Fancy Plotting in R for EDSDers: A tutorial

Tim Riffe

November 18, 2011

Abstract

One of the strong points of R is its graphical power. This document is not a complete tutorial to R graphics. Rather it's an ad hoc collection of tips and tricks for effective plotting (papers), power-plotting (diagnostics) and beautiful plotting (presentations) in R. A good plot for a presentation is different than a good plot for a publication and so forth. In demography and other disciplines it is important to maximize the so-called information-to-ink ratio. I'll also include some thoughts on good form for presentations.

1 base vs lattice vs ggplot2

There are several *systems* for graphics in R. The two main power-houses are `lattice` and `ggplot2`. I am in a minority because I prefer `base` graphics. If you know enough about any of these systems, you find out that they are all perfectly capable of doing the same things. My advice is to choose your weapon, and learn it well. When you are begining to learn R (during the EDSD), do not waste your time trying to figure them all out. Just choose one, power through a tutorial for it, and use it for all your assignments. If you have a high standard for your plots and always insist on getting the details right, then by the end of the EDSD year you will be a guru in that graphics system because you will have been forced to creatively use the tools provided in that system. Here is an incomplete summary of the 3 weapons you can choose from:

1. `lattice`: for `lattice` graphics, I recommend the following materials from Prof. Jakoby: `http://polisci.msu.edu/jacoby/icpsr/graphics/`. This includes a pdf tutorial, example R scripts and datasets to execute them. His examples start easy and end up getting very advanced. Here's my run-down on `lattice`, from the little exposure I've had; 1) (+1) if you follow those tutorials and apply the same concepts to your data, you can get started in a single afternoon; 2) (-1) it's a rather self-contained system: you have to learn the lattice way of doing things, so you can't combine base graphics functions with `lattice`; 3) (+1) `lattice` has better default aesthetics than base graphics, and is generally color-blind friendly; 4) (+1) the package is capable of handling massive datasets and can often convert huge data into the plot faster than either of the other two systems; 5) (+1) it can make really cool plot matrices that are useful for diagnostics; 6) (-1) it is a legacy system. Most of its dedicated users have been using it for many years and are experts, and so it has a low presence in current discussion forums, but you can still find answers to questions in old mail lists.

2. `ggplot2`: Every second question in online discussion forums for R is about a package called `ggplot2`. There are many programmers of other languages that use R *only* because it has `ggplot2`. The main idea is to `ggplot2`, as I understand it, is to implement the so-called *grammar of graphics* (hence gg) proposed by Leland Wilkinson. That is good because it formalizes ones approach to graphics (+1), but it's also a disadvantage because you have to learn a self-contained system, like with lattice (-1). Stackexchange has tons of help for `ggplot2` (+1) and it has a rapidy growing user base. The system has aesthetically awesome defaults (+1). I have not experimented with it, so I can't be very helpful. Once you learn how things work, I'm convinced that there's nothing you can't do with it. Daniel purchased a `ggplot2` manual for the EDSD, and I have another copy, if some wants to borrow it. There is also a pdf copy available on request. Also, if anyone is more interested, the R User Group meeting on December 15th will feature an English language presentation of `ggplot2`, so ask if you are interested.

3. `base`: for some reason I never graduated from `base` graphics. That's bad because `ggplot2` is where the party is at, but good because 1) you can

get really proficient in `base` graphics simply by trying (and succeeding) to emulate either lattice or ggplot2, 2) (+1) its easier to invent new plots using primitive tools in `base`, 3) (+1) I have the impression that interactive graphics are easier in base too using the `locator()` function. (+1) Using its primitive tools, I have been able to write functions for plotting Lexis surfaces as triangles rather than as a grid. This is implemented using primitive `base` functions, and likewise for population pyramids and Lexis diagrams. I think even a guru would have to struggle for days to figure out how to do these kinds of custom demography plots in `ggplot2`, but everything follows intuitively in `base`. I'm certain you can do beautiful pyramids in either `lattice` or `ggplot2`, but you'll have to invent that yourself!

That being said, most of the tricks that I can show you now are only valid for `base`, altough 1) color works the same in all systems and 2) most base graphics functions have parallels in `lattice` and `ggplot2`. In order to use the latter two, you need to install them as packages and call them using `library(lattice)`, etc.

2 On the topic of color in R

There are different ways to specify colors in R plots. In general, do not always limit yourself to writting `"red"`, `"green"` and so forth. If you do, then your head quickly runs out of colors and your plots end up looking cheap. This place `http://research.stowers-institute.org/efg/R/Color/Chart/` is a good reference for colors if you just want a quick suggestion.

2.1 Color tip #1: use a palette

One thing that I find helpful for style and consistency is making a palette of colors to use within a project. Define a palette as a vector of colors something like this:

```
> # Chose some nice colors:
> my7cols <- c("gold", "darkturquoise", "maroon1", "olivedrab3",
      "orangered", "slateblue1", "springgreen")
> # let's say they're for identifying countries
> names(my7cols) <- c("IT", "FR", "CZ", "DE", "ES",
      "UK", "DK")
```

Now whenever you go plot something, just use the object `my7cols` to grab the colors by index number, like this: `col=my7cols[1]`; or by name, like this`my7cols["IT"]`. The basic idea is to go go through your work, recycling the same nice palette. You'll need a bigger or smaller palette depending on what you're doing. The point is to avoid inconsistency in your figures: If age groups are indicated by color in more than one plot, then you need to be consistent about which color is for which age/variable/dimension in your data. It's easier 1) to avoid mistakes and 2) to make global changes to your color scheme if you simply define a palette once at the begining of your R figures script[1]. This advice is valid for any of the 3 earlier-mentioned graphics systems. Seems obvious, but it's easy to get sloppy otherwise.

2.2 Color tip #2: use color ramps

A ramp is a continuous color gradient from which you can select colors. Ramps have start and end color, and optionally specified intermediate colors. Color ramps in R are functions. You may be familar with the functions `heat.colors()` or `rainbow()`. These are standard color ramps. Many others are available in packages, and they are also easy to invent. Do:

```
> rainbow(7)
```

[1]Tangential advice: within a project (paper, assignment,thesis), don't cram all of your R work into a single script. It get's long and impossible to navegate. Instead, write a separate R file for the major steps of analysis: 1) reading in data, 2) data pre-processing, 3) model 4) results 5) figures (change these to suit your style) and put these in your project folder. Then get a single file that just calls the other scripts sequentially, using source("path
to
readindata.R")

```
[1] "#FF0000FF" "#FFDB00FF" "#49FF00FF" "#00FF92FF" "#0092FFFF" "#4900FFFF"
[7] "#FF00DBFF"
```

The number 7 is how many colors you want back from the function, spread out evenly over the entire ramp. If you specify more colors, the starting and ending colors will be the same, but the intermediate colors change to new interpolated positions. You see that it spits back 7 colors specified as hexidecimal character strings. It's hard to look at those and imagine what colors they are, but a simple way to guess is to remember the pattern `RRGGBB`, that is to say the first 2 numbers/letters after the `#` are reds, the 3rd and 4th are greens, and the 5th and 6th are blues. The last two are optional, and are for opacity, which I discuss later. Think of `FF` as *full*. So the first color means 100% red, the third is like 1/2 red and full green, and so forth[2]. Color ramps are useful in demography when you want color to stand for a continuous variable. This might be age, time, intensity- anything that you can think of on a continuum. Do not use color gradients to represent qualitatively different things like population subgroups. I use them mostly for mortality (logged) and fertility surfaces[3].

