Axis Plotting

C. Godsalve
email: seagods@hotmail.com

February 21, 2020

Contents

1 The Problem

Axis plotting is a "simple" problem, except that it isn't that simple after all. We shall just think about the x axis in a simple plot. First, we have no idea what the x values are until we read in the data. We have no idea whether the smallest x value is -10^{34}, or 13.44, or 10^{99}. If it is 10^{99}, the largest x value could be $10^{99} + 10^{-45}$. Next, we have the problem of how to divide the x axis up into sensible and easily readable portions so that we can read off numbers. We don't want to see values like 1.3523135, 1.4234124, and such like written on the axis. Values like 1, 1.5, 2.0, make things much easier to read, and then there are the "tick marks" to boot. A well designed axis plotter makes things *look* simple and easy to read, but reading in completely arbitrary data and automatically ending up with a sensible easy to read axis isn't completely trivial.

2 Starting up

Well, to begin at the beginning, we must read in the data. The first (and easy) sub-task will be to find the maximum and minimum values. The range of the values will also be important. At any rate, on reading in the data, we shall easily find $xmin$, $xmax$, and $xrange = xmax - xmin$ (always positive). We can get the thing to exit if the range of values is zero. Of course, the order of magnitudes are completely arbitrary. In order to cope with this we need logarithms, and we shall use base 10 if we want easily understandable axes.

For the moment, we shall concentrate on the range. We shall require an integer $xexp = (int)\log_{10}(xrange)$ and a floating point scale factor $xfact = 10^{xexp}$. But, what do we mean by (int)? In a programming language, this will normally mean the "integer part" of, so $(int)0.1$ or $(int)0.99$ would both be zero. Do we want this, or do we want something else? It is a matter of choice really, we shall choose the "nearest integer lower than" via the C++ $floor$ function.

So, we shall have floating point values of $xexp$ and use the C++ $floor$ function (which returns double or float). We use $floor$ rather than casting to integer, because this neatly sidesteps the following problem. If xrange is 4, say, then $\log_{10}(4) = 0.602$, and $(int)\log_{10}(4) = 0$. Then the scaling is unity. However if $xrange$ were 0.4, then $\log_{10}(0.4) = -0.398$, the integer part of which is also zero. As we make the transition from numbers larger than one to less than one, there is a sudden step-shift needed. We have to subtract one from the log of a number less than one if we are to get the right order of magnitude for scaling. Though this is a minor inconvenience the C++ $floor(x)$ function returns (as float or double) the largest integral value not greater than x. We now have our scale factor $xfact$, so the values of $(x - xmin)/xfact \to (xmax - xmin)/xfact$ are always between 0 and something between 1 and 10.

There are other choices to be made. First, we choose our x axis end-points $xleft$ and $xright$ as they shall appear on the screen. These shall act as defaults which might (but probably won't) be overridden. The next decision we make is to have the x axis range from $xmin$ to $xmax$ so that the data coincide with the axis end-points. (We could force the x axis endpoints to be "nice numbers" and allow the data points to mismatch the axis, our present choice means that there will be nice numbers at the major tick-marks, and the tick-marks won't necessarily match the axis endpoints.)

3 Nice Numbers

So, what about these "nice numbers"? We shall start off by just assuming that $xmin = 0$. We shall think about offsets later. We know that $xrange\_scale = xrange/xfact$, will be

anything from 1 to 10. We don't want the numbers to get too crowded, and we shall deem 10 numbers to be too many. So, if $xrange\_scale$ is anything up to and including 1.4 we shall choose steps of 0.2. The first step up goes from here til $xrange\_scale$ is 2.0, where we use steps of 0.25. Steps of 0.5 shall be used if $xrange\_scale$ is larger still, up to and including 3.5. Steps of 1.0 will be used for still larger $xrange\_scale$, up to and including 8.0, and steps of 2.0 will be the largest step size. Occasionally we shall have as many as nine numbers on the axis, and sometimes as few as five.

With this sorted, we need to start thinking about $xmin$ and $xmax$. First we introduce some booleans which indicate whether the number is greater than or less than zero (true for greater than zero). These shall be $xmin\_zero$, $xmax\_zero$, and $xboth$. The latter shall be true if both $xmin\_zero$ and $xmax\_zero$ are both true *or both false*. If $xboth$ is false, then the y axis shall cut across the x axis. The booleans with the trailing $_zero$s are true if the quantity is *greater than* zero. On top of these we shall need $xmin\_scale = xmin/xminfact$ and $xminexp = (int)floor(\log_{10}(xmin))$ and $xminfact = 10^{xminexp}$.

