

Plotting and data visualization

Visualization means making plots of something you've computed. This hand-out discusses the process of making plots from data. Once you have this, you can explore the R documentation to see the many different kinds of plots and visualizations you can make. R was created by statisticians. These people have some unusual ideas about how to visualize data, particularly statistical data. For that reason, you may have to spend some time figuring out how to get just the plots you want.

Good visualization is essential for most computing projects, even the lightweight ones you get for homework. There are professional standards and visualization practices just as there are programming standards and software practices. They take a little time to follow, but they usually save time in the long run. This is true even if the "long run" is just a two hour homework assignment. You can have points taken off for poor plots and for poor codes.

Basic line plot

A *line plot* is a plot of a collection of (x, y) values. The term "line" plot may come from the idea that you are supposed to connect the dots with lines to see a curve. Figure 1 shows an R script that makes a line plot. It plots the probability density function

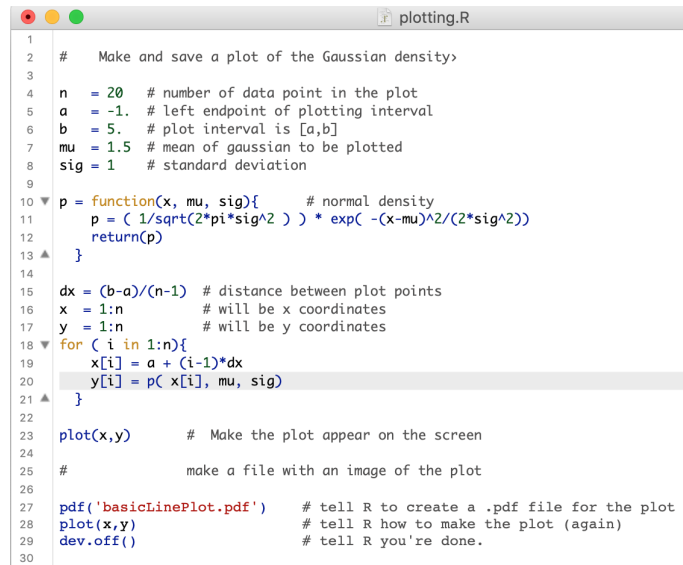
$$p(x) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{(x-\mu)^2}{2\sigma^2}}.$$

The parameters μ and σ are given on lines 7 and 8. The density function p is implemented using the function `p()` defined in lines 10 to 13. The density $p(x)$ is evaluated at n evenly spaced points starting from $x_1 = a$ and ending at $x_n = b$. The parameters n , a , and b are defined on lines 4, 5, and 6.

The code from lines 15 to 21 creates the values to be plotted. The distance between points is $\Delta x = (b - a)/(n - 1)$ (it's $n - 1$ instead of n because both endpoints are included). Lines 16 and 17 create (*initialize*) the arrays of length n that will hold the x and y values. These both are initialized to the numbers $1, 2, \dots, n$, but these values will be *over-written* by the actual values, which happens in the loop starting at line 18. It can be confusing to use the name `x` for an array of x values. A name like `xa` or even `xArray` might be clearer. Line 19 creates the evenly spaced x values. Line 20 *calls* the function `p()` to evaluate the probability density, which will be the y value. In formulas, the data values `x[i]` and `y[i]` represent the point in the plane

$$\begin{aligned} & (x_i, y_i) \\ x_i &= a + (i - 1)\Delta x, \quad \Delta x = \frac{b - a}{n - 1} \\ y_i &= p(x_i). \end{aligned}$$

The R function `plot()` on line 23 opens (or should open, post on the class forum if this doesn't work for you) another window with the (x, y) pairs marked with little circles.



```
1
2 # Make and save a plot of the Gaussian density>
3
4 n = 20 # number of data point in the plot
5 a = -1. # left endpoint of plotting interval
6 b = 5. # plot interval is [a,b]
7 mu = 1.5 # mean of gaussian to be plotted
8 sig = 1 # standard deviation
9
10 p = function(x, mu, sig){ # normal density
11   p = ( 1/sqrt(2*pi*sig^2 ) ) * exp( -(x-mu)^2/(2*sig^2))
12   return(p)
13 }
14
15 dx = (b-a)/(n-1) # distance between plot points
16 x = 1:n # will be x coordinates
17 y = 1:n # will be y coordinates
18 for ( i in 1:n){
19   x[i] = a + (i-1)*dx
20   y[i] = p( x[i], mu, sig)
21 }
22
23 plot(x,y) # Make the plot appear on the screen
24
25 # make a file with an image of the plot
26
27 pdf('basicLinePlot.pdf') # tell R to create a .pdf file for the plot
28 plot(x,y) # tell R how to make the plot (again)
29 dev.off() # tell R you're done.
30
```

Figure 1: An R script that makes a basic line plot.