There is no standard color ramp for mortality surfaces at this time in demography. If you want one, you have to define it, and a legend is necessary for reference. At times you'll need to make a custom legend for it to work well. Here's how to make one (`mxcolors()`):

```
> library(grDevices)  # this package is included in base
> # colors evenly spaced over range(values):
> mxcolors <- colorRampPalette(c("white", "blue", "palegreen",
    "yellow", "red", "purple"))
> # a sequence mx values:
> mxvals <- seq(from = log(1e-05), to = log(1), length.out = 500)

> # image() always wants a matrix to plot:
> COLMAT <- matrix(mxvals, ncol = 1)
> # image() plots a gridded surface:
> image(z = COLMAT, x = mxvals, col = mxcolors(length(COLMAT)),
    axes = FALSE, main = "custom color ramp for e.g. m(x)", xlab
= "m(x) values")
> # defaults to log axis ticks, (negative numbers).
> # need to be tricky to get the labels right:
> axis(1, at = log(c(1e-05, 1e-04, 0.001, 0.01, 0.1,
    1)), labels = c(1e-05, 1e-04, 0.001, 0.01, 0.1, 1))
> box()  # give it a frame
```

[2]When you're feeling too lazy to go to the R Colors webpage, invent something random, keeping this pattern in mind.

[3]You could make a migration surface and you'd be the first!

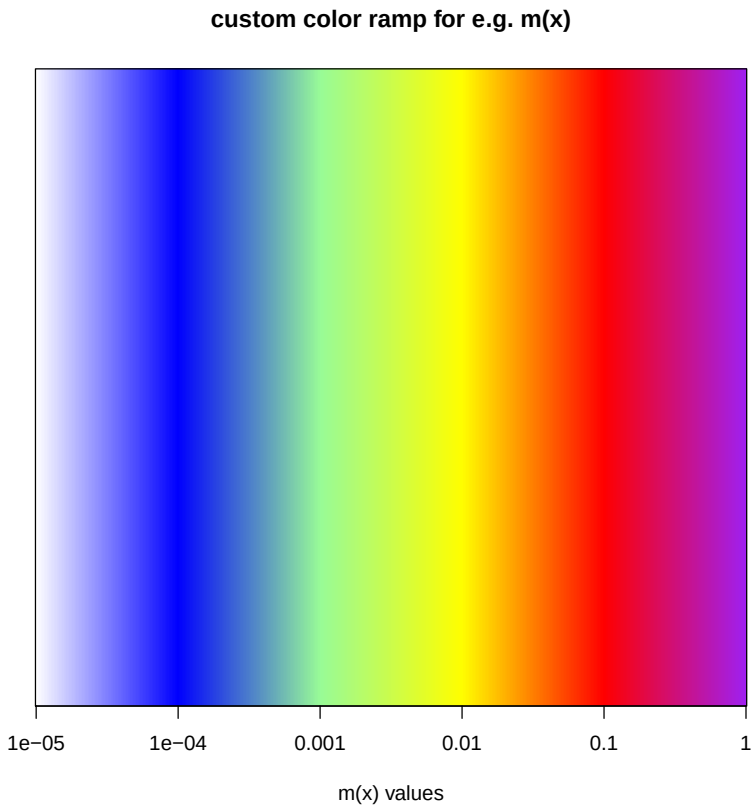

Figure 1: Matching labels to a logged color ramp

2.3 Color tip #3: use transparency

Transparency is useful for managing clutter and/or displaying density in your plots. It allows you to overlap plotted objects, fitting way more in the plot without confusing people's eyes. Let's say you have two different lines fit to data, or a few competing lowess smoothers on a scatterplot, or something like that. If you put in confidence interval lines for each, then your plot suddenly has 6 lines. That gets confusing. You can sort it out a bit with color, but the plot quickly becomes a jungle as the number of plotted lines grows. When that happens, most people just decide not to put in the confidence lines. Bad! Instead, plot the confidence interval as a shaded area using the `polygon()` function, and make them semitransparent so that these regions can overlap with no loss of information.

To use transparency in R you need to specify the color in hexidecimal and add 2 digits to the end. Here's a convenience function to take a named color or a vector of named colors and give them 50% transparency in hexidecimal, where `alpha` means opacity (transparency reversed!):

```
> colalpha <- function(color, alpha) {
     colalphai <- function(color, alpha) {
         paste(rgb(t(col2rgb(color)/255)), alpha, sep = "")
     }
     sapply(color, colalphai, alpha = alpha)
 }
> colalpha(my7cols, 50)
```

```
          IT          FR          CZ          DE          ES          UK
"#FFD70050" "#00CED150" "#FF34B350" "#9ACD3250" "#FF450050" "#836FFF50"
          DK
"#00FF7F50"
```

2.4 An Example

We'll now walk through a semi-realistic example that uses both a palette and transparency to make a hectic plot intelligble. The data used are simulated below, but there's no need to examine it unless you're interested. Most aspects of this data are random, but the mechanism at work within each country subset is similar.

```
> set.seed(236)  # to get consistent random numbers:
> ctry_y <- rev(sort(sample(-100:400, 7))) %InLiNe_IdEnTiFiEr%
     "# some country effects"
> ctry_x <- sort(sample(1:90, 7)) %InLiNe_IdEnTiFiEr%
     "# x shifting for each country too..."
> x <- y <- c()
> for (i in 1:7) {
     # the x range is random too
     xi <- rep(seq(from = 0, to = runif(1, runif(1, 10, 30),
runif(1,
         40, 60)), length.out = 50), 3)
     x <- c(x, ctry_x[i] + xi)
     # y takes the country effect plus obs noise
     # a random country scalar and a random country exponential.
     y <- c(y, ctry_y[i] + runif(150, runif(1, 0, 30), runif(1,
         30, 80)) + runif(1, 0.8, 3) * xi^(runif(150, 1.2, 1.4)))
 }
> # stick together into a data.frame
> sdat <- data.frame(ctry = rep(names(my7cols), each = 150),
     x = x, y = y)
```

Our point of departure plot suffers from 2 things: 1) The noise wins your attention rather than the line and 2) the line is very wrong. We will alter this plot iteratively to learn about R graphical parameters, i.e. this is not stats advice, although you can get an idea of how function syntax works from this. The whole Simpson's Paradox thing is to make the example more interesting, for an exuse for putting lots of lines on the plot, and to get you thinking about Jim's heterogeneity tricks.

```
> # naive linear regression
> LM <- lm(y ~ x, data = sdat)
> plot(sdat$x, sdat$y, main = "*Simpson's paradox*, it can happen
to you!",
     sub = "95% CI", xlab = "meaningless x", ylab = "meaningless
y",
     xlim = c(0, 125), ylim = c(0, 700))
> abline(LM, col = "red", lwd = 2) %InLiNe_IdEnTiFiEr%
     "# regression line (y~x)"
> clim <- as.data.frame(predict(LM, data.frame(x = 0:125),
     level = 0.95, interval = "confidence"))
> # lower then upper CI
> lines(0:125, clim$lwr, col = "red", lty = 2)
> lines(0:125, clim$upr, col = "red", lty = 2)
```

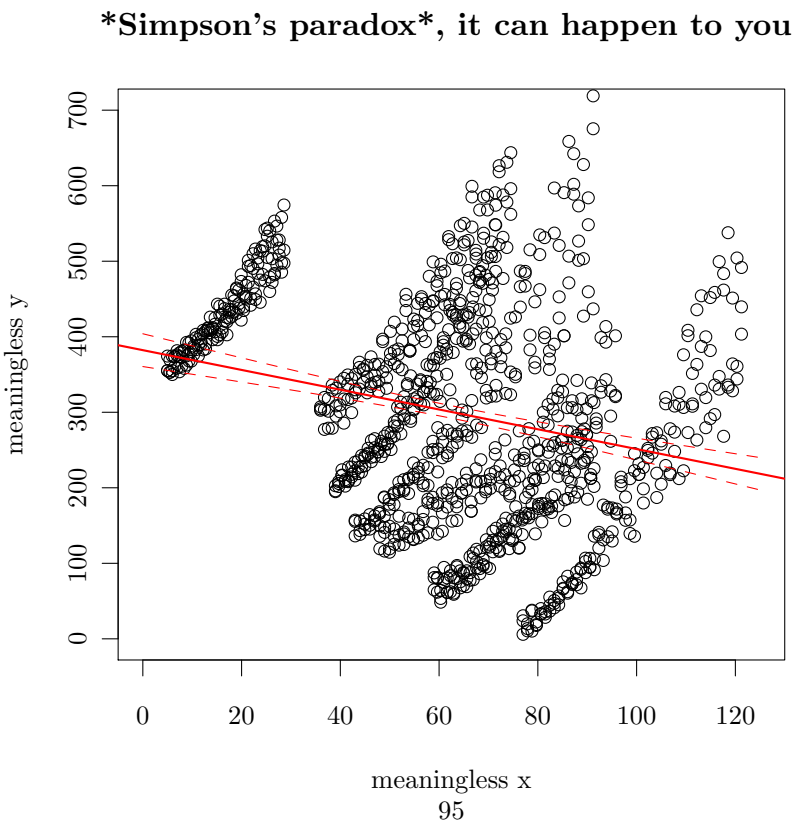

Figure 2: A deceptive scatterplot

There is some mixing in there that is difficult to separate visually, though it's clear that points are somehow grouped, probably by country. A quick and ugly diagnostic, and a plotting function you should frequently use when getting

