Mathematica Tips, Tricks, and Techniques

Two-Dimensional Graphics

Michael A. Morrison

(Version 3.3: February 2, 2000)

> [N]ow look up at any office building, look into living-room windows at night: so many people sitting alone in front of monitors. We lead machine-centered lives; now everyone's life is full of automated tellers, portable phones, pages, keyboards, mice. We live in a contest of the fittest, where the most knowledgeable and skillful win and the rest are discarded; and this is the working life that waits for everybody. Everyone agrees: be a knowledge worker or be left behind.
>
> —*Close to the Machine*,
> by Ellen Ullman

Contents

1 Common practical problems using 2–D graphics. (Everybody)

1.1 I want to use ListPlot to plot pairs of numbers (x, y). I now have all the x values in one list and all the y values in another. What do I do?

Use Transpose to construct a list of data to be plotted.

This trick is easy to understand once you understand that ListPlot wants *a list of lists, where each inner sublist contains one pair of numbers to be plotted.* For instance, suppose you want to plot some experimental data which you've entered in two lists

```
xData = {15, 22.5, 30}
yData = {22 , 27, 33}
```

To use Transpose, you construct a new list whose elements are the two sublists xData and yData. We then feed our new list to Transpose. *Mathematica* returns a list in the form ListPlot needs:

```
dataToPlot = Transpose[{xData,yData}]
{{15,22},{22.5,27},{30,33}}

ListPlot[dataToPlot];
```

We can more clearly show the sequence of steps involved in creating the new list by using the **piping** syntax,

```
dataToPlot = {xData,yData} // Transpose
{{15,22},{22.5,27},{30,33}}
```

Tip If you give ListPlot only one list, it assumes that the variable you want plotted on the horizontal axis varies in unit steps from a minimum value of 1.

Warning The command Transpose will gripe if xData and yData contain different number of elements. You can check this by feeding each sublist to the command Length.

Warning Watch your syntax! The command Transpose will gripe (a lot) if you forget to enclose the sublists in an outer lists; that is, don't enter Transpose[xData,yData] unless you want to see new irritating error messages.

1.2 Rather than plot two curves on the same graph, I want to superimpose two *graphs* on top of one another. How do I do this?

Name your plots. Then use Show.

A very common situation in using *Mathematica* to do physics is the following. You've generated a nice graph—say of some interesting data. Elsewhere in your notebook you've generated another graph related to the first—say of a function you think may explain the data. Now you want to compare the two. *Mathematica* offers several ways to do this. The simplest is to use the command Show, which "shows" its arguments one atop the other.[1] If you've named your data plot something imaginative like dataPlot and your function plot functionPlot, then all you have to do to see them superimposed atop one another is to enter

Show[{dataPlot,functionPlot}].

Tip The trick to efficient use of Show is to assign a name to each of your graphs. Then you can just feed Show the names you assigned and it will do the rest.

Tip You don't have to enclose the sequence of plots you want superimposed in curly brackets, as I did in this example. But I recommend doing so. It does no harm, and it conforms to the general *Mathematica* syntax rule that when you feed a built-in function multiple values for a single argument, you enclose those values in braces—i.e., you use a list.

[1] It's arguments must be what *Mathematica* calls **graphics objects**. That is, you can't Show an integral on top of a sum.

Warning When *Mathematica* superimposes two (or more) plots, it takes the axes labels, plot labels, legends, special ticks, and other formatting information from the *first* plot in the argument list. So if you've created beautiful labels for one of the individual plots you want to superimpose, *be sure the name of the plot with the labels comes first in the argument list to* `Show`.

In the above example, if I have carefully labelled the axes and the plot for `dataPlot` but not for `functionPlot` and I enter

```
Show[{functionPlot,dataPlot}];
```

Mathematica will correctly superimpose the plots but the output won't have any of my labels. The reason is that *Mathematica* took the formatting information from `functionPlot`, which in this example has no labels.[2]

1.3 I want to use `Show` to superimpose two graphs, but I don't want *Mathematica* to show me the individual graphs. Can I suppress the output of the individual plot statements?

