Graphs in Max

This note demonstrates a few techniques for making graphs with Max.

Gathering Data

The web is a great source of data about all manner of things, but there is a current (and understandable) trend to present data in graphical and even interactive forms. If we want to make our own presentations, we need the original, raw data, which can take a lot of snooping. We are looking for text data, which will usually be in a table such as figure 1.

```
Total Precipitation         12-Month Period Ending in Month 12
    YEARS : 1890 - 2015
   AVERAGE           21.204
   SIGMA (RMS)        7.102
   COEFF OF VAR       0.335
   SKEWNESS           0.730
   MEDIAN            19.930
   MAXIMUM VALUE     44.110
   MINIMUM VALUE      7.060
   NUMBER OBS          117.
 YEAR 1896. VALUE =   20.010
 YEAR 1897. VALUE =   24.960
 YEAR 1898. VALUE =   13.050
 YEAR 1899. VALUE =    7.060
 YEAR 1900. VALUE =   20.830
 YEAR 1901. VALUE =   19.840
 YEAR 1902. VALUE =   19.620
 YEAR 1903. VALUE =   20.460
 YEAR 1904. VALUE =   20.310
 YEAR 1905. VALUE =   24.080
```

Figure 1. Source: http://climate.umn.edu/doc/historical.htm

This is going to require some work before we can use it, but it's not as bad as some. This preparation is best done in a text editing program like TextWrangler[1]. Fancy editors like word cannot be counted on to provide plain text documents that we can load into Max. This is what must be done:

- Remove all headers—everything that is not the data. (Keep a complete copy though.)
- Remove unnecessary columns. In this case, a find and replace all operation was sufficient, but sometimes you have to resort to GREP[2]. It is possible to just ignore unneeded columns in our Max patch, but the size of these files often becomes an issue.

[1] Free from http://www.barebones.com/products/textwrangler/

[2] GREP stands for Global Regular Expression Print, a scheme for finding text based on format rather than content. TextWrangler does GREP, one of its many charms.

The final version of this data looks like figure 2.

```
1896.    20.010
1897.    24.960
1898.    13.050
1899.     7.060
1900.    20.830
1901.    19.840
1902.    19.620
1903.    20.460
1904.    20.310
1905.    24.080
1906.    18.670
```

Figure 2.

We bring text into Max with the text object. Text can take an argument for the file name, and responds to a read command.

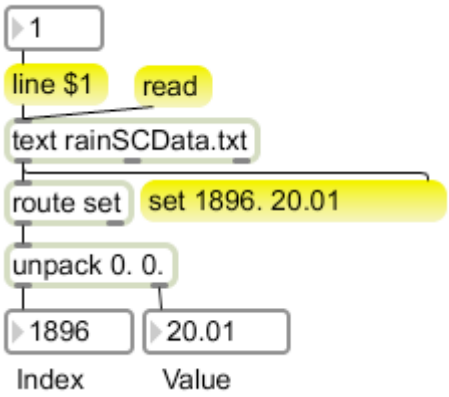

Figure 3.

Once the file is in a text object, you can double click on the object to open a simple editor. The contents of text objects can be extracted line by line as shown in figure 3. Each line comes out as a list beginning with the word set. (This was originally designed to work with prehistoric message boxes which require a set message to change contents.) The route set object followed by unpack will parse the line into individual values for each column. This particular file includes index numbers (the year the value refers to in this case) which will make the patch fairly simple.

Line Graph

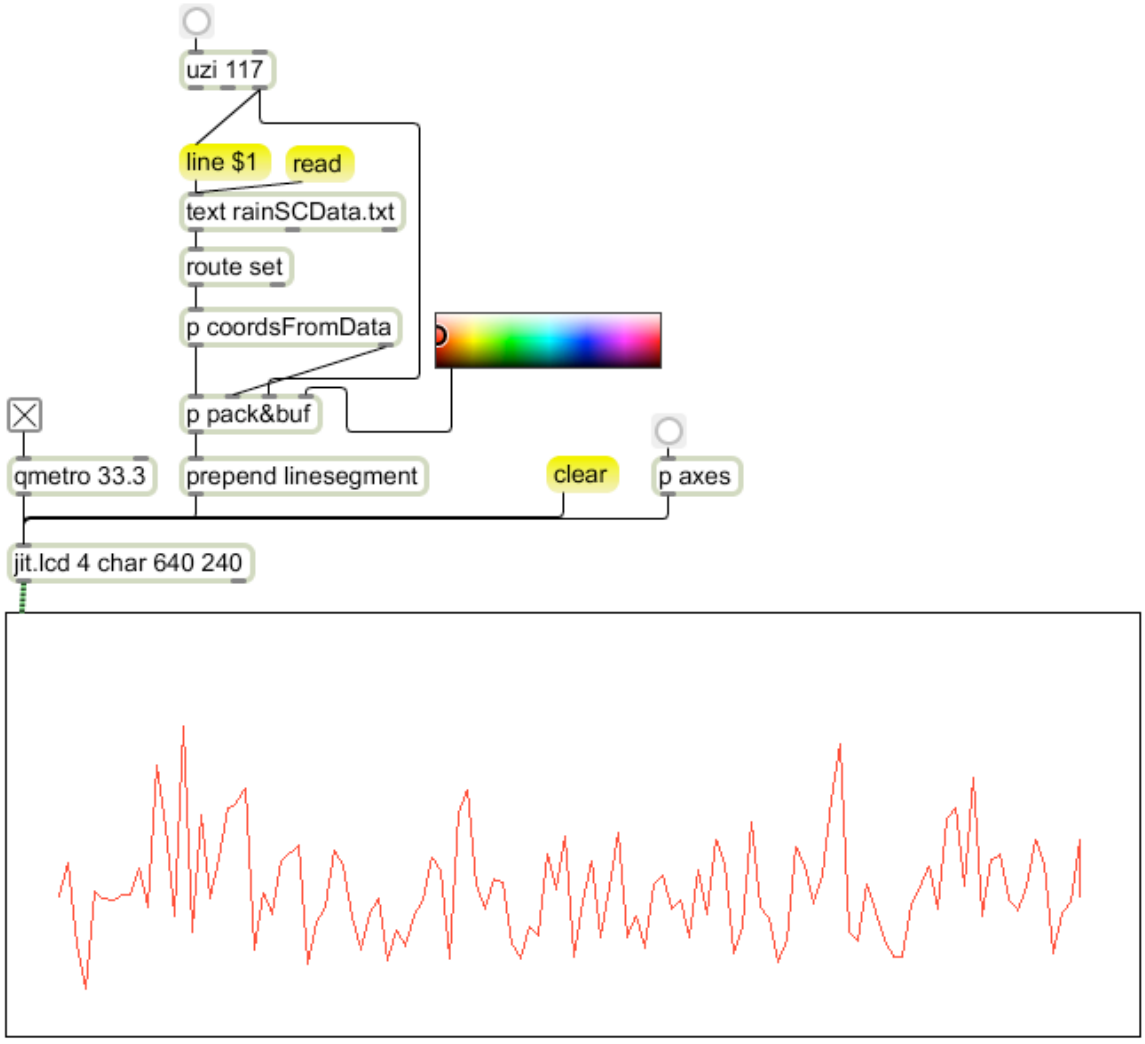

Figure 4.

