APPENDIX 1. HOW TO USE MATHEMATICA

Mathematica is a high-level mathematical computing software package that is available from Wolfram Research Incorporated. These instructions are written for the current Mathematica version (8)at the time of this writing. However, the online help is excellent, so if you want to take advantage of new functions, or if there are minor syntax changes, use the help commands. Also, Mathematica has a number of databases (stored on-line) that include information on chemistry, physics, social sciences, economics, etc., which can be readily imported for various types of analysis. One way to get started with Mathematica is to go through the introductory videos and work along with them. Here, specific tasks are presented that you will need to execute throughout this book (generating functions, plotting, numerical analysis, curve fitting), but there are other excellent features including signal processing, graph theory and topology.

One nice feature of Mathematica is that functions can be written on the fly instead of making a list (vector) of x-values, and then operating on them to transport each x-value to a y-value. This is sometimes referred to as "symbolic math". Another nice feature is that the commands and results can be built into a single file (referred to as a "notebook"), rather than writing a script which executes a series of commands that put "results" (such as numbers, plots, and fits) into separate spreadsheets and windows. These Mathematica notebooks (designated with an .nb extensions) can be used to prepare full reports (and homework assignments) in a single document. Finally, these notebooks can be prettied up much more than a simple Matlab script file, with Titles, text, and images, and also with palates that make equations (fractions, exponents, square roots, sums, and integrals) look much more like we would write them, rather than as line-by-line computer script. And still, for the purists who think more stripped down computer code is the way to go, all commands can be entered line-by-line using the Mathematica programming language. In my opinion Mathematica is easiest for students to learn and for generating documents, but Matlab may be easier to learn if you are already a fluent programmer. Matlab and Scilab are quite similar, and although Scilab is much less developed, it is is free, whereas Mathematica and Matlab can be quite expensive (though many universities and laboratories have site licenses for them). Note also, the cost of Matlab has "hidden costs", because many of the "toolbox" add-ons greatly expand the functionality, but they are not inexpensive.

As with the other appendices for mathematical analysis packages, in the instructions below, commands typed by the user in Mathematica are given in `Courier font`, with responses from Mathematica *`italicized`* when appropriate. Comments in the standard Helvetica font to are sometimes given to the right.

Getting started in Mathematica

Once you install Mathematica on your machine (which works for PC and MAC, simply click the Mathematica icon to launch the program (again, you must be connected to the internet). You will see a welcome window, that has some links to tutorials (these

are useful), but for the purposes of this appendix, you should start a new notebook. In the upper left corner, there is a button that says "Create new…Notebook". Click the notebook part and a new notebook will come up. Another way to do this is from the drag down menu "file/new/notebook". Also to the left of the welcome screen it shows some notebooks you have worked on before, so if you want to continue work with one of them, just click that notebook.

Once you have created a new notebook you are given a clean slate. You can put in titles, text, equations, calculations, plots, tables, etc. There are many ways that you can do these things in Mathematica, depending on if you like pull-down menus, keyboard shortcuts, or formatting palettes. I will give some examples from each, and you can choose which one suits you. Imagine you are writing a homework assignment. The first thing you might want to do is give it a title, your name, and a date. Do this by adding text. One way is to go to the "Format" drag-down menu, and go to "style". You will see a lot of options for Titles, subsections, etc. Select "title" (Format/Style/Title) and enter the text for your problem set: "Problem set 1: The first law of thermodynamics". Then either hit a down arrow or click in the white space below what you have written. This gives you a new "cell" to type more information in. Make a subtitle (Format/Style/Subtitle) and type your name and the date. If you want you can use a simple return to separate the name and date. Mathematica does not interpret a return as a command for execution, because it wants you to be able to use it as a text editor as well as a mathematics engine (to execute a command, you hold "shift-enter"—more on that later). If you don't like the way your text is formatted, open the Palettes menu (top right) and select "Writing Palette". There you can center, change font size, type color, bold/italic., *et cetera*.

Doing arithmetic and simple calculations with built-in functions.

Clearly, being able to able to write pretty, formatted text is not the reason to use Mathematica. It is a bonus, but the point is to do math calculations. There are a few general rules you need to learn to do math in Mathematica.

1. Mathematica has built-in operators for arithmetic (see below) that can be used in line with numbers as you would in normal writing (3+2, for example).

2. Mathematica has lots of commands and built-in functions, and they all start with a capital letter (e.g., `Log`, for example, not `log`). And when the command is a combination of two words (or word fragments), both words (or fragments) are capitalized. Examples include `PlotRange` or `ExpToTrig`.

3. Typically, when you use a command in Mathematica, it is followed by a square bracket […] where the stuff inside the bracket is what the command is operating on.

4. Lists in Mathematica are made by curly brackets `{xxx, yyy, …}` where the elements `xxx, yyy, …` are treated as separate objects. Lists are very similar to vectors, and share some (but not all) characteristics of the vectors in Matlab and Scilab (many of which make them easier to use). We will use lists a lot to compute and analyze data. Lists of lists (nested lists) have multiple curly braces (see below). These nested lists have the structure of matrices.

5. To group your operations together, use parenthesis, for example `(4+6)`. This is in part because the other types of brackets have been used up with functions and lists (see above).

6. When you execute a Mathematica command, you must hit "`shift`" and "`return`" (or `shift` and `enter`) at the same time. Just hitting return makes Mathematica think you just want a new line within a cell. Here, I will designate this as `<shift-enter>` for a while, so you get in the habit of it

7. If you want to issue a set of similar commands within a single cell, must use a "return" between each. This will give you two multiple output cells, one for each input.

8. You can suppress the output of a command using semicolons `(;)` at the end of the command. This is particularly good when your command generates a very long list that will fill the screen. It is also nice for multi-line (multi-command; see 7. above) input cells.

With these rules we can get started with some simple examples of input and output in Mathematica notebooks. If you type `1+1` and then hit shift enter in your notebook, you get the following in your notebook:[1]

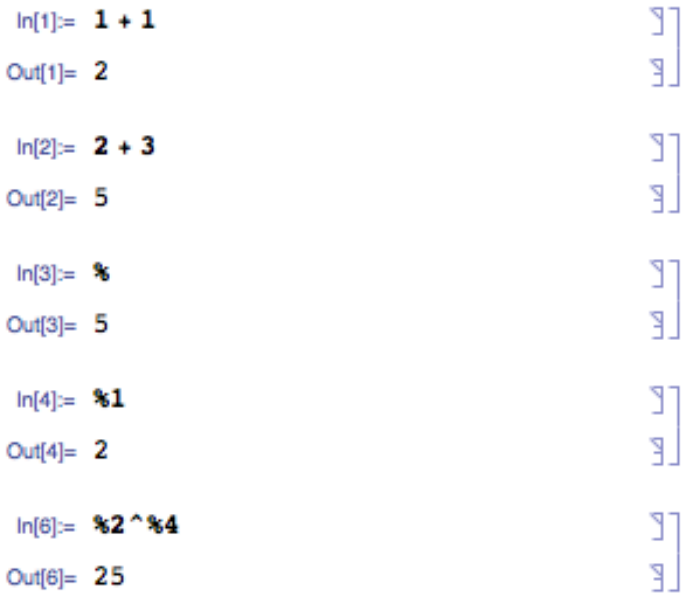
First, you get a blue `In[1]:=` that tells you what your input was, and second you get blue `Out[1]=` that tells you what the output is (in this case *2*). The lines on the right are "cells"—the right-most one shows the first cell, and the left ones show input and output.

To make a second cell, just mouse-click below the lower line (the line with the plus tab on the left) or down-arrow until you see the line. You could next add 2 plus 3, and your notebook would look like this:

[1] For the first few examples, I have included screenshots of the Mathematica notebook to give you a sense of how cells are structured. Thereafter, I will just switch to listing `commands given` and *`output returned`*.

On the right, Mathematica designates another set of cells, and on the left, it increments the input and output numbers to 2. This would keep going as you added (and executed) more cells. You can save your notebook under file/save, but you should note that when you open it back up at a later date, none of the commands will be executed. Although variables were defined and commands were executed in the previous session, in the new session they will not be there until you execute the newly opened notebook. You can do this line-by-line, but an easier way to do it is with the Evaluation/Evaluate notebook.

One feature of Mathematica is that you can refer to the output of the previous command using the percent (`%`) sign, and to earlier outputs using the % followed by the output number, and use these outputs to do further calculations:



A word of warning with this approach is in order. If you refer to results by line number (e.g., `%4`), save the notebook, and quit Mathematica, the next time you run the notebook in a new Mathematica session, the command numbers may change and you will get an incorrect result. These numbers are usually reliable within a single session, but not necessarily between sessions.