Sure! Use the `DisplayFunction` option.

Once you get the hang of `Show`, you'll be using it a lot. Sure as shootin', you'll encounter the following situation. You know in advance that you want to superimpose several graphs. But the individual graphs aren't interesting; you want *Mathematica* to show you *only* composite graph.

There are two steps to accomplishing this. Both are achieved with the **option** `DisplayFunction`, which is accepted by all *Mathematica* graphics commands. Essentially, `DisplayFunction` tells *Mathematica* where to show the graph it constructs as a result of your graphics command. Like all options, it has the form of a **replacement rule**. All you need to know now about the `DisplayFunction` option is two possible values; each value accomplishes one step for you.

1. Tell *Mathematica* to *suppress the graphical output from the commands that create the individual graphs.* To do this, give the option `DisplayFunction->Identity` to each of these commands.

2. Tell *Mathematica* to *show you the composite graph it construct by superimposing the individual graphs.* To do this, give `Show` the option `DisplayFunction->$DisplayFunction` to `Show`, after the list of names of plots you want to superimpose.

Tip Always conclude your graphics commands with a semicolon *after the last square bracket.* Doing so will keep *Mathematica* from cluttering up your notebook with useless output lines like `-Graphics-`.

Warning Don't forget the dollar sign on the value of `DisplayFunction`. This quantity is what *Mathematica* calls a **global variable**, and the names of all global variables begin with a dollar sign.

[2] This description is a little imprecise. What *Mathematica* actually does is to *literally* superimpose the arguments atop one another in such a way that the first graph is on top of the second is on top of the third, etc. So when it superimposed `functionPlot` on top of `dataPlot`, the (blank) axes and plot labels for `functionPlot` covered up my elegant labels for `dataPlot`.

Here's the sequence of commands that creates *only* a composite plot for the example in Question 1.2. Note that I've named the function to which I want to compare the data `f[x]`; the definition of this function isn't shown here.

```
dataPlot = ListPlot[dataToPlot,
           DisplayFunction->Identity];

functionPlot = Plot[f[x],
               {x, 0, 30},
           DisplayFunction->Identity];

Show[{functionPlot,dataPlot},
     DisplayFunction->$DisplayFunction];
```

2 Elementary Formatting. (Everybody)

2.1 How can I make *Mathematica* draw a nice little frame around my plots?

Use the option `Frame->True`.

Warning If you use this option, then you can't use `AxesLabel` to label your axes. Instead, you must use `FrameLabel`.

2.2 I'm plotting several curves, but I can't tell one from another! How can I make *Mathematica* use different line types for different curves on the same graph?

Use the `PlotStyle` option.

This option lets you specify the format of each curve generated when you ask one of the *Mathematica* plotting commands to graph several functions. If you don't specify this option, then *Mathematica* will draw all your curves with a solid black line. With `PlotStyle`, you can make lines different colors, dashed or dotted, of different thicknesses, etc. Like all options, we specify `PlotStyle` as a replacement rule. *The right hand side of this particular rule must be a list.*

For instance, the following command generates a plot of $\sin(x)$ as a red dashed curve.

```
Plot[ Sin[x],
      {x, 0, Pi},
      PlotStyle->{ Red, Dashing[{0.02}] }];
```

Warning To specify colors by name, as in the above example, you must load the package `Graphics‘Master‘` in the bookkeeping section of your notebook!

Tip Always use `PlotStyle` to ensure that each curve on your graph is clearly distinguished from the others.

Warning The only mildly tricky thing about `PlotStyle` is the syntax used to specify dashed lines. The command `Dashing` takes one or two arguments; *in either case, you must enclose the argument(s)—even if there is only one—in curly brackets. That is, you must alwasy feed `Dashing` a list.*

To control the line types in a graph that consists of several curves, we just include different formatting commands in a list on the right hand side of `PlotSytle`. Specifically, we include formatting commands in sublists on the right hand side. Each sublists contains the formatting commands we want *Mathematica* to use for a single curve. *Mathematica* will use the first style commands in the first sublist to plot the first curve in the function list, the second style commands for the second curve, and so forth. If you want one of the curves drawn in *Mathematica*'s default style, just include an empty sublist `{}` in the appropriate slot in the `PlotStyle` list.

