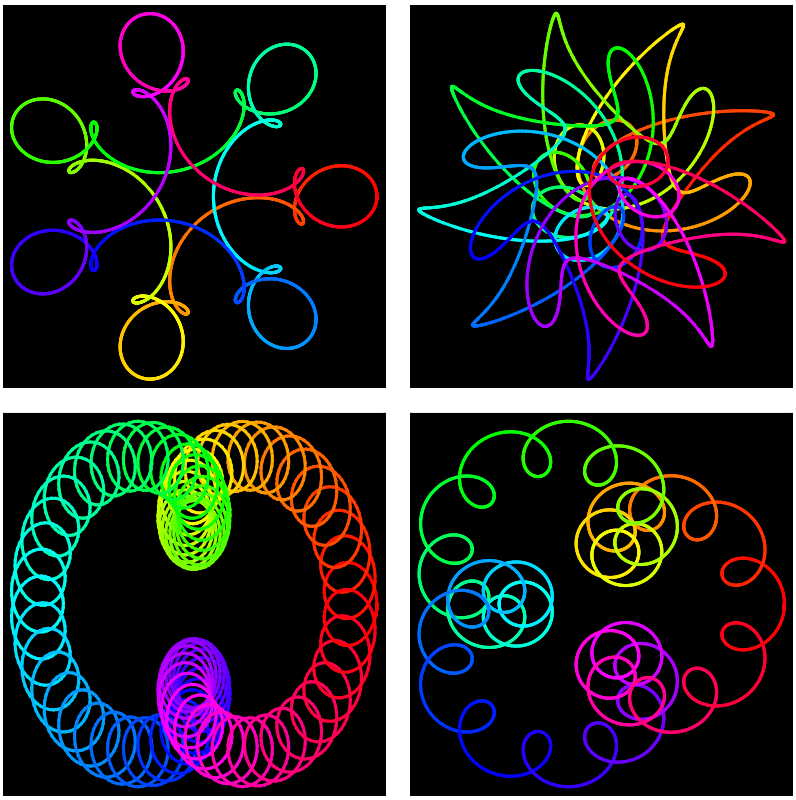
1 Basic Plotting



Placing wheels on wheels on wheels and giving them different rates of spin leads to some interesting parametric plots. The images show four examples. They arise from the values below, clockwise from upper left, as explained in §1.7.

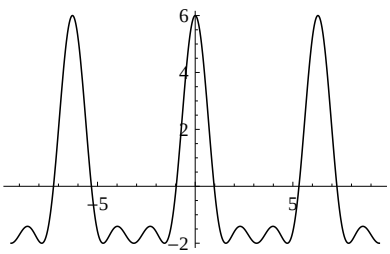
Radii	Speeds	Offsets
$\{1, \frac{1}{2}, \frac{1}{4}\}$	$\{-2, 5, 19\}$	$\{0, 0, 0\}$
$\{1, 0.8, 0.4, 0.2, 0.4, 0.2\}$	$\{1, 10, -17, -26, 28, 37\}$	$\{0, -\frac{\pi}{2}, -\frac{\pi}{2}, 0, 0, \frac{\pi}{2}\}$
$\{1, \frac{1}{2}, \frac{1}{4}\}$	$\{1, 4, 31\}$	$\{0, 0, 0\}$
$\{1, \frac{1}{2}, \frac{1}{4}\}$	$\{1, 3, 80\}$	$\{0, 0, 0\}$

This chapter provides an introduction to the fundamental two-dimensional plotting functions of *Mathematica.* As often happens, even these simple functions can lead to interesting observations about familiar mathematical constructions. Animations can be enlightening and the chapter includes an introduction to the generation of animations using `Manipulate`.

1.1 Plot

The basic plotting command, `Plot`, is simple to use.

```
Plot[3 Cos[x] + 2 Cos[2 x] + Cos[3 x], {x, -3 π, 3 π}]
```



As with all *Mathematica* commands, the output can be highly customized by using options. The great benefit of the option method is that the order in which the options are placed does not matter. There are many options to `Plot`; here are their names. The output below uses boldface for the ones that I feel every user should learn about.

```
Options[Plot][[All, 1]]
```

```
{AlignmentPoint, AspectRatio, Axes, AxesLabel, AxesOrigin,
 AxesStyle, Background, BaselinePosition, BaseStyle,
 ClippingStyle, ColorFunction, ColorFunctionScaling, ColorOutput,
 ContentSelectable, DefaultAxesStyle, DefaultBaseStyle,
 DefaultFrameStyle, DefaultLabelStyle, DisplayFunction, Epilog,
 Evaluated, EvaluationMonitor, Exclusions, ExclusionsStyle,
 Filling, FillingStyle, FormatType, Frame, FrameLabel, FrameStyle,
 FrameTicks, FrameTicksStyle, GridLines, GridLinesStyle,
 ImageMargins, ImagePadding, ImageSize, LabelStyle,
 MaxRecursion, Mesh, MeshFunctions, MeshShading, MeshStyle,
 Method, PerformanceGoal, PlotLabel, PlotPoints, PlotRange,
 PlotRangeClipping, PlotRangePadding, PlotRegion, PlotStyle, Prolog,
 RegionFunction, RotateLabel, Ticks, TicksStyle, WorkingPrecision}
```