Other commands for arithmetic and calculations:

Many of the calculations in Mathematic are intuitive, and you could probably guess at the symbols and syntax. Some common ones are listed here, but there are way to many to list, and you can find them quickly with the on-line help.

Arithmetic

Operation	symbol or function	Example (and what it equals)
Addition	+	`10+7 (17)`
Subtraction	-	`10-7 (3)`
Multiplication	*	`10*7 (70)`
Division	/	`10/7 (10/7)`
Exponentiation	^	`10^7 (10,000,000)`

Note that the output in the division example is the same as the input:

```
In[1]:= 10 / 7
```

$$\text{Out[1]}= \frac{10}{7}$$

This is because Mathematica gives exact answers unless you tell it otherwise. Sometimes you would rather see an approximate result, but in cases like 10/7, the value is a rational number that goes on forever. To get an approximate result, type //N after the input line (N for "numerical approximation"). This tells Mathematica to give you the result to six significant figures[2]:

```
In[2]:= 10 / 7 // N
Out[2]= 1.42857
```

Transcendental functions

Operation	symbol or function	Example (and what it equals)
Square root	`Sqrt[]`	`Sqrt[9] (3)`
Natural log (ln) of x	`Log[x]`	`Log[15]//N` (2.70805)
Base 10 log of x	`Log[b,x]`	`Log[10,15] (1.17609)`
e to the power of x	`Exp[x]`	`E[5]` (148.413)
Sine function of x	`Sin[x]`	`Sin[Pi]` (0)[3]
Cosine function of x	`Cos[x]`	`Cos[Pi]` (-1)
Inverse sine	`AcrSin[x]`	`Arcsin[1]` ($\pi/2$)

[2] The `//N` is a shortcut to a function that can be run on its own. So for example, the same could be achieved by saying `N[Sqrt[10/7]]`. Furthermore, the default of six significant figures can be changed by specifying the number desired after the argument of the N function. So if you want only three significant figures, you could enter `N[Sqrt[10/7],3]`.

[3] Note that the trigonometric functions calculate assuming the argument is in radians. Work with a degree value, you can use the function Degree[x], which converts to radians.

In addition, Mathematica has some ***built-in constants*** that are useful to know:

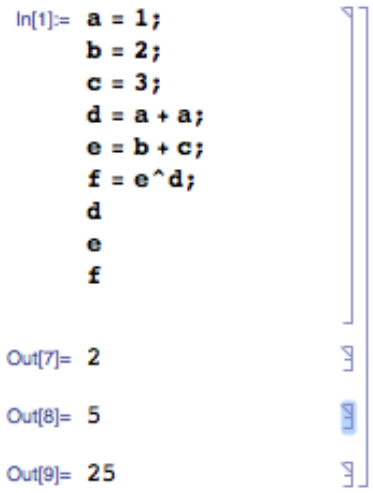
Constant	symbol or function	Example (and what it equals)
π	`Pi`	`Pi//N` (3.14159)
e	`E`	`E//N` (2.71828)
∞	`Infinity`	`Infinity//N` (∞)

Defining variables of different data types.

There are several types of objects that can be defined in Mathematica. These include scalar variables that are assigned a single numerical value (real or complex), string variables, lists (which are similar to vectors in Matlab if they are lists of numbers, but they can also be lists of strings, graphics, functions, etc.), arrays (similar to matrices in Matlab), and graphics of various types.

Scalar variables

The following example uses the assignment of variables to illustrate the simple calculation above, and can be run within a single cell with line breaks:



The variables a, b, and c are assigned the numerical values 1, 2, and 3 using an equals (=) sign. Like in Matlab and Scilab, the use of semicolons is very useful for suppressing unwanted output. As you can see from the output numbers [7, 8, 9], the calculations still exist (for example, in example above, if you entered `%5` as input, you would get out 5 (b+c).

Lists.

Lists are really important in Mathematica calculations. Lists are a series of numbers (real or complex), or other items, such as text strings, graphics, etc. In their simplest form (numbers), they have a lot of features in common with vectors. However,

some of the features of vectors that can be a nuisance for data analysis do not apply to Mathematica lists. A little more on this later.

The simplest way to define a list is to put each object in the list in a set of curly braces, separated by commas:

```
{1,2,3,4,5}     <shift-enter>          returns
Out[1]:=  {1,2,3,4,5}
```

Another very similar way to make a list is the command "`List`".

```
List[1,2,3,4,5]     <shift-enter>          returns
Out[2]:=  {1,2,3,4,5}
```

In other words it is the same thing. These lists can be assigned to variables (again, starting with lowercase is a good idea). For example,

```
mylist={1,2,3,4,5}     <shift-enter>              returns
Out[3]:=  {1,2,3,4,5}
```

Again, it is the same thing. But now, you can call it back because you have created a variable that it is assigned to. If you say

```
mylist     <shift-enter>          Mathematica returns
Out[4]:=  {1,2,3,4,5}
```

The advantage is that Mathematica can easily manipulate your list. If you want the square of the integers in `mylist`, type

```
mylist^2     <shift-enter>              Mathematica returns
Out[5]:= {1, 4, 9, 16, 25}
```

Similarly, the factorial of the numbers in your list can be had by typing

```
Factorial[mylist] <shift-enter>     or
mylist! <shift-enter>               In both cases, Mathematica returns
Out[6]:= {1, 2, 6, 24, 120}
```

And unlike in Matlab and Scillab, the elements of lists (of the same list) can be directly multiplied without any special characters to indicate "element-by-element" multiplication, and not a vector product. For example,

```
lista={1,2,3,4,5};
listb={5, 4, 3, 2, 1};
listc=lista+listb;
listc          <shift-enter>          returns
```

```
Out[7]:= {5, 8, 9, 8, 5}
```

Automated lists

Lists are handy because they can be used as a range of x values for a function that operates on each element of the list. For example, we can create a list of volumes and then calculate pressures, given a temperature value, the number of molecules (or moles), and an equation of state (e.g., an ideal gas law, a van der waals). We then might want to plot the results, or even add some noise to the data, and see how well we can extract constants (like van der waals a and b constants). In short, we might want to simulate some data, and see what can be done with it. Rather than tediously typing in the each of the many volumes we want to evaluate pressure, there are more efficient list generators that can simplify the task. One is called the "`Range`" function, and it is good for integers. If you enter `Range[n]`, Mathematica will return a list of integers starting at one and going up to `n`.

For example, if you type

`Range[12] <shift-enter>` and Mathematica returns

```
Out[1]:={1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12}
```

If you want to start the list at an integer `i` other than 1, and end it at integer `n`, you can type 20, you can type `Range[i,n]`

For example, if you want a list that goes from 3 to 17, type

`Range[3,17] <shift-enter>` and Mathematica returns

```
Out[2]:= {3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17}
```

And if you want a list that does not go in increments of 1, you can specify smaller (or larger) increments for successive elements within the list. This is done with the command `Range[i,n,j]`, where the list starts at `i`, goes to `n`, and the numbers in the list change by increment `j`. For example, if you want a list that goes from 3 to 7 in steps of 0.2, type

`Range[3,7,0.2] <shift-enter>` and Mathematica returns

```
Out[3]:={3., 3.2, 3.4, 3.6, 3.8, 4., 4.2, 4.4, 4.6, 4.8, 5.,
5.2, 5.4, 5.6, 5.8, 6., 6.2, 6.4, 6.6, 6.8, 7.}
```

Importantly, the starting and ending values don't need to be integers, but it is good that you specify an increment that actually gets you to the ending value. Here is an example where the list starts at 0.01, and increments by units 0f 0.01 to a final value of 1.0:

```
Range[0.01,1,0.01]  <shift-enter>
```

and Mathematica returns

```
Out[4]:={0.01, 0.02, 0.03, 0.04, 0.05, 0.06, 0.07, 0.08, 0.09, 0.10,
   0.11, 0.12, 0.13, 0.14, 0.15, 0.16, 0.17, 0.18, 0.19, 0.20, 0.21,
   0.22, 0.23, 0.24, 0.25, 0.26, 0.27, 0.28, 0.29, 0.30, 0.31, 0.32,
   0.33, 0.34, 0.35, 0.36, 0.37, 0.38, 0.39, 0.40, 0.41, 0.42, 0.43,
   0.44, 0.45, 0.46, 0.47, 0.48, 0.49, 0.50, 0.51, 0.52, 0.53, 0.54,
   0.55, 0.56, 0.57, 0.58, 0.59, 0.60, 0.61, 0.62, 0.63, 0.64, 0.65,
   0.66, 0.67, 0.68, 0.69, 0.70, 0.71, 0.72, 0.73, 0.74, 0.75, 0.76,
   0.77, 0.78, 0.79, 0.80, 0.81, 0.82, 0.83, 0.84, 0.85, 0.86, 0.87,
   0.88, 0.89, 0.90, 0.91, 0.92, 0.93, 0.94, 0.95, 0.96, 0.97, 0.98,
   0.99, 1.00}
```

It is not very pretty, but one thing you can do is assign it to a variable to use later, and you can suppress the output with a semicolon.