In figure 4, the data is read line by line by an uzi object and graphed by linesegments. Two processes are necessary to do this—the data must be converted into X Y coordinates and the coordinates must be formatted to give a clean drawing. The coordsFromData subpatch does the first task:

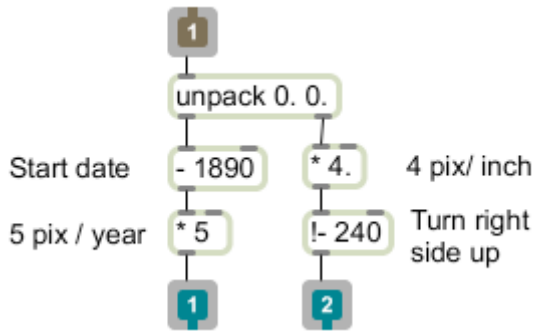

Figure 5. CoordsFromData subpatch

Figure 5 shows the contents of coordsFromData. This determines how the data will be graphed, and must be customized for every situation. Choosing the best way to graph values is an art form by itself. There are many useful books on designing graphs, such as "Show me the Numbers" by Stephen Few (Analytics Press 2004).

This subpatch works on the list from the text object, which is unpacked for independent processing. In this example, the data from the left outlet of the unpack will be the year index values, which start with 1896. Subtracting 1890 will give a few pixels space at the left margin of the graph. That value is then multiplied by 5 to give the proper spacing. The multiplier is chosen to give enough pixels per datum to fill the graph—here there are 117 values to graph, so a 5 pixel interval will nearly fill a width of 640. (If the data does not include index values, the index output of uzi (less one) can be used.)

The right outlet provides rainfall values, which range up to nearly 50 inches but are mostly around 30 inches. 4 pixels per inch will give an attractive vertical spread. These numbers must be subtracted from the coordinate of the bottom of the window (240) to turn the graph right side up.

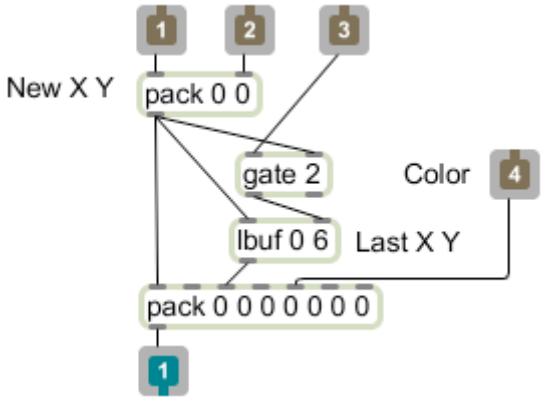

Figure 6. Pack&buf subpatch

The coordinates produced by coordsFromData must be formatted for the linesegment command. This requires two pair of coordinates: the last destination and the new one. To remember the last coordinates, the pack&buf subpatch stores the incoming coordinates in an lbuf object[3]. Lbuf holds a list until a new one comes in. Lbuf must be primed for the first segment to be drawn (it normally outputs 0 0 when it receives its first list of the day). This priming is provided by the gate object. Inlet 3 receives the index number from the uzi that is scanning the file. This number will be 1 for the first point drawn. If the gate receives a 1 in the control (left) inlet, numbers received in the right inlet will come out the left outlet. Other numbers will steer data out the right. In this case, when the gate is set to 1, the coordinates will be stored in lbuf via the right inlet and sent out when the coordinates arrive again at the left inlet. Thus the coordinates will

[3] This is one of my Lobjects, available from ftp://arts.uscs.edu/pub/ems/Lobjects/

match on the first line--this will cause linesegment to draw a point. All of this draws the lines shown in figure 4.

Labels

The next task is to draw labels on the axes to define the graph values. This is in the axes subpatch which is triggered by a bang and generates commands to jit.lcd.

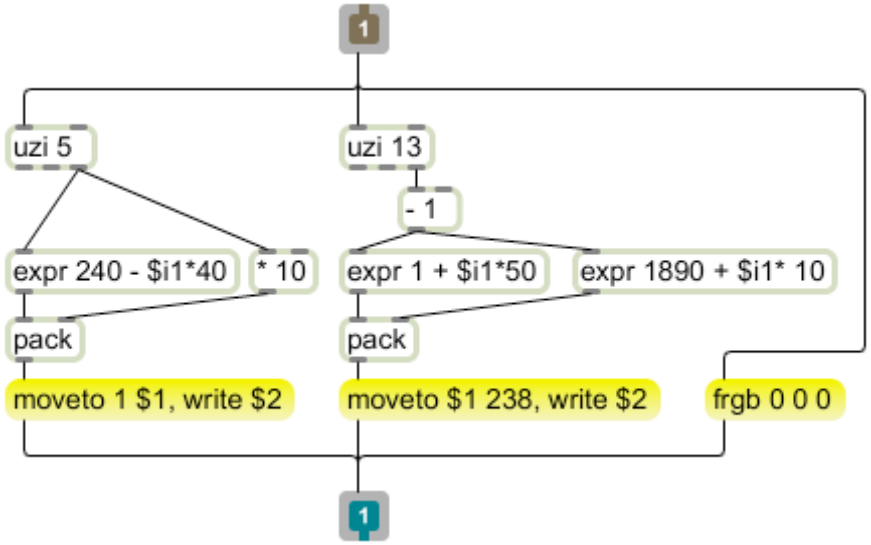

Figure 7. Axes subpatch

There are three parts to this subpatch. On the right (first to execute) the drawing color is set to black. The center section labels the horizontal axis. Uzi will produce numbers from 1 to 13, which are changed to be 0 to 12 so we can easily calculate the years and positions for each year[4]. The year will be the starting year followed in increments of 10. These will be spaced 50 pixels apart, starting with a 1 pixel offset. The values for year and position are packed and applied to a two part message. The message moveto positions the pen one pixel above the bottom and the command write prints the year as text. The third section of the patch draws the left index in the same manner. The result is visible in figure 8.

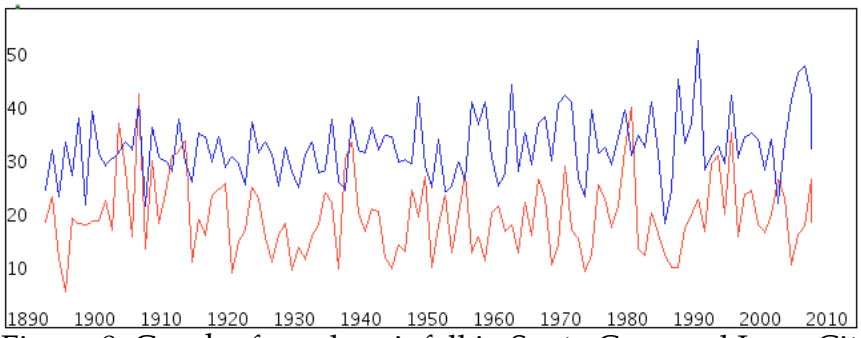

Figure 8. Graph of yearly rainfall in Santa Cruz and Iowa City

[4] This is why programmers prefer to count starting with zero.