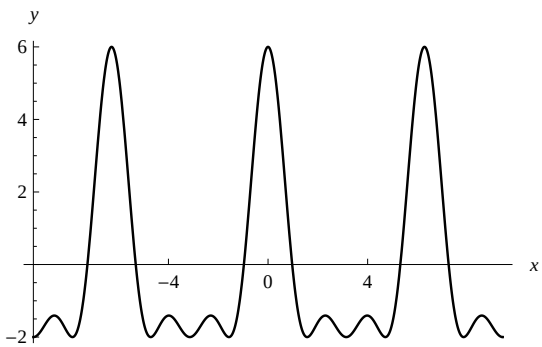
Several of these will be discussed here, but for more information on any of them, start with the usage message, as follows.

```
?PlotRange
```

PlotRange is an option for graphics functions
that specifies what range of coordinates to include in a plot. ≫

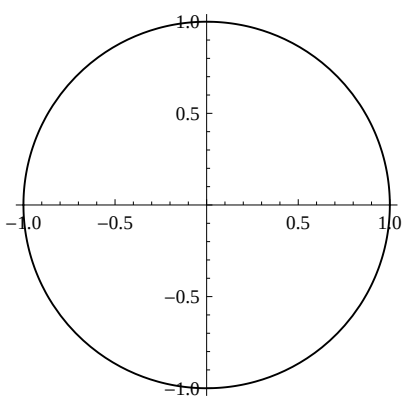
`Plot` uses several internal algorithms that one should become familiar with. First of all, `Plot` tries to determine the region of visual interest and restrict the plotting range to that region. This can be overridden by setting `PlotRange` to `All` or to a specific interval for the vertical range (and also the horizontal range if desired). One can also control the location of the axes origin, label the axes, and so on. When a `Thickness` setting is used, the parameter refers to the proportion of the horizontal span, and so it changes when the image is expanded.

```
Plot[3 Cos[x] + 2 Cos[2 x] + Cos[3 x], {x, -3 π, 3 π},
 AxesOrigin → {-3 π, 0}, Ticks → {{-4, 0, 4}, Automatic},
 AxesLabel → {x, y}, PlotStyle → {Thickness[0.005], Black}]
```



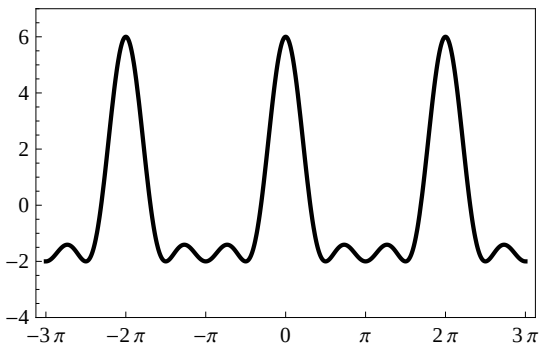
The default aspect ratio is the reciprocal of the golden ratio. Often one wants an aspect ratio that yields visual equality in the scales on the two axes. That is done as follows; without the last option the graph would appear to be an ellipse. Note that in this example we plot two functions.

```
Plot[{-1, 1} √(1 - x^2), {x, -1, 1},
 PlotStyle → {Thickness[0.005], Black}, AspectRatio → Automatic]
```



The axes often interfere with a clear view of the graph. While one can move them by using the AxesOrigin option, it is usually better to remove the axes entirely and add a frame. In the example that follows we do this, and we also use the π character to get the Greek letter in the tick marks. To get the π on screen, type ESC p ESC (or click on π in the BasicMathInput palette). Note that Thick can be used as a style, but it is an absolute setting, and does not change as the graphic is resized. The syntax of FrameTicks was changed in version 6 from {bottom, left, top, right} to {{left, right}, {bottom, top}}; both work.

```
Plot[3 Cos[x] + 2 Cos[2 x] + Cos[3 x],
 {x, -3 π, 3 π}, PlotRange → {-4, 7}, Frame → True,
 FrameTicks → {{Automatic, None}, {Range[-3 π, 3 π, π], None}},
 Axes → None, PlotStyle → {Thick, Black}]
```



Frames are so nice that we now make them the default for all the plotting functions we will discuss in the rest of this chapter, and also change some styles.

```
SetOptions[{Plot, ListPlot, ParametricPlot, PolarPlot, ListLinePlot},
  Frame → True, Axes → None,
  FrameTicks → {{Automatic, None}, {Automatic, None}}];
SetOptions[{Plot, ParametricPlot, PolarPlot, ListPlot, ListLinePlot},
  PlotStyle → {{Thick, Black}}];
```

1.2 An ArcSin Curiosity

It is often convenient to define the function one is interested in, and that is done simply as follows.