Here is a little notebook which does just that, calculating the volume for an expansion.

```
In[1]:= V = Range[0.01, 1, 0.01];

In[2]:= V
Out[2]= {0.01, 0.02, 0.03, 0.04, 0.05, 0.06, 0.07, 0.08, 0.09, 0.1, 0.11, 0.12, 0.13, 0.14, 0.15,
  0.16, 0.17, 0.18, 0.19, 0.2, 0.21, 0.22, 0.23, 0.24, 0.25, 0.26, 0.27, 0.28, 0.29,
  0.3, 0.31, 0.32, 0.33, 0.34, 0.35, 0.36, 0.37, 0.38, 0.39, 0.4, 0.41, 0.42, 0.43,
  0.44, 0.45, 0.46, 0.47, 0.48, 0.49, 0.5, 0.51, 0.52, 0.53, 0.54, 0.55, 0.56, 0.57,
  0.58, 0.59, 0.6, 0.61, 0.62, 0.63, 0.64, 0.65, 0.66, 0.67, 0.68, 0.69, 0.7, 0.71,
  0.72, 0.73, 0.74, 0.75, 0.76, 0.77, 0.78, 0.79, 0.8, 0.81, 0.82, 0.83, 0.84, 0.85,
  0.86, 0.87, 0.88, 0.89, 0.9, 0.91, 0.92, 0.93, 0.94, 0.95, 0.96, 0.97, 0.98, 0.99, 1.}
```

In the second line I typed

```
V          <shift-enter>
```

so you can see that indeed, a variable is associated with the list that has the name "V". A handy command that you will use in calculations that involve combining different lists in various ways is

```
Length[{list}]       <shift-enter>
```

which gives the number of elements in the list. For example, the number of volumes in the list V above can be calculated by saying

```
In[15]:= Length[V]
Out[15]= 100
```

Now we can manipulate the list to calculate the pressure during a reversible isothermal transformation over this volume. If the gas is ideal, we can use the equation

$$p = \frac{nRT}{V}$$

To calculate the isotherm. To do so we need to pick values for *T* and *n*, and use the known value for *R* (about 8.314 in units of Joules mol^{-1} K^{-1}). The following notebook does this[4]:

```
In[1]:= V = Range[0.01, 1, 0.01];

In[4]:= R = 8.314;

In[5]:= T = 300;

In[7]:= n = 0.01;

In[9]:= p = n * R * T / V;

In[10]:= p

Out[10]= {2494.2, 1247.1, 831.4, 623.55, 498.84, 415.7, 356.314, 311.775, 277.133, 249.42,
  226.745, 207.85, 191.862, 178.157, 166.28, 155.888, 146.718, 138.567, 131.274,
  124.71, 118.771, 113.373, 108.443, 103.925, 99.768, 95.9308, 92.3778, 89.0786,
  86.0069, 83.14, 80.4581, 77.9438, 75.5818, 73.3588, 71.2629, 69.2833, 67.4108,
  65.6368, 63.9538, 62.355, 60.8341, 59.3857, 58.0047, 56.6864, 55.4267, 54.2217,
  53.0681, 51.9625, 50.902, 49.884, 48.9059, 47.9654, 47.0604, 46.1889, 45.3491,
  44.5393, 43.7579, 43.0034, 42.2746, 41.57, 40.8885, 40.229, 39.5905, 38.9719,
  38.3723, 37.7909, 37.2269, 36.6794, 36.1478, 35.6314, 35.1296, 34.6417, 34.1671,
  33.7054, 33.256, 32.8184, 32.3922, 31.9769, 31.5722, 31.1775, 30.7926, 30.4171,
  30.0506, 29.6929, 29.3435, 29.0023, 28.669, 28.3432, 28.0247, 27.7133, 27.4088,
  27.1109, 26.8194, 26.534, 26.2547, 25.9812, 25.7134, 25.451, 25.1939, 24.942}
```

The last line is to simply show that the product of the calculation is a list, which is a good thing, since our goal was to get p at a bunch of different volumes. And it is clearly decreasing with increasing V, which matches our intuition. In the section on plotting, we will show how we can plot this data.

A second way to make an automated list is with the command "`Table`". `Table` is extremely versatile, and can generate lists of all sorts of objects including graphics arrays where one parameter is changed among the different list elements. `Table` works like `Range`, but it uses a function or an expression to generate the list. In addition to specifying starting and ending values and a step-size, an index (or counter) is defined, and it is used in an expression. In general, the format is

```
Table[expression, {i, imin, imax, step}]
```

[4] Note if you are wondering why the numbers in the "In[]" and "Out[]" statements don't increment as 1, 2, 3, 4 etc., it is because I messed up on some of them when I typed them the first time, and had to rerun a cell a second time. This is why you need to be careful calling a cell output by number (e.g. %23), which may evaluate a different cell the next time you run it. Assigning results like lists to variables is a really good way to get around this.

As an example, the pressure data can be generated above without the vector V, using the command

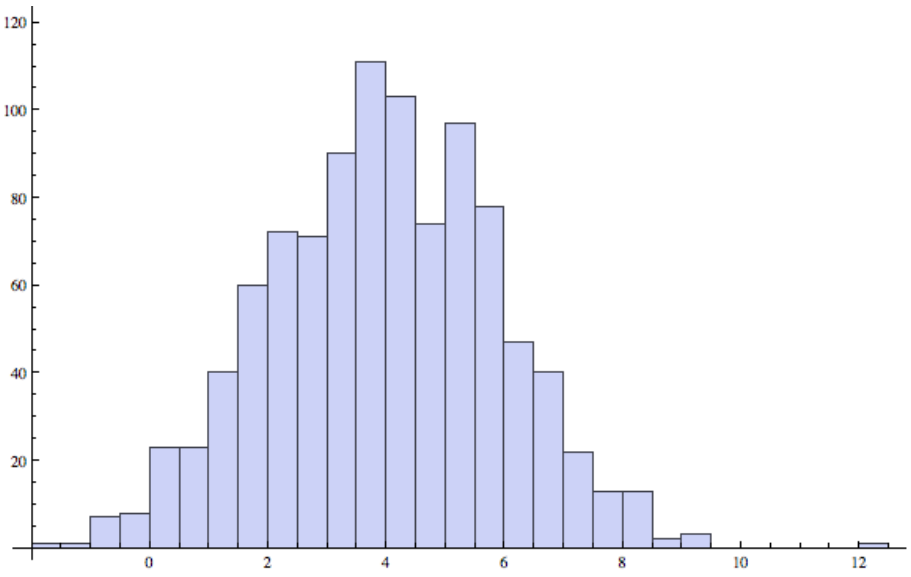
```
Table[n*R*T/V,{V, 0.01, 1, 0.01}]
```

Generating lists of random numbers.

The function we want here is `RandomVariate[NormalDistribution[]]`. This samples a single random number from a Gaussian Distribution centered on zero with a standard deviation of one. But if you say

```
RandomVariate[NormalDistribution[4,2],10^3]
```

you get a list of 1000 numbers randomly selected from a normal distribution with a mean of 4, and a standard deviation of 2. Note that this can always be directed to a variable by writing `variable=` in front of the command above. Plotting this list using the `Histogram[…]` function gives



Manipulating elements in lists.

There are many times when you will want to be able to obtain values for specific elements, or a range of elements, from one or more lists. This can either be done using the command "Part", or by using multiple square brackets. For a simple flat list like

```
mylist=2*Range[12]
{2, 4, 6, 8, 10, 12, 14, 16, 18, 20, 22, 24}
```

```
Part[mylist,3]
6
```

returns the third element in the list.

The same result is obtained by the shortcut `mylist[[3]]`. If you want a range of elements from the list, for example, the 4th through the 8th, you can either enter

```
Part[mylist,4;;8]          or          mylist[[4;;8]]
{8, 10, 12, 14, 16}
```

Often one has a nested list, and wants a single column or a single row. Working from the outside in, the first element of a nested list is the first row of data. For example, in the list

```
littlelist={{a, b}, {c, d}}
```

the first element is {a, b}. Thus, saying either

`littlelist[[2]]` or `Part[littlelist,2]` returns the row

```
{c, d}
```

To get the first element of the second row, say

`littlelist[[2,1]]` or `Part[littlelist,2,1]` which returns

```
c
```

Notice that the command works from left to right, saying "take the second element (a row), and from that, take the first element" (a number). This works in the same way as elements of a matrix are specified (e.g. a_{ij} is the ith row, jth column element).