to know your data is `pairs()`, which plots a matrix of bivariate plots:

```
> # try this, plot not included in document
> # pairs(y~x+country,data=sdat)
```

Back to the original scatter, it's now clear how we need to split the data visually. Let's use the `pch` parameter to make the points solid (19), make them stand out less by using transparency with the function `colalpha()` defined above, and separate them using our color palette defined earlier. One way to do this efficiently is to iterate over a vector of country codes (you can iterate over just about anything in R!). If this were a big computational task, I would not recommend using a for loop, but there are plenty of cases where it really makes no difference whether you use a for loop or not in R. For fancy plotting, I use them quite a bit.

```
> # define empty plot of required dimensions
> plot(NULL, type = "n", xlim = c(0, 125), ylim = c(0,
    700), xlab = "meaningless x", ylab = "meaningless y")
> # iterate over country names to add points:
> for (i in names(my7cols)) {
    ind <- sdat$ctry == i
    points(sdat$x[ind], sdat$y[ind], pch = 19, col =
colalpha(my7cols[i],
        30))
 }
> legend("bottomleft", fill = my7cols, legend = names(my7cols))
```

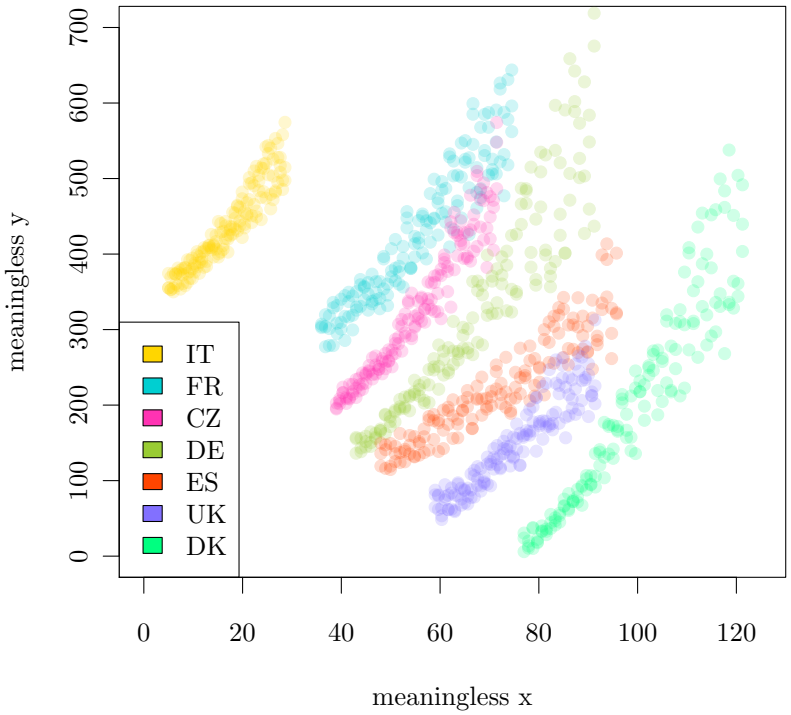

Figure 3: Assign to colors to subsets in a loop

It being clearly the case that each country shows a similar but shifted pattern, we can do away with the naive regression line and and control for country. This code chunk does this (still easily improved upon) regression and grabs us a few useful points for plottng in the next code chunk. I'll explain a bit the strange-looking loop. What I want to do for the plot is draw a line over each colored cloud of points, but I don't want it to cross the entire plot. This will be done with the function `segments()`, which like most functions in R is vectorized, meaning we can supply vectors as arguments and it will repeat same task running element-wise simultaneously down each of the argument vectors. `segments()` wants the x and y for the points forming each end of the segment, so this for-loop selects an appropriate x max and min for each country, and then finds the corresponding model-predicted y values. First we use `range()` to grab the min and max x values for a given country and stick them into `xi`. Then we stick it into the model formula, where `LM["x"]` is the slope coefficient, and `LM[paste("ctry",ctryi,sep="")]` grabs the additive country coefficient. `paste()` is used to concatenate character strings in R and is one of the most useful functions you can know.

```
> LM <- unlist(lm(y ~ x + ctry, data = sdat)$coef)
```

```
> LM["ctryCZ"] <- 0  # a hack to make life easier
> xmin <- xmax <- ymin <- ymax <- c()
> for (i in 1:7) {
      ctryi <- names(my7cols)[i]
      ind <- sdat$ctry == ctryi
      xi <- range(sdat$x[ind])
      xmin[i] <- xi[1] - 5
      ymin[i] <- LM[1] + LM["x"] * (xi[1] - 5) + LM[paste("ctry",
          ctryi, sep = "")]
      xmax[i] <- xi[2] + 5
      ymax[i] <- LM[1] + LM["x"] * (xi[2] + 5) + LM[paste("ctry",
          ctryi, sep = "")]
 }
```

Now we have four vectors ready and can replot, drawing country-specific predicted line segments using `segments()`.

```
> # define empty plot of required dimensions
> plot(NULL, type = "n", xlim = c(0, 125), ylim = c(0,
      700), xlab = "meaningless x", ylab = "meaningless y")
> # iterate over country names to add points:
> for (i in names(my7cols)) {
      ind <- sdat$ctry == i
      points(sdat$x[ind], sdat$y[ind], pch = 19, col =
colalpha(my7cols[i],
          30))
 }
> segments(xmin, ymin, xmax, ymax, my7cols, lwd = 2)
> legend("bottomleft", fill = my7cols, legend = names(my7cols))
```

