Notes on creating gradients.

Chroma.js

mix(c1,c2,d,cSpace) returns a color d along the line from c1 to c2 in the specified color space.

Brewer.<name> Returns the color brewer scale associate with that name

(has breaks and k-means clusters, but only in 1D)

F=scale(cArray) //creates a scale function. F(d) returns the color at d.

F = scale(cArray).domain(dArray) //assigns distances in dArray to cArray colors

F = scale(cArray).domain(dArray).mode(cSpace) //’lab’, ‘lch’

F = scale(‘brewerName’)

Scale().correctLightness() Redistribute the lightness to be uniform (I suspect)

B= Bezier(cArray) //does a Bezier interpolation, looks like it risks doing too much smoothing

Cubehelix() //a twist through color space,

.limits(data,’k’,5) But I think this is 1D data only.

Model

Create interpolation points using the palette tool. Use the various scale operators to generate more colors. Need:

* nSteps: read/write text box. This shows the number of points to generate
* colorspace to use for interpolation: lab, lch
* CorrectLightness (operator).
* Reverse order (operator) added to sort section
* A sequential palette has some key colors and some interpolated colors. They are in two separate arrays in the palette. It is optional to show the key colors
  + We want to assert it is a sequential, but display and edit the key colors like any other palette
  + We want to generate a sequence, view and edit it only
  + We want to generate a sequence, view it and its seeds, edit the seeds, regenerate
* Model
  + State.colors follows the display option (input, interpolated, both). We take advantage of being able to get to the input colors via state.palette.eColors
  + Added a Copy To Input button to update the input with the interpolated
  + Added a Correct Lightness checkbox. This function is applied as the palette is generated.
* A diverging palette is defined by two sequential palettes and a center. Could we do this with two pointers?
* Read treats a sequential palette like any other. Defines the input colors.
* Write writes out state.colors. At this point it does nothing smart at all about distinguishing the input from the generated. User keeps track of that.
* Reset will go back to the original input colors. To progress from the interpolated, need to copy them back into the input (eColors), then copy the palette (eColors goes to oColors)

Scaling and filtering

* Map range[a\* b\*] to x,y.
* Given aMin, aMax, bMin, bMax, pWidth, pHeight.
  + Find AtoW, BtoH. (Max-Min)/wSize
  + From aMin and bMin, compute orig in a,b, then convert to pixels using function above
* Created display controls for this as a global field for now, want to add controls soon.
* Filtering should filter the data based on ranges of L,C and H.
* Fit to data check box is any easy way to implement some of the most critical cases right away (formatting lights, tinted grays, etc.)

Display

We have three display procs

* AllPalettes: Let’s assume this will show the input colors only for now
* displayColorspace: The LAB plots.
  + Input only: Works by default
  + Generated only: needs to use a different array. Would just setting state.colors to the gColors work?
  + Both: This is the tricky one. Ideally, we have a way to distinguish the two. Need to first plot the gColors, then overlay the input
  + Selection: index into state.colors. If we let state.colors be gColors, then select/deselect might work.
* Reset needs to clear the gColors