To get a *column*, use the "`all,`j" command to indicate that you will be selecting an element from the jth column. For the medium sized list,

```
mediumlist={{a, b, c}, {d, e, f}, {g, h, i}}
```

you can get the entire second column by saying

```
mediumlist[[All,2]  or   Part[mediumlist,All,2]   gives
{b, e, h}
```

If you want the second and third column, you can say

```
mediumlist[[All, 2;;3]   or   Part[mediumlist,All,2;;3]
```

which gives

```
{{b, c}, {e, f}, {h, i}}
```

This could also be done with a comma and curly braces to specify columns 2 and 3, i.e.,

```
mediumlist[[All, {2,3}]   or    Part[mediumlist,All,{2,3}]
```

This last approach is useful if you want non-adjacent columns, e.g., columns 1 and 3:

```
mediumlist[[All, {1,3}]
```

Finally, there are times when you want to delete a row or a column from a nested list (for many data files, the first row is a text string of what the variables are—good to know, but bad for plotting and calculations). If you want to delete only the first row (or first element in a non-nested list), you can use the "`Rest`" command:

```
Rest[mediumlist]          returns
{{d, e, f}, {g, h, i}}
```

The command "`Take`" can be used similarly, but allows several rows to be skipped (either from the beginning or end of a nested list).

Separating lists into nested lists can be done with the command "`Partition[list,n]`", where `n` is the number of elements you want in each sublist. In its simplest form, the number of elements in the list must be an integer multiple of `n`. `Partition` keeps the elements in order. For example, to break mylist into a nested list with three sublists, four elements each,

```
Partition[mylist,4]       returns
{{2, 4, 6, 8}, {10, 12, 14, 16}, {18, 20, 22, 24}}
```

The command "`Flatten`" undoes what `Partition` does.

There are many other ways to manipulate *single* lists, for example, deleting elements, removing elements. See the Mathematica documentation center for more information.

It is often useful to **combine lists together**. For example, let's combine `mylist` above with another list containing odd integers

```
anotherlist=mylist-1
{1, 3, 5, 7, 9, 11, 13, 15, 17, 19, 21, 23}
```

To take lists of the same length and interleave the elements, use the command "Riffle":

```
Riffle[anotherlist,mylist]
```

```
{1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18,
19, 20, 21, 22, 23, 24}
```

(Note that Riffle is sensitive to the order that you specify your lists. `Riffle[mylist,anotherlist]` gives a different result. If you then Partition the riffled list above into sublists with two elements, the result is that the two lists are combined into sublists containing the first, second, third…element from each list:

```
Partition[Riffle[anotherlist,mylist],2]
{{1, 2}, {3, 4}, {5, 6}, {7, 8}, {9, 10}, {11, 12}, {13, 14},
{15, 16}, {17, 18}, {19, 20}, {21, 22}, {23, 24}}
```

(note that the inner command, `Riffle`, is done first, followed by the second, `Partition`). This is handy when you want to combine an x vector and a y vector for fitting.

Reading from and writing to files

One of the important applications for Mathematica is analysis of data sets (sometimes large data sets) collected outside Mathematica. Another is generation of data in Mathematica for analysis by other programs. In both cases, external files are required. The two operations used for this are "`Import`" and "`Export`". These commands will read (and write) files from (and to) your home directory (e.g., /Users/doug). Following the description of these commands is a section for how to change the "working" directory so that you don't have to move files to and from their final destination.

Importing data from files. The command used to read data sets into Mathematica from external files is "`Import`". For example, if you want to read a file from your computer that is named "filename.ext" that has a single column of numbers, and put it into a Mathematica list, you say

```
Import["filename.ext","List"]
```

The extension describes how the elements of the list are separated. Tab-separated elements are imported from `.txt` files. Comma separated values are imported from `.csv` files. Space-separated values are imported from `.prn` files.

When you import a file with data arranged in multiple columns and you want to preserve the column structure, you say

```
Import["mybigfile.txt","Table"]
```

which returns

```
{{1, 1}, {2, 4}, {3, 9}, {4, 16}}
```

This generates a nested list. You will use this kind of "`Table`" importing a lot when you bring experimental measurements into Mathematica for analysis, especially for displaying data as a set of points using "`ListPlot`", and curve-fitting. If you don't include the "`Table`" command, Mathematica imports the contents of the file, but it just treats it as a string of numbers.

A full list of allowable extensions can be seen by entering `$ImportFormats` in Mathematica. The safest bet for importing and exporting is a simple text (txt) file, which can be read by all other programs, and does not have any extra formatting (that can produce unintended, and thus undesirable, consequences) when importing back into Mathematica. Tab-delimited data imports as separate columns, as does comma-separated data.

Exporting data to files. Exporting works much the same way as importing. If you have a list, for example,

```
mylist={1,2,3,4,5,6}
```

and you want to save it to a text file, you can say

```
Export["myfile.txt",mylist]
```

This will create a file named `mylist.txt`. Note, the file does not need to exist beforehand, so you can call it anything you want, but if it does exist, it will overwrite without asking) that has each element of the list on a single row, i.e., it will be a column of data. If instead, you have a list of lists, e.g.,

```
mybiglist={{1,1},{2,4},{3,9},{4,16}}
```

and you want to save it as a file with a table format, type

```
Export["mybigfile.txt",mybiglist,"Table"]
```

Produces a file named `mybigfile.txt` in your home director with the contents

```
1       1
2       4
3       9
4       16
```

Allowable extensions for export can be seen by entering `$ExportFormats` in Mathematica. Note that export can also be used to create graphics files, with formats such as JPEG, PNG, PDF, TIFF. This is one way to save plot output for use in other applications. See more on the Export command for graphics (and Image resolution!) in the plotting section below.

Changing the directory for import and export. Typically, your work is being done in a subdirectory of your file system. It is rather cumbersome to move files you need to import to Mathematica up to the default (home) directory, and put exported files back down when you are done (and if you forget, you get a very cluttered home directory very fast!). A good solution is to change the directory. This is done with the `SetDirectory["dir"]` command, e.g.,

```
SetDirectory["/Users/doug/book/Chapter 2/Figures"]
```

The format of the directory path can vary a little from one operating system to another (the above is for a Mac). To make sure you have really set the directory you want, you can say

```
Directory[]                  which returns
/Users/doug/book/Chapter 2/Figures
```

You can list the contents of your current (working) directory with the command

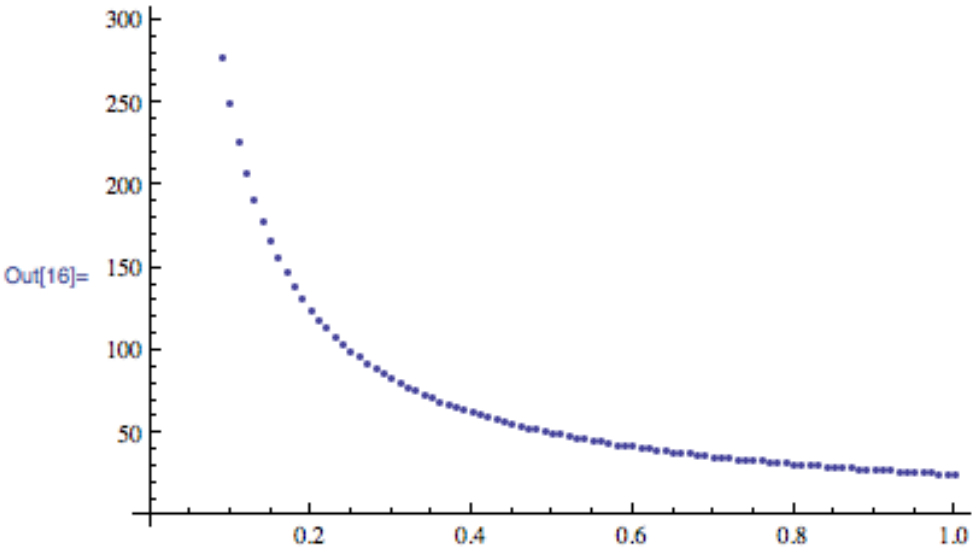
```
Filenames[]
```

Plotting in Mathematica

The simplest and most common plot that can be made in Mathematica is the **two-dimensional plot** (an x-y plot). There are two basic plots that can be made. One is of the type one would generate with discrete data points, like the pressure-volume lists we generated above for an ideal gas. The other plots functions directly.