Warning If you give *Mathematica* more curves than you do sublists, it will cycle through your sublists. Hence *Mathematica* will plot more than one curve with the same set of formatting commands—resulting in a

confusing plot in which two or more curves look alike. If, however, you give more style sublists than you do curves, *Mathematica* will just ignore the extra sublists.

All this is easily understood by example. Let's extend the above example to show three curves.

```
Plot[ {Sin[x], Cos[x], x^2}, {x, 0, Pi},
      PlotStyle->{
         {},
         {Red, Thickness[0.01], Dashing[{0.02}]},
         {Blue, Dashing[{0.01}]}
      }];
```

In this example, the list of style commands specified with the `PlotStyle` option instructs *Mathematica* to proceed as follows. Plot $\sin x$ using its default style (a solid black line). Plot $\cos x$ using a thick red medium-dashed curve. Plot x^2 using a blue small-dashed curve.

Tip Colors are neat, but don't use only colors to distinguish line types. Colors, obviously, print in color only on a color printer. You never know when you'll want to print output on a black-and-white printer. So whether or not you use colors, get in the habit of distinguishing curves using specifying different thickness or dashing forms.

2.3 *Mathematica* insists on putting a vertical line near the left vertical axis of my plot. How do I get rid of this?

Use the `PlotRange` option to control the limits of the horizontal axes.

This jolly little artifact of *Mathematica*'s awesome graphics commands is a nuisance that appears whenever the left limit of the *horizontal* axis is not a convenient number for *Mathematica*'s automatic tick assignment algorithm. To get rid of it, take charge of the axis limits yourself. Suppose you want the horizontal axis to run from $x = 0$ to $x = 3$. The easiest way to ensure this is to use the option

`PlotRange->{{0,3},Automatic}`.

The first sublist in this option contains the lower and upper limits of the horizontal axis (here, the x axis); the second sublist contains the same information for the vertical axis. If you're happy to let *Mathematica* determine one of these limits for you, just insert `Automatic` in that place in the list. (If you want *Mathematica* to determine both ranges, you don't need this option!) Of course, you can control the vertical range also. For instance, if in the same graph you want the vertical axis to run from $y = 0$ to $y = 100$, then change this option to

`PlotRange->{{0,3},{0,100}}`.

Warning Watch your syntax! You must give the `PlotRange` option a list of lists—unless one of the ranges is `Automatic`.

Tip If you want to force *Mathematica* to show you the entire range of the function you're plotting, set `PlotRange` to `All`. This overrides whatever decisions *Mathematica* has made about your function and shows you the whole thing. I usually use this option not as an end in itself, since the whole plot may be very hard to interpret physically, but rather as a precursor to manipulating the plot range to show me what I want to see about the function.)

3 Coping with error messages from *Mathematica* plot commands.

3.1 When I try to plot my function, *Mathematica* spews out errors that claim that my function is "not a machine-size real number." What's going on?

Your function probably contains either units or symbols which have no values assigned to them.

This instance—the most common error generated by graphics commands—is a rare case where a *Mathematica* error command means exactly what it says. The signature of this message is `Plot::"plnr"`. When you see this (usually in profusion) where you expected to see a plot, go back to the plot command and cut-and-paste your function into a nearby cell. Execute the function and scrutinize the output for any *non-numeric* quantities. Then use assignment statements before the plot command and/or replacement rules in the argument list to ensure that the function you're feeding the command is numeric at all values of the argument.

Warning *Mathematica*'s graphics commands require arguments that are numerical except for the independent variables with respect to which these arguments plotted.

4 Efficient Plotting with *Mathematica*. (Intermediate)

4.1 *Mathematica* is taking an unconscionably long time to generate my plot. What can I do to nudge it along?

You've come to the right place! The answer's easy: use `Evaluate`! To learn more, read the other questions and answers in this section.