Figure 8 also shows how easy it is to add more functions to this graph-- just read another compatible set of data and change the color.

Bar Graph

A bar graph is a simpler patch. The coordinates only need a minor tweak to work with paintrect. They are applied to a pack for the left and top values, and the right is calculated as 4 pixels more than left. The bottom value is entered into the pack and left alone.

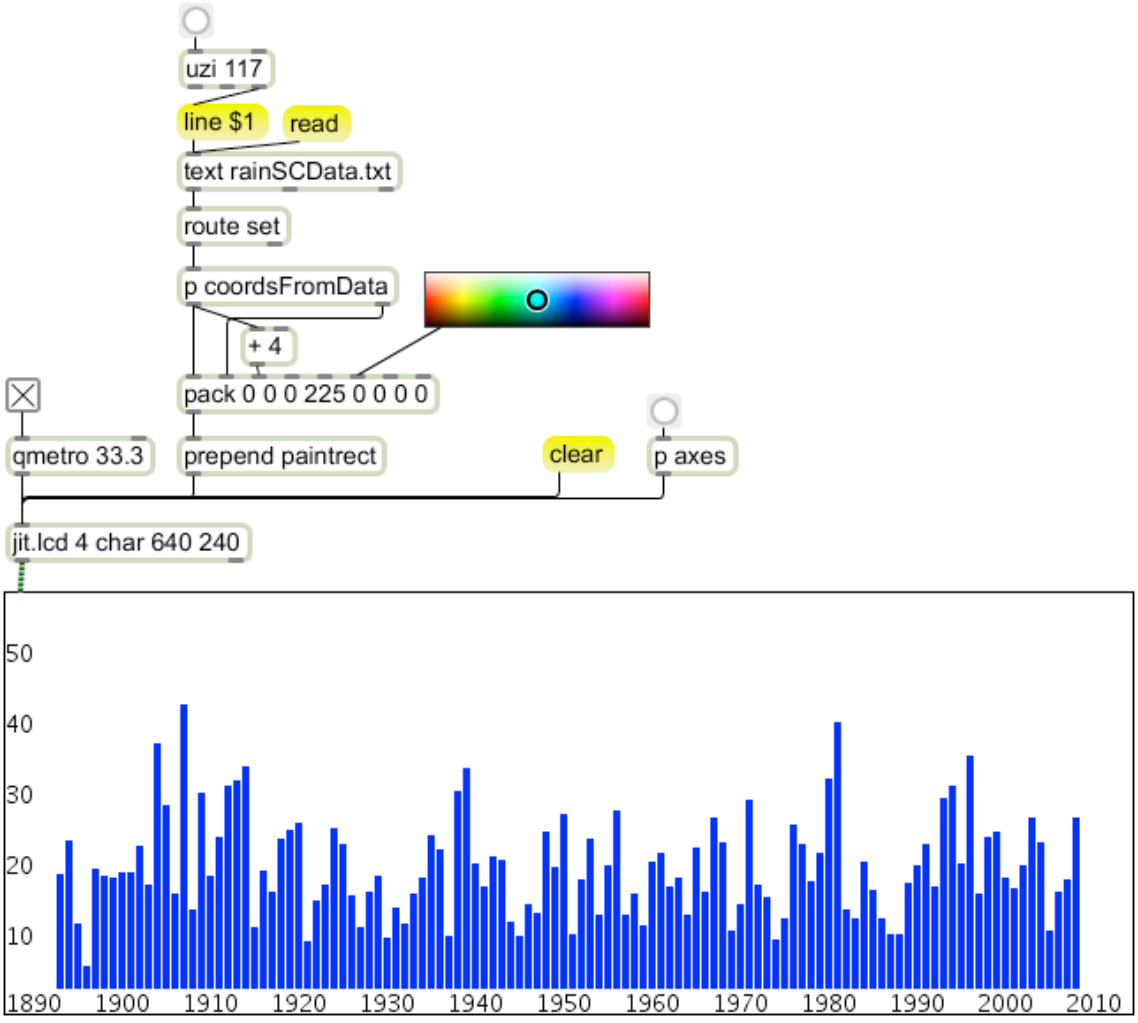

Figure 9. Rainfall in Santa Cruz

Pie Charts

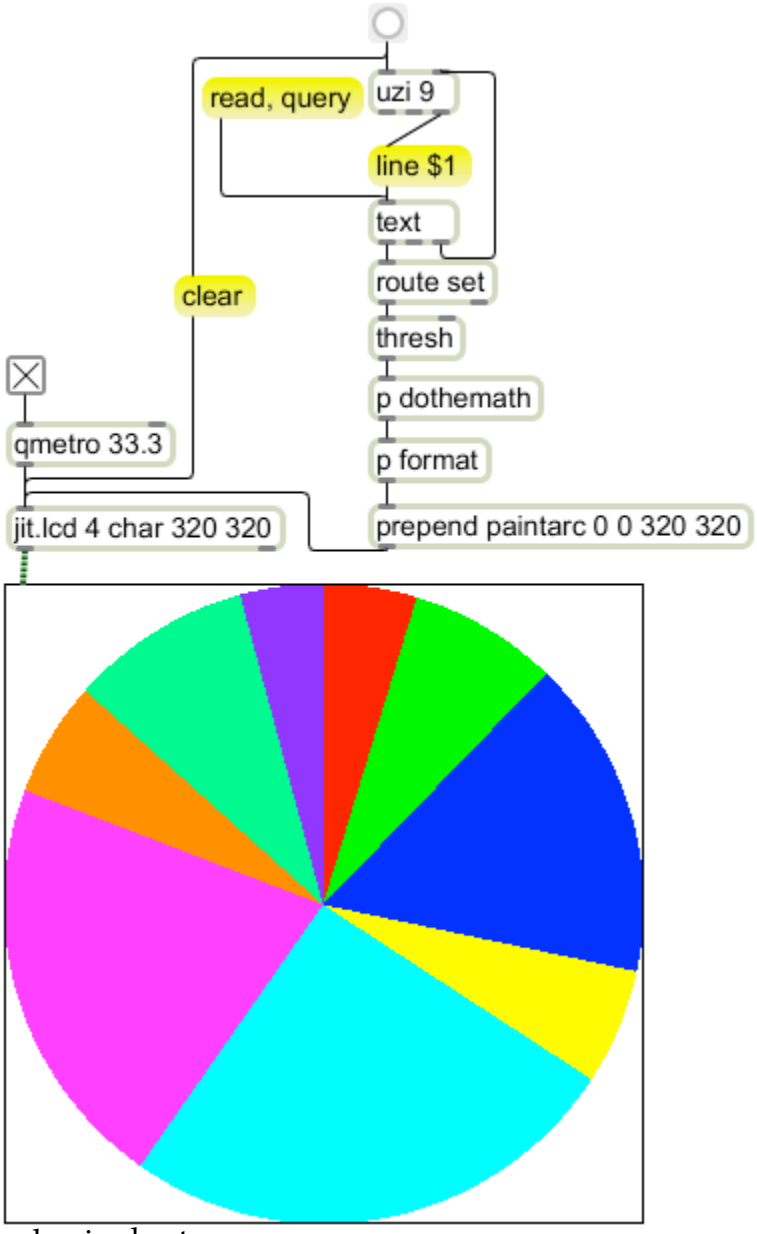

Figure 10. Sample pie chart

A pie chart is easy to make using the paintarc feature of jit.lcd. Paintarc requires 9 arguments. These are: Left, top, right, and bottom of the enclosing box followed by angle start (degrees) angle width, and color.