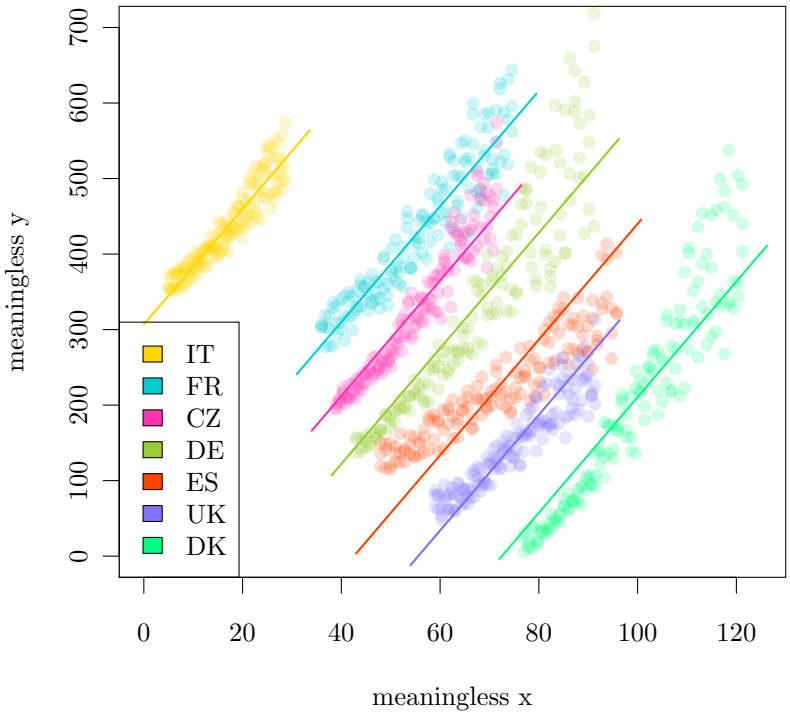

Figure 4: Fitting multiple OLS to subsets, assuming linearity and equal slopes

You could without much effort add lines for confidence intervals to these lines, using the same steps as for the naive(r) regression that we started with. For that you'd want to throw the `predict()` steps into a loop as well to be able to calculate separate lines for each country. Even better would be to refit the model allowing slopes to vary between countries, and even better still would be to allow a non-linear fit, since the within-country pattern is exponential, rather than linear. For your reference, you can do this using the `nlme()` function in the package `MASS` or with `glmer()` in `lme4a`, and there are multiple free online tutorials for doing that kind of thing. Instead of doing that, we'll pretend we don't know the true pattern to each point cloud, and we'll jump to non-parametric fitting. There are many ways to do this. You can find a good primer with John Fox's non-parametric regression tutorial here: `http://cran.r-project.org/doc/contrib/Fox-Companion/appendix-nonparametric-regression.pdf`. Really all we want is to put some decent-looking confidence bands on some decent-fitting line decribing each cluster. This we'll do using the `loess()` function, which, along with the spline family of functions, is extremely useful in demography. Here we just want an informative plot, but really you can use non-parametric function to smooth any kind of noisy data, e.g. ASFR curves for small areas, or to infer single age rates from 5-year age groups, etc.

Another good choice in that situation is to simply make all of your confidence bands a transparent light grey. To keep everything clear, plot points first, then the confidence bands, then the predicted fitted lines. Here's an example that iterates over everything:

```
> # define empty plot of required dimensions
> plot(NULL, type = "n", xlim = c(0, 125), ylim = c(0,
    700), xlab = "meaningless x", ylab = "meaningless y", main =
"color grid background\nsemitransparent 95% CI\nloess smoothing
over points")
> # par['usr'] = coords of user area
> # make a light grey rectangle
> rect(par("usr")[1], par("usr")[3], par("usr")[2],
    par("usr")[4], col = "#EBEBEB")
> # plot gridlines at ticks
> abline(v = axTicks(side = 1), col = "white")
> abline(h = axTicks(side = 2), col = "white")
```

The above chunk gets the plot area set up. The following piece repeats the same routine for each country subset of points.

```
> # iterate over country names to add points:
> for (i in names(my7cols)) {
    ind <- sdat$ctry == i
    points(sdat$x[ind], sdat$y[ind], pch = 19, col =
colalpha(my7cols[i],
        30))
    # fit loess
    lo.i <- loess(y ~ x, data = sdat, subset = ctry == i)
    x <- seq(min(sdat$x[ind]), max(sdat$x[ind]), length.out =
50)
    xnew <- data.frame(x = x)
    # predict center and s.e.
    pred.i <- predict(lo.i, newdata = xnew, se = TRUE)
    fit <- unlist(pred.i["fit"])
    # 1.96*se = 95% conf
    ci <- 1.96 * unlist(pred.i["se.fit"])
    # polygon explained in text
    polygon(x = c(x, rev(x)), y = c(fit + ci, rev(fit - ci)),
        col = "#44444430", border = NA)
    # line for fit
    lines(x, fit, col = my7cols[i], lwd = 2)
 }
> legend("bottomleft", fill = my7cols, legend = names(my7cols))
```

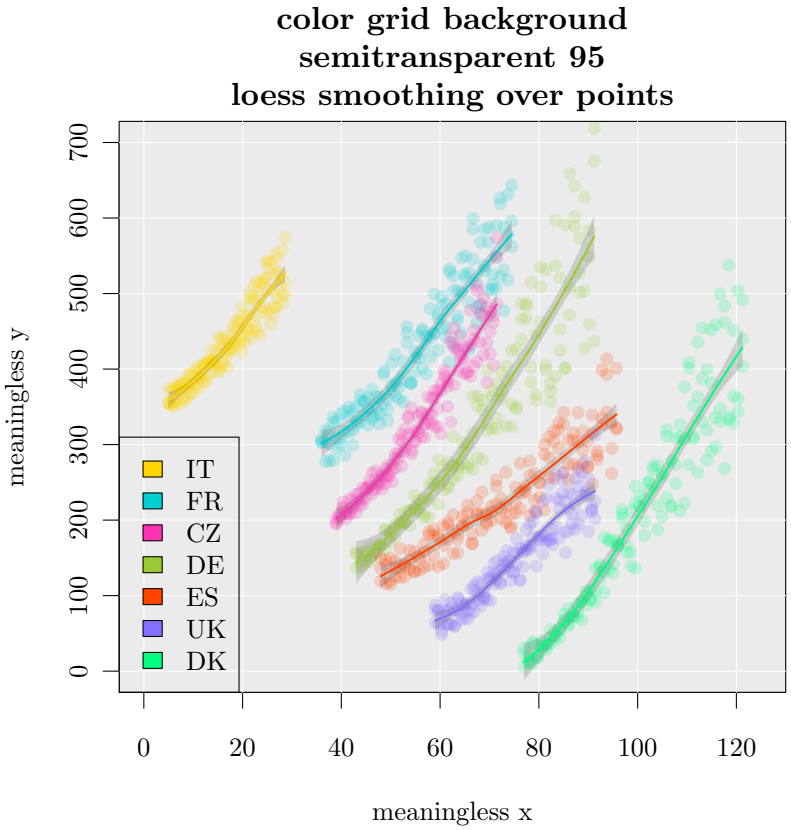

Figure 5: Adding loess smoothers with confidence bands to subsets

The background grid probably introduced you to the `axTicks()` function, which is pretty self-explanitory. This function is sometimes called by `plot()`- we use it to make sure our grid lines are on the same ticks. Then we plot the colored points in the same way as before, also semi transparent. Then we fit a loess line[4]. We want to extract from this the predicted fit and standard errors (of the loess fit). The `predict()` function takes the `newdata` argument, which are the x values for which we want predictions, which must be supplied as a `data.frame`. These values must be within the range of the original data, since this is a local regression (splines can go beyond the data range). I went ahead and multiplied the standard errors by 1.96 to get out to the 95% level (`ci`).

The way most eyes like to see confidence intervals plotted, rather than lines, is as a shaded region. It's a matter of preference, of course, but most R users don't bother to figure out the syntax. My idea is to use `polygon()` so simply draw the CI as a simple shape. Each x value for the prediction has a high and low estimate, thus we need to give each x twice. Think of yourself as drawing a line *around* the circumference of the area you want to shade: you need to

[4]you can make the line more or less sensistive to noise by setting the `span` argument, which we left at the default (.75)

go one way, then come back the other way, hence (`rev()`). The x argument, `c(x,rev(x))`, does this for us. The y argument must be given in the same order: first left to right over the top, then right to left around the bottom `c(fit+ci,rev(fit-ci))`. Your starting and ending points are automatically connected. `col` refers to the fill color and `border` is the shape outline. Last, we draw a thick colored line for the smoother itself.

A mini-trick is also in the title: any argument that gets sent to `text()`, in this case `main`, inserts a line break whenever it sees `"{}n"` in the middle of the text. That's how the multiline title gets accomplished.

3 Demographic Surfaces in R

Whenever you have data cross-classified along three or more continuous dimensions, you should think about looking at it as a surface. These might not even be strictly demographic dimensions! You could in principal plot a surface of mortality by income over time [5]. Demographers generally choose to plot the variable of interest over age and time. You could also however plot a marriage or fertility surface where age of female runs over one axis and age of male over another- these are rarely seen in the wild, but have a straightforward interpretation. I mention all these atypical examples precisely so that you can start to think creatlively with the powerful tools at your fingertips: You can get really great ideas just by looking at your data in a different way. In any case, data that are cross-tabulated like this are easy to find or can easily be aggregated from microdata. I'll start this part by running over the simplest surface functions in R, and how to get the most out of them, then we'll do some examples with real data (and make some more color ramps).