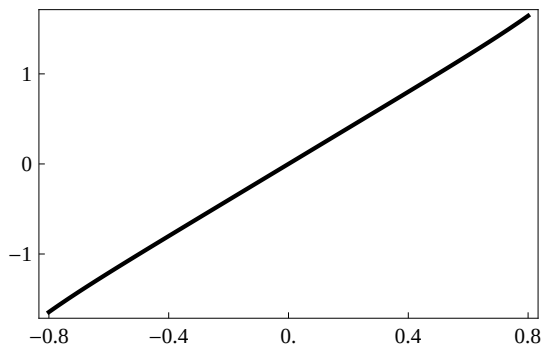
```
f[x_] := Sin[x] + ArcSin[x]
```

The syntax here is as follows: x_ means that f can apply to anything, which will be given the temporary name x. The := means that this is a delayed assignment, and the right side is not to be looked at until f is actually called. It is natural to wonder whether something like f[x_] = x + 3 would work.

Sometimes it would, but if `x` had a prior assignment to, say, 17, then `f` would return 20 for all values of its argument. The reason is that the = takes effect immediately, so the rule becomes, essentially, `f[anything] = 20`. So always use delayed assignments when defining functions (and don't use them when a simple assignment, such as `a = 5`, will do).

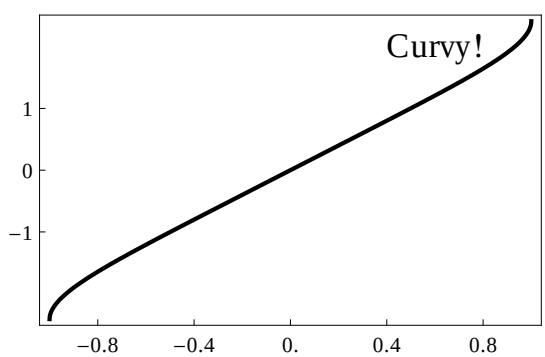
Now here is a very simple plot: the sum of sine and arcsine.

```
Plot[f[x], {x, -0.8, 0.8},
 FrameTicks → {{{-1, 0, 1}, None}, {Range[-0.8, 0.8, 0.4], None}}]
```



This is surely too simple! Is it correct? Can the graph of this function really be just a straight line? Of course not, as a plot from −1 to 1 will show. But it is remarkable how straight the graph is between −0.8 and 0.8. This example, which was pointed out to me by John Schue, becomes much clearer if we examine the Taylor series of the function being graphed. The next graph shows the full domain; `Epilog` and `Text` have been used to add text in a specific size.

```
Plot[f[x], {x, -1, 1},
 FrameTicks → {{{-1, 0, 1}, None}, {Range[-0.8, 0.8, 0.4], None}},
 Epilog → Text[Style["Curvy!", FontSize → 16], {0.6, 2.}]]
```



And here is the Taylor series about 0. The 0 indicates that the series is centered about 0 (such series are often called Maclaurin series) and the 14 specifies the highest power sought.

```
ser = Series[f[x], {x, 0, 14}]
```

$$2x + \frac{x^5}{12} + \frac{2x^7}{45} + \frac{5513x^9}{181\,440} + \frac{2537x^{11}}{113\,400} + \frac{4\,156\,001x^{13}}{239\,500\,800} + O[x]^{15}$$

The `Series` command gives its output in a special form, with a big-Oh error term. If only the partial sum of the series is wanted, then use `Normal`.

```
Normal[ser]
```

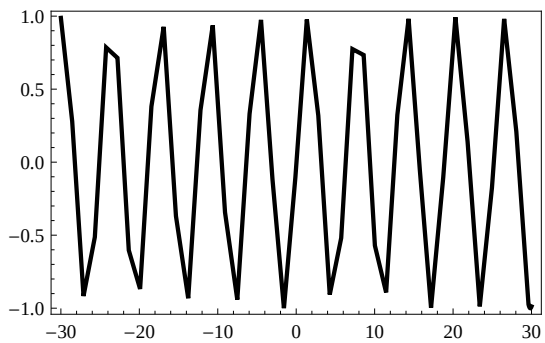
$$2x + \frac{x^5}{12} + \frac{2x^7}{45} + \frac{5513x^9}{181\,440} + \frac{2537x^{11}}{113\,400} + \frac{4\,156\,001x^{13}}{239\,500\,800}$$

In any case, it is now clear what is happening. There is a coincidental cancellation of the third-degree terms when sine and arcsine are added, and the function is close to being linear. The contribution of the fifth- and higher-order terms, whose coefficients are rather small on the interval $[-0.85,\ 0.85]$, is very small.

