

# Doing More than Just the Basics with SAS/Graph and R

## Tips, Tricks, and Techniques

Rafe M. J. Donahue, Ph.D.

Associate Director, Statistics  
BioMimetic Therapeutics, Inc.  
Franklin, TN

Adjunct Associate Professor  
Vanderbilt University Medical Center  
Department of Biostatistics  
Nashville, TN

Version: in development  
April 2009

This text was developed as part of the course notes for the course Fundamental Statistical Concepts in Presenting Data; Principles for Constructing Better Graphics, as presented Rafe Donahue as part of the American Statistical Association's LearnSTAT program in April 2009 (and possibly beyond).

This document was prepared in color in Portable Document Format (pdf) with page sizes of 8.5in by 11in, in a deliberate spread format. As such, there are "left" pages and "right" pages. Odd pages are on the right; even pages are on the left.

Some elements of certain figures span opposing pages of a spread. Therefore, when printing, as printers have difficulty printing to the physical edge of the page, care must be taken to ensure that all the content makes it onto the printed page. The easiest way to do this, outside of taking this to a printing house and having them print on larger sheets and trim down to 8.5-by-11, is to print using the "Fit to Printable Area" option under Page Scaling, when printing from Adobe Acrobat. Duplex printing, with the binding location along the left long edge, at the highest possible level of resolution will allow the printed output to be closest to what was desired by the author during development.

Note that this document is in development, implying that the author is aware of the very high possibility of there being a plethora of typos, grammatical errors, misspellings, and whole-things-wrong. Gently alerting the author (via email at [rafe.donahue@vanderbilt.edu](mailto:rafe.donahue@vanderbilt.edu) or in person) of needed changes will greatly improve upcoming versions. Thank you.

“How do you make those graphs? What software do you use? What options do you use to get the software to put all those things on the page? My graphs from my stats package don’t look like that. I know what I want to do but I just don’t know how to do it; how can I get these stat packages to give me what I want?”

After the 2008 JSM course, I got lots of questions like these; people want to know how to make software make good graphs. This document is an attempt to answer some of those questions.

This document will present some examples of building graphics in SAS/Graph and R, two systems I have worked with for years. My conjecture, although I cannot say it with certainty, is that other statistics packages have similar functionality. Making them do what you want them to do requires patience and time and lots and lots of reading. There is no magic bullet.

We’ll start with drawing a plot with SAS/Graph and then modify its defaults and make it better. Along the way I will discuss issues that will arise with how the code runs and how SAS works and whatnot.

Then we’ll start over and do the whole thing all over again with R.

Some notes are in order before we start.

1. I do statistical programming batch mode. What that means is that I do not believe in dragging and dropping, clicking and pointing, in GUIs, or the mouse. I am old-school. I use a vi clone text editor and submit code via line commands or the “batch submit” command in a Windows Explorer pull-down menu.

Whether I am programming in R or in SAS or in whatever, my general M.O. stays the same: a window for editing, a window for submitting code, a window for checking some kind of log, a window for viewing results, and whatever other windows are needed for other output (eg, a pdf file reader). I find trying to remember the current state of things in an interactive session too difficult; I make mistakes and overwrite data sets or data frames. I forget that I have set or unset options or parameters.

So I program in batch and work iteratively: write, save, submit, examine; write, save, submit, examine; write, save, submit, examine. Lather, rinse, repeat.

In general this affords me a clean slate each time I run a piece of code and makes my programs more repeatable.

You do what you like. If you can make the interactive GUI work, more power to you. I have trouble with it so I don’t do it. I work a lot faster in batch.

2. What I have to say is most likely correct but certainly not complete. What follows is a way that I work. There are likely better and more efficient and easier ways to do all these things. I am not the final authority on things SAS/Graph and R. Learning these things is a life’s work. What I have to present is what works for me, now, here. What I present will be what I understand to be true; the future might prove me wrong or point out that my understanding or mental model is not quite right. So be it.

What this means is that I don't want to spend lots of time entertaining things like "Did you know that you can do it this way?" or "Did you know that you can do it that way?". Either (a) I did know and decided not to discuss it or (b) I didn't know and thus didn't put it in. Either way, it's not in here.

I'm certain that almost all of this can be improved or changed for a *particular* purpose. I am seeking *understanding* so as to all these tools to be used for *many* purposes.

3. The SAS code I am running is Version 9.1.3, service pack 4, running on a Windows server. In batch mode. The R code is release 2.8.1, running on a Windows XP Dell laptop. In batch mode. It is Spring in Tennessee, I'm snacking on Harvest Peach yogurt and wearing a grey hoodie.

4. Examples are cut-and-pasted from my programming and output editor windows and formatted here in Adobe InDesign. I take the liberty of not providing absolutely everything in the output (like line numbers or time and date stamps) or program (like inconsequential options). I'll make every effort set off the code and output to make it look like code and output but I likely will not always modify the typeset form of keywords and such when they fall in the text.

What this means is that in the text I will write things like "this came from `proc print`" or "this is the output from the `order` function" instead of "this came from `proc print`" or "this is the output from the `order()` function". That might happen in a future iteration, but I don't see it happening now. Context should aid in resolving any ambiguities of language.

5. I don't use many R packages or functions or SAS macros that I have not written myself. So that means that most of you will see will be very basic code. Fancy is nice but if you don't understand what the fancy is allowing you to avoid, it is hard to understand the value.

Furthermore, just because an R package or SAS macro exists on the web doesn't mean that it really does what it is supposed to do. My general principle with statistical software is to stay as close to the base functionality as possible. You may disagree but that won't change what you see in the pages that follow.

6. I am assuming that the reader of this text has working knowledge of SAS and R at a fairly advanced level. Understanding ideas in SAS such as how the data step works or how macro variables work and ideas in R such as functions and different types of objects behave is assumed. I am not writing this to teach SAS or R; I am writing this to help SAS and R users make better graphics in these tools.

7. This is not a competition. Both of these software packages can be measured on multivariate scales and each has components where they win and lose.

My goal is to demonstrate what I have found one can do with these packages. I will not allow this to be an opportunity to bash or praise either package. I believe that seeing what both packages can do, and how they do it, can only help people using either package.

Clinical trials collect all sorts of data on the subjects enrolled in these studies. A recent study collected subject weight at the start of the trial. Our first example is going to be a graphical investigation of the empirical cumulative distribution function for the weights in the clinical trial. We'll start with SAS and then use R.

The first 10 weight observations look like this, as generated by a proc print:

Obs	pt	weight
1	41S01	186
2	41S02	215
3	41S03	265
4	41S04	190
5	41S05	265
6	41S06	168
7	41S07	180
8	41S08	240
9	41S09	233
10	41S10	190

There are 334 observations in the data set. I will reveal other variables at our disposal as the example unfolds but for now we have 334 weight readings and we want to draw an empirical cumulative distribution function of those weights.

R has an `ecdf` function that allows us to jump right to the plotting but, alas, SAS does not (at least not in 9.1.3; if it does, that's not a big deal, as generating the `ecdf` is rather simple). So we'll run a `proc freq` and generate the list of support points of the weight distribution and the corresponding cumulative percentages:

```
proc freq data=thedata noprint;
  tables weight / out=freqs outcum;
run;

proc print data=freqs;
run;
```

The `noprint` option to `proc freq` suppresses the printing and the `out=freqs` option in the `tables` statement generates an output data set called `freqs`. The `outcum` option specifies that the output data set contains the cumulative frequencies and percents, instead of just the counts at each support point. The first 10 observations look like this:

Obs	weight	COUNT	PERCENT	CUM_FREQ	CUM_PCT
1	52	1	0.29940	1	0.299
2	58	1	0.29940	2	0.599
3	59	2	0.59880	4	1.198
4	65	1	0.29940	5	1.497
5	67.1	1	0.29940	6	1.796
6	71	4	1.19760	10	2.994
7	75	1	0.29940	11	3.293
8	77	1	0.29940	12	3.593
9	78	1	0.29940	13	3.892
10	79	1	0.29940	14	4.192

What we will want to plot will be the cumulative percentages (`cum_pct`) as a function of the weights (`weight`). We'll want to use the SAS/Graph functionality; the corresponding procedure is called `gplot`.

Of course, before we get started, we have to set up all the graphics stuff. I'm sure this is the first place that people struggle with graphing software. With either SAS/Graph or R, we are subject to the defaults, which we may or may not like. We want to be able to control what we are getting, so we will specify things directly. We could just submit the proc gplot SAS code and hope for the best but we don't want to do that. We're grown-ups now, we want to understand what is going on.

The relevant SAS code that I used looks like this:

```
options papersize=(4.125 3.000);

goptions dev=sasprtc
        target=sasprtc
        gunit=pct
        ftext="Times New Roman"
        htext=10 pt
        ;

ods pdf file="f:\sas_data\testing\sasplotsforcourse\plot01.pdf"
        notoc bookmarkgen=no bookmarklist=none contents=no;
```

The options statement tells SAS that I want the output on paper 4.125 inches by 3.000 inches. (I want it to fit nice on these pages. No sense making them too big.)

The goptions specify all sorts of graphics options for SAS. dev= specifies the device driver. Keep in mind that statistical graphing programs do not make graphs; they generate codes that devices use to make graphs. Thus, SAS/Graph does not make graphs but, in this case, generates code that a pdf viewer (eg, Adobe Acrobat) can use to make a graph. The dev parameter tells SAS/Graph what device we will be using to eventually make the graphic.

Our device, sasprtc, is the SAS default color printer. This is what the SAS Output Delivery System (ODS) likes to use.

The target parameter is a trick that SAS uses to make one device generate a graphic as it would appear on another device. In our case, these are the same, but if we want to see how our graphic would look on, say, an HP pen plotter, we would tell it that the target was the pen plotter but the device is the sasprtc device.

gunit specifies the units we will use to specify dimensions. We are going to use percent of the graphics area but we could use inches or centimeters or points.

ftext says we will use a Times New Roman font for our lettering and htext says that those letters will be, by default, 10 points high; we will use 10-point Times lettering.

The ods statement says that we will generate pdf output and route it to the file specified by the file= option. We do not want a table of contents (notoc), we want no bookmarks generated, we want no bookmarklist, and no contents. I like my plots clean.

So, here, finally, is the plot command:

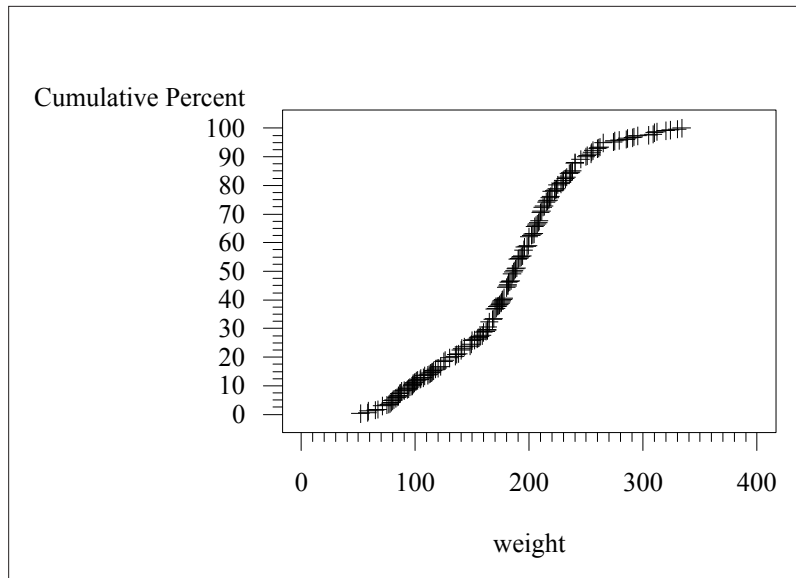
```
proc gplot data=freqs;
  plot cum_pct*weight;
run;
quit;
```

The gplot procedure uses the freqs data we generated before and requests a plot of cum\_pct as a function of weight. The run statement submits those statements for processing and then we follow it with a quit statement. Why?