4.2 What does the command `Evaluate` do?

This command lets you force *Mathematica* to evaluate an argument to a function. It is relevant when wrapped around an argument to a function.

To understand the power of `Evaluate`, you need to understand a feature of many of *Mathematica*'s graphics (and numerical) commands: they don't evaluate their arguments. This strange statement is best explained by illustrate. Suppose you've defined a function `f[x]` and now want to plot it. When you enter

```
Plot[f[x],{x,0,Pi}];
```

Mathematica first looks at the argument (here, `f[x]`) to determine whether it's a *list*. If so, it prepares to plot the several functions it expects to find in the list. If not, it *assumes* that the argument is an expression *that is numeric except for the independent variable with respect of which you're plotting*, here the variable x. *Mathematica* then determines a grid of values of this variable between the specified upper and lower limits (values of x between 0 and π). Finally, it plugs those values one at a time into the argument *and then evaluates the argument*.

But if you type

```
Plot[Evaluate[f[x]],{x,0,Pi}];
```

then once *Mathematica* has determined that its argument isn't a list, it will evaluate `f[x]` *before plugging in values of* x. In practice, this means that *Mathematica* will perform any algebraic simplifications it can on `f[x]` only once. Without `Evaluate`, it will perform these simplifications *for every value of* x *in the grid it determines*, which may waste gigantic amounts of CPU time.[3]

4.3 When is it *advisable* to use `Evaluate` with *Mathematica* graphics?

When you want to plot one or more functions that are algebraically complicated or computationally demanding.

The plot commands in Version 3.0 of *Mathematica* are much more forgiving than those in previous ver-

[3] For more advanced discussions of `Evaluate`, see §5.3.3 of *Programming in Mathematica, Third Edition* by Roman Maeder, (New York: Addison Wesley Longman, 1997).and §7.2.2 of *Power Programming with Mathematica: The Kernel* by David B. Wagner, (New York: McGraw Hill, 1996)..

sions. Usually, they will generate plots under circumstances where earlier versions would croak Nevertheless, you can almost always speed up the generation of plots by wrapping the *argument* to the graphics command in `Evaluate`. How much this tactic speeds up plot generation depends on the analytic complexity of the function you want to plot. For example, `Evaluate` has almost no effect on the plot of a simple sin function, as you can verify by entering the following:[4]

```
Plot[ Sin[x], {x,0,Pi}]; // Timing
Plot[ Evaluate[Sin[x]], {x,0,Pi}]; // Timing
```

But if the function is more complicated, its effect can be dramatic.

For instance, here is a function that arises in the solution of Laplace's equation for a standard boundary-value problem in electricity and magnetism:[5]

$$V(x,y) = \frac{4}{\pi} \sum_{m=0}^{m_{\max}} \frac{1}{2m-1} \sin[(2m-1)\pi y]\, e^{-(2m-1)\pi x}. \tag{1}$$

The sum, which formally runs to $m_{\max} = \infty$, in practice must be truncated at some finite $m_{\max}$. An interesting question about such functions is, how many terms must be included to generate the function to sufficient accuracy? This question is easily answered with *Mathematica* by defining a function that generates the function, which we'll call `v`, and including `mMax` as one of its arguments: `v[x_,y_,mMax_]`. Doing so and executing these two commands shows the dramatic effect of `Evaluate`:

```
Plot[ v[0,y,100], {y,0,1}];
Plot[ Evaluate[v[0,y,100]], {y,0,1}];
```

In both cases, *Mathematica* draws the plot. But the former (without `Evaluate`) takes *eight times longer* than the latter! Since some such truncated sums require many hundreds or thousands (or more) terms to achieve sufficient precision, using `Evaluate` becomes essential to retaining one's sanity!

Tip Use of `Evaluate` to save CPU time and internal memory becomes more important when plotting several functions (i.e., when the argument to your graphics command is a list) than when plotting a single function.

4.4 When is it *essential* to use `Evaluate` with *Mathematica* graphics?

When plotting a *named* function (or list of functions) and when the argument to a graphics command uses `Table`.