1.3 Adaptive Plotting

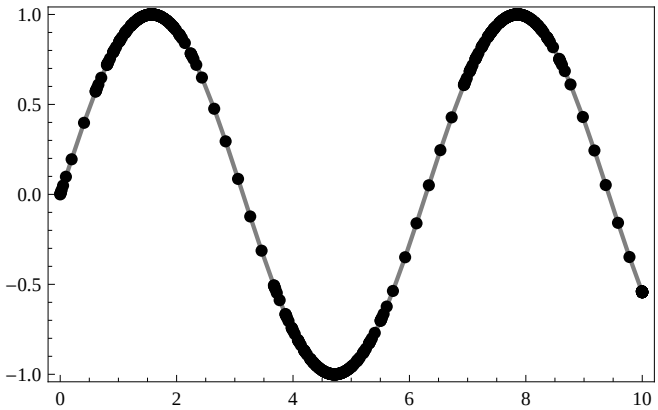
The general idea underlying the basic plotting algorithm is as follows. The function is evaluated at 50 uniformly spaced *x*-values and successive line segments are examined. If the angle between two consecutive segments is less than 5°, the algorithm is happy and moves on; if not, it subdivides (the maximum number of subdivisions is controlled by the `MaxRecursion` option setting) until the angle criterion is reached. The following example shows how a too-loose setting can lead to an inaccurate plot.

```
Plot[Sin[x], {x, -30, 30}, PlotPoints → 6, MaxRecursion → 3]
```



In order to see this adaptive plotting mechanism in action we can use the `Mesh → All` setting. The `Mesh` option is more commonly used for meshes sitting on surfaces, but it works in this context as well, and defines points on the graph.

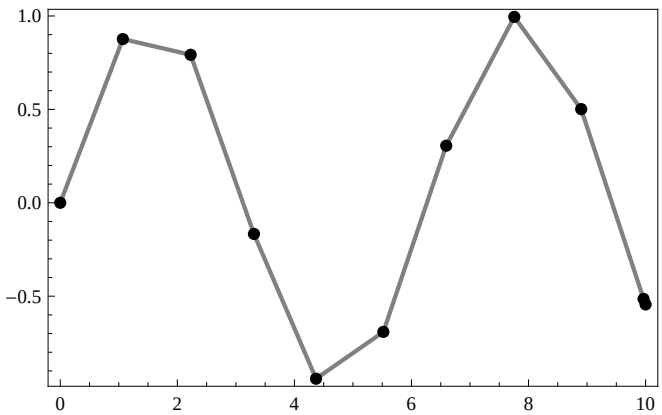
```
Plot[Sin[x], {x, 0, 10}, Mesh → All,
 MeshStyle → PointSize[0.02], PlotStyle → {Thick, Gray}]
```



One can see that more points were examined in regions where the graph bends quickly.

Here is one way knowledge of some of these internals can be used to advantage. Suppose one has a function that takes a long time to compute: perhaps it comes from a differential equation. One wants to visualize, say, 10 values. One can make a table of values and use `ListPlot`, to be discussed shortly. But one can also just use `Plot` and its options to make sure that only 10 points are plotted. There is a slight bug here as two points are plotted at the right end.

```
Plot[Sin[x], {x, 0, 10}, Mesh → All, MeshStyle → PointSize[0.02],
 PlotStyle → {Thick, Gray}, PlotPoints → 10, MaxRecursion → 0]
```



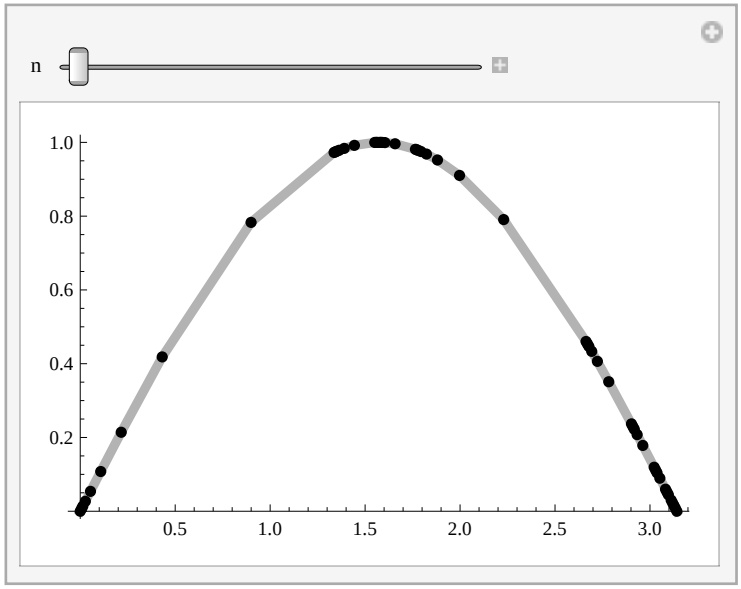
The `Reap` and `Sow` mechanism show the actual points used. Here a semicolon is used after the `Plot` command to suppress the graphic output.

```
Reap[p = Plot[Sin[x], {x, 0, 10}, PlotPoints → 10,
    MaxRecursion → 0, EvaluationMonitor :→ Sow[x]];]
```

```
{Null, {{1.11111 × 10^-6, 1.06867, 2.22723, 3.30902,
   4.36958, 5.52004, 6.59372, 7.75729, 8.89964, 9.96521, 10.}}}
```