The answer is that I really don't know, but I think of the quit statement as a statement that kind of flushes all the stuff out of the system. In interactive SAS, I'm pretty sure that you can submit multiple plot requests ("plot y\*x; run; plot w\*x; run; plot effect\*cause; run;") without having to resubmit the proc gplot statement. The quit, in essence, kills the proc gplot that is running. In general, it is a good idea to put quit in places where things need to be done before moving on, certainly if there are input/output tasks (like writing graphics to file). If you really want to know the full answer, call the SAS technical support people. I'm sure they'll tell you. But whatever it means, we're going to use it here.

[One note here is worth pointing out: In SAS' proc gplot, we plot by saying "plot y\*x", listing the vertical component first and then the horizontal. The way to remember that is to note that y axis is typically to the left of the x axis and hence comes first. In R when doing general plotting, the standard setting is plot(x,y), listing the independent variable first followed by the dependent variable.]

After running that code, we look at the resultant pdf file and see the image shown here. I added the line around the outside of the plot so that we can see the boundaries of what the software produced. Note that it is exactly 4.125 in by 3.000 in (unless it has been scaled when printing). But wow, that's not particularly pretty.



Well, these are just the defaults. It appears that the default plotting symbol is a little cross and that there is some algorithm for determining the axis values and the tick marks. The gplot routine didn't know this was supposed to be an empirical cumulative distribution function so it didn't know to make the connected step function line.

Ok, it is what it is. It is far from perfect but we can make adjustments. Let's see how we can improve it. It is really not very hard.

The first step will be to improve the axes and the plotting symbol. To do that we specify axis statements and a symbol statement:

```
axis1 order=(0 to 350 by 50);
axis2 order=(0 to 100 by 20);

symbol1 value=point interpol=stepj1 color=blue;
```

The axis1 statement defines an axis that we will use for the weight values on the horizontal axis. The axis2 statement will be used for the cumulative percents. There are other options available other than the order option but we won't use any right now. One thing at a time. Of course, all of these options are explained in excruciating detail in the documentation.

Note that just specifying the definitions of the axes will not put them on the plot. We will need to tell proc gplot to use them when plotting the data.

The symbol statement says that we want the thing that we are drawing to be a blue point and the interpolation between the data points (the line that connects the data points) should be a step function, vertically joined, with the data point on the left edge of the step.

This `interpol` option is really powerful, as it can be used to do more than just connect the dots. It can be used to determine how to present the y values when more than one exists for a given x value. For example, it can be used to compute a regression or to plot the high, low, and close values on each day for a stock price. There are pages and pages of examples in the documentation.

When we specify the `symbol1` statement, the ‘1’ means that this definition will be applied to the first symbol needed in each plotting request. If we would need more than one symbol, we could specify a `symbol2` and `symbol3` and so on.

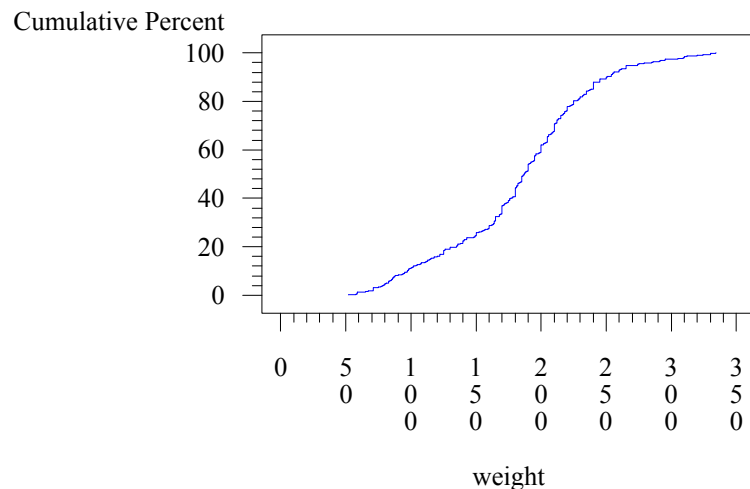
So, we submit the code like this:

```
proc gplot data=freqs;
  plot cum_pct*weight / haxis=axis1 vaxis=axis2;
run;
quit;
```

and we get the following output (I opted for no boundary line this time.).

Ok, this is better but still in need of some work. The label on the y-axis is really messing up the figure as it requires a whole ton of white space out to the left edge. And the vertical orientation of the x-axis labels is really bad. But we told the program to go from 0 to 350 by 50 with 10-point Times New Roman so it did what it was told.

Furthermore, we would like our curve to start at zero on the x-axis with a y-value of zero cumulative percent and then run along the x-axis until it hits the smallest datum. And at the top of the weight range, we want things to hang out at 100% until the edge of the plot.



To accomplish this, we are going to add those extrema points to our data set so that they can be plotted. Since we might change the low and high values of those points in the future (right now we have them set at 0 and 350; in the future they might be at 50 and 400 or something completely different, who knows?), we will practice some wise programming techniques and place them in macro variables so that we can use them where ever we see fit. (You’ll see soon why this is a good idea.)

```
%let xbottom=0;
%let xtop=350;
%let xstep=50;
```

Now we add the extra points to our `freqs` data set, putting placeholders on the top and bottom. We’ll start with the `freqs` data set and use it as the basis for the `freqsplus` data set. All we’ll do is output each observation to the `freqsplus` data set as that data step implicitly loops through the `freqs` data set. That’s the “output” statement.

But just once, we need to insert the two extra data points, or observations, into the `freqsplus` data set. To do this, we will employ the `_n_` variable that is implicit in the data step. This variable tracks the current observation being worked. When we are working on the first observation (`_n_=1`), we will coerce a weight variable value of the value of the macro variable `xbottom`, a count of 0, a percent of 0, and cumulative percent of zero, and then spit out that new placeholder observation. This will be the point that will start our ECDF at a cumulative percent of zero.

We will also generate a cumulative percentage of 100% at the value of the `xtop` macro variable. The count at this value should be zero, along with the percentage, but the cumulative count should be the same as the value of the at the final `x` value. Since I have been working with this data set, I know that the cumulative percentage here is 334 but that's really a dangerous programming practice. In reality, I should compute and use the cumulative frequency from within the program itself. I'm being lazy here, and it could bite me in the end.

```
data freqsplus;
  set freqs;
  output;
  if _n_=1 then do;
    weight=&xbottom; count=0; percent=0; cum_pct=0; output;
    weight=&xtop; count=0; percent=0.000; cum_freq=334; cum_pct=100; output;
  end;
run;
```

The first ten observation of the `freqsplus` data set look like this:

Obs	weight	COUNT	PERCENT	CUM_FREQ	CUM_PCT
1	52	1	0.29940	1	0.299
2	0	0	0.00000	1	0.000
3	350	0	0.00000	334	100.000
4	58	1	0.29940	2	0.599
5	59	2	0.59880	4	1.198
6	65	1	0.29940	5	1.497
7	67.1	1	0.29940	6	1.796
8	71	4	1.19760	10	2.994
9	75	1	0.29940	11	3.293
10	77	1	0.29940	12	3.593

Note that observations 2 and 3 are the ones we added. Before we plot them, we are going to want to resort them in order from small to large weight, as we are connecting them. If we were to leave them in this order, the line would start at a weight of 52, jump back to 0, climb to 350, and then go back to 58. This would not be good, so we will sort by weight (and then `cum_pct` just in case).

```
proc sort data=freqsplus out=freqsplus;
  by weight cum_pct;
run;
```

[There really is no end to the way one can add these observations to the data set. One could build a separate data set and append it and then sort. One could add a grouping variable to the data set and then use `first.` and `last.` to drop the variables in the right spot. Some kind of merge might work. `Proc sql` is an option if you know how to use it. Heck, one could export the data set to a spreadsheet, type the values in, and then read it back in. The point here is that if we want our plot lines to extend to both edges, we need to add those points.]

We need to update the `x-axis` definition to include the new range of values based on the macro variables. We will also force the axis to draw only one minor tick mark between the major ones that exist at the multiples of 50, and we will specify a label.

```
axis1 order=(&xbottom to &xtop by &xstep)
  minor=(number=1)
  label=("Weight (lb)")
;
```

We also need to improve the y-axis label. We will place the label at an angle of 90 (rotated counter-clockwise) to the default position of horizontal. And we are going to scrap the minor tick marks all together here. There is no need to repeat the symbol definition but it is nice to have it nearby, so we will restate it here for our own purposes. And then we will submit the proc gplot request.

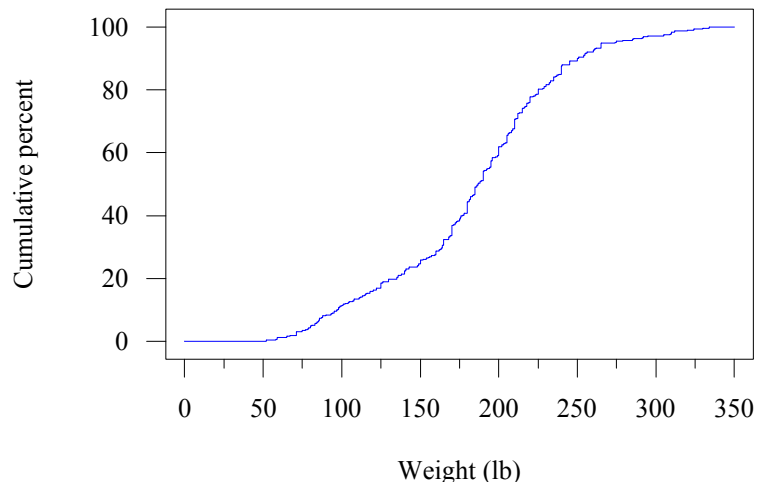
```
axis2 order=(0 to 100 by 20)
      label=(a=90 "Cumulative percent")
      minor=none
      ;

symbol1 value=point interpol=stepj1 color=blue;

proc gplot data=freqsplus;
  plot cum_pct*weight / haxis=axis1 vaxis=axis2;
run;
quit;
```

The result is an improvement. Turning the label vertical has helped with the cramped spacing, although we generally are opposed to forcing the user to turn his or her head to read text on our plots. We may still change this. Note the single minor tick mark between the major ticks on the x-axis and the absence of minor ticks on the y-axis. The line joining the points now extends all the way down to zero at the left and up to 100% at the right.

Now we can see some other details that we might or might not like to change. Note that there are no reference lines in case we would want to estimate, say, the 80th percentile point. And there is space between the edge of the plotting region and the terminal points on the axes. That is, there is space to the left of 0 and to the right of 350 on the x-axis, space that is mirrored on the vertical axis as well.



There is an option in the axis statement (we won't be using it here yet but I thought I would tell you anyway) called offset. In the axis statement you specify something like "offset=(0.25 in, 0.25 in)" to specify that you want a quarter-inch space on the edges.

Sometimes offset is good, like if you are worried that some plotting element might otherwise fall outside the region but other times it can be a nuisance. Regardless, let's see what else we can do.

I like light-colored backgrounds to frame displays, instead of a line around the outside, so the next step is that we will add that coloring and some subtle reference lines in the background.

The first step will be to define some colors. SAS/Graph and R come preloaded with lots of colors but I like to define my own because I am a control freak. One way to specify colors is to use the RGB color scheme. In both SAS and R, colors can be specified by using a six-digit hexadecimal code. Each level of the code corresponds to one of  $16^6=16,777,216$  colors. The mapping scheme works as follows.

Two digits each are assigned to the basic colors red, green, and blue. Mixing these colors completely and evenly gives white. Turning all these colors off gives black.

The values of each of the pairs can run from 00 to ff. Since these are hexadecimal, 0 through 9 map to 0 through 9 and then a, b, c, d, e, and f map to 10, 11, 12, 13, 14, and 15, respectively. Therefore, ‘00’ means  $0*16 + 0 = 0$ , ‘40’ means  $4*16 + 0 = 64$ , ‘b6’ means  $11*16 + 6 = 182$ , and ‘ff’ means  $15*16 + 15 = 255$ .