I type

```
> source("plotting.R") [enter]
```

at the prompt in the command window (the console) and up pops a window pictured in Figure 2. If the script were to end here, there would be no record of the plot except the screen capture in the figure, which is not a good way to save plots. Lines 27 through 29 tell R to make a .pdf file of the plot. Line 27 creates an R *device* that records all your plotting commands. In this case, the device is a “pdf writer”, which creates a .pdf file. The name of the file will be `basicLinePlot.pdf`. You could make a .jpeg file or a .gif file, but .pdf usually results in a higher quality plot. Line 28 is the plotting command. Unfortunately, it has to be repeated as the one on line 23 happened before the “device” was created. Often it takes more than one command to make a plot. You have to repeat all of them. I usually do this with copy and paste. Line 29 turns the “device” off, which stops the recording and creates the actual .pdf file.

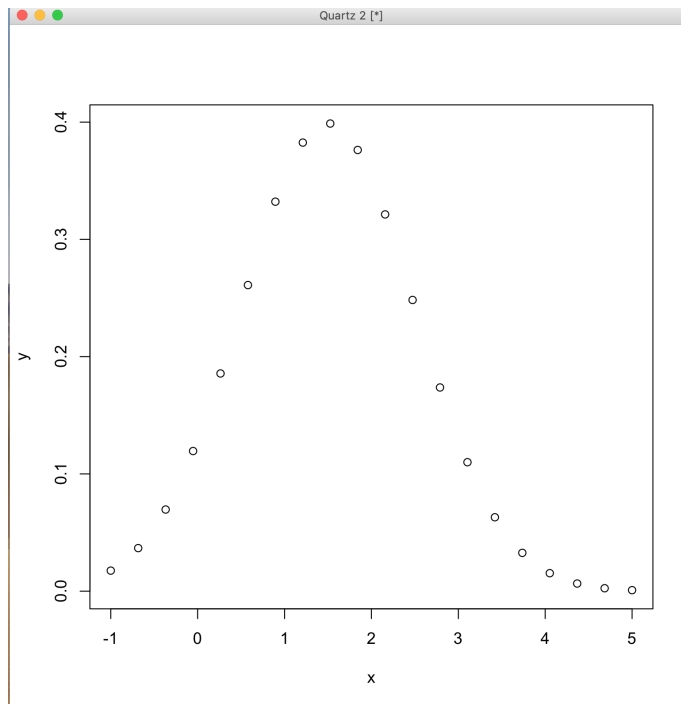


Figure 2: The basic plot made with the code of Figure 1

Good plotting practice

Figure 3 has the basic plot code from figure 1 and more stuff to make it more useful. Figure 4 has the file `fancyPlot.pdf` that this script saves. It has a *plot title* on the top that says what the plot is and gives some parameter values. The character string `title` is created by line 23 using the `sprintf()` function. The values of μ and σ are embedded in this string. Line 24 creates a subtitle, which appears at the bottom of the plot. Lines 28 to 33 are an expanded call to the R function `plot()`. One new thing is that the command takes more than one line. You saw this in the beginning with *continuation lines*. The continuation is automatic when it is obvious to the interpreter that the command is not finished. In this case, it is obvious because the function `plot(. . .)` has to end with a close paren `)`. Until then, the command is not finished. It would have been possible to put the whole command on one line (if you delete the comments), but that line would have been long and hard to read. All R programming style guides say to avoid long lines in this way. The comment on line 29, starting with the hashtag character `#`, is ignored by the interpreter. It is there to make the program easier to understand by the person reading the code.

The arguments on lines 29 to 33 are of the form *keyword = value*. On line 29, `main` is a keyword whose value is `title`. Many R functions have a large

number of possible arguments that have *default* values. If the command doesn't specify a value of the keyword, then R gives it the default value. The default value of `main` is the empty string `""`. The `plot()` command on line 23 of Figure 1 doesn't specify a value of `main` so the plot title comes out as the empty string. That empty string is actually at the top of Figure 2, but you can't see it because it's blank. The other arguments, given on lines 30 to 34 in Figure 3 also have default value `""`. Saying `main = title` overrides the default with the string `title`. If you look in the documentation of the R `plot()` function, you will see that there are many other arguments with default values that we still aren't overriding. Some of them change colors (black lines to blue, say), symbols (circles to triangles, say), line thickness, etc. You can spend a lot of time making plots fancy. Line 33 tells R to put faint grid lines in the plot. These make it easier to read function values from the plot. Line 34 calls the R function `lines()`, which creates line segments between each pair of points (x_i, y_i) and (x_{i+1}, y_{i+1}) .