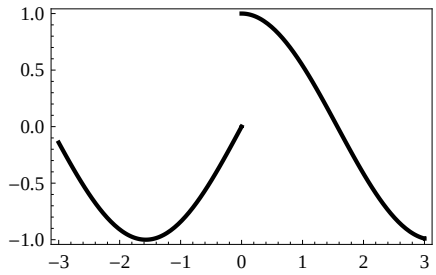
One usually will not have to fuss with these parameters. There is an option called `PerformanceGoal` that one can set to either `"Quality"` or `"Speed"`. The following manipulation shows how one can get more precise control over the method (here we vary the angle between segments that stops the subdivision process), and also is our first example of how `Manipulate` can be used to create an animation. In general we will omit the output of `Manipulate` since it must be manipulated in the live window to be appreciated. In the output of a `Manipulate` one can move the slider to change the value of `n`, or get more options by clicking on the small icon to the right of the slider.

```
Manipulate[Plot[Sin[x], {x, 0, π},
  Method → {Refinement → {ControlValue → (90 - n) °}}, Mesh → All,
  MeshStyle → {Red, PointSize[Large]}, PlotPoints → 8], {n, 0, 90, 2}]
```



Often one wants to combine different types of plots, and this can be done with `Show`. Note that the intermediate plots are not generated; there is no need to use the `DisplayFunction` option to suppress the plots.

```
Show[{Plot[Sin[x], {x, -3, 0}], Plot[Cos[x], {x, 0, 3}]},
 PlotRange → All]
```



In some advanced work one might find it useful to have the actual points that make up the plot. For example, one might want to use the curve, or part of the curve, in another graphics construction and just using the plot itself is not flexible enough.

We showed above how Sow and Reap can be used to get the points, but if one already has the plot in hand (p was defined above) one can grab them from that object as follows.

```
Short[InputForm[p]]
```

```
Graphics[{{{}, {}, {Hue[0.67, 0.6,
  0.6], <<2>>, Line[{{1.111111111111111*^-6,
  <<1>>}, <<9>>, {<<2>>}}]}}}, {<<8>>}]
```

```
Short[Cases[p, _Line, ∞]]
```

$$\{\text{Line}[\{\{1.11111\times10^{-6}, 1.11111\times10^{-6}\}, \ll9\gg, \{10., -0.54402\}\}]\}$$

Here is how one can efficiently turn the Line object to a list. Be aware that the plot might have several Line objects.

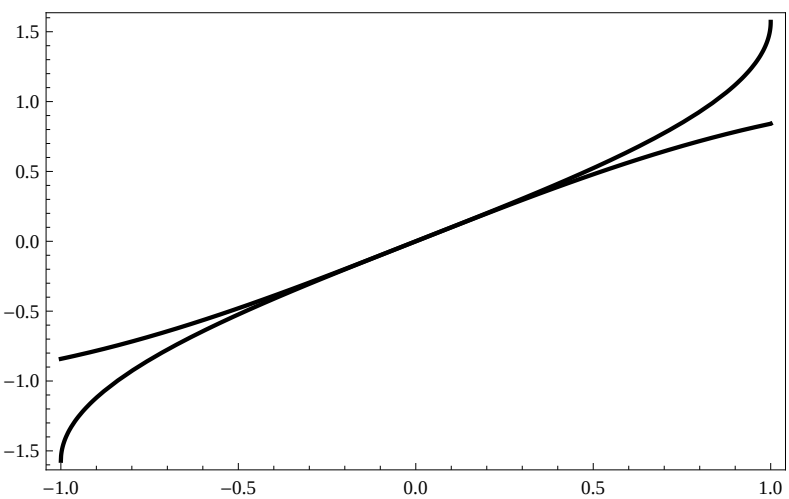
```
Short[Cases[p, Line[x_] :> x, ∞]]
```

$$\{\{\{1.11111\times10^{-6}, 1.11111\times10^{-6}\}, \ll9\gg, \{10., -0.54402\}\}\}$$