The three pairs are concatenated as rrggbb, so the first pair is the red component, the middle pair is the green component, and the last pair is the blue component. So, if we want white, we need red all the way on, green all the way on, and blue all the way on. So, white will be ‘ffffff’. Similarly, pure red is ‘ff0000’, pure green is ‘00ff00’, and pure blue is ‘0000ff’.

I like greys and since greys are balanced colors across red, green, and blue, all three of the components being equal is going to give me shades of grey. I will need two greys for background of the plot and some reference lines, so I am going to use ‘d8d8d8’ for light grey and ‘b0b0b0’ for dark grey. It is left as an exercise for the reader to show that light grey is actually lighter than dark grey.

SAS/Graph requires us to prefix a ‘cx’ to the front of our color specifications, (R, we will see, will require a different prefix), so our macro variables will look like this:

```
%let ltgrey = cxd8d8d8;
%let dkgrey = cxb0b0b0;
```

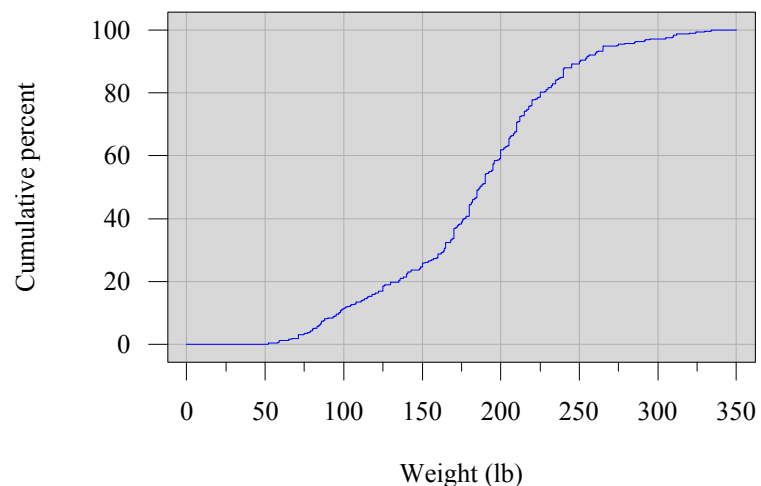
There is a functionality in the plot statement that allows us to draw horizontal and vertical reference lines at all the major tick marks by specifying “autohref” and “autovref”. We can specify the color of the authorefs and autovrefs with “cautohref” and “cautovref”, respectively. There is also a specification for coloring the background of the frame by specifying a cframe= option.

```
proc gplot data=freqsplus;
  plot cum_pct*weight / haxis=axis1 vaxis=axis2
                        autohref autovref
                        cautohref=&dkgrey
                        cautovref=&dkgrey
                        cframe=&ltgrey
                        ;
run;
quit;
```

Sweet! Now that looks like a real graph. In the end, we might worry about the line weights and such, but this is pretty nice.

But, perfect is the enemy of the good, so we should see what else we can do. I really don’t like the boundary on the box being in black while all the other plot frame components are in lighter greys. Come to think of it, *I really don’t need the tick marks and boundary lines at all.* I going to try to remove them.

There is an option in the axis statement called “style”. One can specify a number from 0 to 46 for the type of line to



draw for the axis, with 0 being no axis line. Furthermore, unless the `noframe` option is set, the style number impacts the entire frame around the plotting area.

So, our axis and `gplot` statements now look like this:

```
axis1 order=(&xbottom to &xtop by &xstep)
      minor=(number=1)
      label=("Weight (lb)")
      style=0
      ;
axis2 order=(0 to 100 by 20)
      label=(a=90 "Cumulative percent")
      minor=none
      style=0
      ;

proc gplot data=freqsplus;
  plot cum_pct*weight / haxis=axis1 vaxis=axis2
                       autohref autovref
                       cautohref=&dkgrey
                       cautohref=&dkgrey
                       cframe=&ltgrey
                       ;
run;
quit;
```

And the plot resultant plot is on the right.

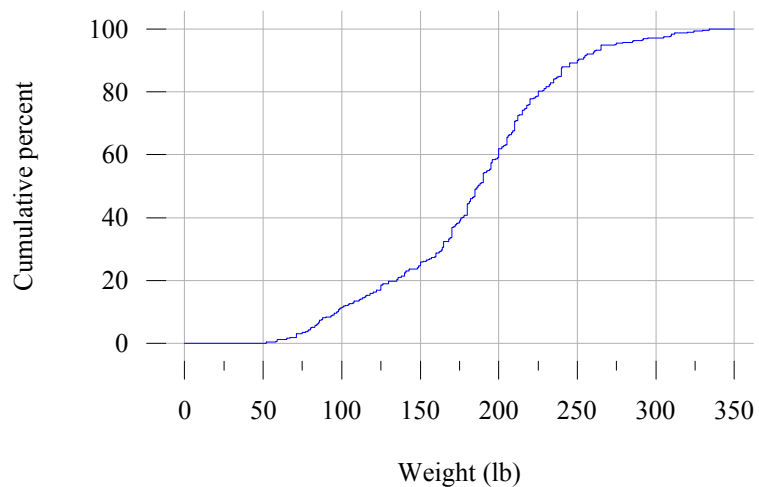
Well, that certainly wasn't expected. We got rid of the axis lines but the other things hung around. The tick marks are still in black and are still redundant. Our grey frame has disappeared, probably because there is no box to contain it. Hmmm. What to do?

What we can do is to turn off the major and minor tick marks completely and then change the axis color to the light grey, the same color as the background. This will create the illusion of there being no boundary.

We still want the label in black and the numbers indicating the reference lines in black so we will specify those directly.

The new axis statements are this:

```
axis1 order=(&xbottom to &xtop by &xstep)
      minor=none
      label=(color=black "Weight (lb)")
      major=none
      color=&ltgrey
```



```

        value=(color=black)
    ;
axis2 order=(0 to 100 by 20)
      label=(a=90 color=black "Cumulative percent")
      minor=none
      major=none
      color=&lt;grey
      value=(color=black)
    ;

```

Notice that we stated that the color of the entire axis should be our light grey macro value. (Wasn't that a good idea to put that in a variable? Now if we need to change it, we only change it in one place!) Since we still want the values on the axis black, we have specified the color in the value parameter. Inside of those parentheses could go oodles of descriptors for the values including fonts and sizes and all manners of pathology. We're not going to do that, but the documentation has all sorts of fun things that can be done.

Submitting the code with those axis statements gives us the next adaptation of the plot:

```

proc gplot data=freqsplus;

    plot cum_pct*weight / haxis=axis1 vaxis=axis2
                        autohref autovref
                        cautohref=&dkgrey
                        cautovref=&dkgrey
                        cframe=&lt;grey
                        ;

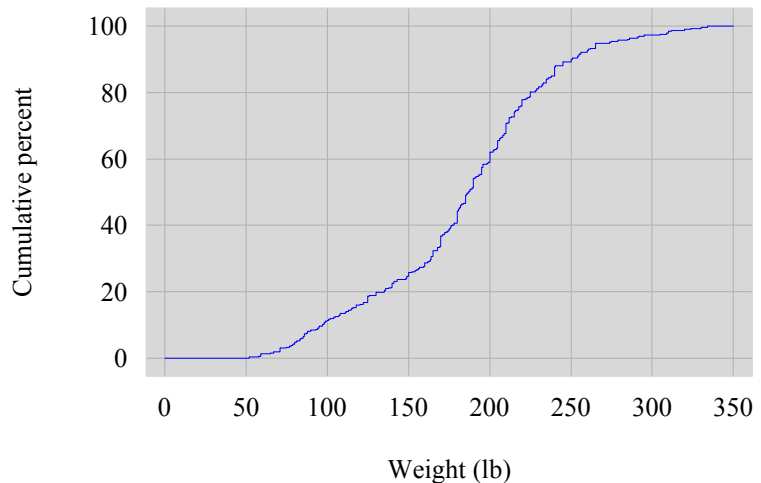
run;
quit;

```

Ok, very nice. But now we can see that the offsets on both of the axes are too big, as there is all that extra space around the outside.

We'll add an offset specification to the axis statements and rerun the code.

Note that all of these iterations we are making are trial and error steps. Our goal is understanding the distribution, in this case shown by the ECDF. If we start looking at the data in the plot, we see (and have seen for some time) that there are two relatively uniform parts to the distribution of weights. The first part runs from approximately 50 to 160 pounds, and the second runs from approximately 160 to 250 pounds. Above 250 pounds we have some right skew in the data. Remember that in a cumulative distribution function, steepness equals dataness. The steepness is relatively constant from 50 to 160 and again from 160 to 250.



Looking closely, however, we can see some steep spikes in the curve, probably falling on multiples of 5 and 10 pounds, evidence of point mass at these weights, which in turn is evidence of digit preference. As we continue to work with these data, we will investigate these fundamental components of the distribution and their sources of variation.

So, the new axis statements with the offset and the proc gplot:

```
axis1 order=(&xbottom to &xtop by &xstep)
      minor=none
      label=(color=black "Weight (lb)")
      major=none
      color=&ltgrey
      value=(color=black)
      offset=(0,0)
      ;
axis2 order=(0 to 100 by 20)
      label=(a=90 color=black "Cumulative percent")
      minor=none
      major=none
      color=&ltgrey
      value=(color=black)
      offset=(0,0)
      ;

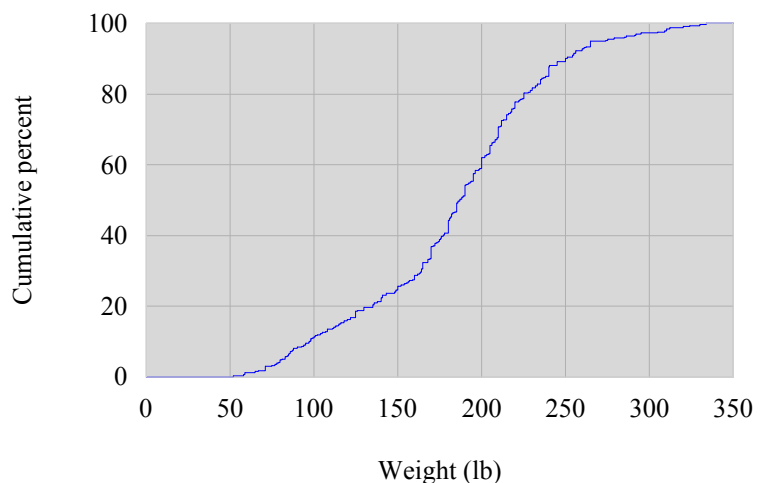
proc gplot data=freqsplus;
  plot cum_pct*weight / haxis=axis1 vaxis=axis2
    autohref autovref
    cautohref=&dkgrey
    cautovref=&dkgrey
    cframe=&ltgrey
    ;

run;
quit;
```

And the plot, at the right.

Well, the offsets are now zero, but there is one big problem: our data are obscured by the light grey boundary around the plotting frame. So, that's not good. We don't ever want to hide the data with plot residue.

Well, let's try the style=0 trick again to see what it will look like; the axis statements get even bigger:



```
axis1 order=(&xbottom to &xtop by &xstep)
      minor=none
      label=(color=black "Weight (lb)")
      major=none
      color=&ltgrey
```

```

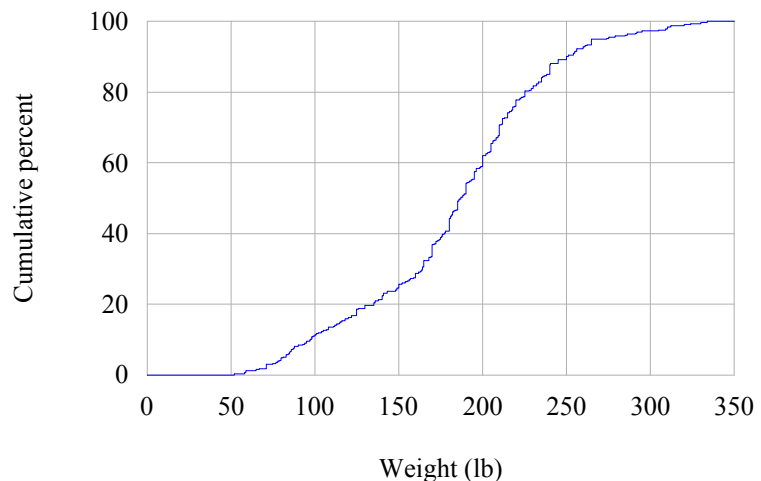
value=(color=black)
offset=(0,0)
style=0
;
axis2 order=(0 to 100 by 20)
label=(a=90 color=black "Cumulative percent")
minor=none
major=none
color=&lt;grey
value=(color=black)
offset=(0,0)
style=0
;
proc gplot data=freqsplus;
  plot cum_pct*weight / haxis=axis1 vaxis=axis2
                        autohref autovref
                        cautohref=&dkgrey
                        cautohref=&dkgrey
                        cframe=&lt;grey
                        ;
run;
quit;

```

And the plot is not what we want, of course. since the colored frame has disappeared.