The tricky part of pie charting is figuring out the angles to represent each value. A pie chart is only useful for a dozen or so values, so it is reasonable to process them in a list. The list can be generated from a text file formatted with one value per line. Then the uzi and text object we have used before gather the values into a list with the thresh object.

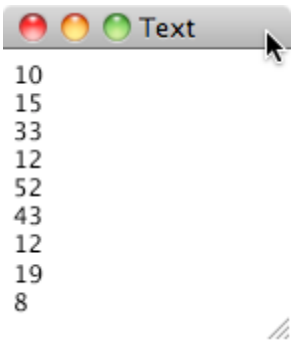

Figure 11. Data to be charted.

The values used in figure 10 are shown in figure 11. In order to display them in a pie chart we must know what percentage each value is of the total. That action is shown inside the dothemath subpatch. This is not rocket surgery—the sum of all values in the list are divided into 360 to get a conversion factor that equates all values to a full circle. Each member of the list is then multiplied by this factor.

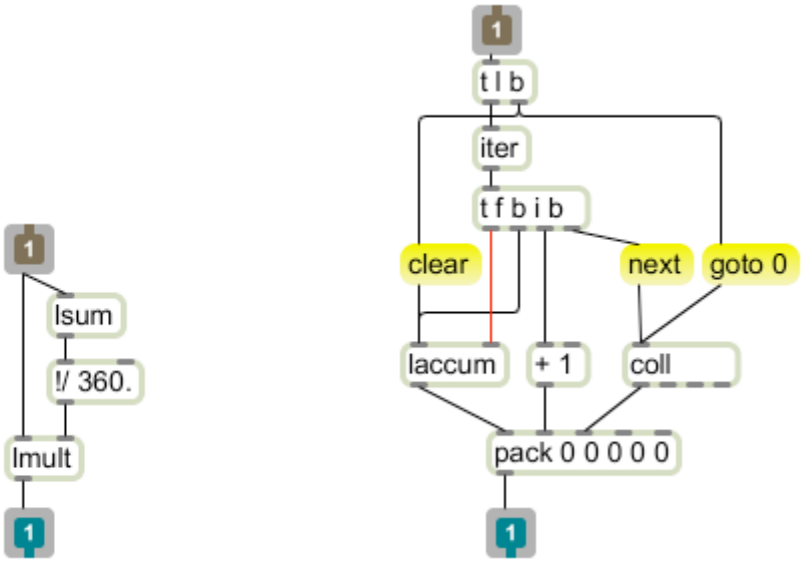

Figure 12. Dothemath and format subpatches

The format subpatch is more complex, primarily because of the need to keep operations in the correct order. The data enters as a list. The list is fed to a trigger object (t) that bangs to send a goto 0 message to the coll and a clear message to laccum. These reset the objects to an initial state. The list is then passed on to iter. Iter breaks the list into a stream of angles. Each value then triggers the following actions:

1. A color is packed into the last three positions of a five element list. The colors are stored in a coll[5] so that the order of items in a list will determine the color.

[5] Remember to open the coll inspecter and check "Save data with patcher".

2. The angle plus one is packed into the second spot in the list. The one is added to make sure the wedges overlap slightly. Otherwise there may be white spots.
3. The current value in laccum is packed in and the list sent out. We are not done however—laccum keeps a running total of the angles processed so far. The new angle needs to be added in for the next go 'round so the itered value is passed out again, this time as a float[6].

Once this list leaves the format subpatch, the command paintarc and the coordinates of the enclosing rectangle are prepended.

Labels

Adding automatic labels to the pie chart is fairly complex. The labels must be printed in a circle with the end of the label near the center of each wedge. Figure 13 shows one way to format the data files to include labels.

Figure 13. Sample data with labels

There's no particular advantage to putting the labels after the data, but it looks neater.

Figure 14 shows changes to the main patch. There is a new subpatch named labels that needs a lot of information. The messages to this subpatch are, from right to left:

- A bang to reset objects in the subpatch when a new plot is started.
- The names split off from each line in the file. The 's' in an unpack object means a symbol is expected here.
- A item number supplied by the format subpatch as the wedge drawing commands are assembled. This is available inside the format from an outlet on the coll that selects colors.
- The list of angles from the format subpatch.

The contents of the labels subpatch are shown in figure 15.

[6] If the angle were to be passed as an int, the fractional part of each would be lost and the total would be less than 360.