3.1 Functions for plotting surfaces in R

1. `image()` base (`graphics`). Simplest- Be careful which axis is which if your data happen to be square. `heat.colors(12)` is the default color ramp
2. `heatmap()` base (`stats`)- puts a dendogram (cluster tree) in the margins too. Beware- it might reorganize your data according to the dendogram! Do not use this for demographic surfaces- it's for finding clusters in your data!
3. `image.plot()` `fields` package. Like `image()`, but includes legend. Default color ramp goes from cool to hot colors (`tim.colors(64)`)
4. `levelplot()` `lattice` package. Transposes data (columns in y, rows in x). Default color ramp is cyan to pink
5. `contourplot()` in `lattice` package. Plots isolines like a topo-map. For color, add `region=TRUE`. Also transposed.
6. `geom_tile` in `ggplot2` package. No matrices: organize your data as a data.frame with 3 columns "age", "year", and "z". See example later on.

In general, you need to pay attention to whether surface plotting functions need your data as a matrix, data.frame, in gridded or long format. The examples here use data from the HMD for the USA, 1933-2007 (75 years), ages 0-110+ (111 ages). We start with a matrix object, `MxMat`, extracted from the HMD and organized to have years across the columns and ages going down the rows (as opposed to the default long format given by the HMD).

```
> load("data//MxMat.RData")
> MxMat[1:4, 1:6]
```

[5] you'd probably be the first, and a righteous demographer indeed

```
      1933     1934     1935     1936     1937     1938
0 0.054391 0.059665 0.053952 0.055168 0.053667 0.050897
1 0.008746 0.009953 0.008258 0.008141 0.007856 0.007369
2 0.004013 0.004540 0.003789 0.003887 0.003717 0.003322
3 0.002868 0.002979 0.002756 0.002692 0.002490 0.002250
```

3.1.1 image()

Let's jump straight into examples. `image()` is probably the most primitive and flexible function for plotting surfaces, but its defaults aren't necessarily the most convenient for for Lexis-like surfaces. That is to say, you can do whatever you want with it, if you have the time. Function input must be a `matrix` (no `data.frames`). If your data matrix has years across the columns and ages going down the rows, beware: it will be plotted transposed, with years going down the y-axis and ages going over the x axis, and you will not necessarily notice this because 1) axes are scaled to go from 0 to 1 and 2) data are stretched to fit into the plot region evenly. For Lexis surfaces you always want proportionality between the rendering of age and years. This we can force by specifying an aspect ratio of 1 with `asp=1`.

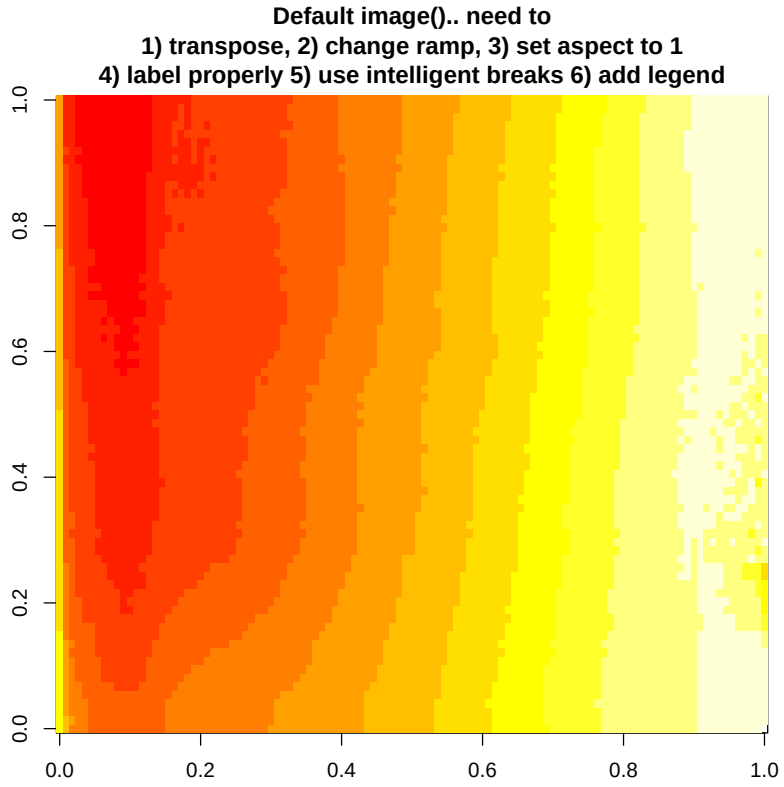

Figure 6: image() function, leaving all defaults.

```
> load("data//MxMat.RData")
> image(log(MxMat))
```

Now we'll run through a variety of pointers for how to get the most out of the `image` function. Many of these will be little details, and we'll do a large amount of *overplotting* (layering) to get the details.

First we set value breaks. Remember that one the one hand you want to render your data in logged form, but you want legend labels to be nice round intuitive numbers. Always specify break points if you'll be 1) making more than one surface plot or 2) want to have a clean and understandable legend. Be sure to specify 1 more break point than you have colors, and that all of your data are contained within your extreme breakpoints. I wanted 20 colors here, so I chose 21 evenly spaced breaks over the range `log(1e-5)` to `0`, which will therefore include several nice clean points to be picked out for labelling later: 1e-5, 1e-4, 1e-3, .01, .1, 1 (along with some ugly numbers in between).

Second, we reverse the `heat.colors()` because to most people red is more intense than yellow (don't argue this point with chemists). We are sure this time to take the transpose `t()` of the matrix, so that ages run up the y axis. The default axes plot ugly, so we switch them off with `axes = FALSE` and then manually add on axes later with `axis()`. We also give our own bounding box. One very fine point to get the cells properly centered is to remember that the data begin in 1933, but end at the *end* of 2007, so we specify the x range from 1933 to 2008, rather than 2007. If you specify up until 2007, you'll notice that individual cells overlap years and ages. Likewise, add 1 to your highest age. In this way we have x and y values that cleanly *bound* the data grid.

```
> brks <- approx(log(c(1e-05, 1)), n = 21)$y
> image(x = 1933:2008, y = 0:111, z = t(log(MxMat)),
     col = rev(heat.colors(20)), asp = 1, axes = FALSE, xlab =
"Year",
     ylab = "", breaks = brks, main = "US females, ln(Mx),
1933-2007 (HMD)")
> rect(1933, 0, 2008, 111, xpd = T)
> axis(1, at = seq(from = 1940, to = 2000, by = 10),
     cex = 0.75)
> axis(2, at = seq(from = 0, to = 110, by = 10), pos = 1933)
> mtext("Age", 2, -2, xpd = T)
```

Unfortunately `image()` does not offer a legend, but we'll go through and design a nice one here, in case you need to make one for future reference. By the way, we'll be doing *manual* rescaling and shifting in order to get this thing looking good. If you're OK using some else's default color strip legend, then jump to `image.plot()`, which simply gives you one. Honestly this one is better, though:

```
> # the legend is a bit tricky to do by hand:
> legendymax <- 100
```

```
> legendymin <- 10
> legendxleft <- 2012
> legendxright <- 2020
> legendy <- seq(from = legendymax, to = legendymin,
    length.out = 21)
> rect(legendxleft, legendy[-1], legendxright, legendy[-21],
    col = heat.colors(20), xpd = T, border = NA)
> rect(legendxleft, legendymin, legendxright, legendymax)
> # actually just the ticks are tricky:
> legendlabs <- c(1, 0.1, 0.01, 0.001, 1e-04, 1e-05)
> legendticks <- log(legendlabs)
> legendticksscale <- (legendymax -
legendymin)/diff(range(legendticks))
> legendticks <- (legendticks + abs(min(legendticks))) *
    legendticksscale + legendymin
> segments(legendxright, legendticks, legendxright +
    2, legendticks)
> text(legendxright + 1, legendticks, legendlabs, pos = 4)
```

The above code chunk could very easily be converted into a function, but we're not that far along yet. Summarizing, we specify the upper and lower limits of the area where we want to put the legend (over in the right margin, so the coordinates actually go beyond the data). Then we want 21 evenly spaced points to draw the color ramp into (`legendy`), and we can go ahead and draw the colored rectangles (`rect()` is vectorized, no need to loop). Remember we want clean labels (`legendlabs`), but that colors are mapped into log space. Furthermore, we need to spread our ticks out over a different range (in plot coordinates) than that directly given by `log(legendlabs)`. That's why we make `legendticksscale` (it stretches our ticks out over the given range), and we shift the whole series upward. Whew! With `legendticks` defined, we can draw the ticks manually with `segments()`, which is also vectorized, and likewise with the labels (`text()`).