Here's an example of one of the most common—and irritating—glitches newcomers to *Mathematica* encounter. You want to plot a list of functions—say, $\sin(nx)$ for several values of n. Begin an efficient *Mathematica* user, you know that you can use `Table` to generate the list of functions. So you type

```
Plot[ Table[ Sin[n*x], {n,1,7}],
   {x, 0, Pi}];
```

and are enraged when *Mathematica* spits out a stream of error messages informing you that

```
Table[Sin[n x], {n, 1, 7}] is not a machine-size real number
```

at various values of x where you know darn well it is a machine-size real number. What went wrong?

Remember how graphics commands work. *Mathematica* tried to insert values of x into the argument `Table[ Sin[n*x], {n,1,7}]`. This attempt resulted in an unevaluated quantity that, indeed, is a table, not a number. We can avoid the problem in two ways: the inefficient way is to just list the functions; the easy, efficient way is to wrap the argument in `Evaluate`:

```
Plot[ {Sin[x], Sin[2x], Sin[3x], Sin[4x], Sin[5x], Sin[6x], Sin[7x]},
```

[4] Actually, the effect is deleterious! Since the second command asks *Mathematica* to perform a useless chore, it actually takes longer than the first command. Try it and see!

[5] This function is the electric potential of two grounded semi-infinite parallel electrodes. See Tam, §5.2.


```
    {x,0,Pi}];

Plot[ Evaluate[ Table[ Sin[n *x], {n,1,7}]],
   {x, 0, Pi}];
```

The second way works because you've told *Mathematica* to `Evaluate Table[ Sin[n *x], {n,1,7}]` *before it starts substituting values of x into this argument.* When it does so, it (internally) generates the list `{Sin[x], Sin[2x], Sin[3x], Sin[4x], Sin[5x], Sin[6x], Sin[7x]}`, *then* starts plugging in numbers. So to *Mathematica*, the two forms shown above are equivalent. The second, however, is easier to read, saves typing, and minimizes typos—good things all.

Incidentally, you may try (for some perverse reason) to "trick" *Mathematica* by naming the list of functions, then graphing it, as

```
functionList =
   {Sin[x], Sin[2x], Sin[3x], Sin[4x], Sin[5x], Sin[6x], Sin[7x]};

Plot[functionList, {x,0,Pi}];
```

Naughty, naughty. Try this and watch *Mathematica* spew errors at you. Then try wrapping `functionList` in `Evaluate` and watch *Mathematica* do what you want. (It's not nice to fool mother *Mathematica*.)

4.5 What's the most efficient way to plot the `InterpolatingFunctions` returned by *Mathematica*'s numerical differential-equation solver?

Use `Evaluate`!

The many wonderful commands *Mathematica* uses to solve equations *numberically* return *Mathematica* objects called **interpolating functions**. Interpolating functions are easier to manipulate than tables of numbers (the alternative), and can be fed directly into commands to plot, numerically integrate, or otherwise process solutions to equations. You don't really need to know in detail what an interpolating function is so long as you are careful in its use. Most often, what you want to do with an interpolating function is to plot it. Nothing could be easier: you just feed it to the appropriate graphics command as the argument. But because of the way graphics commands deal with arguments, generating the graph may take `Math` a long time—unless you use `Evaluate`!

`Evaluate` is a smart little command. If you feed it an `InterpolatingFunction`, it will perform a simplification (using *Mathematica*'s internal compiler) that will enable it to plot (or otherwise process) the function much more efficiently. For instance, here are the *Mathematica* commands to solve the set of simultaneous differential equations

$$\frac{d}{dt}x(t) = -y(t) - x(t)^2 \tag{2a}$$

$$\frac{d}{dt}y(t) = 2x(t) = y(x) \tag{2b}$$

subject to the initial conditions

$$x(0) = y(0) = 1. \tag{2c}$$

```
differentialEqns  =
    {x'[t]  == -y[t] - x[t]^2,
     y'[t]  ==  2*x[t] - y[t],
     x[0] == y[0] == 1};

soln = NDSolve[ differentialEqns, {x,y}, {t,0,50}]
```

Either of the following *Mathematica* commands will generate a parametric plot of the solution,

```
ParametricPlot[ {x[t],y[t]} /. soln,
  {t,0,50},
  PlotRange->All];

ParametricPlot[ Evaluate[ {x[t],y[t]} /. soln],
  {t,0,50},
  PlotRange->All];
```

But the second command forces *Mathematica* to *compile* the function internally, so it generates the plot 33% faster than the first command! Again, the magnitude of this effect—and the importance of using `Evaluate`—grows dramatically with the complexity of the functions being plotted.