At this point, we have several choices, with one of them the obvious “one of these plots should be good enough; we’re done.” But why should we do what the software allows instead of what we want to do? We were supposed to have had flying cars by the year 2000 and we don’t. Cannot I get a plot that does what I want it to do?

The answer is that, yes, I can get what I want. I just need to do something more primitive and sneaky. The trick will be to employ the annotate facility.



The annotate facility can be used to do all sorts of really fancy things, but most of the uses I have seen have focused on just annotating plots. For example, perhaps we want to take the plot we just produces and put an annotation on it; perhaps we just want to write “Steepness equal dataness” in the upper left corner. We could use annotate to do that. And, from what I have seen, that is pretty much the only sort of thing that people do with the annotate facility.

The smoke-trails plot on page 50 of Fundamental Statistical Concepts was done entirely within the annotate facility because getting regular SAS/Graph to do what I wanted typically got me really close, but not quite there. (Relax, we will see similar things with R, and functionality similar to annotate.)

The fundamental strength of annotate is that it employs a “go here, do this, in this way” mentality. What we will do is create a data set that contains not data *per se*, but instead *instructions* for where to go on the plot, what to do when there, and how to carry it out. And all those computations and such will be based on the data.

So, asking ourselves what it is we want from the graphic now, we simply want SAS to go to the origin (“go here”) and then draw a rectangle to the upper left corner (“do this”) and make sure the rectangle is light grey and behind the other graphic elements (“in this way”). How hard could that be?

We will build an “annotate data set” that will contain these commands. An annotate data set has to have a number of predefined variables that will tell SAS the go here, do this, in this way instructions. Then, when we call the proc gplot routine, we will tell SAS the name of the annotate data set to use with the plotting step.

First, the data step to construct the data set and then the explanation:

```
data makebk;
  length function style color $ 8 text $ 25;
  retain xsys ysys hsys `2';

  function="move";
  x=&xbottom; y=0;
  output;

  function="bar"; x=&xtop; y=100;
  color="&lt;grey"; line=3; style="solid";
  when="b";
  output;
run;
```

What is the data step doing?

The data makebk statement is just naming the data set. I am calling it makebk; makebk stands for “make background”.

The length function is assigning lengths to some character variables. The variables function, style, and color have length 8 and the variable text has length 25. The function variable will hold an instruction, either a “go” or “do” instruction. The style and color variables will be part of the “in this way” component.

The retain statement specifies three character variables (xsys, ysys, and hsys) and assigns them each the value of ‘2’. I will talk more about these later but note that they will have the value of ‘2’ on ever observation in the data set.

The function=’move’ statement assigns the variable function the value ‘move’. So we are telling the program to “go” somewhere. And where would that be? That is answered in the next line where we assign a variable x the value of xbottom and a variable y the value 0. We are tell the program to go to the origin. The output statement writes that observation to the data set makebk.

The next line defines the variable function to be “bar”; after the previous observation moved to the origin, this observation will draw a “bar”, essentially a rectangle. Where will the opposite corner of that rectangle be? It will be at the new values of x and y: xtop and 100, the top right corner of our plot.

The next line defines a color (light grey), a way to draw the border around that bar (line=3 means no line), a style for filling the bar (make it solid, as opposed to empty or something else).

The when=’b’ assignment tell SAS to make sure it does all this stuff before doing the other graphing stuff so that the other elements will be place on top of our little grey rectangle. The other options is ‘a’ for ‘after’, which means it will put it on top of the other graphic elements. We don’t want that.

Then we output the second observation and we are done!

The annotate data set looks like this:

Obs	function	style	color	text	xsys	ysys	hsys	x	y	line	when
1	move				2	2	2	0	0	.	
2	bar	solid	cx8d8d8		2	2	2	350	100	3	b

The first observation has the move value for the function and the x and y values of 0. The second observation has our “draw a solid light grey bar to the point (350,100) and make it without a border and do it before drawing the other graphic things” command. (Note that we didn’t specify anything for the text variable. There is a function value that we could assign called ‘label’ that would use that argument. I just like to keep those variables around if I need them.)

An important thing to note about annotate data sets is that if an observation possesses a function that does not use a specific variable, then that variable is ignored for that function. So, since we didn’t need the text variable for our move or bar commands, we could have put “padding” into the text variable and it wouldn’t have made any difference.

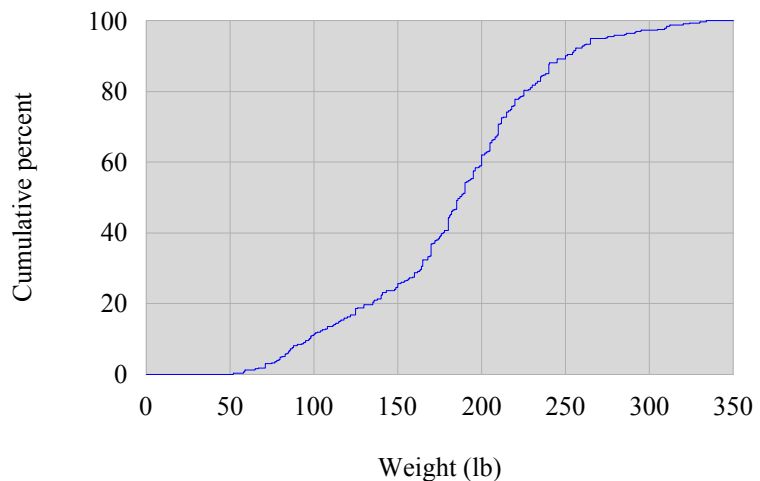
Furthermore, there can be other variables in the data set and if the annotate facility doesn’t need them, they are ignored.

So, to use the annotate data set that we just built, we just specify it with the plot command in the gplot statement.

```
proc gplot data=freqsplus;
  plot cum_pct*weight / haxis=axis1 vaxis=axis2
                        autohref autovref
                        cautohref=&dkgrey
                        cautovref=&dkgrey
                        cframe=&ltgrey
                        anno=makebk
                        ;
run;
quit;
```

Our annotated plot is at the right. Note that the data line now sits atop the grey box! Whee!

With proc gplot, there are two places we could have put the anno= parameter. It can be placed with the procedure options in the proc gplot line, along with the data=freqsplus option. If so, this annotate data set would be applied to all the plots that would come out of this procedure until we kill the procedure with the run; quit; at the end. The way we have placed the anno= options, as an option to the plot statement, it will only be applied to this plot command. Recall that we could tell the gplot procedure to run multiple plot statements. This multiple annotating allows one lots of control over what gets added to which plots, a very powerful feature.



A word before we go further (and we *will* go even further): I never explained the xsys, ysys, and hsys variables. These variables tell annotate which general area to use for plotting.

There are three areas of interest in a SAS/Graph plot. The largest area is the “graphics output area”. This includes everything on the space assigned for the graphic. A subset of the graphics output area is the “procedure output area”, an area where the procedure writes things. Note that the difference between the graphics output area and the procedure output area is essentially a border around the procedure output area where things like footnotes and titles are placed. Within the procedure output area is the “data area”, a subset that corresponds to the plotting region.

Each of these region can be accessed relative to exact coordinates or percentage coordinates and the xsys and ysys variable specify which system is in use. We used ‘2’ for both the xsys and ysys variables meaning we were going to be addressing the data area and we were going to use the same values as specified for the axes. Had we used ‘1’ for xsys and ysys, we could have specified the units in percentage of the axis. Thus, if we had set xsys and ysys to ‘1’, we could have gotten the same grey background box by setting the first x and y to 0 and 0 and the second x and y to 100 and 100.

If we would want to put things outside the axes, we could have used ‘3’, which means “percent of area for procedure output area”. Note that (0,0) for this coordinate system is a point below and to the left of the plot and axes.

There are a total of twelve coordinate schemes, six for the absolute specification of values for the cross of percent or specified units with the three output areas and six for the relative specification of values across that six-tuple. Relative units mean that we don’t tell annotate the absolute position, we tell it the relative position given the current state.

All of this is in the documentation, of course.

So, now that we know so much about annotate data sets, let’s get greedy. We are going to add some content to the plot from the last run by adding little bars along the bottom that show the raw count of subjects at each support point. The support points are in the freqs data set; all we need to do is tell annotate how to plot the points.

Here is the data step code to generate the extra bars; explanation will follow:

```
data extrabars;
  set freqs(keep=weight count);

  length function style color $ 8 text $ 25;
  retain xsys ysys hsys `2' when "b";

  barwidth=1.00;

  x=weight-(0.5*barwidth); y=0; function="move"; output;
  x=weight+(0.5*barwidth); y=count; function='bar';
  style="solid"; color="red"; line=3; output;

run;

data bkandbars;
  set makebk extrabars;
run;
```

At the beginning we have the name assignment (extrabars) as we had before but in this data set we will start with the freqs data set, since that one has the weights and counts. We keep the weights and counts, just to keep things neat.

The length and retain statements are as before so that needs no explanation.

Next is a barwidth definition. We set barwidth equal to 1.00 for starters because we might want it different later. We will use this number to specify how wide each little bar is at each point. You will see in the next two statements how

we will use `barwidth`.

For each element, call it  $s$ , of the support set, we will make a bar that has lower left value at  $(s-1/2, 0)$  and upper right value at  $(s+1/2, \text{count}(s))$ , where  $\text{count}(s)$  is the count of subjects at that weight. Since we have specified `barwidth`, we can easily change this from  $\pm 1/2$  to  $\pm$  anything, in case we might want, say, some horizontal spacing between the vertical bars. As it stands now, there should be no spacing between individual bars that differ by one pound, as they both extend exactly to the midpoint.

By this time, you know how to read the data step code for building the `annotate` data set. We set function and location of the left bottom and tell `annotate` to go there ('`move`'), then we set the function and the upper right location and tell `annotate` to make a solid, red, non-outlined bar.

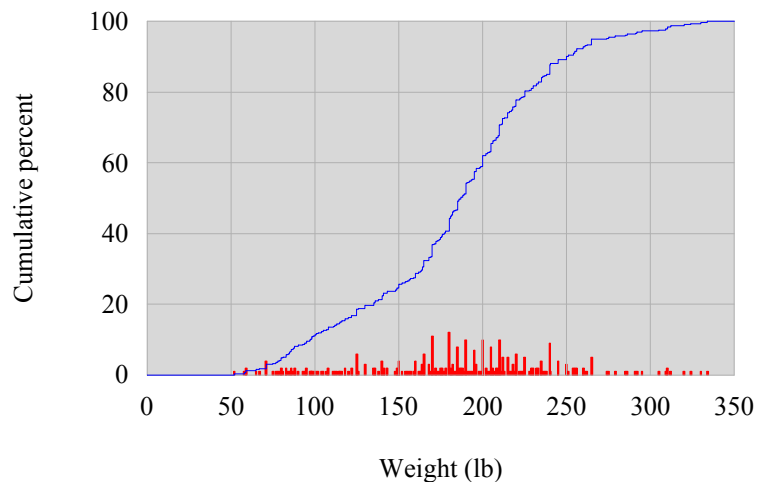
Easy. Done.