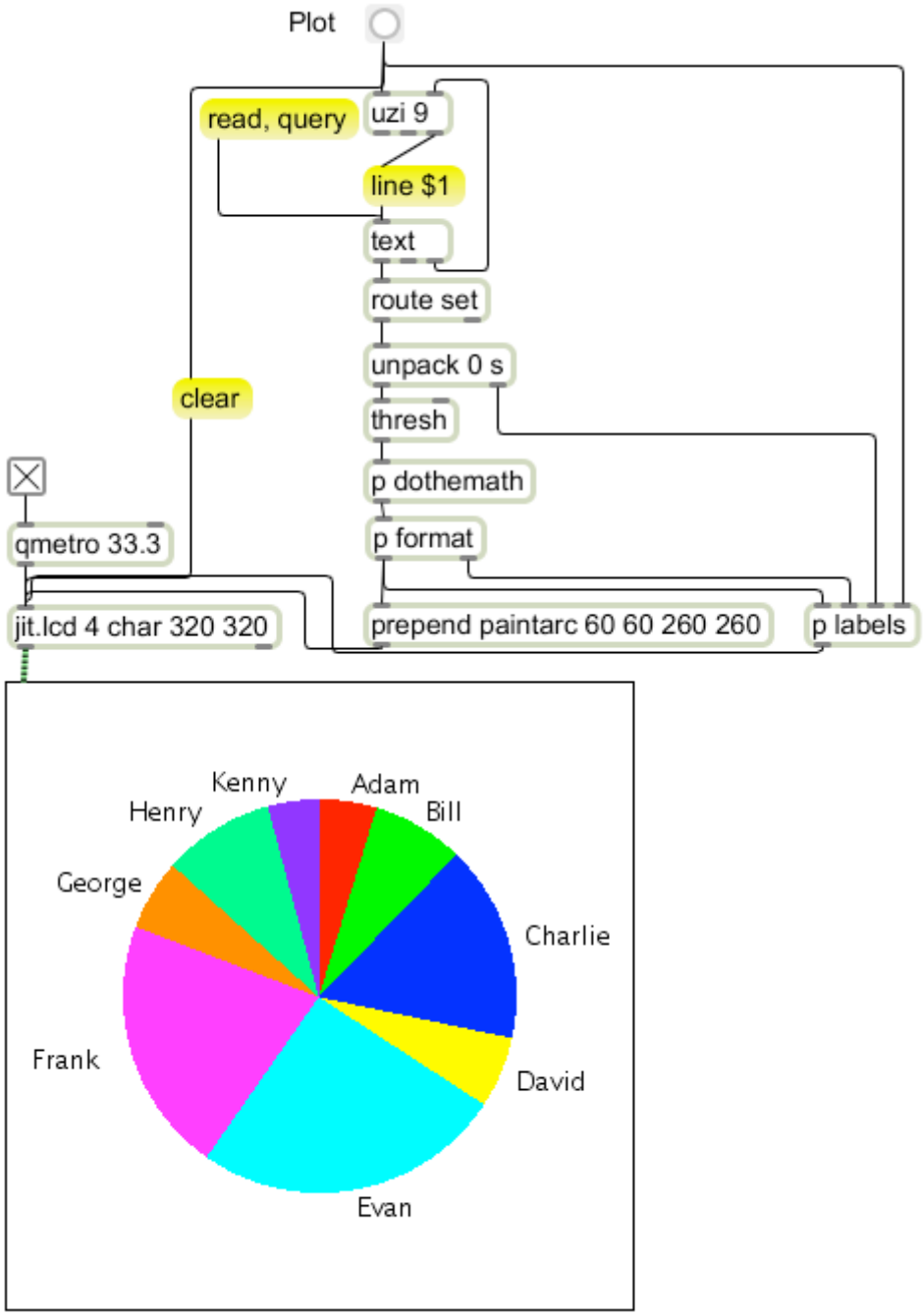

Figure 14. Pie chart with labels

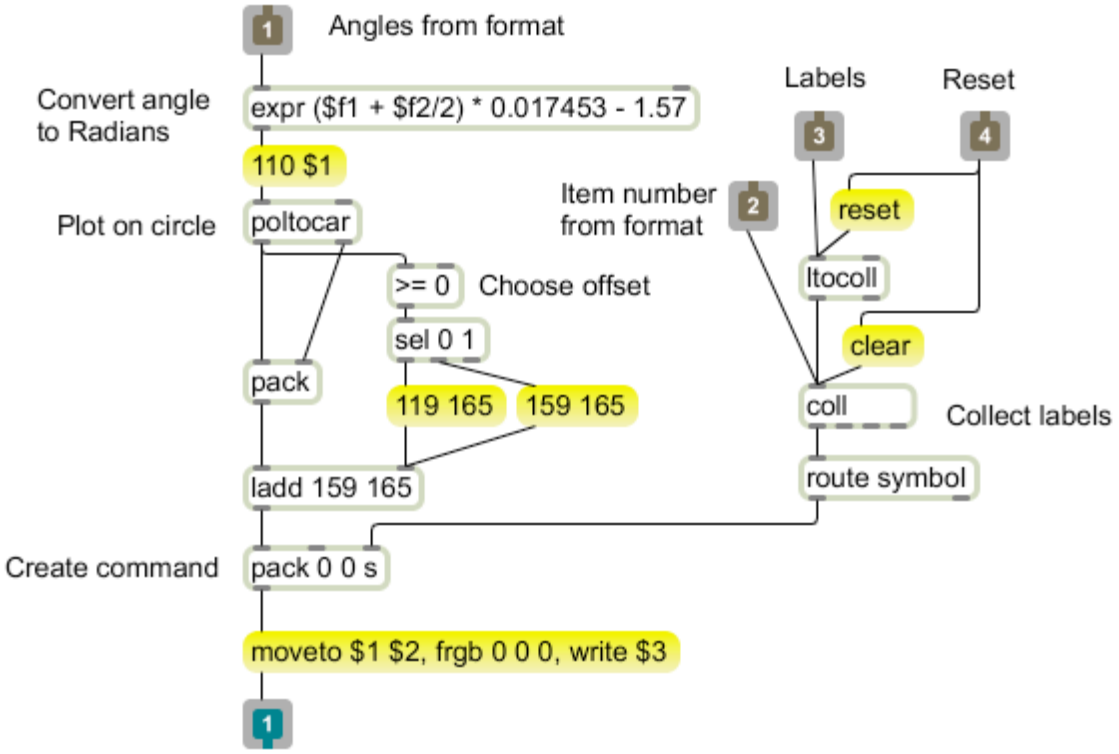

Figure 15. Labels subpatch

There are two parts to the labels subpatch. The mechanism on the right collects the labels from the file as the uzi in figure 14 steps through the lines. Ltocoll is an object specifically designed for the task of entering itered data into a coll. For each item that comes in, it creates an address and outputs a list of the address and the item. When the format subpatch in figure 14 sends an item number, this coll will produce the associated name, which is packed as the third item in a list. The route object is there to strip the word "symbol" which always precedes a symbol output from a coll. (Another prehistoric feature of Max.)

The left hand of figure 15 converts the angles produced by the format subpatch to radians. Since the wedge is drawn from the starting angle ($f1 in the expr) along an arc $f2 degrees wide, the center of the wedge is at $f1+ $f2/2. The long fraction converts degrees to radians. Finally 1.57 is subtracted[7] from the new angle, because while paintarc counts from 0 degrees at the top, poltocar places 0 at the right.

This angle is converted to Cartesian coordinates with a fixed radius of 110 pixels. This is offset to the center of the graph with one wrinkle: if the label is on the left of the figure (X will be negative) an extra offset to the left is used to make room for the end of the label to clear the graph. In an extra fancy version this offset would be made variable to match labels of different lengths.

[7] It's subtracted, because poltocar produces positive Y for positive angles, but positive Y is counted from the top of the window down.

Jit.Graph

Figure 16. Rate of snow accumulation in Greenland by depth in ice core

Jit.graph is a powerful graphing tool within limits—it works best for dense data such as audio waveforms. For small sets of data it is just too coarse a display. In figure 16 a jit.graph is displaying the rate of snow accumulation in Greenland for the last 50,000 years.

Jit.graph takes its data from a 1 dimension array containing the data. These numbers are mapped 1 per pixel across the height (default 240) of an output matrix. The @rangelo and @rangehi attributes determine the vertical scale. The width of the output matrix is determined by the length of the input matrix—in this patch the length is automatically set when "query" is sent to the text object. Text sends a pointer to the first empty line of the loaded file from the right outlet. Since text counts the first line as one, we have to subtract one from this value to get the size of the matrix.

The matrix snowfall is stuffed with setcell messages. This takes an index (from uzi with one subtracted) and a val, which here comes from the second column of the data.