1.4 Plotting Tables and Tabling Plots

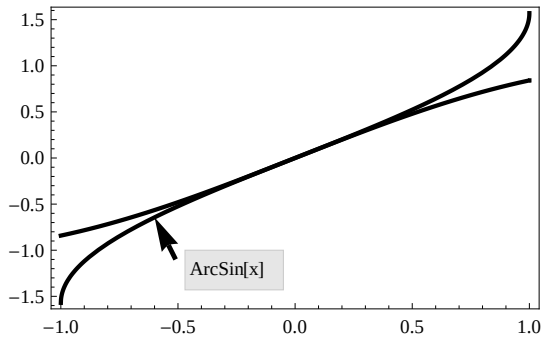
One often wants to plot several functions at once. If the functions are given explicitly, it is easy.

```
Plot[{Sin[x], ArcSin[x]}, {x, -1, 1}]
```



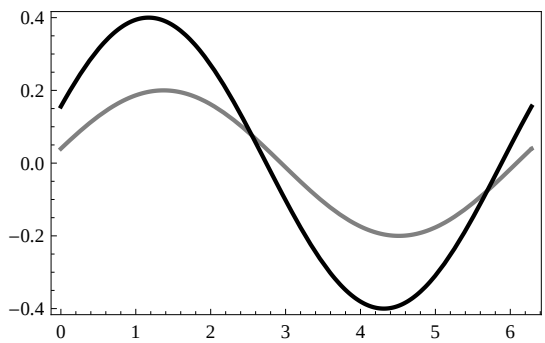
If instead one adds a Tooltip wrapper, as follows, then the result is such that when the mouse passes over the graph the name of the graph is shown.

```
Plot[Tooltip[{Sin[x], ArcSin[x]}], {x, -1, 1}]
```



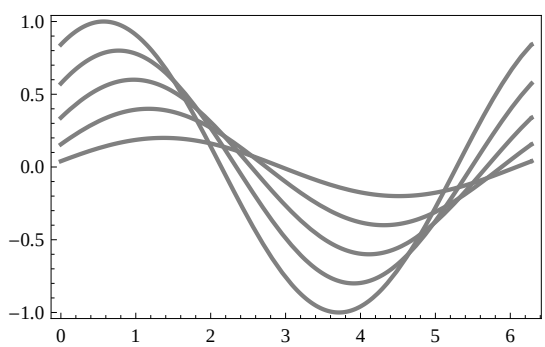
`Plot` considers a list generated by a `Table` command to be a little different than an actual list given explicitly as, say, `{f1, f2}`. The former is viewed as a function from $\mathbb{R}$ to $\mathbb{R}^n$. The latter is viewed as a list of distinct single-variable functions. From a pure plotting perspective the difference is invisible. But it can show up in some of the options. In the next command the `Evaluate` wrapper means that the table is evaluated into a list of two functions; then one is shown gray and the other black.

```
Plot[Evaluate[Table[c Sin[x + c], {c, 0.2, 0.4, 0.2}]],
 {x, 0, 2 π}, PlotStyle → {{Gray, Thick}, {Black, Thick}}]
```



But without that option, the function being plotted is viewed as a single function from the reals to the plane, and so it has only one style; the second style directive is ignored.

```
Plot[Table[c Sin[x + c], {c, 0.2, 1, 0.2}],
 {x, 0, 2 π}, PlotStyle → {{Gray, Thick}, {Black, Thick}}]
```



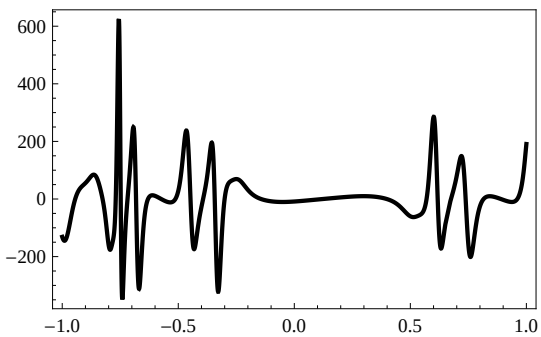
An important use of forced evaluation comes when plotting derivatives. The naive approach leads to an error message.

```
Plot[D[x^2, x], {x, 0, 1}]
```