Lines 28 to 34 make a plot appear on the computer screen with all these features. Lines 42 to 48 make the same plot go into a .pdf file. I created lines 42 to 48 by copy/paste from lines 28 to 34. The comments should say that I did that. These comments may be the most helpful in the whole script, since anyone who wants to change the plot has to make sure the change happens in both places. It's too easy to change lines 28 to 34 and then forget to copy/paste the changed code to the part that makes the .pdf file.

```

1
2 # Make and save a plot of the Gaussian density>
3
4 n = 20 # number of data point in the plot
5 a = -1. # left endpoint of plotting interval
6 b = 5. # plot interval is [a,b]
7 mu = 1.5 # mean of gaussian to be plotted
8 sig = 1. # standard deviation
9
10 p = function(x, mu, sig){ # normal density
11   p = ( 1/sqrt(2*pi*sig^2 ) ) * exp( -(x-mu)^2/(2*sig^2))
12   return(p)
13 }
14
15 dx = (b-a)/(n-1) # distance between plot points
16 x = 1:n # will be x coordinates
17 y = 1:n # will be y coordinates
18 for ( i in 1:n){
19   x[i] = a + (i-1)*dx
20   y[i] = p( x[i], mu, sig)
21 }
22
23 title = sprintf("Gaussian with mean %6.2f and standard deviation %6.2f", mu, sig)
24 subtitle = sprintf("Using %4d points", n)
25
26 # Plot commands
27
28 plot( x, y,
29   main = title, # title at the top of the plot
30   sub = subtitle, # subtitle at the bottom
31   xlab = "X axis", # label of the horizontal axis
32   ylab = "probability density", # label of the vertical axis
33   panel.first = grid() # make a grid on the plot
34   lines(x,y) # plot lines between the data points
35
36 # make a file with an image of the plot
37
38 pdf('fancyPlot.pdf') # tell R to create a .pdf file for the plot
39
40 # Plot commands, copy and paste from above
41
42 plot( x, y,
43   main = title, # title at the top of the plot
44   sub = subtitle, # subtitle at the bottom
45   xlab = "X axis", # label of the horizontal axis
46   ylab = "probability density", # label of the vertical axis
47   panel.first = grid() # make a grid on the plot
48   lines(x,y) # plot lines between the data points
49   dev.off() # tell R you're done.
50

```

Figure 3: An R script that makes a basic line plot.

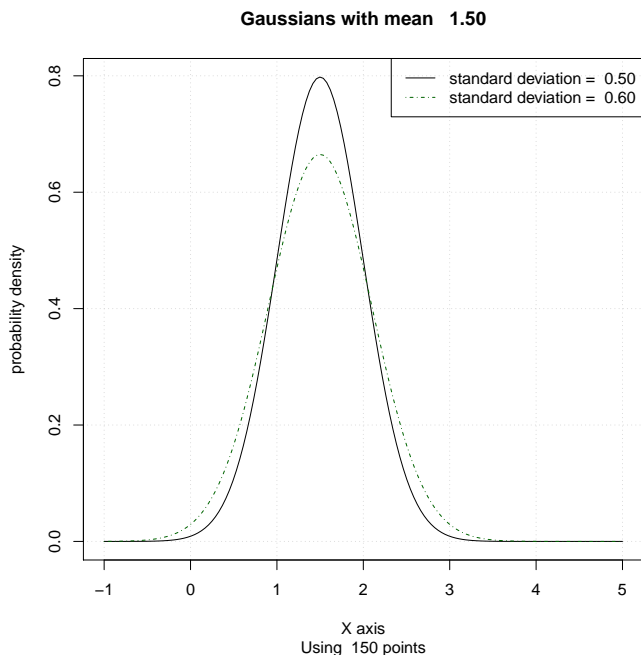


Figure 4: An image of the file `fancyPlot.pdf`.

Automation is an important principle in lightweight scientific computing, which is what we're doing. Once you're created a script like this, you want to play with it. In this case, you might want to know what other Gaussian probability density functions look like. Suppose, for example, we want to see what happens with standard deviation $\sigma = .5$. We change line 8 of the script in Figure 3 to `sig = .5`, then type `> source("plotting.R")` [enter]. You get the plot in Figure 5. The title of this plot records that $\sigma = .5$. Imagine the situation if we had not put the value of σ in the title. We would have the two plots, one with $\sigma = 1$ and the other with $\sigma = .5$, and we would immediately forget which is which. The title contains the information about the calculation that makes it possible to look at a collection of plots and know which is which. Informative titles are crucial in any presentation, even in homework. You will lose points if your plots don't have informative titles with the important parameters.