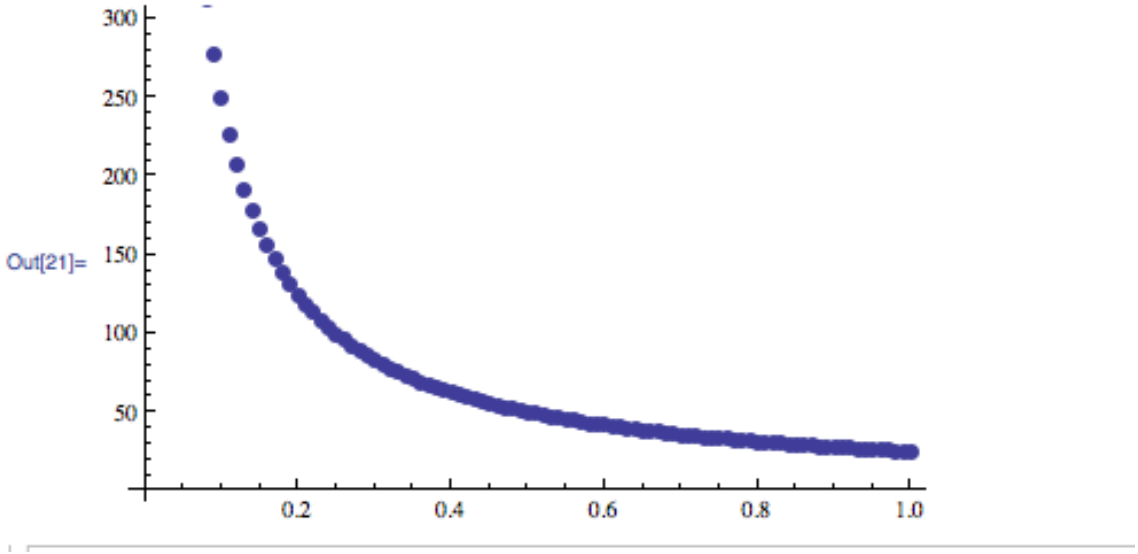
Plotting discrete sets of data. This is done with the function "`ListPlot[argument]`". If the argument of ListPlot is simply a list of numbers (e.g., `{1,4,9,16,25}`), the numbers will just be evenly spaced with x values of `{1,2,3,4,5}`. For our *p-V* relationship above, the volume values, which we would like plotted on the x-axis, do not cover this range, but go from 0.01 to 1 in steps of 0.01. This can be specified as the option `DataRange->{`x_{min}`,`x_{max}`}` in `ListPlot`. For example, for our p-V relationship above, we would type

```
In[16]:= ListPlot[p, DataRange → {0.01, 1}]
```



If the points seem a little too small (the default is 0.008 the fraction of the overall width of the graph [i.e., 0.8%]) you can change by modifying the "Plotstyle":

```
In[21]:= ListPlot[p, DataRange → {0.01, 1}, PlotStyle → PointSize[0.02]]
```

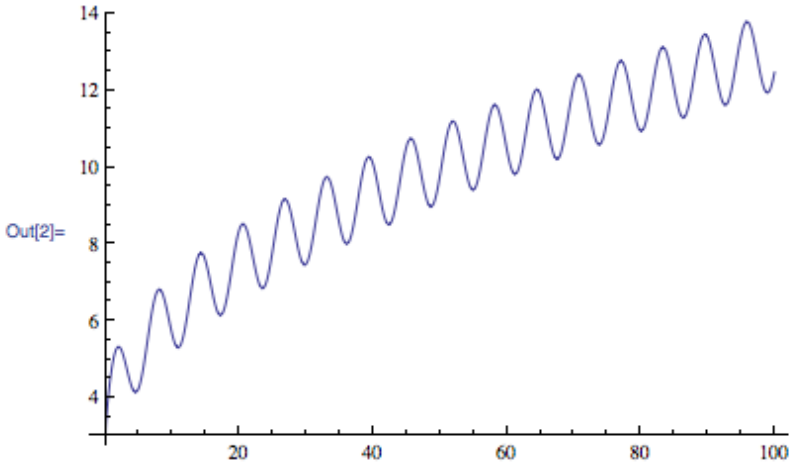


Plotting functions directly:

Another situation where plotting is instructive is to investigate directly how certain functions behave, and to compare them to each other. One way to do this would be to generate a bunch of data values through lists (like our volume list), transform it to a dependent variable (like we did to get pressure), and "connect the dots". But this is a lot

of work. One nice feature is that Mathematica lets you plot "functions" [e.g. $y=3+x^{0.5}+\sin(x)$] directly. The general syntax for this is `Plot[f([x], {x, xmin, xmax}]`. Here is an example for the function given above:[5]

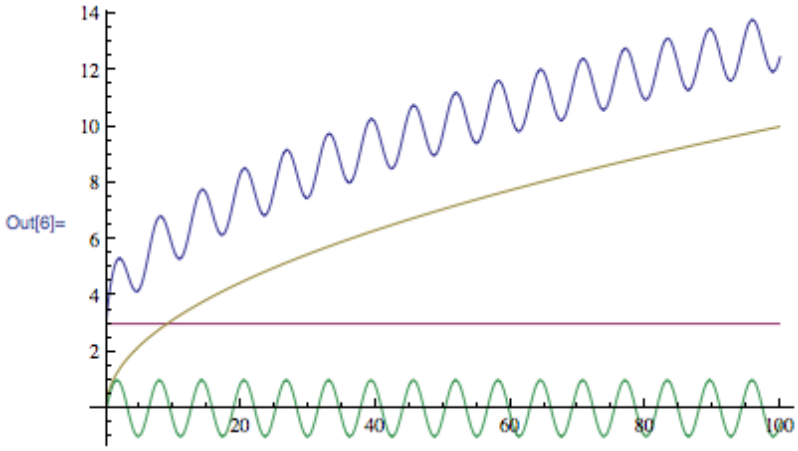
```
Plot[3 + x^0.5 + Sin[x], {x, 0, 100}]   <shift-enter>  returns
```



In the example above, it is pretty clear what the parts of the function do: the sin(x) part gives the wiggle, the square-root of x (i.e. x^0.5) gives the gradual rise, and the 3 gives an offset to the whole function. But in many cases, it is not so easy to see how parts of a function combine to give the whole. In this case it is instructive to plot the parts along with the whole. As is often the case, there are several ways to do this in Mathematica. One way is that the command "`Plot`" allows you to plot a *list* of functions at the same time, in a single command. As always, lists are put in curly braces {} and are separated by commas. So for example, the function could be plotted above, along with the three individual terms in the sum using the command

```
Plot[{3 + x^0.5 + Sin[x], 3, x^0.5, Sin[x]}, {x, 0, 100}]
```

[5] Note that in the within the plot command, which requires square brackets, there are additional square brackets following the Sin command, since it is also a function. In a sense, these are nested functions, and can be nested as deeply as one likes.



The plot above gives a clear sense of how the sin, the square root, and the offset combine to give the whole function. Mathematica uses a default set of colors to draw each function to a different color, which can help distinguish when things get complicated. It also chooses the y-axis scale to show the entire, but not a lot more, which is ideal for visualization (note the x axis scale is set by the command `{x,0,100}`.

The method above for plotting multiple different functions can get a little cumbersome, the more functions you want to include. There is a different way to plot multiple functions that is a little bit simpler, though it takes multiple lines. This is done by plotting each function individually, and assigning the plot to a user-defined variable. For example, for a bimolecular association reaction, which has the form (Chapter 7)

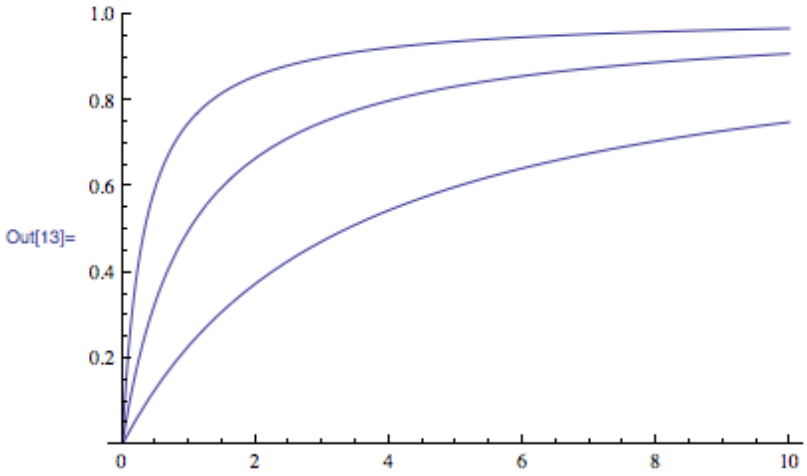
$$f_{bound} = \frac{K[x]}{1+K[x]}$$

where *[x]* is free ligand concentration and *K* is the binding constant (units M^{-1}), three different plots with three different binding constants can be defined as follows:

```
p1=Plot[0.3*x/(1+0.3*x),{x,0,10},PlotRange->{0,1}]
p2=Plot[1.0*x/(1+1.0*x),{x,0,10},PlotRange->{0,1}]
p3=Plot[3.0*x/(1+3.0*x),{x,0,10},PlotRange->{0,1}]
```

This generates three plots (which are not shown here), but more importantly, they are now assigned to the user-defined variables, p1, p2, and p3. The command `Show[…]` can then be used to show multiple curves on the same set of axes as follows:

```
Show[p1, p2, p3, PlotRange -> {0, 1}]
```



In all of the plotting commands in this example, the option `PlotRange->{0,1}` was inserted so that when the plots were combined with the `Show` command, they span the entire x-axis. You should play with what happens when this option is deleted from the four lines of code above.