The trick to using jit.graph is in finding the right dimensions to make a clear display. The output is always as wide as the input data, but that is stretched or squeezed to fit the display area in the window. There are several modes of display, which helps a lot. Mode 3 is a bar graph.

Interaction with jit.graph

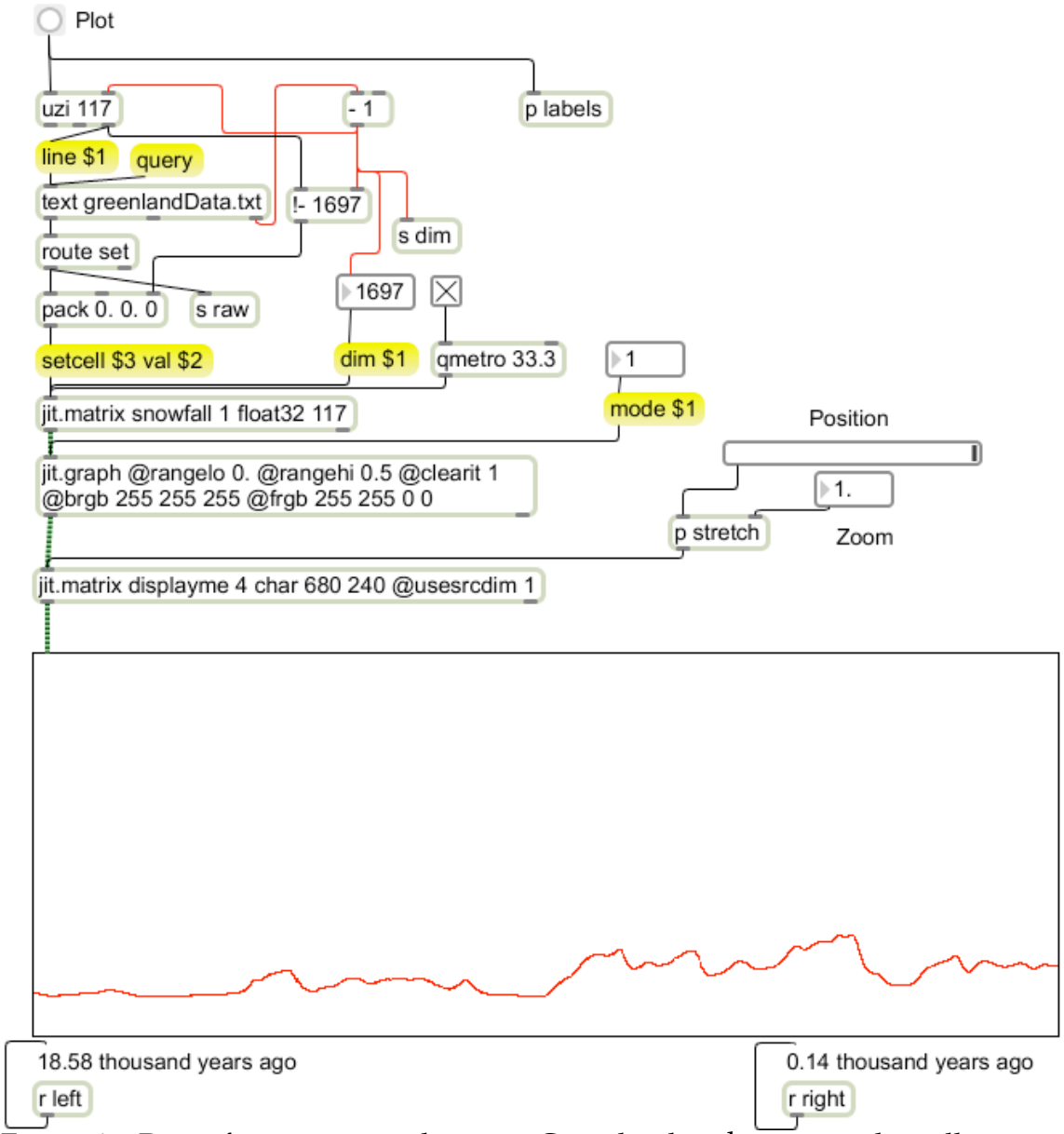

Figure 17. Rate of snow accumulation in Greenland with zoom and scroll

Figure 17 revises the presentation of figure 16 and includes some interaction. The revision deals with a quirk in the data. This is a graph of snow accumulation rates over 50 millennia. Since the data was taken from ice cores, it is backwards—to display the data chronologically, we have to reverse the data. This is easy to do at the text dump stage. Figure 17 calculates the address in the snowfall matrix by subtracting the uzi index from the size of the data. This fills the matrix backwards.

```
2.  Accumulation rate in central Greenland

Column 1: Age (thousand years before present)
Column 2: Accumulation rate (m. ice/year)

          Age          Accumulation
        0.144043         0.244106
        0.172852         0.246155
         0.20166         0.248822
        0.230469         0.249856
        0.259277         0.249943
```

Figure 18.

Since the output of jit.graph is a stream of matrices, we can update the display interactively. The snowfall data consists of 1697 points and is hard to see in detail. All of the data is in the jit.graph output, but it is squeezed to fit a window 640 points wide. The solution is to provide the ability to scroll through the data and to zoom in and out.

The manipulation is made possible by adding an intermediate matrix "displayme". This matrix has source dimensions enabled, so it can copy selected sections of the jit.graph output. The scrolling operation is contained in the stretch subpatch shown in figure 19. The role of this subpatch is to calculate values for source dimension start and source dimension end.

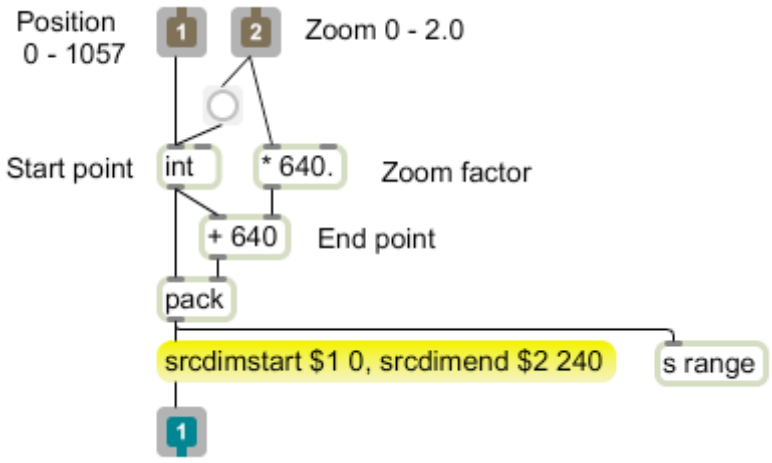

Figure 19. Stretch subpatch

The default is to display the first 640 points. This is with start position 0 and zoom at 1.0. Increasing the start point will move through the data range. Changing the zoom will change the width of the source rectangle, which will be interpolated to fit into a matrix 640 by 240. I've used a slider to control position and a number box for zoom. These would probably be repositioned under the graph in presentation mode.