It is possible to type plot titles and other information by hand instead of putting it into a script. That is, you don't have to *automate* your work by putting it into a script. Many of the examples you find by web searching questions on R are typed directly at the command line in the console (command) window. You should do this as little as possible. Non-automated work may be quicker the first time, because you skip the complexity of writing a script and typing `source("...")` [enter] to run it. Scientific computing involves exploration. You want to see what happens if you change something. Exploring, at

the command line, involves typing the same or very similar commands over and over. It's much quicker to change a few lines in a script and run it again. For example, the curve in Figure 5 doesn't give a very clear picture of graph of $p(x)$ because the points x_i are too far apart. That's easy to fix by using more points. Let's see whether $n = 50$ is enough points. Change line 4 to `n = 50` and `source` the script. The new plot is better but not great. Let's try $n = 80$. That looks good enough to me. This plot is in Figure 6. Imagine how long it would taken to do this trial and error experiment typing the plot titles by hand. Would you have bothered to do it?

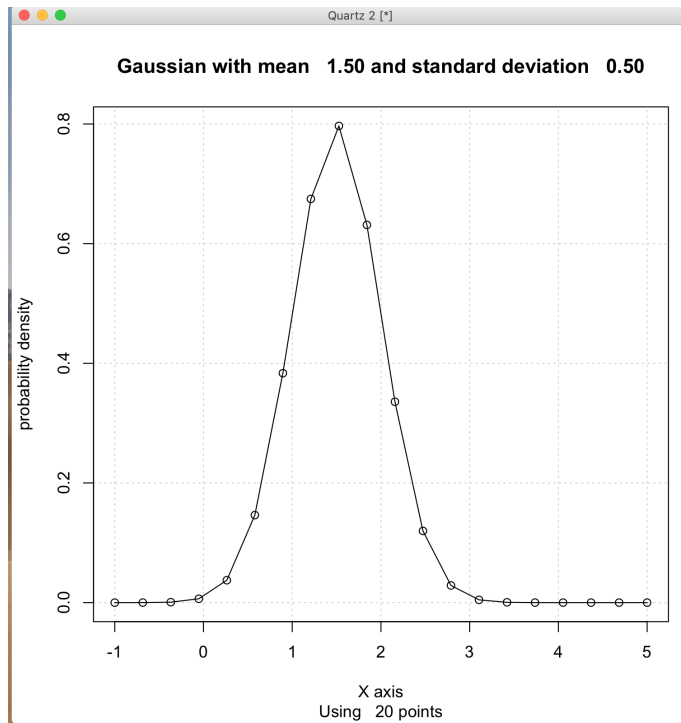


Figure 5: Just changing line 8 to `sig = .5`.

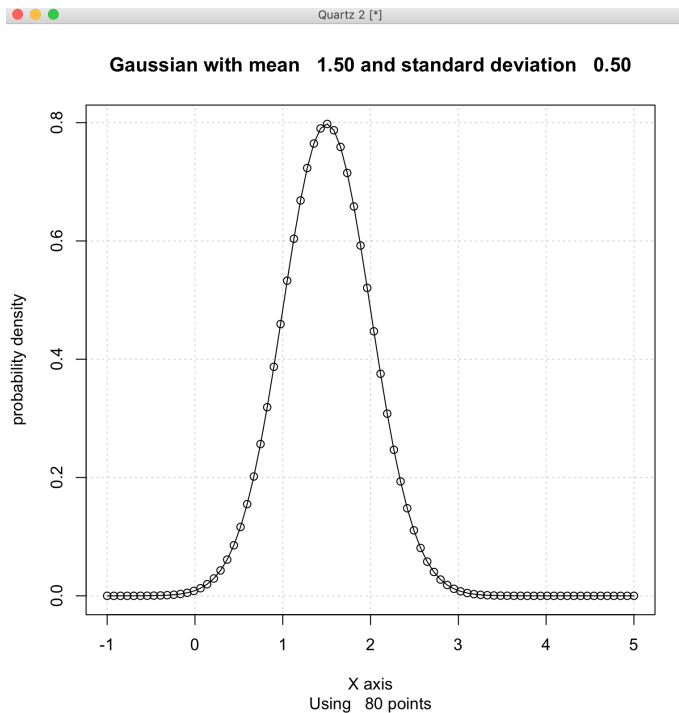


Figure 6: Just changing line 4 to $n = 80$.