This however just gets us the major tick marks. Ocassionally, to make such scales clear to the viewer, you'll want to plot minor tick marks in such a way as to make clear that the scale is log (or remind the reader what that means). This little code chunk puts in the minor ticks, by 1) getting a clean sequence from e.g. 1 to .1 (1, .9, .8, .7, .6, .5, .4, .3, .2, .1), then logging, scaling and shifting it prior to drawing the ticks. When completed, it does the same for a clean sequence from .1 to .01, and so forth, until the intermediate legend ranges are filled up nicely. If and when you want to add log ticks that work like these, you'll have to find a manual solution, such as this, as there is no standard or common way of doing it.

```
> # and if you really want to be clear about the log scale:
> for (i in 1:5) {
    yi <- log(seq(legendlabs[i], legendlabs[i + 1], length.out =
10))
    yi <- (yi + abs(min(yi))) * legendticksscale +
rev(legendticks)[i]
```

```
    segments(legendxright, yi, legendxright + 1, yi)
}
```

Finally, another nice detail is to overlay Lexis reference lines. The horizontal and vertical lines are straightforward, but the diagonals are taken care of with a rather hacky but effective solution: we know we'll need 18 or 19 diagonals in order to cover the whole surface (number of vertical reference lines plus number of horizontal reference lines), and each will have a lower left and an upper right coordinate. `cmat` is just a container for these coordinates, in order the columns hold: x left, y low, x right, y high. At first they are given coordinates as if the surface drawn were not bounded by years, the the function `cslicer()` does proportional snipping (to maintain the 45 degree angle) to make sure that none of the lines go beyond the surface. If and when you want to add reference lines to your surface, you'll always have to think of some kind of hack because there is no standard way to do it. My mantra: if it looks good, adds information to the plot and doesn't cost a lot of ink, then do it. Reference lines in this case turn a shapeless ocean of death into an interpretable Lexis surface.

```
> segments(1933, seq(0, 110, by = 10), 2008, seq(0,
    110, by = 10), col = "#00000030")
> segments(seq(1940, 2000, by = 10), 0, seq(1940, 2000,
    by = 10), 111, col = "#00000030")
> # cohort lines are always a head-scratcher:
> cmat <- matrix(ncol = 4, nrow = 18)
> cmat[, 1] <- seq(1830, 2000, by = 10)
> cmat[, 2] <- rep(0, 18)
> cmat[, 3] <- cmat[, 1] + 111
> cmat[, 4] <- 111
> # the x argument is a row of cmat
> cslicer <- function(x) {
    if (x[1] < 1933) {
        d <- 1933 - x[1]
        x[1] <- 1933
        x[2] <- 0 + d
    }
    if (x[3] > 2008) {
        d <- x[3] - 2008
        x[3] <- 2008
        x[4] <- x[4] - d
    }
    x
}
> cmat <- apply(cmat, 1, cslicer)
> segments(cmat[1, ], cmat[2, ], cmat[3, ], cmat[4,
    ], col = "#00000030")
```

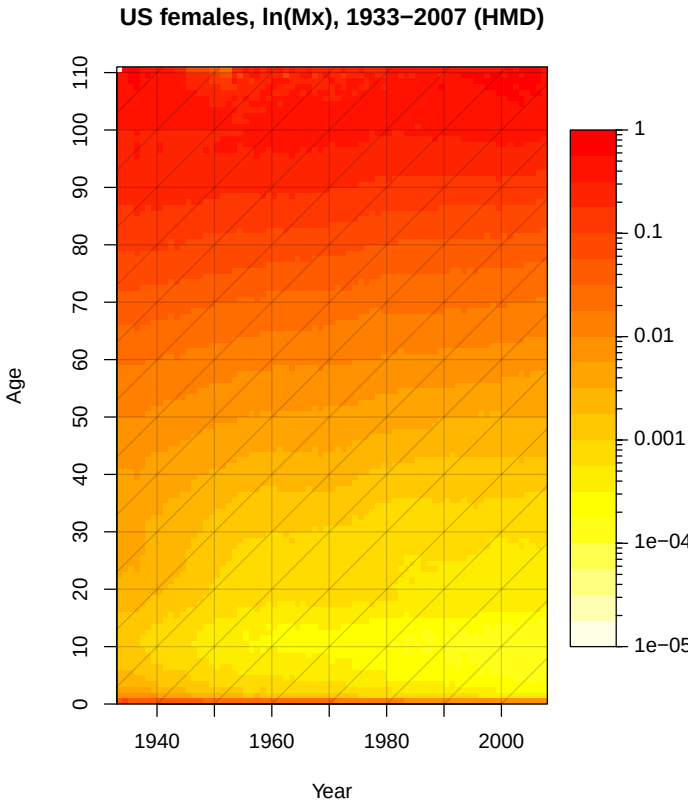

Figure 7: image() function, tweaked to produce a Lexis surface.

In brief, most of the jewlery for this Lexis surface was acheived doing creative overlaying. Of course, it's not all always necessary, and will depend on what you'll want to point out from the surface. If you'll be talking about cohort patterns, do yourself a favor by labelling the cohort reference lines (probably moving the legend over to the left).

3.1.2 image.plot()

You can at least save part of the headache dealing with the legend by jumping straight to `image.plot()` from the `fields` package. `image.plot()` simply calls `image()`, from above, and is basically a convenience function just for the sake of the legend. The default color ramp is divergent [6], meaning that it goes from cool colors at low values to warm colors for high values- at first this is intuitive, but then you start thinking: Mortality is always positive, and all mortality is, well, mortality, so why use cool colors at all when there's room for improvement? Here's what the function will spit back if you do minimal argument specification, accepting all the defaults:

[6] it's called `tim.colors()`, no relation

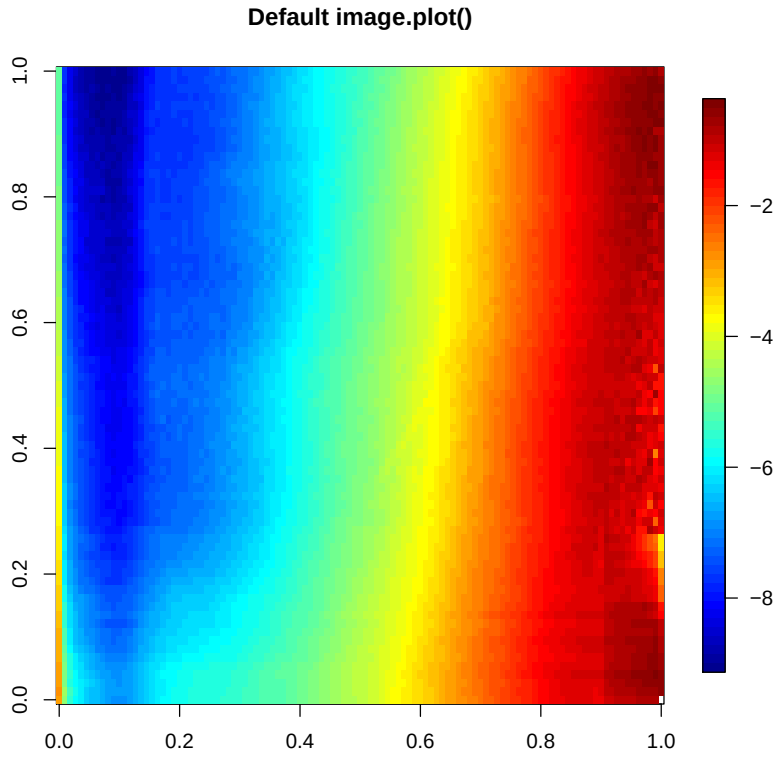

Figure 8: image.plot() function, leaving all defaults.

```
> library(fields)
> load("data//MxMat.RData")
> image.plot(log(MxMat), main = "Default image.plot()")
```

