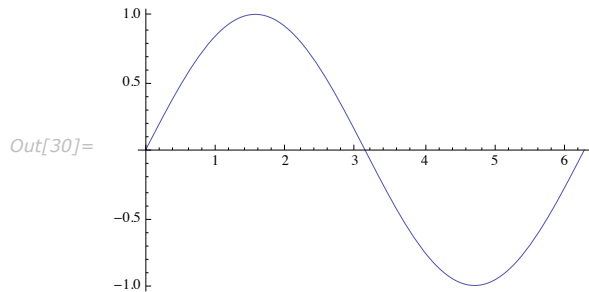
```
Out[29]=  
```

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]}
```



`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]
```

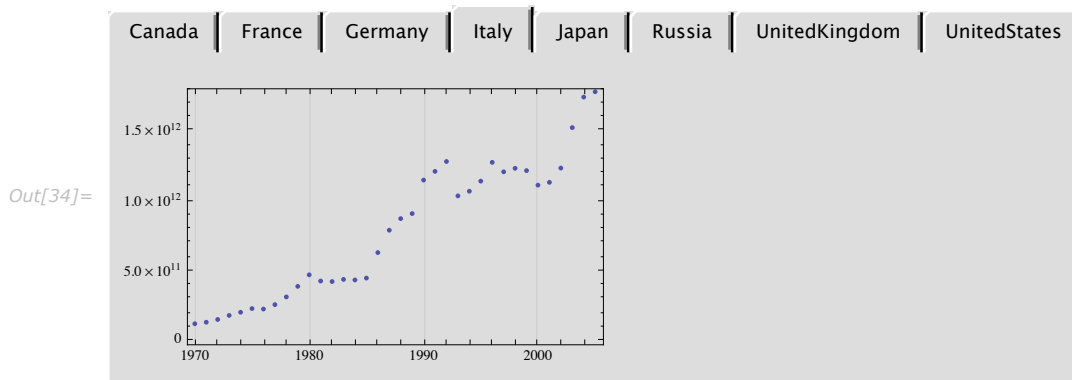
```
Out[32]= ▼ Title
          Contents
```

`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
```

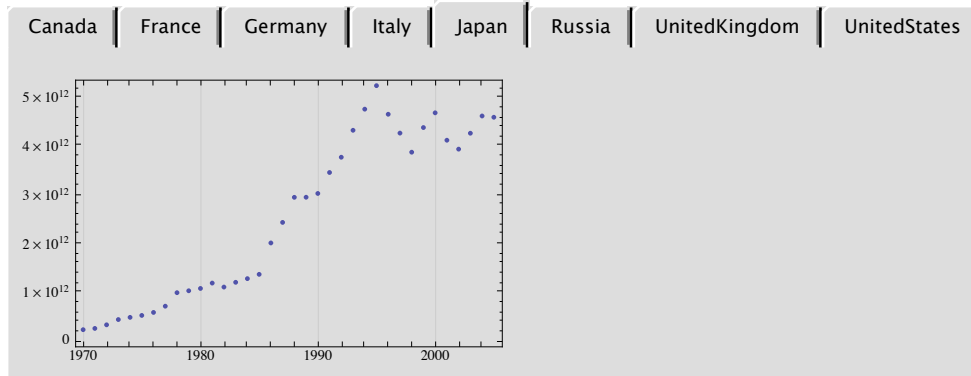
```
In[34]:= TabView[Map[# → DateListPlot[CountryData[#, {"GDP"}, {1970, 2005}]] &,
                  CountryData["GroupOf8"]], 4]
```



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"]
```