One of the down-sides of the `Show[{plot list}]` example above is that all three curves are the same color. This makes it hard to tell your reader (your TA, professor, or colleagues) which curve corresponds to which value of the binding constant *K*. But there are other problems with the plot. First, the numbers on the axes are kind of small, and hard to see. Second, there are no labels on the axes. It is very uninformative to present a plot without labeling the axes. Fortunately, there are options that can be included in plot commands (or even at the start of a notebook, if you know in advance that you would like to change some variables (font, font size, axis style) for all of your plots. The next section describes ways to improve the clarity and look of Mathematica plots.

Making 2-D plots beautiful

The basic command for plotting above is "`Plot[f(x),{x,xmin,xmax}]`". Additional options can be added before the last square bracket to change the appearance of the plot. That is, "`Plot[f(x),{x,xmin,xmax},option1,option2…]`". To get a full list of options that can be modified, you should look under the "options" for "graphics" in the help center, which most all apply to plots.

Changing the contents of what is plotted. "`PlotStyle->`" can be used to change the contents of what is plotted (i.e., the (color, , dashing, thickness, reflectance, etc. of points, lines, and surfaces.). This is done with "directives" that `PlotStyle` points to. These directives include

`Color` can be one of Mathematica's default colors[6], or it can be defined by various color schemes such as rgb, hsb, cmyk.

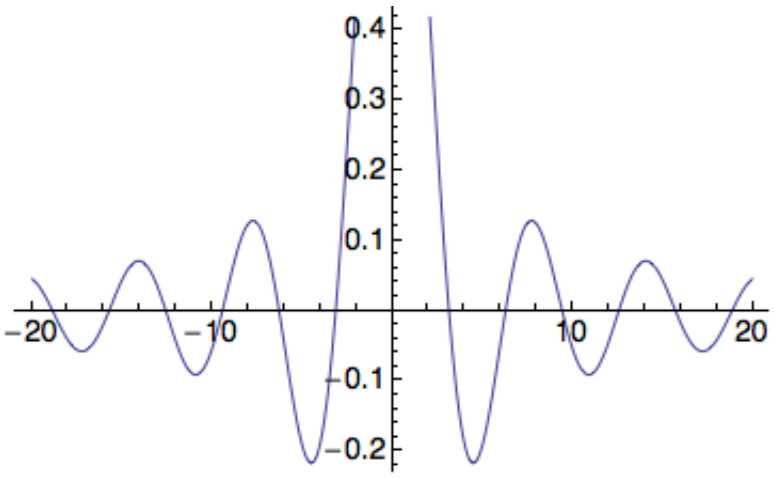
`Thickness[r]` changes the thickness of lines. r is the fraction of the width of the total plot.

`Pointsize[d]` changes the size of markers, for example, the ListPlot points, where d is the fraction of the width of the total plot.

`Dashing[{r1,r2}]` makes lines dashed. The values of r_i are, as above, the fraction of the total width of the plot. When two r values are given, they specify the length of the dash and the white space connecting to the next dash. To avoid messing with these values, can simply be given as `Tiny`, `Small`, `Medium`, or `Large`.

So for example, the simple curve

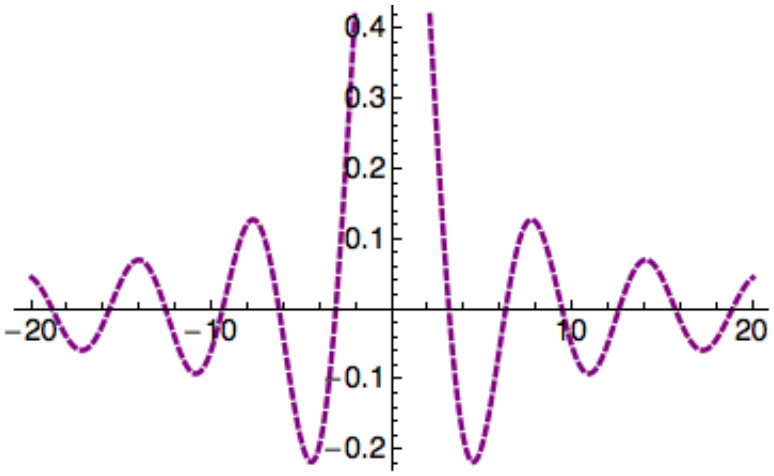
```
Plot[Sinc[x],{x,-20,20}]
```



can be made fancier by adding the PlotStyle option:

```
Plot[Sinc[x],{x,-20,20},
PlotStyle-> {Purple,Thickness[0.007],Dashing->{0.01,.01}}]
```

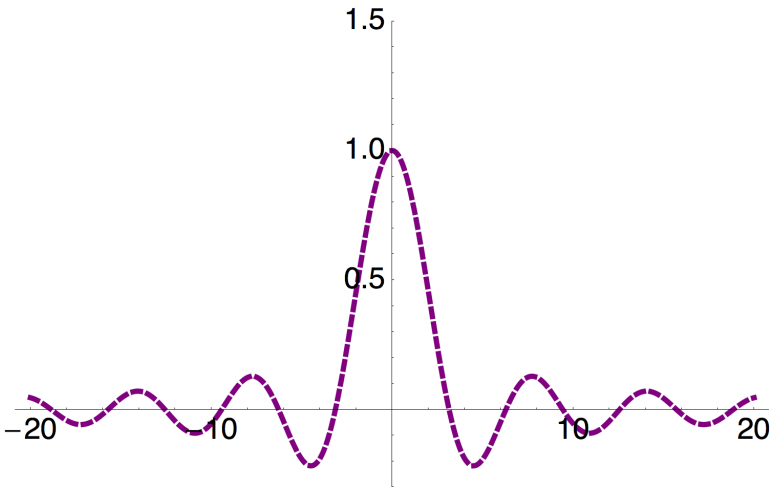
[6] Mathematica's default colors are Red, Green, Blue, Black, White, Gray, Cyan, Magenta, Yellow, Brown, Orange, Pink, Purple, LightRed, LightGreen, LightBlue, LightGray, LightCyan, LightMagenta, LightYellow, LightBrown, LightOrange, LightPink, and LightPurple.



Changing the region that is plotted. Mathematica tries to show what it things will be the most informative *y* region in a plot. However, sometimes its choice cuts off part of the plot that you are interested in. You can change the axis limits of your plot with the `PlotRange` command. If just one pair of numbers is given as a list, it will change the dependent variable range only (y_{min} and y_{max} for a 2D plot, z_{min} and z_{max} for a 3d plot. If instead two ranges are given for a 2D plot will change both x and y; three ranges will change x, y, and z for a 3D plot. If you simply want to be sure all your data is showing, say `PlotRange->All`.

Here is an example where the range of the plot above is changed so that you can see the top of the Sinc(0)=1 peak:

```
Plot[Sinc[x],{x,-20,20},PlotStyle->
{Purple,Thickness[0.007],Dashing->{0.01,.01}},PlotRange->{-
0.3,1.5}]
```



Adding labels to plots. In addition to showing the lines, points, or surfaces of a plot, there is important information that should be included in a plot, so that you and other readers can tell what they are looking at. For example, a 2-dimensional plot

should indicate what quantities are being plotted, what the units are, and what different lines mean. In addition, it is often helpful to give an overall title to the plot. These commands are given below.

To put a title across the center of the plot using the command

```
PlotLabel->"Text"
```

A specific example will be given below.

If you want to put labels on the axes of your plot, there are two ways to do it.