Informative plots

Good plots convey a lot of information. Put multiple curves in the same plot. These could be theory and data, or plots with different parameter values. This makes it easier to compare the curves. Use color or line style or symbols to distinguish different curves. Use a legend in the plot to label the curves. Choose scales and plot styles to make plots easy to read.

It can be frustrating and time consuming to get plots to look the way you want. Information on the web is either sketchy or too detailed. Examples are poorly documented and explained, and they don't always work. The language itself can be clumsy. R is worse than Python or Matlab in these ways: clumsy graphics routines, poor documentation.

Download the file `plotting.R` next to this handout. Save it with your other R scripts and open it in the R script editor so you can see the line numbers. Figure 7 has the plot with two curves. The density with $\sigma = .5$ is the solid black line. The density with $\sigma = .6$ is the dotted green line (dark green may be hard to tell from black, but it is different). The legend in the top right box says which curve is which. Putting the two curves together in the same plot makes it easy to see the differences. The density with larger σ is wider (obviously). Also, the peak density is lower. The area under both curves is 1, so the wider

curve has to have a lower peak. It may seem surprising: the peak seems a lot lower although the wider curve is only a little wider.

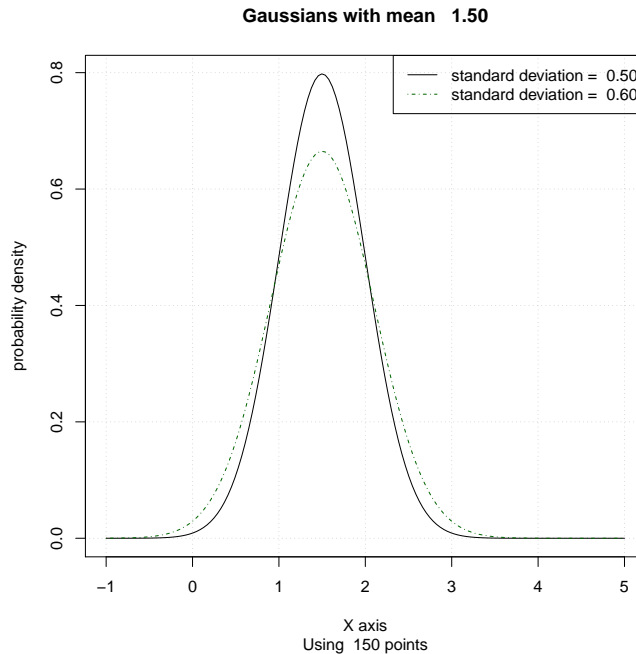


Figure 7: Two Gaussian densities plotted together.

lines 2-4 Whenever you post code or give it to someone else, it should have some explanation at the top – who wrote it, when, and why.

lines 7-12 The parameter values have changed and the = signs don't line up any longer. There are two σ parameters σ_1 and σ_2 for the two plots. Note that the new value $n = 150$ appears in the plot subtitle. If $n = 80$ had been hard wired, I probably would have forgotten to change it both places.

lines 14-17 The function that evaluates the Gaussian PDF is the same. That shows the value of defining functions with arguments.

lines 20-22 The x values are the same, but there are two sets of y values.

lines 26-27 Call the Gaussian density function twice, with the two σ parameters.

lines 33-37 Different lines can have different **types** and different colors. Use this to make it easy to see which line is which on the graph.

lines 39-40 The text of the legend should have the parameter(s) defining each curve.

lines 42-44 The R plot function `legend()` takes arrays as arguments. Each array has one data member for each curve. The R function `c(...)` creates an array with the arguments as data members. For example, `types` will be an array of length 2 with `types[1]` having the value `type1`, etc. Legends and colors are character strings (the `"..."` on lines 33, 34, and the `sprintf()` on lines 39 and 40). Line types (solid, dashed, dotted, mixed) are integers.

line 55 This gets rid of the circles marking the data points in Figure 6.

lines 56-57 Specify the style (type) and color of the first curve.

lines 63-66 Put the legend in the top right corner of the plot. For some reason, `legends` is an argument you just give, but `col` (for *colors*) and `lty` (for *line type*) have to be given by name.

lines 77-97 Once the plot commands worked (after hours of fighting with the computer) I could copy/paste them to lines 77-97 to make the .pdf file. There is an R function `copy2pdf` that should make this easier, but I wasted hours trying to make it work before giving up. *Help!*