To see why we need these, consider data sets varying from 1234543.005 to 1234543.2005., -7000.1 to -6999.0, and 3.17827 to 3.6. In the first case, $xrange = 0.2$, $xexp = -1$, and $xfact = 0.1$, with $xrange\_scale = 2.0$. Now, $xmin = 1234543.005$ and $xminexp = 6.0$. Now, we don't want numbers like 1234543.25 printed on our axes. The numbers would be too long, and overlap (or have to be printed so as to be too small to read). What we want to do instead is label the x axis as $x - 1234540.0$ and have the axis numbers as 3.25, 3.5, 3.75, ..., so how do we do that automatically? In the second case, we want to label the axis as $x + 7000$ and have numbers 0.2, 0.4, 0.6, 0.8, and 1.0. In the last case, we just want to label the x axis as x, and our numbers will be 3.2, 3.3, 3.4,... and so on.

This is where $xminexp$ comes in. If $xminexp$ is two more than $xexp$, we shall add or subtract. Suppose we subtract, how do we automate what number to subtract? We have $xexp = -1$, and $xminexp = 6$. The difference is seven, which tells us that the numbers are far too long to print. We shall only allow a difference of two before adding or subtracting from the x axis label. So, $xmin\_scale = 1.234543005$. We multiply this by $10^{xminexp-xexp-2}$ to get 123454.3005. Why the minus two? Because we start adding and subtracting when $xrange$ is more than two orders of magnitude less than $xmin$. Taking the $floor$ of this gives us 123454, and multiplying by 10 gives us the number $xsubtract$ to subtract. The same shall apply to the second set, except we must take a bit of care with the signs.

As well as $xsubtract$, we may need to multiply (or divide) the numbers on the axis. We might read in $xtext$ to describe what the data on the x axis means, and $xunits$. Suppose we have "resistance" in "ohms", but the data may be in tens of thousands. The actual axis text will be modified by an internal string $xtextmod$ so we shall see "resistance/1e4 ohms" written on the axis, and the axis numbers will be 1, 2, 3 and 4 (for instance).

In the last set, there is no addition or subtraction as $xmin = 3.17827$ is only one order

of magnitude larger that that of $xrange = 0.42173$ with $xrange\_scale = 4.2173$. This gives us (according to our rules given above) steps of 1.0 when scaled, or 0.1 when scaled back again. We divide $xmin$ by this step size to get 31.7827, and this time we use the C++ *ceil* function to give us 32, and scale back again to 3.2. So, the numbers on the axis will be 3.2, 3.3, 3.4, 3.5, and 3.6, we shall call these values xtick. The tick-marks get placed at $xleft + (xright - xleft) * (xtick - xmin)/xrange$.

So, we have arrived at how to automate what numbers to print on the axis, and where to print them and the major tick-marks. The major tick-marks have scaled intervals of 0.2, 0.25, 0.5, 1.0, or 2.0. Minor tick-marks shall either split up each major tick-mark interval into four (for 0.2 or 2.0), or five for the others.

4 Some OpenGL and C++ Details

We are using OpenGL, and in general we might need our axes, and the text that goes with them, to be rotated, or zoomed in. and so on. To do this we use <*gltt*>. That is openGL's utilities for using <*truetype*> fonts. So, somewhere along the line, we include all the headers with lines like

```
#include "/usr/include/gltt/FTBitmapFont.h"
```

— just find the directory where these headers are, and include the lot. If you can't find the headers, that means that gltt has not been installed. Then we need stuff like

```
FTFace myface;
int iface=myface.open("/usr/share/fonts/TTF/times/Timeg.ttf");
GLTTFont  font1(&myface);
```

So, we declare a freetype font and call it font1. We open the font telling it what truetype font to use, and call the constructor (GLTTFont). Of course, you need the truetype fonts installed, and need to know where they are on your system. As well as GLTTFont, there are GLTTOutlineFont, GLTTPixmapFont, and GLTTBitmapFont. All this is for the header files. In the programme you need to create the font with a pointsize to tell OpenGL what size it is. For this we have something like

```
int pointsize=20;
font1.create(pointsize);
```

To use the font, we use openGL's translate and rotate to get where we need to be, then

```
font1.output(Xstring.c_str);
```

This immediately brings us to the next bit. What exactly is $Xstring$? This will be a C++ string object, and not just a C string. The C++ string object has various member functionns. GLTT's font output needs a C string, and the C++ string object has a member function $c\_str()$ that returns just such a C string.

Now, for axes, we generally need to put numbers into the C string objects. To do this, it is convenient to use the C++ standard library string-stream. This acts in a manner that is similar to iostream for standard input output, and fstream for handling files. So, we need a #include <sstream>, and then

```
ostringstream Xout;  //declare an output stringstream
Xout << setprecision(3) << scientific;
     //sets the precision and tells it to use scientific
     //notation in subsequent calls
Xout  << x;   // a floating point x is sent to Xout;
x_string=Xout.str();   //Xout.str() returns a C++ string representation of x
font1.output(x_string.c_str());   //  x_string.c_cstr() returns a C string
```

So all this gives a fairly easy way to get a truetype font set up, use a string stream to get numbers into strings and format them, and get the font to output.