The data step that follows simply concatenates the two `annotate` data sets together. Then we run the code as before.

```
proc gplot data=freqsplus;
  plot cum_pct*weight / haxis=axis1 vaxis=axis2
                        autohref autovref
                        cautohref=&dkgrey
                        cautovref=&dkgrey
                        anno=bkandbars
                        ;
run;
quit;
```

Is that cool or what!?! Now we have both the ECDF and the support set, along with the counts on the bottom. And now we can see the digit preferences for the pleasing multiples of 5 and 10 in the red spikes.

If we look carefully, we can see one tiny problem: the reference lines actually got placed on top of our support set bars. This is not good as the data should sit atop the graphic structural components.



And further, note that the reference lines at the edge of each axis are really unnecessary, as the edge of the light grey frame tells me I am at the boundary.

So, let's redo the reference lines so they sit behind the red spikes and only install lines for the interior references. We'll also add some reference lines to aid in counting the mass at each support point.

We will also kill off the vertical axis label and put that information in the title as we now have two things being plotted at each weight value.

The final code follows.

```
data reflines;
  length function style color $ 8 text $ 25;
```

```

retain xsys ysys hsys `2' when `b";

do x=50 to 300 by 50;
  y=0; function="move"; output;
  y=100; function="draw"; color="&dkgrey"; size=1; output;
end;
do y=5 to 15 by 5, 20 to 80 by 20;
  x=&xbottom; function="move"; output;
  x=&xtop; function="draw"; color="&dkgrey"; size=1; output;
end;
run;

data annods;
  set makebk reflines extrabars;
run;

axis1 order=(&xbottom to &xtop by &xstep)
  minor=none
  label=(color=black "Weight (lb)")
  major=none
  color=&ltgrey
  value=(color=black)
  offset=(0,0)
  style=0
  ;

axis2 order=(0 to 100 by 20)
  label=none
  minor=none
  major=none
  color=&ltgrey
  value=(color=black)
  offset=(0,0)
  style=0
  ;

symbol1 value=point interpol=stepj1 color=blue;

title h=10pt
  c=blue "Cumulative percent "
  c=black "and "
  c=red "raw counts "
  c=black "for patient weights"
  ;

proc gplot data=freqsplus;
  plot cum_pct*weight / haxis=axis1 vaxis=axis2
    anno=annods
    ;
run;
quit;

```

Some items are noteworthy. First, we used a new value for the function variable in the data set reflines. The draw command draws a line from the most recent x and y values to the specified x and y values. Furthermore, we introduced the size variable. In the context of the draw command, size tells how wide to make the line.

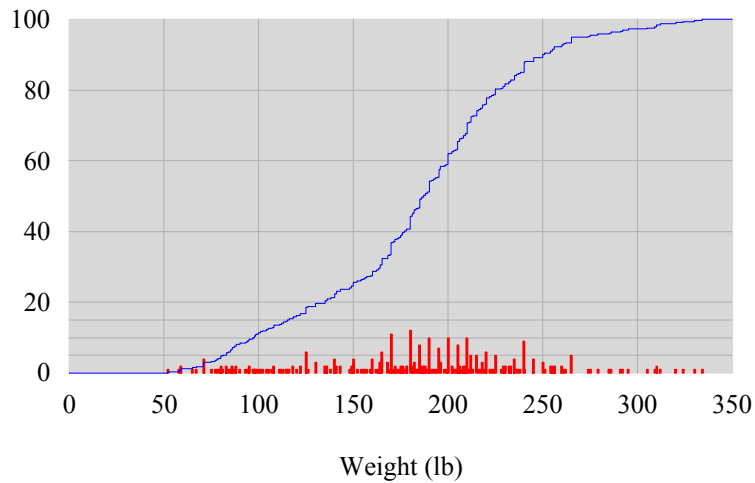
When we constructed the complete annotate data set with the “set makebk reflines extrabars” data set, we placed the three individual annotate data sets in an order that would draw the grey rectangle first, followed by the reference lines, followed by the support points. And all this drawing took place prior to drawing the actual plot.

We used a title statement that assigned different colors to the different words. We did this to provide color cues for the different things plotted on the graphic.

Lastly, we removed all those automatic reference lines since we didn’t need them.

And here is our plot.

Cumulative percent and raw counts for patient weights



Let's make the same plot with R. While in SAS, we needed to compute the cumulative frequencies and then make the plot of the empirical cumulative distribution function, in R there is a function in the stats package that will create an ecdf object for us; calling the plot function on this ecdf object will produce the default ecdf plot. While this seems like it will be the perfect situation, we will find that this built-in function might be considered a blessing and a curse, a double-edged sword, or your favorite good-and-bad cliché object.

The data exist in a comma delimited file, so we will read them in with a read.csv function:

```
thedata=read.csv(file="exampledata.csv"
                 ,header=TRUE
                 ,as.is=TRUE
                 ,sep=",")
)
```

While technically this has nothing to do with the graphic itself, I included this function in the conversation to point out the as.is parameter. This parameter prevents the default behavior of converting certain character variables to factors. Since the pt id data are character data, they will be read in that way.

Now, you might like to read in your character data as factors; I don't. If you do like to do that, then leave out the resetting of as.is. I don't really care. But just be careful because we might need those pt id values as characters sometime.

Calling the print function shows us the data frame we have created:

	pt	weight
1	41S01	186.00
2	41S02	215.00
3	41S03	265.00
4	41S04	190.00
5	41S05	265.00
6	41S06	168.00
7	41S07	180.00
8	41S08	240.00
9	41S09	233.00
10	41S10	190.00

We are really just interested in the weight values; the pt id values are just indexing the weights. Drawing the default plot will be easy once we set up the file to receive the plot.

Again, we like pdf output, so we will use the pdf function:

```
pdf(file="u:/my documents/jsm2008/updatesandcode/rplotsforcourse/plot01.pdf"
    ,height=3.000
    ,width=4.125
    )
```

We specify here the output file for the graphic and the height and width in inches. Note that we are using the same dimensions as we did in the SAS example so that we get a small plot that will be easy to fit on these pages.

[We could specify other devices here, like bitmap devices jpeg, png, and bmp, or something vector-based, like postscript. We will not be pursuing these other devices at this time but may discuss differences between bitmaps and vectors later on.]

We also need to set some graphics parameters, like we did in SAS with the goptions. In R we use the par function. The

par function specifies oodles of graphics parameters; we will specify only 4 here at the outset:

```
par(family="Times", font=1, cex=10/14.4, las=1)
```

What we have set here is the type family (Times) and have specified the basic font (font=1), instead of something bold (font=2) or italic (font=3) or bold italic (font=4). The las=1 setting specifies the style of the axis labels; the one means that they are always horizontal, so our readers do not have to tilt their heads.

We also scaled all the text and symbols (cex=10/14.4) by 10/14.4, or approximately 69.44%. Why would we do such a thing?

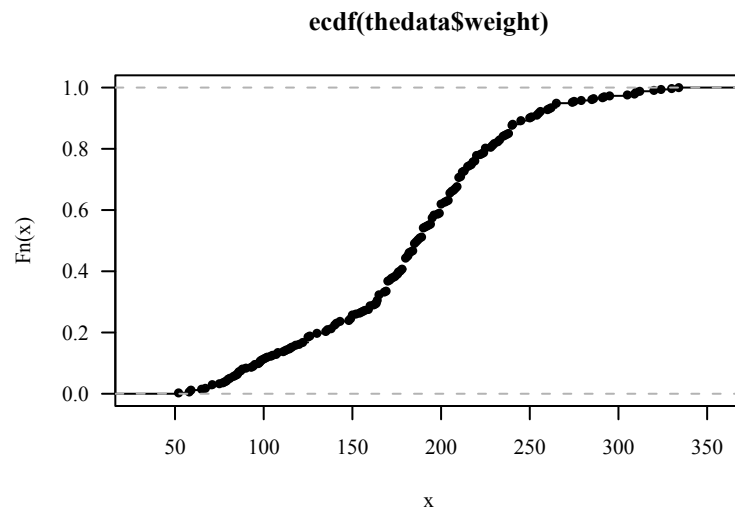
One can print the current values of the par parameters with print(par()). Doing so will reveal that the default character height "cin" was 0.2 inches. At 72 points per inch, that gives us the 14.4 points, the reading of the parameter "cra". Unfortunately, this parameter is read-only. Hence, the cex magnification parameter can be set to 10/14.4 so that when we multiply the cra value by the cex value, we get 10 points. Thus, this way we get an approximate 10 point Times font.

Now we can run the plot of the ecdf of the weights

```
plot(ecdf(thedata$weight))
```

and we get the plot at the right.

And that's certainly fine for a default plot. But, because we can, we are going to do some modifications. We have several things to accomplish. We are going to get rid of the default graphic symbol and replace it with a tiny dot. We are going to force the horizontal axis to run from 0 to 350, as we did before. We are also going to force the ecdf curve into a step function. As it stands now, each step simply hangs in space, secured to the actual datum at the left end of each step. We want real steps; should we be forced to march up the curve, we don't want to risk falling through.



[Note that the defaults in R with the ecdf automatically add the boundary points that we needed to add manually in doing the SAS version we did previously. This benefit comes at the price, though, of having to deal with those extra points when we only want the support points when we will draw the bars at the end.]

To accomplish this, we can use the following arguments to the plot function:

```
plot(ecdf(thedata$weight)
     ,xlim=c(0,350)
     ,ylim=c(0,1)
     ,pch="."
     ,verticals=TRUE
     )
```

Here the xlim provides a vector that specifies the minimum and maximum values for the horizontal axis. The ylim does the same for the vertical axis. (Note here that R is plotting the cumulative distribution function on a 0-1 scale.)

In the SAS example, we plotted percentages from 0-100. (Tomato-tomahto.) The `pch` parameter specifies the plotting character. The `“.”` specification makes a really tiny dot, essentially as small as the device can go. (Exact details are in the help files.) The `verticals=TRUE` option is what gives us support in the steps.

It is important to note here that not all these options are available in all calls to the plot function. Since the `ecdf` is an `ecdf` object, the generic plot function calls the `plot.ecdf` function. The `plot.ecdf` function, in turn, is actually a wrapper based on the `plot.stepfun` function. Exactly which options work with which function is sometimes a matter of trial and error.

Our cleaned up plot is shown at the right.

This is much nicer, as the plot symbols are out of the way. And the steps won't let us fall through.

But like we did before, we are going to spend some time cleaning up the axes, modifying tick marks and labels and such. We will also make the `ecdf` line blue.