In order to get the surface to render properly, you'll have to do all the same tricks as for `image()`, so I'll jump right to decent argument specification. The only argument that changes from the prior example was `axis.args=list(blabla)`, which is the part that tells the legend ticks to be spaced evenly over the log scale, but to be labeled with not-logged numbers. Notice that I told it to include 1 and .0001 and it still didn't: it's a bit less flexible. Another downside is that you don't really know where the legend is in coordinates, so you cannot really do the extra tweaking in terms of adding more ticks, or putting in the missing labels, unless you're willing to look into the source code and figure out how it gets positioned. Other adjustments include transposing the data, fixing the aspect ratio to 1 and providing even breaks. As for Lexis reference lines, that would work in the exact same way as for `image()`, no strings attached: the coordinates are identical. The Lexis reference line code is repeated but not shown.

```
> brks <- approx(log(c(1e-05, 1)), n = 21)$y
```

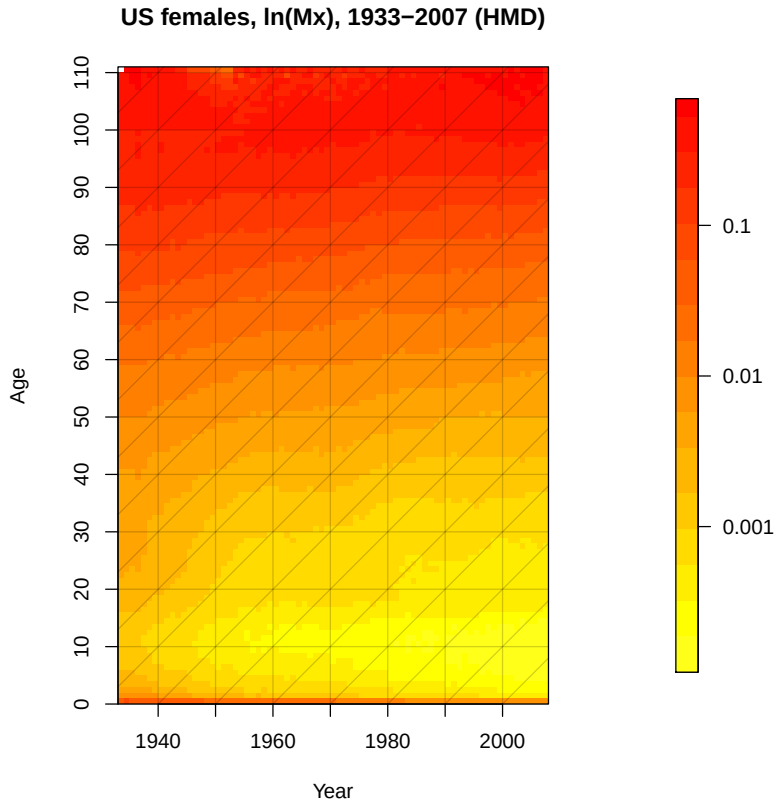

Figure 9: image.plot() function, tweaked to produce a Lexis surface.

```
> legendlabs <- c(1, 0.1, 0.01, 0.001, 1e-04, 1e-05)
> image.plot(x = 1933:2008, y = 0:111, z = t(log(MxMat)),
     col = rev(heat.colors(20)), asp = 1, axes = FALSE, xlab =
"Year",
     ylab = "", breaks = brks, main = "US females, ln(Mx),
1933-2007 (HMD)",
     axis.args = list(at = log(legendlabs), labels = legendlabs))

> rect(1933, 0, 2008, 111, xpd = T)
> axis(1, at = seq(from = 1940, to = 2000, by = 10))
> axis(2, at = seq(from = 0, to = 110, by = 10), pos = 1933)
> mtext("Age", 2, xpd = T)
```

In short, `image.plot()` is a time-saver, and might be the preferred function if you're powering through your data for diagnostics. The extra goodies on the legend are of course still possible, but not worth our effort, so we lose a bit of legend clarity and aesthetics[7].

[7] if anyone invests in perfecting the legend, let me know and I'll gladly update this tutorial

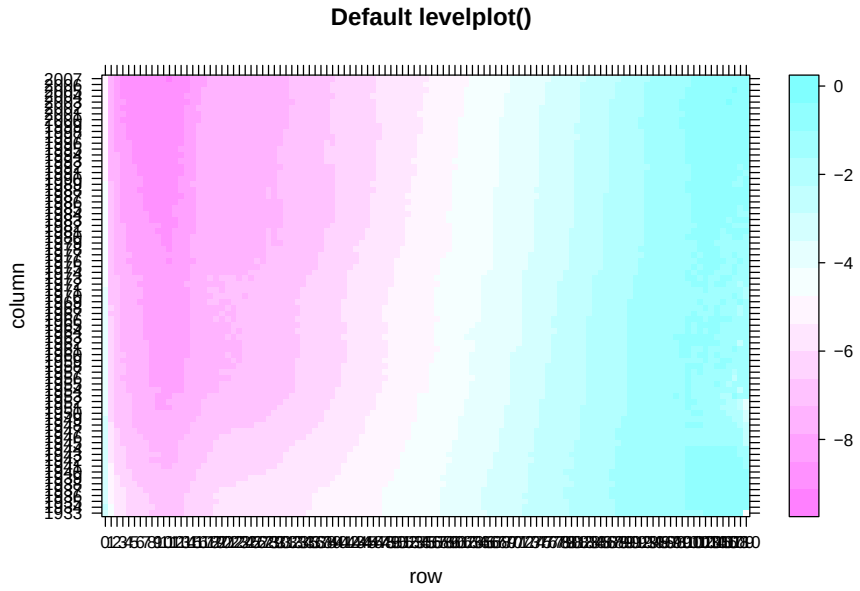

Figure 10: `levelplot()` function, leaving all defaults.

3.1.3 levelplot()

Levelplot is the `lattice` answer to plotting surfaces. If you really want to understand the guts of what is going on this code, then you should really read the Jakoby material linked earlier in this document. I personally do not use `lattice` often, precicely because the specification of axes, labels and overplotting is a chore. Here I first display the default output so that you're not shocked when you first try it. On the positive side, note that the aspect ratio by default maintains a 45 degree angle over x and y values, plus there is a color strip legend by default. This we like. The default color ramp is not so useful, nor are the crowded tick labels.

```
> library(lattice)
> levelplot(log(MxMat), main = "Default levelplot()")
```

The steps in this code are in need of some explanation, I admit. First, you need to know that lattice does not (easily) let you modify a plot once it's made, so you need to (or it's best to) specify everything from within a single

and you'll get credit.

function call. For this reason, all of the Lexis reference line overplotting needs to get pre-written into a custom function, `lexisreferencelines()`, called a *panel* function that later gets called within the main lattice function, in this case `levelplot()`. To understand this more, please refer to the online material mentioned before. The other quirks I'll explain here. First, `levelplot()` will take a variety of things as its main data argument, not necessarily a matrix[8]. Here we gave it a matrix (the transpose of `log(MxMat)`), just as with `image()`, but we need requires a few extra steps in order to get the axes right.