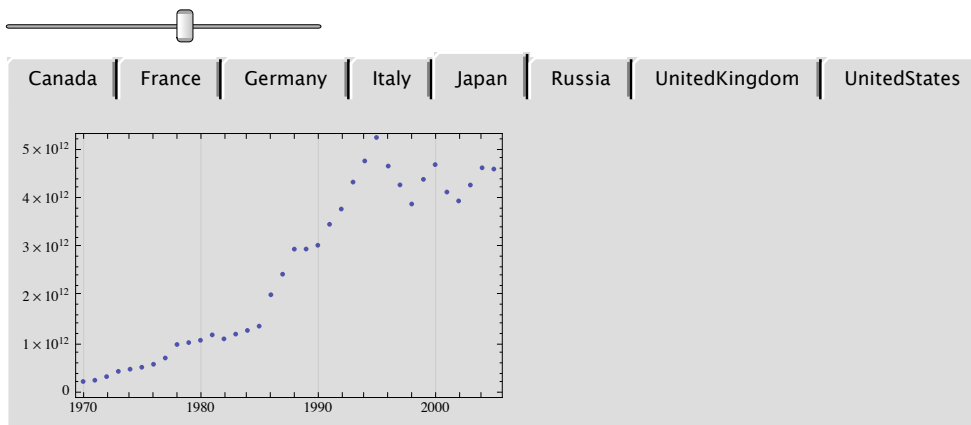
Out[35]=



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.

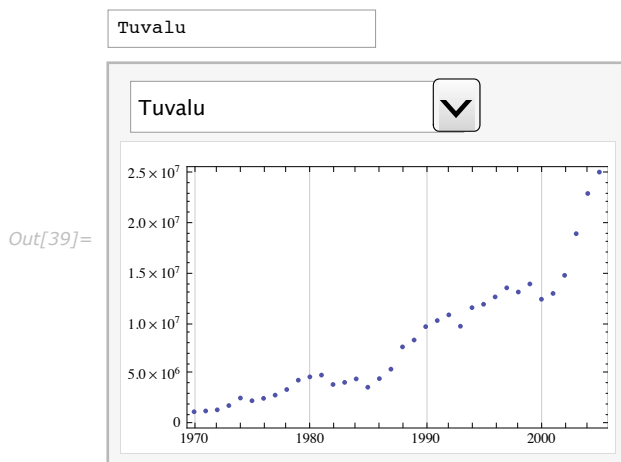
```
In[38]:= DynamicModule[{index = 1},
Column[{Slider[Dynamic[index], {1, 8, 1}],
TabView[Map[# -> DateListPlot[CountryData[#, {"GDP"}, {1970, 2005}]]] &,
CountryData["GroupOf8"], Dynamic[index]]]}]
```

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]]]
```



Controlling whether a View Changes Size

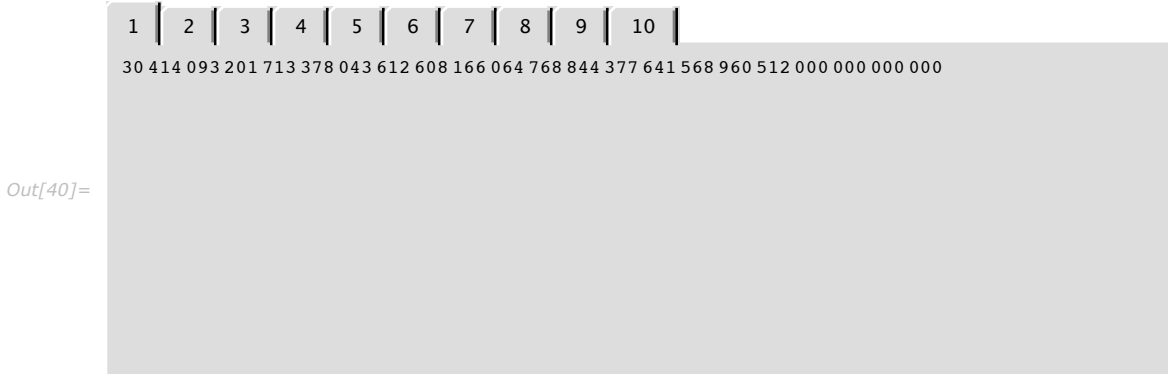
Views always display one of several alternate panes. In determining the overall size of the View, there are two obvious alternatives: make the View big enough to hold the currently displayed pane, or make it big enough so that it never has to change size when switching between panes (i.e., as big as 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.)

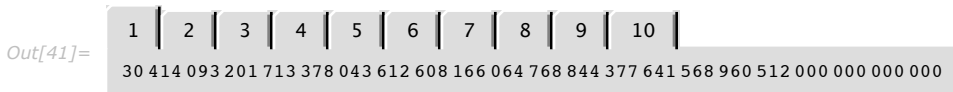
The behavior of any View can be changed using the `ImageSize` option. `ImageSize -> All` means make the View as large as the largest pane, while `ImageSize -> Automatic` means make the View only as large as the currently displayed frame, potentially changing size any time the View is switched to a new 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]
```



```
In[41]:= TabView[Table[(50 i)!, {i, 10}], ImageSize -> Automatic]
```