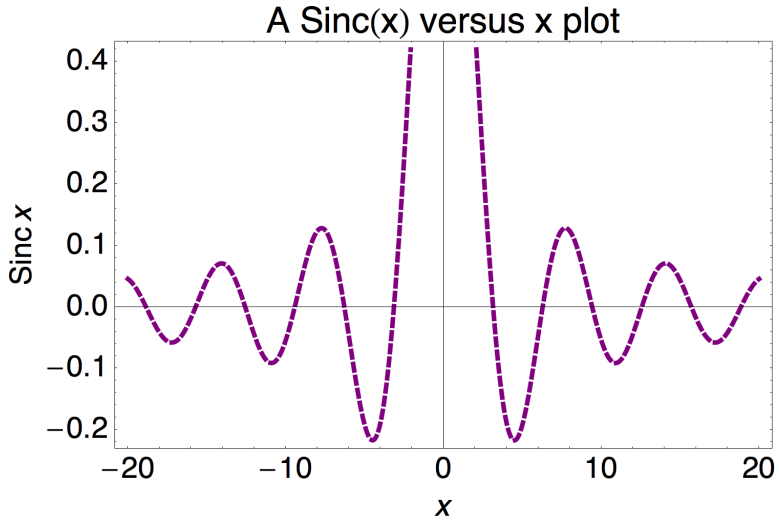
```
AxesLabel->{xlabel,ylabel}
```

puts labels on the x- and y-axes, with the label to the right of the x-axis and directly on top of the y-axis. Unfortunately with this positioning the y-label can look crowded, and the x-label gets lost. A way to get labels in the middle of the axes is to first draw a frame and then use the command `FrameLabel`. The syntax for these commands is

```
Frame->True
FrameLabel->{xlabel,ylabel}
```

Here are some labeling commands that add to the sinc(x) plot from above:

```
Plot[Sinc[x],{x,-20,20},
PlotStyle-> {Purple,Thickness[0.007],Dashing->{0.01,.01}},
PlotLabel->"A Sinc(x) versus x plot",Frame->True,FrameLabel-
>{x,Sinc(x)}]
```



How to set a Mathematica notebook so all your plots have the same features. It is often the case that you will generate multiple plots in a single notebook. In such cases, you will want to change fonts, line thicknesses, etc., the same way for each plot. Rather than entering these commands for each plot, you can set them once at the start of your session. This is done with the command `SetOptions`. For example, you can set the font type and the size for all of the labels using the command.

```
SetOptions[Plot, BaseStyle -> {FontFamily -> "Helvetica",
FontSize -> 14}];
```

Now subsequent text will be in 14-point Helvetica font. But for other graphical output, such as histograms and three dimensional plots, analogous "`SetOptions`" commands will need to be given in which `Plot` is changed to `Histogram` or `Plot3d`.

Exporting plots with high resolution. Many of the plots above was generated simply by selected by clicking on the plot, and writing it out as a file with specific graphic format (.png is nice, but .jpg is fairly universal too). These graphics files can then be imported as a picture into a standard document (like a word or powerpoint file). However, the default resolution on Mathematica graphics is rather low (72 dpi, which is adequate for "screen" resolution, but not for printing or projection), and ends up looking rather pixelated when imported into a document.

To create a graphics file with higher resolution, the command `Export[]` should be used in combination with the command `ImageResolution`. The following command

```
Export["filename.ext", graphic, ImageResolution -> 300]
```

will create a file in your home directory of the appropriate format (specified by `.ext`; e.g., `"filemane.png", "filename.jpg"`) at a resolution of 300 dpi, which should be adequate for most types of production. The three images below (Figure A.XX) were exported with different resolutions.

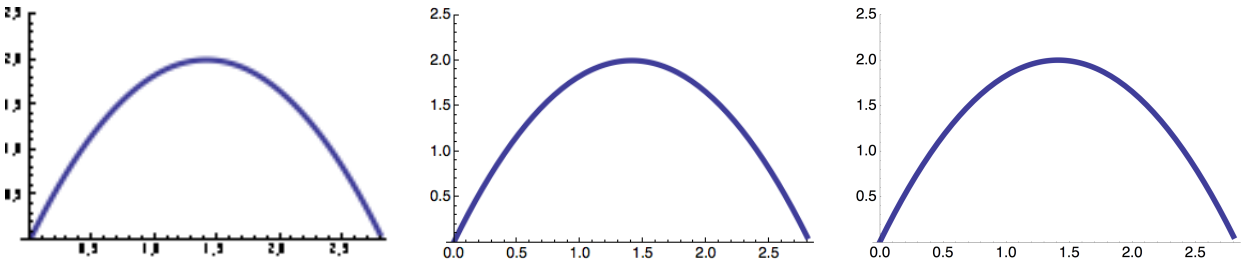


Figure A.XX. Graphics exports with three different ImageResolutions. The file on the left has a resolution of 30 dots per inch (dpi), the plot in the middle has default (screen) resolution of 72 dpi, and the plot on the right has a resolution of 300 dpi.

3D plots.

One nice feature of Mathematica is the ability to generate and manipulate 3-dimensional plots. This allows plotting functions z=f(x,y) of two independent variables. Given that most thermodynamic systems depend on two independent variables, in addition to the amount of material present, such plots are a useful way to visualize thermodynamic quantities. Three-dimensional plots are generated in much the same way that two dimensional plots are generated. The general syntax is

`Plot3D[f(x,y),{x,`x_{min}`,`x_{max}`},{y,`y_{min}`,`y_{max}`}]`

`f(x,y)` is an expression that can either be given inside the Plot3D command, or it can be defined as a separate function. The second and third lists define \ the independent variables to plot against and their limits (they do not need to be called x and y). For example, to plot the function

$$f(x,y) = \mathrm{sinc}(x)\mathrm{sinc}(y)$$

where *sinc(x)* equals *(sin{x}/x)*, , the command is

```
Plot3D[Sinc[x]*Sinc[y], {x, -10, 10}, {y, -10, 10}]:
```