The first thing to do will be to define some parameters that we will use more than once, namely the horizontal axis parameters. Like we did with the macro variables in SAS, we define

```
xbottom=0
xtop=350
xstep=50
```

and also set

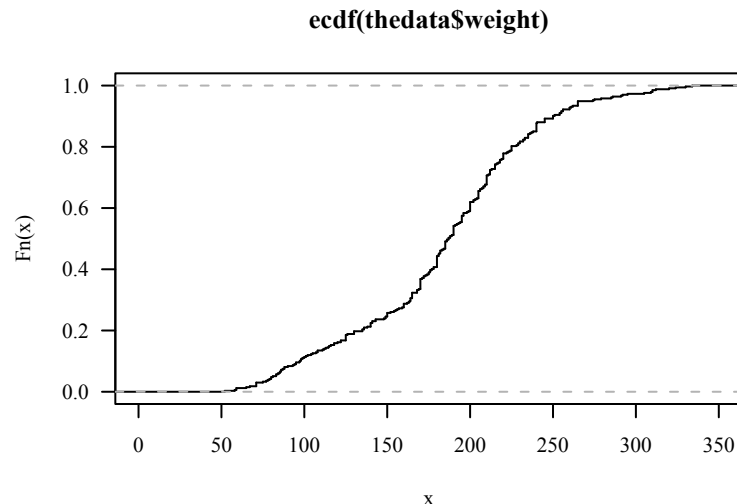
```
par(col="blue")
```

so that the default plotting symbol color will be blue. Then we will call the plot function as before but with some modifications. We will also specify some modifications of the axes and the text in the margins:

```
plot(ecdf(thedata$weight)
     ,xlim=c(xbottom,xtop)
     ,ylim=c(0,1)
     ,pch="."
     ,verticals=TRUE
     ,axes=FALSE
     ,xlab=""
     ,ylab=""
     )
```

```
axis(side=1, at=seq(from=xbottom, to=xtop, by=xstep) )
```

```
mtext(side=1
      ,text="Weight (lb)"
      ,line=3
      ,col="black"
```



)

```
axis(side=2, at=seq(from=0, to=1, by=0.20) )
```

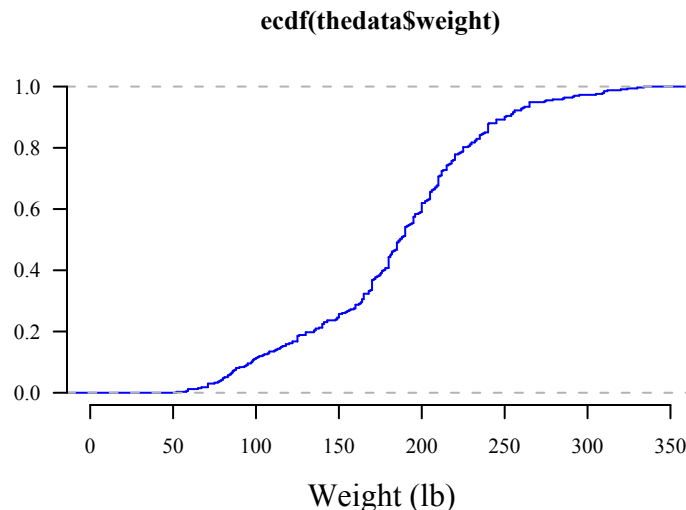
To the plot function we added the axes option, along with xlab and ylab. The axes=FALSE option turns off the automatic production of the axes so that without further instructions, we wouldn't get any axes at all. The xlab and ylab values are both set null, eliminating any labelling text from the horizontal and vertical axes.

We take over control of the axis production through the two axis function calls. Some of the parameters to the axis function are "side", which can take values 1, 2, 3, and 4 (corresponding to the bottom, left, top, and right axes), and "at", which specifies locations for the tick marks. Here we used the xbottom, xtop, and xstep values we defined earlier and also specified the breaks for the vertical axis, even though they are the same as the values that were determined by the automatic algorithm used for determining the ticks. Specifying this directly helps us remember that we are controlling the tick locations; we will not be subject to the whims of the computer should we try different data or different plot sizes.

Also present is the mtext function, a function that puts text in the margin. We tell it the side (side=1 implies the bottom axis), the text we want written, on which line we want that text (line=3), and the color. We explicitly specify the color here since we have been playing around with other color settings in the par; if we fail to specify a color here, R will use the default color value from the par function, which we indicated as blue. If we want black text here now, we need to declare it.

These functions produce the plot shown at right. Note that, compared to the previous plot, the color of the curve is now blue and that the labels have been changed on the axes. Also note that since we turned off the default axes, the prison surrounding the data field has been opened and the offsets on the axes is evident.

We still face the default, not-particularly user-friendly title, and we still don't have our grey background. Also, the plotting routine has drawn grey dashed horizontal lines at 0 and 1 on top of our data; somewhere we will have to remove those.



Let's define our light grey and dark grey colors, as we did in the SAS example. RGB colors in R can be indicated with a six-digit hexadecimal code; however, in SAS the colors are prefixed with "cx" while in R such colors are prefixed with "#".

There is an option in the par function called "bg", which stands for background. Setting bg equal to a color makes the background that color. The default is "transparent" (not "white"). Transparent colors will allow other things beneath them to show through, should this graphic ever be placed in a situation (like this document!) where there may be other visual elements on the page.

There is also a function called grid, which draws a grid at the tick marks. We modify our code, by adding the definitions of our colors and by calling the par and grid functions (and also condense it somewhat for sake of vertical space [In general, I like things with one options per line so as to make it easier to comment out a single option. Feel free to program as you like.]) and rerun our plot:

```
ltgrey="#d8d8d8"
dkgrey="#b0b0b0"

par(bg=ltgrey)

plot(ecdf(thedata$weight)
     ,xlim=c(0,350), ylim=c(0,1)
     ,pch=".", verticals=TRUE
     ,axes=FALSE, xlab="", ylab="")

grid(col=dkgrey, lty=1)

axis(side=1, at=seq(from=xbottom, to=xtop, by=xstep))

mtext(side=1, text="Weight (lb)", line=3, col="black")

axis(side=2, at=seq(from=0, to=1, by=0.20))
```

Well, rats; that's not what we wanted. First of all, the `bg` parameter in the `par` function made the entire background colored, not just the data field. Second, our grid is placed on top of the data.

Furthermore, the offset now obviously needs fixing and the line weights are too thick all around.

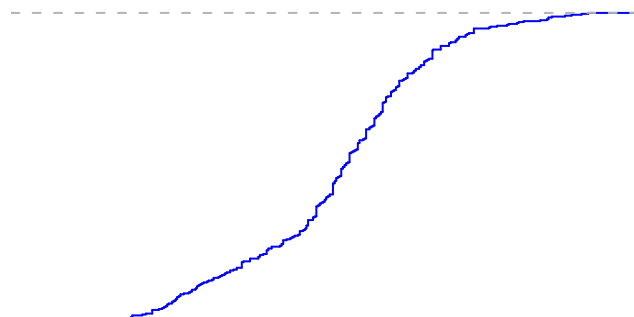
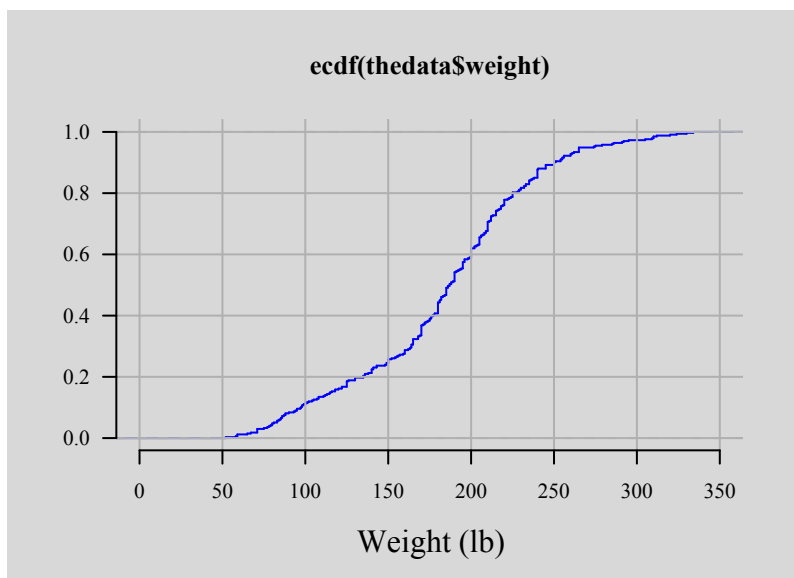
What we need to do is to draw the graphic elements in the right order, starting at the background and working our way to the data on the top.

So, let's do just that. We will reset the `bg` parameter in the `par` function and then build up the plot bit by bit finally getting what we want.

```
par(bg="transparent")
```

Now draw a plot with just the data, no axes, no labels, no annotations of any kind. We do this with the `axes=FALSE`, the `xlab` and `ylab` specifications, and the `ann=FALSE` option. `ann=FALSE` tells high-level plotting functions to not annotate plots with axis titles and overall titles.

```
plot(ecdf(thedata$weight)
     ,xlim=c(0,350)
     ,ylim=c(0,1)
     ,pch="."
     ,verticals=TRUE
     ,axes=FALSE
     ,xlab=""
     ,ylab=""
     ,ann=FALSE)
```



produces only a bare-bones plot with the data and the reference lines.

This, then, is our starting point.

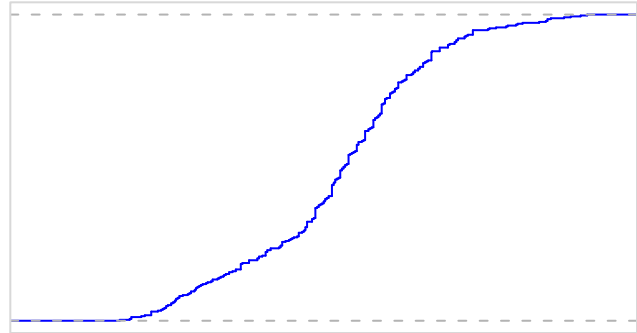
To this starting point, we will add the light grey background box, by using the box function. The box function “draws a box around the current plot the in the given color and linetype.”

Following the plot call, we will call the box function:

```
box(which="plot", col=ltgrey)
```

which produces a light grey box around the “plot” area. (The other options available are “figure”, “inner”, and “outer”.)

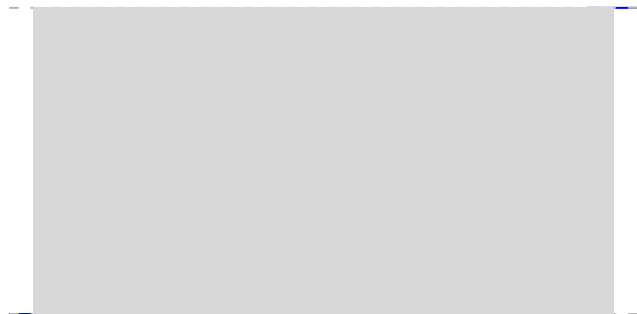
Hmmm. That’s not quite right either. We want a shaded box and it is not obvious to me how to do that with the box function.



We will scrap the box function and instead run the bare plot followed by the rect function. The rect function draws a rectangle at the specified coordinates and colors or shades it as desired.

```
plot(ecdf(thedata$weight)
     ,xlim=c(0,350)
     ,ylim=c(0,1)
     ,pch="."
     ,verticals=TRUE
     ,axes=FALSE
     ,xlab=""
     ,ylab=""
     ,ann=FALSE)
```

```
rect(xleft=xbottom
     ,xright=xtop
     ,ybottom=0
     ,ytop=1
     ,border=NA
     ,col=ltgrey)
```



produces the grey rectangle on top of the bare-bones plot. Note that the reference lines at 0 and 1 are visible where they extend into the horizontal axis range. There are also remnants of the ecdf line itself protruding outside the rectangle. Note that we specified “border=NA” in the rect function, meaning that we didn’t want a border. Generally, I’m not a believer in borders since we can see where the edge of the data field is when light grey turns to white.

Even though this is not optimal (we will clean up those extraneous lines later), we can add the grid and axes and margin text as before

```
grid(col=dkgrey, lty=1)
```

```
axis(side=1, at=seq(from=xbottom, to=xtop, by=xstep))
```

```
mtext(side=1
      ,text="Weight (lb)"
      ,line=3)
```

```
, col="black"
)
```

```
axis(side=2, at=seq(from=0, to=1, by=0.20))
```

and produce the next plot.

Note that we still have the heavy lines which need to be lightened and the axes need to be softened overall. We will remove the axis lines by specifying a style for each axis line, and make the width of the lines zero, just for good measure.

To do so, we add `lty=0` and `lwd=0` options to the axis statements, indicating the line type of zero (“blank”) and a line width of zero.