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").

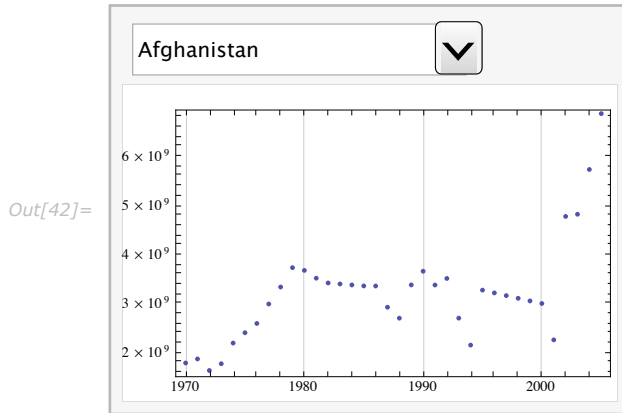
All the View objects fully support `Dynamic` content in any positions where it makes sense. Consider this `MenuView` example.

```
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]
```



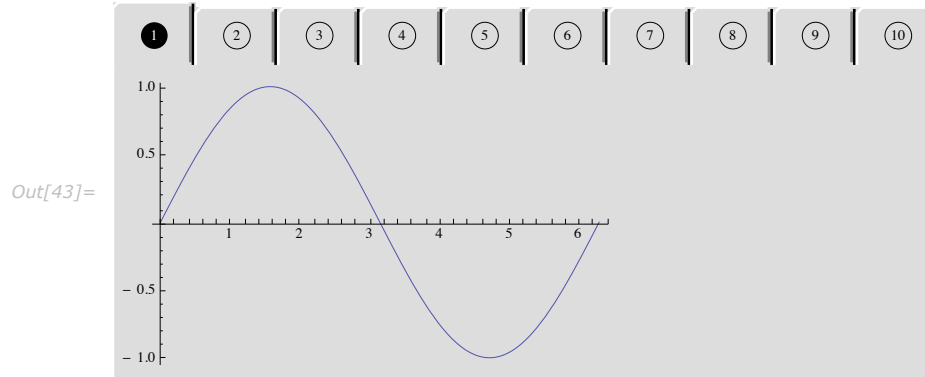
The trade-off is that each new country selected computes the GDP plot on the fly. Fortunately this generally happens so fast as to be unnoticeable. Errors for a particular country are displayed only if that country is selected.

Note that with the setting `ImageSize-> All` (the default for all views except `OpenView` 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]]]
```



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 `TabViews` 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, `PopupMenuView` versus `PopupMenu`, the functions are essentially identical with the arguments reversed.

```
In[45]:= PopupView[{1, 2, 3, 4}, Dynamic[popPosition]]
```

```
Out[45]= 1 
```

```
In[46]:= PopupMenu[Dynamic[popPosition], {1, 2, 3, 4}]
```

```
Out[46]= 1 
```

Why have both? In the case of `PopupMenuView` and `PopupMenu` it is simply for consistency with the other View and Control functions, though there is the convenience that the second argument of `PopupMenuView` 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`.