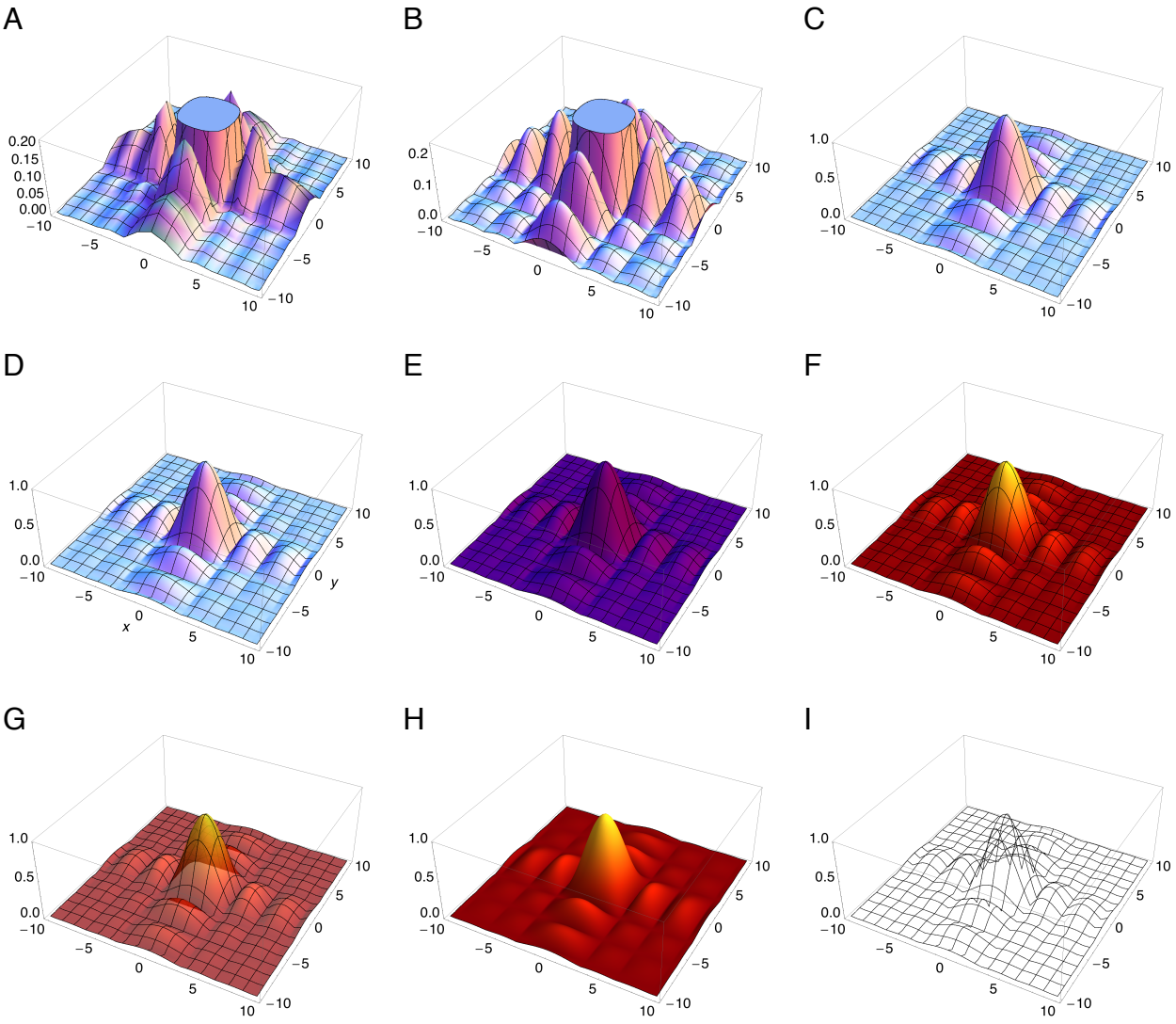


Figure JJ. Three dimensional plotting options.

```
BoxRatios->{1,1,1.5}
AxesEdge -> {{-1, -1}, {-1, -1}, {-1, -1}}
AxesLabel -> {"x", "y", "z"}
Plotpoints->n


ContourPlot
```

`ContourLabels`

Nonlinear least squares fitting.

One of the most important tools we will use in Mathematica is the nonlinear least-squares fitting tool, which allows a model $f(x; p_1, p_2, \ldots)$ to be fitted to observed data ($y_{obs}=y_1, y_2, \ldots, y_n$) by adjusting parameters ($p_1, p_2, \ldots$) to match the model to the data. The method we will use is a method called "least squares". The "least" refers to a minimization, the "squares" refers to the sum of the squared deviation between the model and the data. This deviation, often referred to as a “chi-squared value” can be written as:

$$\chi^2 = \sum_{i=1}^{n} \left\{ y_i - f\left(x_i;\, p_1, p_2, \ldots\right) \right\}^2 \qquad \text{M.1}$$

The algorithm goes through a number of iterations where it adjusts the parameters ($p_1, p_2, \ldots$), and calculates the residual (the term in the curly brackets), and it returns the parameters that minimize χ^2, i.e. those that result in the “least squares”. There are a number of important considerations relating to parameter optimization using least-squares methods; many of these have been discussed extensively in the literature (for example, Johnson, 2008).

If you are familiar with linear regression, then you have some experience with "least squares" approaches to determining parameter values (in that case, the parameters are the slope and intercept of a line). For linear regression, there is an exact analytical formula that can be used to find the slope and intercept that minimize the residual, and thus, χ^2. However, unlike the equation for a line, most of the functions we will fit to data are not linear in their unknown parameters ($p_1, p_2, \ldots$). In general, for such "nonlinear" least squares problems, there is no direct, analytical solution.

Instead, nonlinear least squares methods work by an iterative search mechanism where the parameters are adjusted a little bit, and if χ^2 decreases as a result of the adjustment, the new parameters are kept, and used in another optimization cycle. The process repeats until the parameters do not get better. This process can be considered a search for an optimum (minimum) in *n*-dimensional parameter space. There are a number of problems that can occur in such a search: sets of parameters can be found that provide a local minimum in χ^2, but not a global minimum. That is, another set of parameters result in a lower χ^2 value, but they are not identified because they are far away in parameter space. This is a problem of searching space efficiently. However, a very efficient search of parameter space may be very costly in terms of computing time. A number of different "optimization methods" have been developed, some with speed in mind, and others with thorough searching in mind. These include direct search methods (including Nelder-Mead "Simplex" search, Monte Carlo searching), "gradient" methods, Gauss-Newton, and other methods (see Bevington). One of the most popular methods,

which we encourage you to use for this book, is called the Levenberg-Marquardt method. This method combines features of the different optimization approaches, and as a result, it is both reasonably fast and does a good job of searching broadly (in the language of optimization, it has a "good radius of convergence").

Regardless of the specific method, iterative nonlinear optimization method, most nonlinear least squares methods require a set of “initial guess” parameters to evaluate χ^2. Although these guesses only need to be approximate, they must be close enough that the value of the residual is sensitive to modest adjustments to the parameter values (that is, they need to be within the radius of convergence). Thus, there is sometimes a bit of adjusting that needs to be made to get the initial parameters to where the fitting program can work with them. Otherwise, the fitting algorithm will "diverge" (i.e., fail). Usually you can just look at your experimental data and come up with decent initial guesses that will converge (for example, by eyeballing midpoints, slopes, plateaus in the data). However, sometimes it is helpful to "simulate" the data with the fitting function, and try various combinations of parameter values that make the fitting function look like the data.

The command we will use in Mathematica to fit a model to data using nonlinear least squares is "`NonlinearModelFit`".[7] This function is called using the command

```
NonlinearModelFit[data, {function, constraints}, {fitting params
and guesses},x, options]
```

Format of the data. For a data set with *n* observations, the expression *data* above is a nested list of depth 2, containing *n* sublists. Each sublist corresponds to the values of the independent and dependent variable of the first "point" in the data set, the second to the second point, and so on, up through the n^{th} (i.e., last) point. For example, for a simple fit of a function of a single variable and parameters $y=f(x,p_1, p_2, \ldots)$, the data would have the form

```
{{x1,y1},{x2,y2},{x3,y3},…,{xn,yn}}
```

This format is typically encountered when data is imported using the "Table" option. However, the data may either need to be arranged before (with a spreadsheet or editor) or after (using list manipulation) import to Mathematica. One feature of NonlinearModelFit is that it can fit a function of multiple variables, for example, *z* as a function of *x* and *y*. To do this, the data are modified to

```
{{x1,y1,z1},{x2,y2,z2},{x3,y3,z3},…,{xn,yn,zn}}
```

[7] Another closely related function, "`FindFit`" works almost the same way. However, `NonlinearModelFit` provides a more sophisticated analysis of the results of the fit, such as determination of confidence intervals on parameters, analysis of parameter correlation analysis, and analysis of variance (Anova).

Check to see that this really works. It says so in the documentation, but gives no worked examples.

Format of the Function and constraints. The function is what you are going to fit to the data. It has the form given in Equation M.1 above. You can either enter it directly, or you can define it on a previous line. For example, if the data is fractional saturation of a macromolecule containing multiple sites with ligand at concentration *x*, and you wish to fit a Hill equation (Chapter 7, Equation 7.XX) to the data, fitting for both the binding constant *K* and the Hill coefficient *n* (parameters p_1 and p_2), then you would enter:

```
(K*x)^n/(1+(K*x)^n)
```

If you wish to constrain the fitted parameters to a limited range of values (for instance, equilibrium constants are never negative), you can provide constraints. Constraining a parameter can also be useful when you have strong parameter correlation, especially if you have information on one of the parameter values from another experiment. If you don't wish to use the symbol `x` as your independent variable (or variables x_1, x_2, ...), you need to be sure to enter your chosen symbol (or symbols) following the {Fitting params} instead of `x`.

Format of the fitting params and guesses. These are given as a nested list, with each sublist a two-element list consisting of the parameter name and the initial guess For the example described here with the Hill model, the format would be

```
{{K,200},{n,2}}
```

I would use these initial values if the midpoint of the transition looks like

Options. To see a full list of options that can be entered, go to the the NonlinearModelFit entry in the Documentation center of Mathematica, and open the "more information" section. This is the place where you can set the type of algorithm used for optimization. I would always start with the option

```
Method->LevenbergMarquardt
```

Other things that can be set using "options". For example, weights can be introduced if some *y* values have more error than others[8]. In addition, the maximum number of iterations to do before quitting can be set, and the "convergence criterion" can also be set (the point at which the fitter decides it is done, which is chosen when the decrease in χ^2 is smaller than a specified value).

Combining all these commands for the Hill example above would look like this:

[8] Nonlinear least squares assumes each y-value has the same level of error, that the error is uncorrelated from point to point, and that the error is Gaussian.

```
NonlinearModelFit[data, (K*x)^n/(1+(K*x)^n), {{K,200},{n,2}},x,
Method->LevenbergMarquardt]
```

In addition, you might want to assign the fit to a variable name so that you can analyze various aspects of the fit (compare the fit to the data, check parameter values, plot residuals):

```
myfit=NonlinearModelFit[data, (K*x)^n/(1+(K*x)^n),
{{K,200},{n,2}},x, Method->LevenbergMarquardt]
```

To see examples of this fit, look at the notebooks

Hill plot for fitting no errors.nb

Hill plot for fitting 1pct error.nb