Redrawing with these adjustments, we start again with the raw plot, then the rectangle, and then the extra things:

```
plot(ecdf(thedata$weight)
     ,xlim=c(0,350), ylim=c(0,1)
     ,pch=".", verticals=TRUE
     ,axes=FALSE, xlab="", ylab=""
     ,ann=FALSE)
```

```
rect(xleft=xbottom, xright=xtop, ybottom=0, ytop=1, border=NA, col=ltgrey)
```

```
grid(col=dkgrey, lty=1)
```

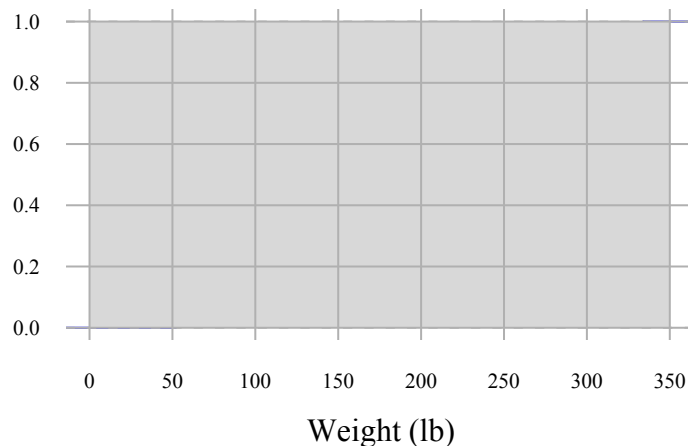
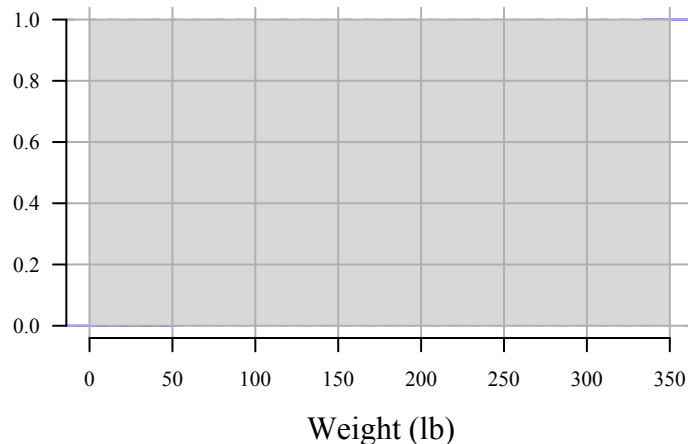
```
axis(side=1, at=seq(from=xbottom, to=xtop, by=xstep)
     ,lty=0, lwd=0)
```

```
mtext(side=1, text="Weight (lb)", line=3, col="black")
```

```
axis(side=2, at=seq(from=0, to=1, by=0.20)
     ,lty=0, lwd=0)
```

giving us a much cleaner plot. I find it remarkable how much cleaner we can make our plots simply by removing all the accoutrements that the default settings place near our data. Of course, we have covered our data; at the end of getting all the ducks in a row with respect to the background information, the last thing we will do is plot the data themselves.

In order to remove the offset that floats around the edges of our data field and makes our plot area look like a fraying



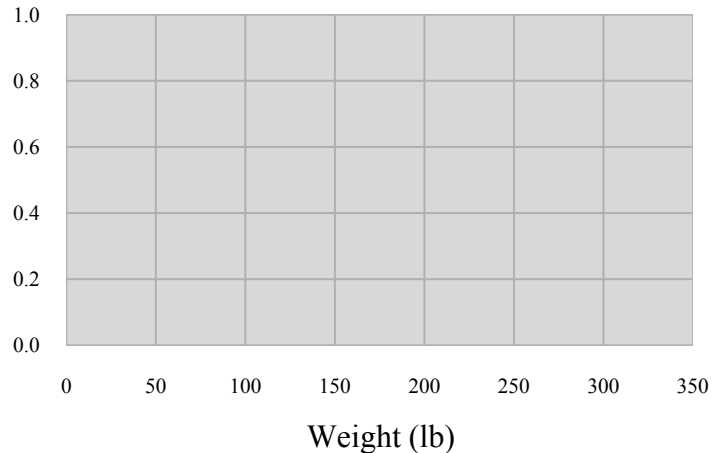
rug, we will adjust some parameters found in the par function. These parameters are the xaxs and yaxs parameters; they control the “style” of the axes.

The default style for the axes is “r”, for regular, meaning that R calculates the range of values on the axis by extending the data range by 4 percent and then finding an axis with “pretty labels” that fits that range. A setting of “i” (internal) finds an axis with “pretty labels” that fit within the range of the data. Three other styles (“e”, “s”, and “d”) are currently not implemented. The specification in the par function is

```
par(xaxs="i", yaxs="i")
```

If we set both the horizontal and vertical axes to style “i” and then rerun the exact code we just ran, then we get the plot on the right, a plot with the offsets removed!

Note that the data and the lines from the ecdf plot are completely covered. Also, the dark grey grid extends around the outline of the light grey rectangle that we drew without a border. We will find a way to deal with this later.

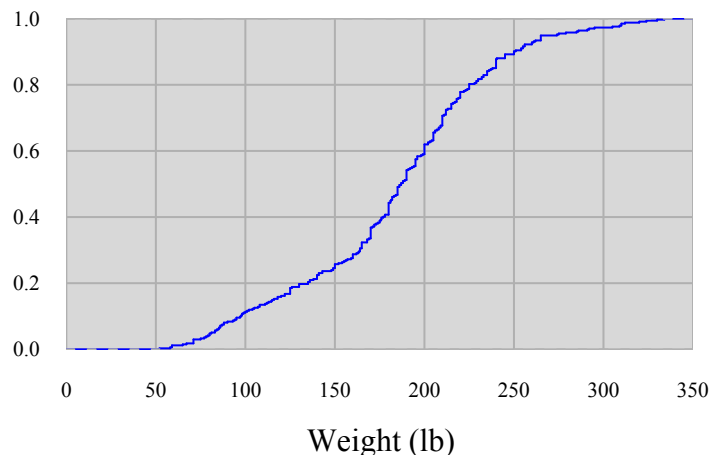


Regardless, we can now draw the data on top of the foundation we have built by rerunning the plot command; however, the plot command typically starts a new plot — what we want to do is to add the points to the current plot (again!) even though they exist in the plot, covered by our foundation.

The trick to replotting the points on the same plot is another option in the plot function: add. We will specify add=TRUE and get the points redrawn on the foundation:

```
plot(ecdf(thedata$weight)
     ,xlim=c(0,350)
     ,ylim=c(0,1)
     ,pch="."
     ,verticals=TRUE
     ,xlab="", ylab=""
     ,ann=FALSE
     ,add=TRUE)
```

Note the add=TRUE declaration in the plot function.



And this is exactly what we want, with one exception. The exception is that the reference lines in the ecdf plot are still present and they are sitting on top of our data. In order to eliminate these lines, we need to dig further into the plot function and, in particular, into how the plot function plots ecdfs.

The plot function is a generic function, meaning that it has a meaning that only becomes clear when one knows what type of object is being “plotted”. When we tell R “ecdf(thedata\$weight)” we are creating an “ecdf” object. The resulting ecdf object carries around with it little labels that tell other parts of R exactly what it is. An ecdf object is a function, but a special type of function, as well. An ecdf object actually belongs to three classes: the ecdf class, the stepfun class, and the function class.

So, when we tell R to “plot” an ecdf, it doesn’t run the plot function on that object, it runs the plot.ecdf function on that object. So, to find out what R is doing, we need to look not at the plot function, but at the plot.ecdf function.

The plot.ecdf function is actually just a wrapper function that runs two other functions in turn. The exact call of the plot.ecdf function, the function that is called when one tells R to “plot(ecdf(thedata\$weight))” is actually defined to be:

```
function (x, ..., ylab = "Fn(x)", verticals = FALSE, col.01line = "gray70")
{
  plot.stepfun(x, ..., ylab = ylab, verticals = verticals)
  abline(h = c(0, 1), col = col.01line, lty = 2)
}
<environment: namespace:stats>
```

We can see then that the plot.ecdf function is first running the plot.stepfun function and then the abline function. The abline function is specifying to draw horizontal lines of color col.01line (which is by default “gray70”) and of type 2 (dashed) at 0 and 1. Aha!

What we will do then to eliminate the horizontal reference lines is to not call the plot function (which actually calls the plot.ecdf function) but instead call the plot.stepfun function directly, bypassing the added abline function which draws the unwanted reference lines.

So, let’s build the whole thing again, adding a main title and making the grid lines and plotting lines narrower by using the lwd=0.5 option in some key locations, like the grid function and the final plot.stepfun function. The code, repeating all those par functions we have implemented, and using the plot.stepfun function instead of the generic plot function or the specific plot.ecdf function is then:

```
par(family="Times", font=1, cex=10/14.4, las=1, col="blue", xaxs="i", yaxs="i")

plot(ecdf(thedata$weight)
     ,xlim=c(0,350), ylim=c(0,1)
     ,pch=".", verticals=TRUE
     ,axes=FALSE, xlab="", ylab=""
     ,ann=FALSE)

rect(xleft=xbottom, xright=xtop, ybottom=0, ytop=1, border=NA, col=ltgrey)

grid(col=dkgrey, lty=1, lwd=0.5)

axis(side=1, at=seq(from=xbottom, to=xtop, by=xstep)
     ,lty=0, lwd=0)

mtext(side=1, text="Weight (lb)", line=3, col="black")

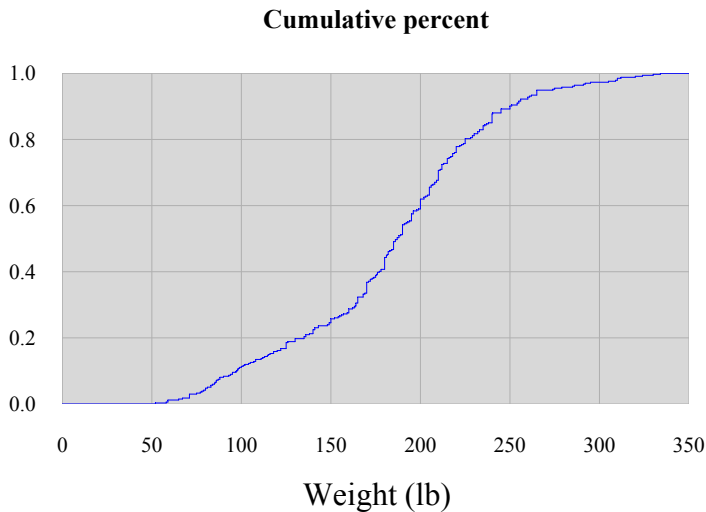
axis(side=2, at=seq(from=0, to=1, by=0.20)
     ,lty=0, lwd=0)

title(main="Cumulative percent")

plot.stepfun(ecdf(thedata$weight)
             ,xlim=c(0,350), ylim=c(0,1)
             ,pch=".", verticals=TRUE
             ,xlab="", ylab=""
             ,ann=FALSE)
```

```
, add=TRUE
, lwd=0.5)
```

which yields a plot that is just what we want:



Well, almost.

Note that the grid is creating a boundary around the light grey rectangle; since we have come this far, why not get it right? Further, we still need to add the count bars at the support points like we did in the SAS version.

In order to accomplish this, we will replace the grid function with some calls to the `abline` function, a function to which we were introduced when we investigated the details of plotting the ecdf. We are also going to change the labels on the vertical axis to reflect percentages instead of fractions. (Personally, I think it makes little difference, as any intelligent user of our graphic should be able to understand that 0.6 means 60%. But we will be adding counts and will be utilizing the same axis so it will be nice to be able to have the axis do double duty. Of course, we could have the right side do the counts and the left side do the percentages but I really don't like double axes...) We'll add the count bars with another `rect` function, but will need to get to the data.

Making our own custom grid will be easy enough. The `abline` function draws lines at, among other things, points specified on the horizontal and vertical axes. So we will remove the grid function and replace it with

```
abline(h=c(seq(from=0.05, to=0.20, by=0.05)
, seq(from=0.40, to=0.80, by=0.20)
)
, col=dkgrey, lwd=0.50)
abline(v=seq(from=50, to=300, by=50)
, col=dkgrey, lwd=0.50)
```

The first call to `abline` draw horizontal lines at 5% to 20% by steps of 5% and also from 40% to 80% by 20%. Note that we are deliberately skipping the lines at 0% and 100%, as there are color boundaries there already.

The second `abline` function draws lines from 50 to 300 every 50 units. Note that we could have computed those points with something like `xbottom+xstep` to `xtop-xstep`, every `xstep` units. But we didn't. If we were going to make this code more universal, we might do something like that. It is often nice to make things work for more than one use.

All of these gridlines will be dark grey and will have width 50% of the default.