It is possible but complex to provide axis labels within jit.graph—that would require a composting stage or a jit.lcd, so I have chosen to take the easy way and display axis labels in comments. Comments can be changed by a set message, which are generated in the labels subpatch, shown in figure 19.

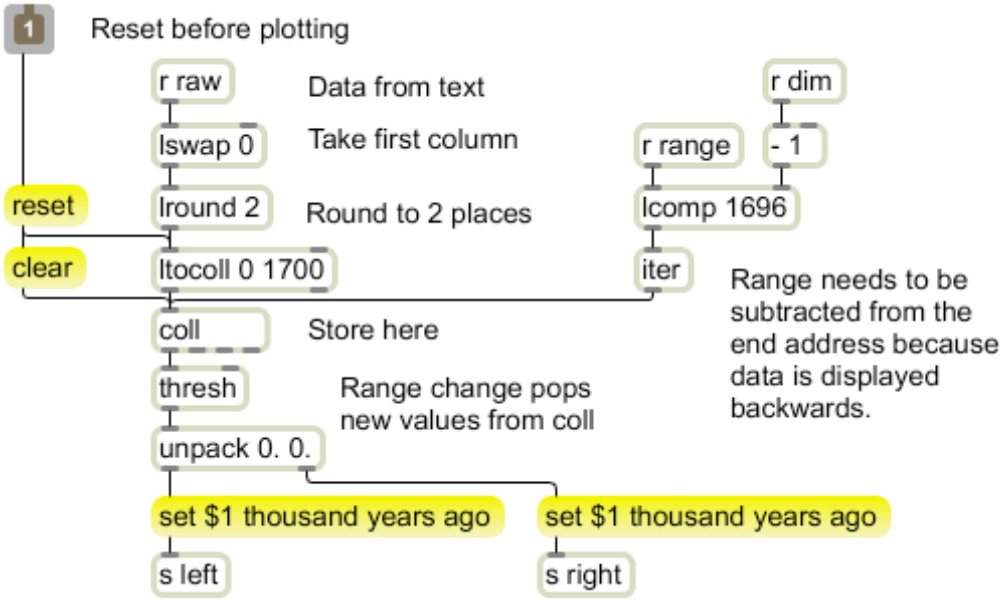

Figure 20. Labels subpatch

The labels subpatch depends on data sent from several sources in the main patch. The receive (r) object named "raw" gets a complete copy of the data extracted from the text object. This is processed by a couple of lobjects. Lswap[8] 0 will pass only the first column of data, which contains the age of each sample. Lround 2 rounds the numbers off to 2 decimal points. These numbers are stuffed into a coll via ltocoll.

The stretch subpatch sends the start and end points of the data to display under the name "range". When the range changes those points will be subtracted from the last address in the data (to compensate for the reversed display) and used to fetch values that will be sent to the left and right captions.

Since jit.graph is producing matrices, it is possible to overlay several, call up different views and even cross fade graphs. See the tutorial on compositing for ideas on how to combine images.

[8] Lswap rearranges lists according to its argument list. The first item in a list is indexed at 0, so lswap 0 2 would pass the first and third items.

2D Mapping

This section is based on material in the book "Visualizing Data" by Ben Fry (O'Reily 2007). This book does a fine job of introducing the art of graph building with code examples in the Processing language. It's not too difficult to convert the methodology into Max operations, and some of the code can be used directly in the mxj or js java objects.

Two dimensional graphs are often used to relate some factor to location. For instance, we often see maps of the country colored in with demographic information. This is reasonably easy to do, because the census bureau releases a lot of economic data organized by zip code, and the post office publishes zip code databases with location information of each office. Ben Fry has posted the latter on his web site for users of his book[9].

```
# 41556,-0.3667764,0.35192886,0.4181981,0.87044954
00210   0.3135056   0.7633538   Portsmouth, NH
00211   0.3135056   0.7633538   Portsmouth, NH
00212   0.3135056   0.7633538   Portsmouth, NH
00213   0.3135056   0.7633538   Portsmouth, NH
00214   0.3135056   0.7633538   Portsmouth, NH
00215   0.3135056   0.7633538   Portsmouth, NH
00501   0.30247012  0.7226447   Holtsville, NY
00544   0.30247012  0.7226447   Holtsville, NY
```

Figure 21. Zips.tsv

Figure 21 shows the beginning of the zip code location file. For each zip code, there is a longitude and latitude and the name of the post office. The longitude and latitude numbers will not look familiar- they have been converted to radians and the longitude is relative to a point at 96°W (a bit west of Houston). They have been further tweaked to give a conic projection map. The top line shows the minimum and maximum for both values, which will be a help when we plot the locations.

This data could be mapped directly from a text object, but it will be much more efficient to use a coll. Figure 22 shows a patch for transferring the information. The procedure then becomes:

1. Clean up the text file in an editor like TextWrangler.
2. Read the file into the text object.
3. 3 adjust the Lswap object to filter for the columns desired. Note that the first item in the output lists will become the address in the coll. Thus Lswap 1 4[10] with figure 21 would produce a coll of names addressed by zipcode.
4. Dump the text. Be prepared to watch the spinning ball for a while.
5. Double click the coll to check the contents- test an interesting address.
6. Click write to save the coll data to disk. You can then use this file name as an argument to coll when you want that data.

[9] www.benfry.com If you find this resource useful, it wouldn't hurt to buy the book.
[10] Lswap 0 would include the word set that the text object sticks on everything.

Figure 22. Conversion of text file to coll.

I used this process to convert figure 21 to a coll of locations for each post office. Figure 23 shows how I plot them.

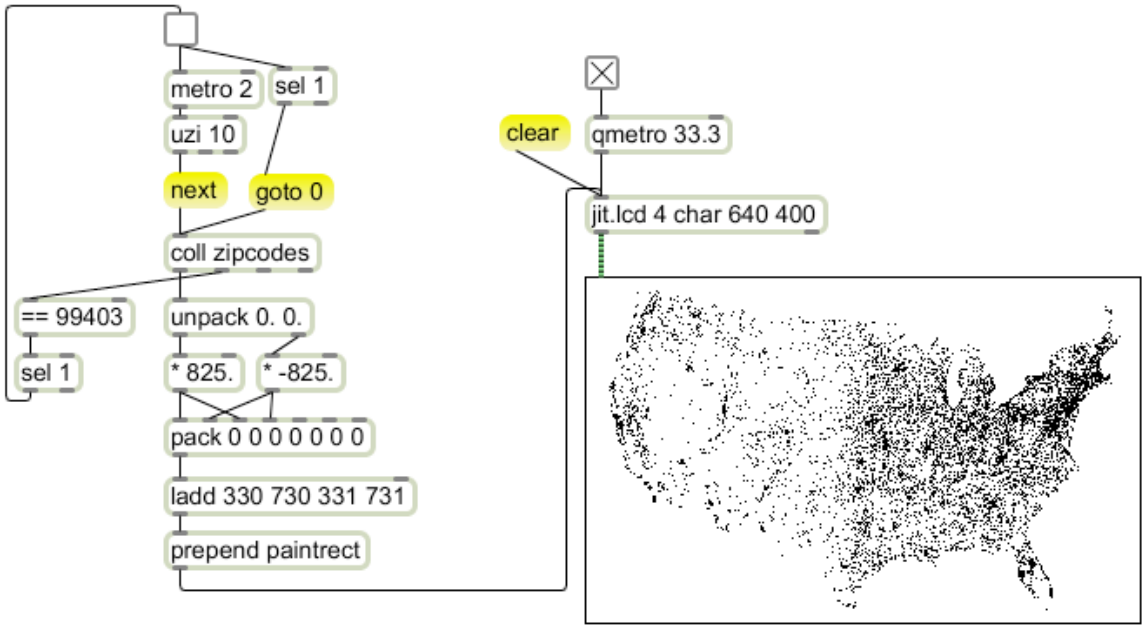

Figure 23. Map of post offices.