General::ivar : 0.000020428571428571424` is not a valid variable. ≫

The logic here is that `Plot` tries a value of *x*, say 0.123, and tries to compute `D[0.123^2, 0.123]`. This cannot be done because 0.123 is not a variable. And even if it could be done, the function being differentiated has become a constant! The way around this is to force evaluation of the plotting function using `Evaluate`. Here is a more complicated example where we iterate a function several times and take the derivative. Forcing the evaluation guarantees that these two algebraic steps are done only once.

```
f[x_] := Sin[x] + 3 e^Cos[x]
Plot[Evaluate[∂x Nest[f, x, 4]], {x, -1, 1}, PlotRange → All]
```



Recall that `Nest[f, x, n]` iterates `f`, with starting value `x`, `n` times, while `NestList` gives the full set of iterates.

```
Clear[f]; Nest[f, x, 4]
```

```
f[f[f[f[x]]]]
```

```
NestList[f, x, 4]
```

```
{x, f[x], f[f[x]], f[f[f[x]]], f[f[f[f[x]]]]}
```

```
NestList[Cos, 0.8, 15]
```

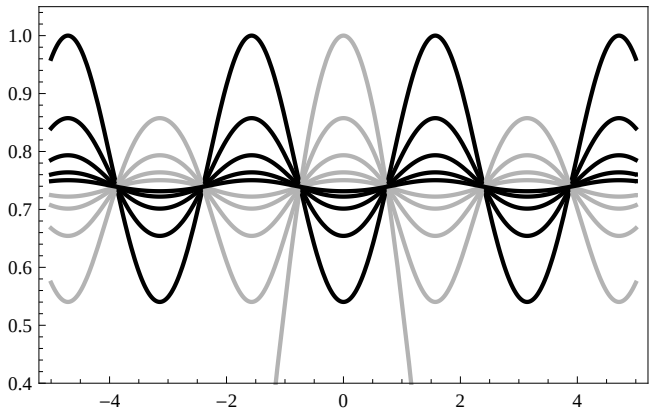
```
{0.8, 0.696707, 0.76696, 0.720024, 0.75179,
 0.730468, 0.744863, 0.735181, 0.741709, 0.737315,
 0.740276, 0.738282, 0.739626, 0.738721, 0.73933, 0.73892}
```

```
Nest[Cos, 0.8, 100]
```

```
0.739085
```

We can plot a family of iterates as follows; `Evaluate` is used to get the styles (gray for odd, black for even) to work. And we add a tooltip so that the mouse shows how many iterates generated each graph.

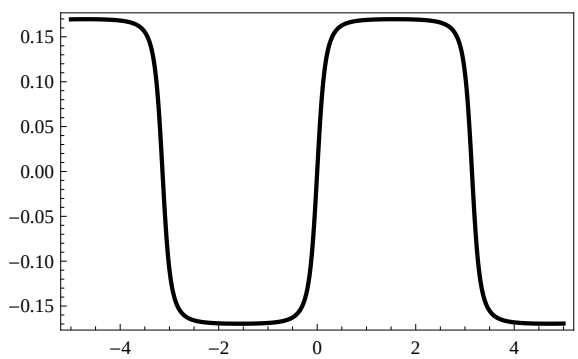
```
Plot[Evaluate[Table[Tooltip[Nest[Cos, x, i], i], {i, 10}]],
 {x, -5, 5}, PlotRange → {0.4, 1.05},
 PlotStyle → Table[{Thick, GrayLevel[If[EvenQ[i], 0, 0.7]]}, {i, 10}]]
```



The wavy behavior is due to the fact that the speed of convergence relates to the size of the derivative, and that varies. But the derivative is never greater than 1, and it is not hard to show (with the help of the mean-value theorem) that iterating the cosine on any starting value leads to the fixed point $p = 0.739085\ldots$, where $\cos p = p$.

The sine function behaves somewhat differently. Iterates converge to 0, a fixed point, but very slowly. Moreover, the graphs of the iterates have a surprising square-wave behavior when one normalizes. For more details on this example, see [GG].

```
Plot[Nest[Sin, x, 100], {x, -5, 5}]
```

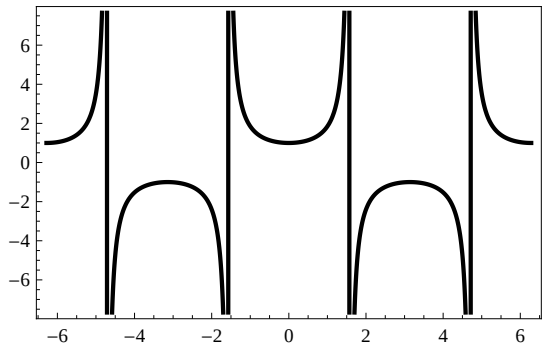


1.5 Dealing with Discontinuities

`Plot` generally assumes that the functions being plotted are continuous, and so the lines one gets when plotting, for example, $\sec x$, may look like asymptotes but really are connecting segments on what is believed to be the graph. That is, the adaptive