On the vertical axis, we will still use the value from 0 to 1 but will relabel them with the `label` option. The axis state-

ment for the vertical axis will look like

```
axis(side=2, at=seq(from=0, to=1, by=0.20)
     ,labels=as.character(seq(from=0, to=100, by=20))
     ,lty=0, lwd=0)
```

We have seen the side and at and lty and lwd options before but the labels option is new. It allows us to put whatever text we want at each of the tick marks defined by the at option. Since the at option in our case is specifying six tick mark locations (we suppressed the ticks themselves), we can specify a sequence of six labels in a vector, in this case we made a character vector of the labels “0”, “20”, “40”, “60”, “80”, and “100”. The labels argument is expecting characters, hence the need for the as.character function.

A couple of short notes here. First, SAS provides similar functionality in the axis statements that we used; we just didn’t use them because we were already working on a 0-100 scale. Second, we can really be mischievous here because there is nothing stopping us from putting labels=as.character(seq(from=100, to=0, by=-20)) here; heck, we could even put labels=c(“Santa”, “pudding”, “pogo stick”, “ear wax”, “baseball”, “Ceasar”) in there if we wanted to do so. Only our integrity is keeping us from being deliberately deceitful.

Lastly, we need the count data. They exist within the ecdf object ecdf(thedata\$weight) but are rather difficult to get to. So we will just count them at each support point. As mentioned previously, there are benefits and detriments to every default designation.

The table function generates counts of the data at each support point. If we run the table function on the weight data and then turn it into a data frame, we will be able to create something we can plot.

First we will create a data frame called counts. Counts will hold the counts at each support point.

```
counts=as.data.frame(table(thedata$weight))
```

We’ll add another variable to the counts data frame. This variable will hold the horizontal values of the support points, but there is some careful trickery needed first. The generation of the counts data frame as we have defined it will have a variable called Var1 which will be the levels of the support points of the weights. This variable, however, will be held in the data frame as a factor, not as a character or a number. As such, it will really be a number from 1 to the number of support points. If we were to use the variable Var1 as the abscissa, we wouldn’t be plotting the weights, we would be plotting the ordinal value of the sorted weights; the lowest weight, 52, wouldn’t be plotted at 52 but rather at the value 1. The second lowest weight, 58, would be plotted at 2. And so on.

To work around this feature, we will create a variable that holds the numeric interpretation of the character interpretation of the Var1 variable:

```
counts$plotx=as.numeric(as.character(counts$Var1))
```

The as.character turns the factor counts\$Var1 into its labelled value; the as.numeric turns that into a number which we can plot. Printing the first 10 rows of the counts data frame shows us the data:

	Var1	Freq	plotx
52	52	1	52.00
58	58	1	58.00
59	59	2	59.00
65	65	1	65.00
67.1	67.1	1	67.10
71	71	4	71.00
75	75	1	75.00
77	77	1	77.00

```
78      78      1  78.00
79      79      1  79.00
```

We see then that we should plot `counts$plotx` on the horizontal axis and use the `counts$Freq` variable in the vertical direction. We'll use those ordered pairs to draw the count bars as follows:

```
barwidth=1.00

rect(xleft=counts$plotx-(barwidth/2)
     ,xright=counts$plotx+(barwidth/2)
     ,ybottom=rep(0,length(counts$Freq))
     ,ytop=counts$Freq/100
     ,border=FALSE
     ,col="red"
     )
```

Each bar will be one unit wide. We specify the bars as having left edges at the support point (`counts$plotx`) minus half the width of the bar (`barwidth/2`), right edges at the support points plus half the width of the bar, bottom edges at 0, and top edges at the count (`counts$Freq`) scaled by 100. Note the importance of the `rep` function in the `ybottom` option; R complains if the lengths of the `xleft`, `xright`, `ybottom`, and `ytop` options are not the same length. We force the `ybottom` to be the right length by repeating the value as many times as there are support points. Further, we scale the top of the bar values by 100 since our plot only ranges vertically from 0 to 1.

We also specify no border (do we really need to discuss this again?) and make the little rectangles red.

We are also going to modify the main title to reflect the new things in the plot.

Now our complete code looks like this:

```
par(family="Times", font=1, cex=10/14.4, las=1, col="blue", xaxs="i", yaxs="i")

plot(ecdf(thedata$weight)
     ,xlim=c(0,350), ylim=c(0,1)
     ,pch=".", verticals=TRUE
     ,axes=FALSE, xlab="", ylab=""
     ,ann=FALSE)

rect(xleft=xbottom,xright=xtop,ybottom=0,ytop=1,border=NA,col=ltgrey)

abline(h=c(seq(from=0.05,to=0.20,by=0.05),seq(from=0.40, to=0.80, by=0.20))
      ,col=dkgrey,lwd=0.50)
abline(v=seq(from=50, to=300, by=50)
      ,col=dkgrey,lwd=0.50)

axis(side=1, at=seq(from=xbottom, to=xtop, by=xstep)
     ,lty=0, lwd=0)

mtext(side=1, text="Weight (lb)", line=3, col="black")

axis(side=2, at=seq(from=0, to=1, by=0.20)
     ,labels=as.character(seq(from=0, to=100, by=20))
     ,lty=0, lwd=0)

title(main="Cumulative percent and raw counts for patient weights")
```

```

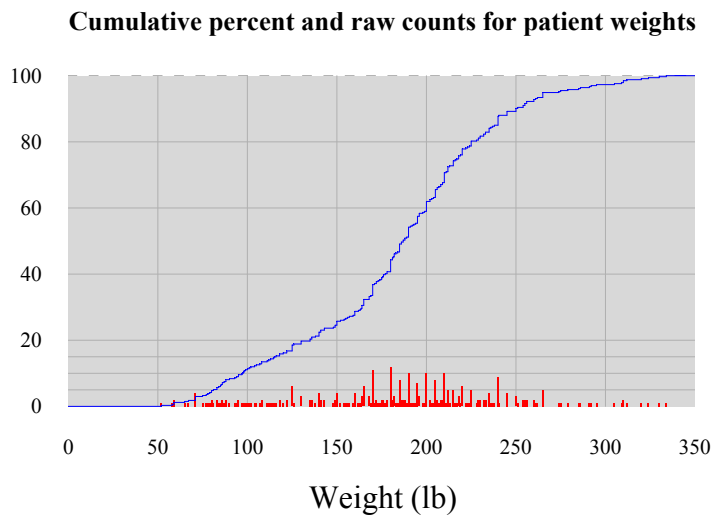
counts=as.data.frame(table(thedata$weight))
counts$plotx=as.numeric(as.character(counts$Var1))
barwidth=1.00

rect(xleft=counts$plotx-(barwidth/2)
     ,xright=counts$plotx+(barwidth/2)
     ,ybottom=rep(0,length(counts$Freq))
     ,ytop=counts$Freq/100
     ,border=FALSE
     ,col="red")

plot.stepfun(ecdf(thedata$weight)
             ,xlim=c(0,350), ylim=c(0,1)
             ,pch=".", verticals=TRUE
             ,xlab="", ylab="")
             ,ann=FALSE
             ,add=TRUE
             ,lwd=0.5)

```

which produces a plot to match the plot we made in the SAS example:



The previous two sections have walked us through making a plot and adding information. We did this first using SAS and then using R. Although these are certainly different software programs, there are a number of lessons concerning similarities and differences that should be exposed.

First and foremost in the list of lessons is the repeat of the statement that statistical graphing programs do not produce graphics; they generate codes that devices use to make graphs. These codes are read by the device drivers and turned into files that other programs turn into graphics.

That being said, understanding the final use for a graphic will dictate the driver that is used to produce it. And since the drivers do not all behave similarly, and they have different defaults with respect to numbers of pixels and pixel and graphic and textual element sizes and so forth, choice of the device driver is one of the first major decisions to make in building a graphic.

Devices fall into two broad categories based on the type of output they produce: bitmap and vector-based. [One might note that SAS/Graph has a bevy of device drivers that drive pen-plotters. I'm not sure where to classify these device drivers as bitmap or vector-based, but I probably should classify them with the vector-based drivers for reasons to be seen soon.]

Bitmap images are images whose data are stored in the form of a pixel-by-pixel color assignment. The data files that store these images contain, essentially, a list of all the pixels and what color they should be. They contain the horizontal location, the vertical location, and the color to be used at the intersection of those points. Various algorithms exist to compress this information. For example, suppose there is a bitmap image with 1,000 rows and 1,000 columns for a grand total of 1,000,000 pixels (1 megapixel). If all the pixels in the top 6 rows are the same color, say "blue", the compression algorithm might notice this and, instead of storing 6,000 pixels that all say "blue", it might say "pixels (1,1) through (6,1000) are all blue". This saves space in the image file but, as is often the case, doesn't really save tons of space, although it is considered "lossless" compression.

Other compression algorithms suffer from loss. For example, the algorithm might average the values of neighboring pixels and produce a smaller image by assigning all the pixels in this neighborhood that same average value. The important thing to remember about compression algorithms is that lossless compression algorithms lose information that cannot ever be regained.

The biggest problem with bitmap images is scalability. If an image is 1,200 pixels by 1,200 pixels, then that image is restricted to be 1.44 megapixels, regardless of the size of the pixels. Therefore, if I were to take this image and print it at 1,200 dpi (dots per inch) on a standard color printer, I would get an image approximately the size of a postage stamp. If I were to blow up this image to the size of a building, say 100 ft by 100 ft, the image would look "grainy" and "pixelated". This pixelation is a result of the stretching of the size of the pixels: 1,000 pixels spread across 100 ft requires those pixels to be each one-tenth of a foot (a deci-foot?), over an inch in width (or height).

That is a dramatic example but it becomes an issue when viewing images on computer screens, as computer screens have abysmally low resolution, typically only 72 or 96 dpi, resulting in less than 1,000 pixels, in each direction, on a

typical computer monitor. Standard television resolution is 480 lines of resolution vertically; high definition television is only 720 lines. Blue-ray generates a whopping 1080 vertical lines.

Yet, when the screen sizes get large, all of these electronic displays suffer pixelation; these displays are blown away by run-of-the-mill ink-jet printers. This is one reason that paper is such a wonderful data display medium. Of course, motion is a problem with paper; no one can deny that.

Vector-based images, however, do not suffer the problems of pixelation. Instead of pixel-by-pixel instructions like bitmap images, vector images provide a “go here, do this, in this way” instruction system, much like we saw with the annotate facility in SAS and the `rect` and `abline` functions in R.

A vector image will contain, say, an instruction to draw a red line from point A to point B. The program that renders the image then decides how to manifest that instruction into a line you and I can see; which pixels to turn on and which ones to turn off. As one zooms in with the computer using a vector image, the edges of the image remain clear and sharp, as the program is doing the pixel control of the image.

One can render vector images as bitmap images but going the other direction is difficult, as the algorithm that maps the pixels to vector instructions must make guesses. Of course, this conversion is rarely, if ever, perfect.

All of this is why I like using vector image formats, like portable document format, or pdf, and why I use these device drivers in SAS/Graph and R.

The second big lesson learned revolves around the defaults that all statistical graphics programs define. For both SAS/Graph and R, the defaults were adequate but not what we wanted. Getting the defaults to give us what we wanted sometimes was a matter of adjusting plotting options like axes and plotting regions and titles and labels.

When we wanted slightly more complicated graphics, we were forced, however, to go to more complicated utilities. In SAS/Graph, we pulled out the annotate facility. In R, we essentially covered up what R gave us and built things from the ground up.

The lesson here then, if we seek to do anything even remotely non-traditional, is to start with the idea that we will likely have to build our graphics from the ground up. Furthermore, anytime that we are thinking of make in a plot, we should start with a straight-edge and pencil mentality: how would I draw this if I only had simple drawing tools like paper, colored ink, a straight-edge, and a compass.

The value of the computer comes in doing lots of really boring, highly-detailed, highly repetitive tasks. The computer has no trouble drawing a thousand circles of varying radii at certain x,y locations and coloring them according to the group they represent or plotting time accurate to the second across the range of a year. So let's use the computer for what it does well and not let ourselves be restricted by defaults that pre-packaged chart styles demand or decree.