This uses our standard trick for drawing rectangles. Since the locations are expressed as fractions, they need to be multiplied by some factor to fill the jit.lcd. The factor used in figure 23 leaves a 24 pixel margin at the left and right edges. That can be calculated from the header line of figure 21: The difference between the leftmost and rightmost point is 0.71875026. If we divide that into 592 (width of finished image) we get 825. The offset values in the ladd have to be found by trial and error, because the image is not quite centered in the window. The vertical offset was particularly tricky, since it refers to the equator, which is well

below the window. Any change in the scaling factor also required a change in offset. The reason for the fussing is apparent in figure 24.

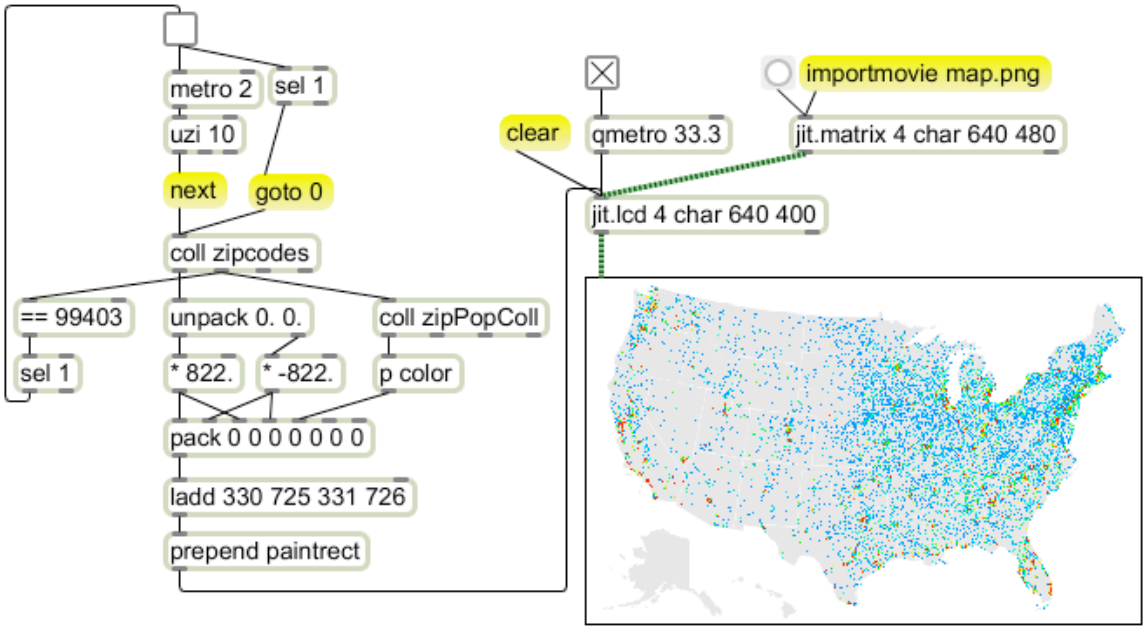

Figure 24. Map of population by zip code

Here I have added an image of a US map (which is also available from www.benfry.com). The map dimensions are 640 by 400, and I rescaled the dots over this map several times to get them to match. The colors in this map come from a second coll that has population of each zip code addressed by the zip code. Thus, while the metro is stepping through the coll with locations, the address coming from the second outlet pops the population value out of the second coll. This chooses a color (using the hsl method) to use in drawing the dot. The result is shown full size in figure 25.

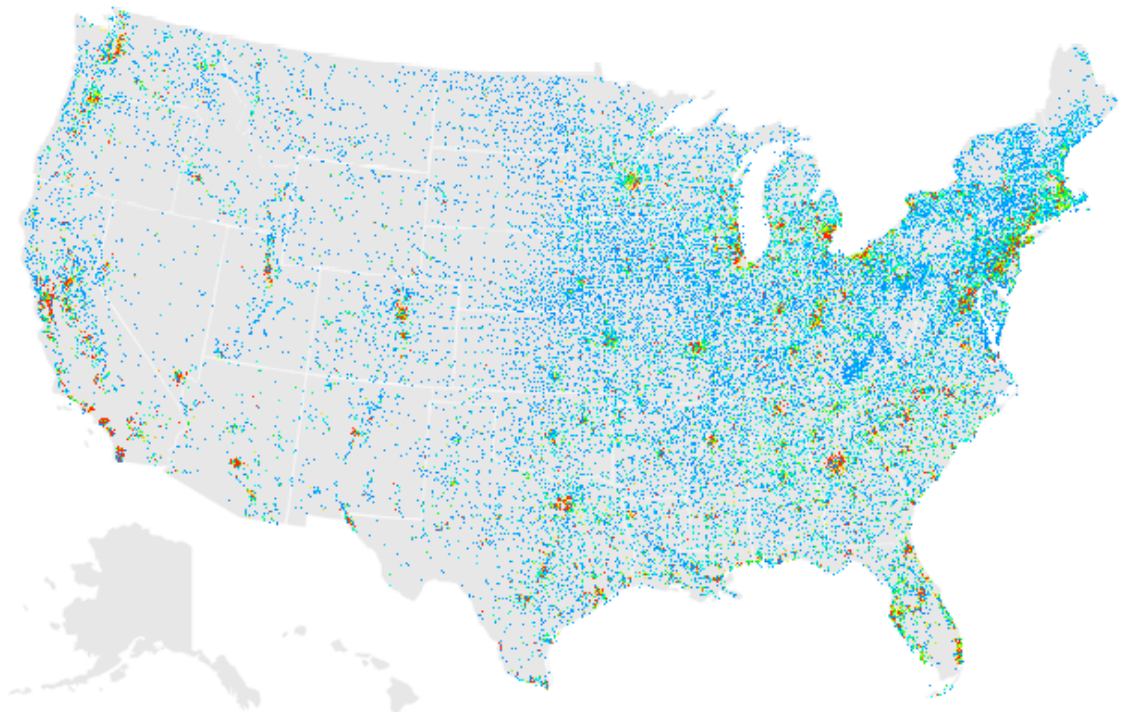

Figure 25. US population by zip code