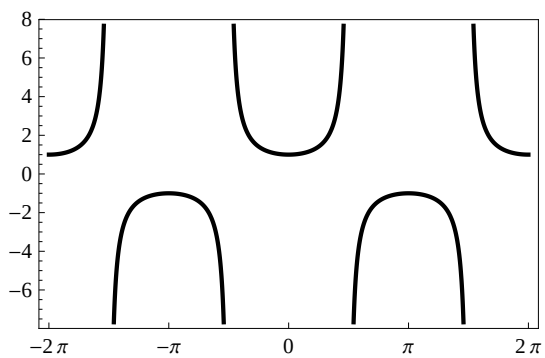
routine subdivided several times to try to get a smooth curve, but then gave up. Add the `PlotRange → All` option to this command to see the whole story.

```
Plot[Sec[x], {x, -2 π, 2 π}]
```



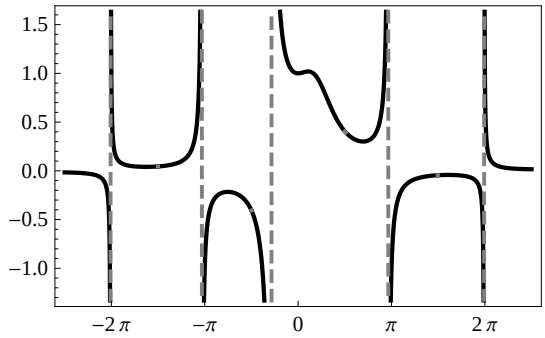
However, a new option called `Exclusions` allows one to specify points to be excluded. So if one knows the discontinuities, one can avoid them.

```
Plot[Sec[x], {x, -2 π, 2 π}, Exclusions → Range[-3 π/2, 3 π/2, π],
 FrameTicks → {{Automatic, None}, {Range[-2 π, 2 π, π], None}}]
```



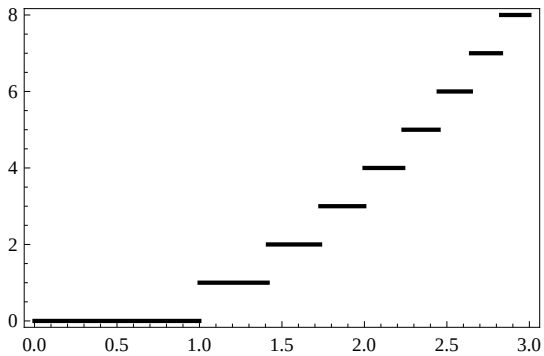
We can specify the exclusions by equations, so that the discontinuities can be found automatically. We can also specify styles for the excluded values, which is a way of bringing asymptotes into the picture. While `ExclusionsStyle → Red` or `ExclusionsStyle → {Red}` work fine, if one has several style items, one needs an extra layer of nesting.

```
Plot[Sec[x]/(1 + x^2 Tan[x]), {x, -5 π/2, 5 π/2},
 Exclusions → {Cos[x] == 0, 1 + x^2 Tan[x] == 0},
 ExclusionsStyle → {{Dashing[0.02], Gray, Thickness[0.008]}},
 FrameTicks → {{Automatic, None}, {Range[-2 π, 2 π, π], None}}]
```



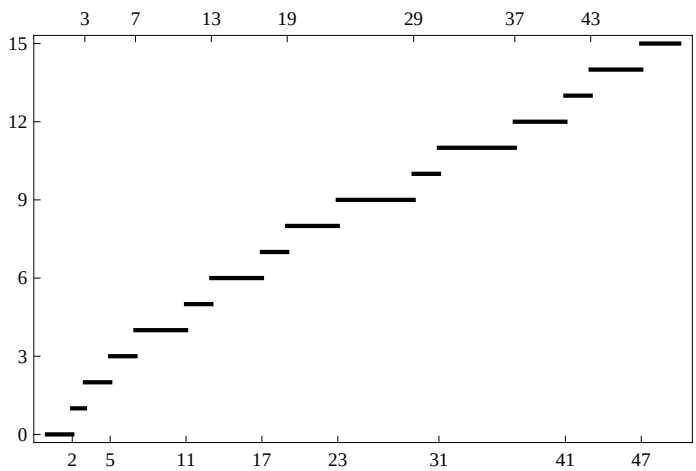
Sometimes `Plot` can discover the correct points to exclude, as with the following two examples.

```
Plot[Floor[x^2], {x, 0, 3}]
```



Sometimes the piecewise recognition requires some help, as in the following example (`PrimePi[x]` gives the number of primes less than or equal to *x*).

```
Plot[PrimePi[x], {x, 0, 50}, "SymbolicPiecewiseSubdivision" → True]
```



In this case, it is simpler to just specify the discontinuities by adding the option `Exclusions → Range[50]`.