4.6 Is it ever a *mistake* to use `Evaluate`?

It depends on what you mean by "mistake."

I know of no instance where using `Evaluate` causes *Mathematica* to screw up. At worst, if you wrap `Evaluate` around an argument that is so simple that *Mathematica* can't do anything with it, you'll just cause *Mathematica* to take a bit *longer* to generate your plot than it would have otherwise. But if the function is that simple, the amount of CPU time we're talking about is trivial anyway.

4.7 Can I use `Evaluate` to speed up other *Mathematica* commands?

Sure—if they process their arguments the same way graphics commands do.

The second most common use of `Evaluate` I know of it so speed up numerical integration. The command `NIntegrate` handles its arguments just like graphics commands do. For instance, suppose you want to evaluate numerically the integral of two spherical harmonics, $\int_\Omega Y_{2,0}(\theta,\varphi)Y_{3,0}(\theta,\varphi)\,d\Omega$, where Ω refers to the unit sphere, the domain of which is $0 \le \theta \le \pi$ and $0 \le \phi \le 2\pi$.[6] Here are two ways to ask *Mathematica* to carry out this chore:

[6]This isn't a very smart thing to want to do. First, the spherical harmonics $Y_{\ell\, m_\ell}(\theta,\varphi)$ are orthogonal with respect to ℓ, so you know in advance that this argument is zero. Second, if you forget, you can ask *Mathematica* to evaluate this integral analytically, which it can do in about 2 seconds. Still, this example illustrates my point.


```
NIntegrate[
    Evaluate[
      SphericalHarmonicY[2, 0, theta, phi]*
      SphericalHarmonicY[3, 0, theta, phi]],
    {theta,0,Pi}, {phi,0,2Pi}]

NIntegrate[
     SphericalHarmonicY[2, 0, theta, phi]*
     SphericalHarmonicY[3, 0,theta, phi],
        {theta,0,Pi}, {phi,0,2Pi}]
```

The first of these commands executes (on my machine) in about 26 seconds. The second takes so long that I gave up and aborted the command. The reason is that the first command forces *Mathematica* to execute the following algebraic simplification *only once, before it starts plugging in values of* θ *and* ϕ:

$$Y_{2,0}(\theta,\varphi)Y_{3,0}(\theta,\varphi) = \frac{\sqrt{35}}{16\pi}\cos\theta(-1+3\cos^2\theta)(-3+5\cos^2\theta). \tag{3}$$

With the second, *Mathematica* performs this simplification *for every value of* θ *and* ϕ *it thinks it needs to generate the numerical integral you asked for*. Grossly inefficient!

Tip To check whether your favorite numerical command processes arguments like graphics commands do, use the command `Attributes` to see if the command has the attribute `HoldAll`. If so, then you may want to use `Evaluate` to take charge of the evaluation order.

4.8 Can I use `Evaluate` to force *Mathematica* to carry out analytic commands it otherwise can't perform?

Absolutely not. This isn't what `Evaluate` does.

All `Evaluate` does is influence *the order in which Mathematica performs commands it can perform*. If you enter a (legitimate) *Mathematica* command and the kernel returns the command unevaluated, that means *Mathematica* could not perform the command. You're out of luck. For instance, *Mathematica* can't evaluate $\int \tan(\sin x)\,dx$, so it just returns your command:

```
In[1] := Integrate[Tan[Sin[x]],x]
Out[1] := Integrate[Tan[Sin[x]],x]
```

Mathematica has done its best. Wrapping this command in `Evaluate` won't help. Your only recourse is to seek a numerical value using `NIntegrate`. But that's another story.