When specifying a matrix, by default `levelplot()` will consider your x and y values to start at 0 and count upward, which is *almost* correct for ages and very far off for years. This we rectify by supplying the `row.values` and `column.values` arguments. Like other surface functions in R, each cell will be centered on its assigned x and y value, and so in order to render properly, we shift the x and y values by .5, but specify the x and y limits to stretch over the entire data range (1993-2008 (end of 2007), and 0-111 (110+)). This is a very fine point, easy to not notice, but essential to get right if you want the image to be exact. Next we specify the color ramp with `color.regions`, just as we did `col` in the other functions. Color strip legend arguments get sent as a list to the `colorkey` argument: notice that I give the same values, and they are all correctly plotted (better than `image.plot()` already). The `scales` argument is the way to be precise about axis labelling in `lattice`: it is also give as a list, with elements x and y. If omitted, the plot renders just fine, but you only get axis labels every 20 ages/years[9]. Finally, and most importantly, we specify a panel function, in this case our `lexisreferencelines` function. Panel functions tend to follow this form, and can be very complex if desired.

```
> library(lattice)
> brks <- approx(log(c(1e-05, 1)), n = 21)$y
> legendlabs <- c(1, 0.1, 0.01, 0.001, 1e-04, 1e-05)
> # panel function for inside levelplot():
> lexisreferencelines <- function(...) {
     panel.levelplot(...)
     # cohort lines (same cmat as before)
     panel.segments(cmat[1, ], cmat[2, ], cmat[3, ], cmat[4, ],
         col = "#00000040")
     # period lines
     panel.segments(seq(1940, 2000, by = 10), 0, seq(1940, 2000,
         by = 10), 111, col = "#00000040")
     # age lines
     panel.segments(1933, seq(0, 111, by = 10), 2008, seq(0, 111,
         by = 10), col = "#00000040")
 }
> # the main surface function:
> print(levelplot(t(log(MxMat)), row.values = 1933.5:2007.5,
     column.values = 0.5:110.5, xlim = c(1933, 2008), ylim = c(0,
```

[8] This could also be done with a formula and `data.frame` in long format, which is not shown here.

[9] Actually we also got rather rough labels with `axis()` before, but this could have been fixed by manually labelling with `text()`.

```
        111), at = brks, col.regions = rev(heat.colors(20)),
    colorkey = list(at = brks, labels = list(at =
log(legendlabs),
        labels = legendlabs)), scales = list(x = list(at =
seq(1940,
        2000, by = 10), labels = seq(1940, 2000, by = 10)), y =
list(at = seq(0,
        110, by = 10)), labels = seq(0, 110, by = 10)), xlab =
"Year",
    ylab = "Age", main = "Same surface plot using lattice
levelplot()",
    panel = lexisreferencelines))
```

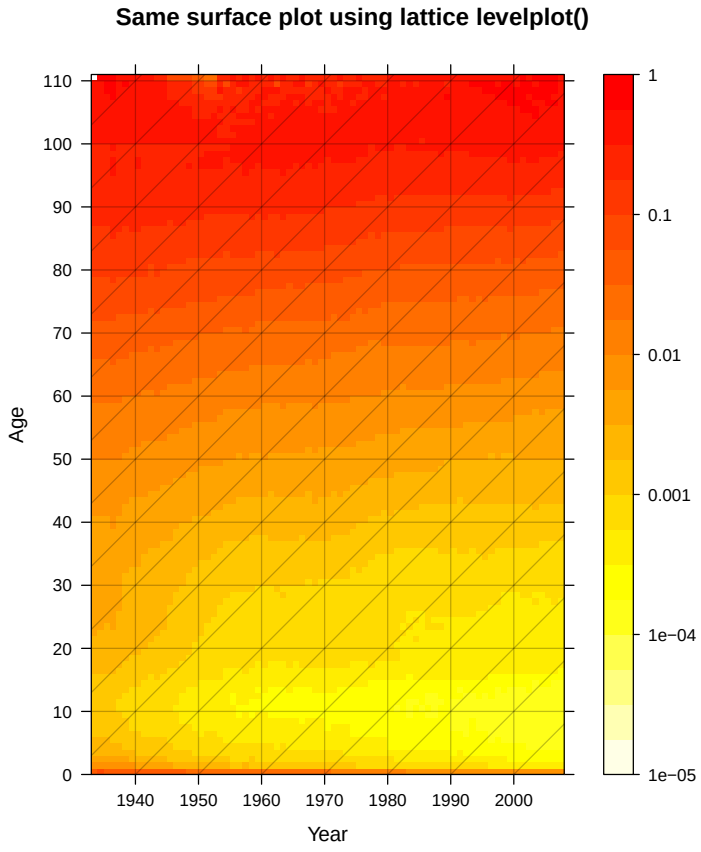

Figure 11: `levelplot()` function, tweaked to produce a Lexis surface.

Sometimes you see surfaces displayed via contour plots that resemble topographic maps. These may or may not be desirable for your particular surface. The `contourplot()` function in lattice is a companion to `levelplot()`, and works in much the same way. A shortcut that I used to produce the below plot was to simply add the argument `contour=TRUE` to the `levelplot()` list of arguments. Notice that the contour lines themselves are smooth interpolations and

do not follow cell borders exactly. This can often help piont out details in the surface, most often period effects, that can go unnoticed on color-only surfaces. Local maxima and minima are also marked. In my opinion the following plot is locally more legible, but on the whole suffers from information overload, since the contour lines pick up noise from ages. You could clean up the image by not overlaying Lexis lines, but then its difficult for your eyes to keep track of age, period and cohort. Whether or not to use contour lines is thus a matter of judgement.

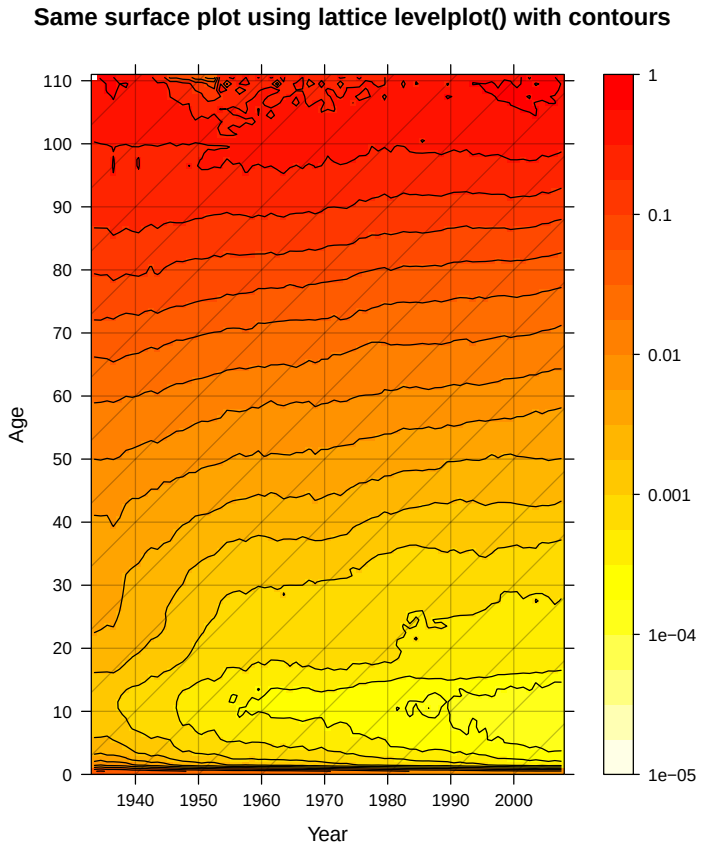

Figure 12: `levelplot()` function, tweaked to produce a Lexis surface, with contour lines.

This concludes our brief foray into `lattice` territory. Notice this took considerably fewer lines of code to produce than did the nearly identical `image()` version. The only drawback is the extra time spent having to learn its idioms, as opposed to brute force intuitive base graphics.