# Fancy Plotting in R for EDSDers: A tutorial

Tim Riffe

November 14, 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 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 color

There are different ways to specify colors in R plots. In general, do not always limit yourself to always 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 script. 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] "#FF0000FF" "#FFDB00FF" "#49FF00FF" "#00FF92FF" "#0092FFFF"
[6] "#4900FFFF" "#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[1]. 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.

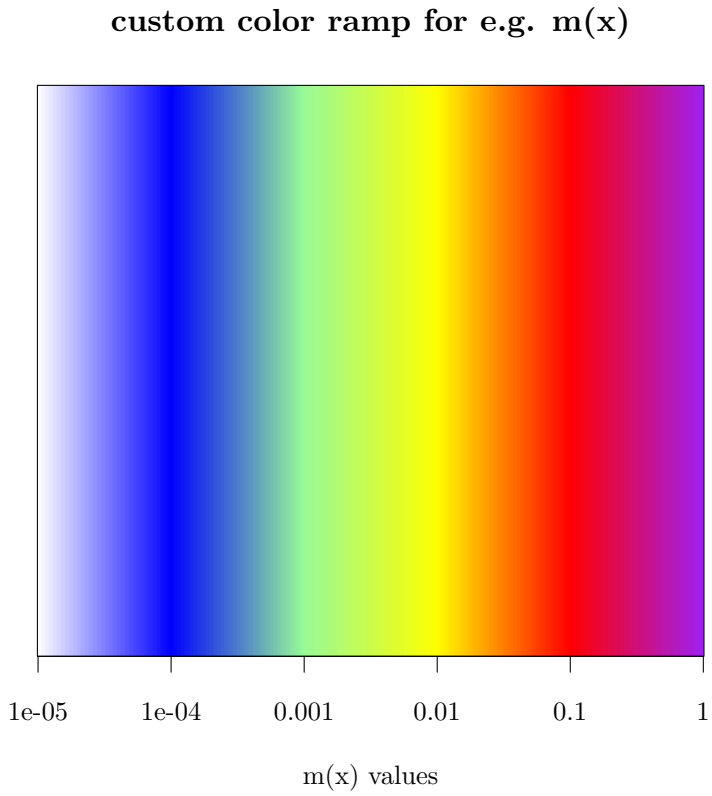
[1] When you're feeling too lazy to go to the R Colors webpage, invent something random, keeping this pattern in mind.

I use them mostly for mortality (logged) and fertility surfaces[2].

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:

```
> 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]You could make a migration surface and you'd be the first!




## 2.3 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
"#FFD70050" "#00CED150" "#FF34B350" "#9ACD3250" "#FF450050"
           UK          DK
"#836FFF50" "#00FF7F50"
```

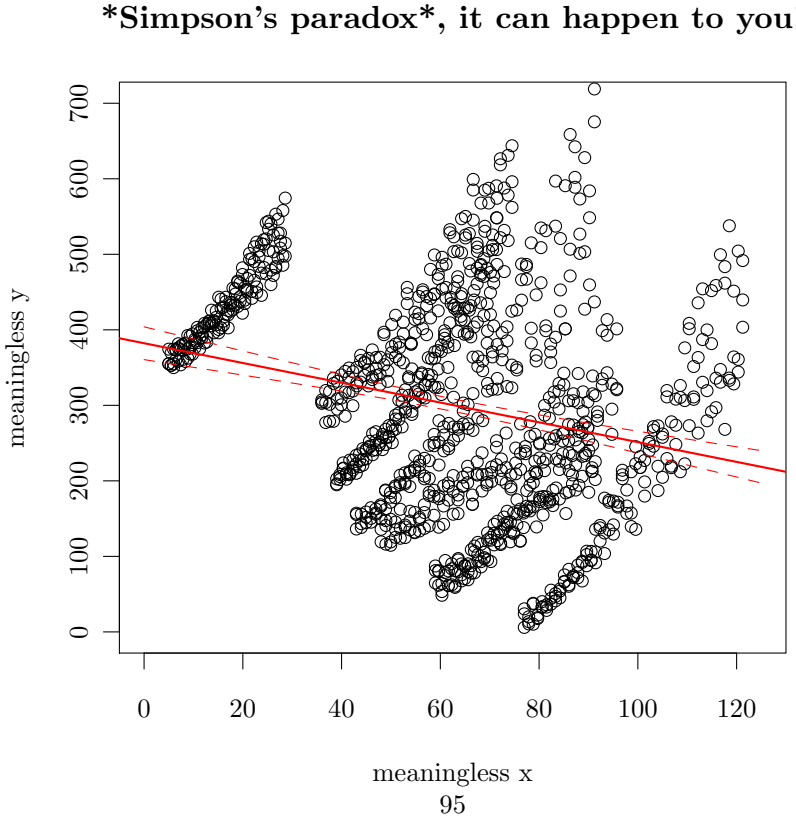
## 2.4 An Example

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)
```

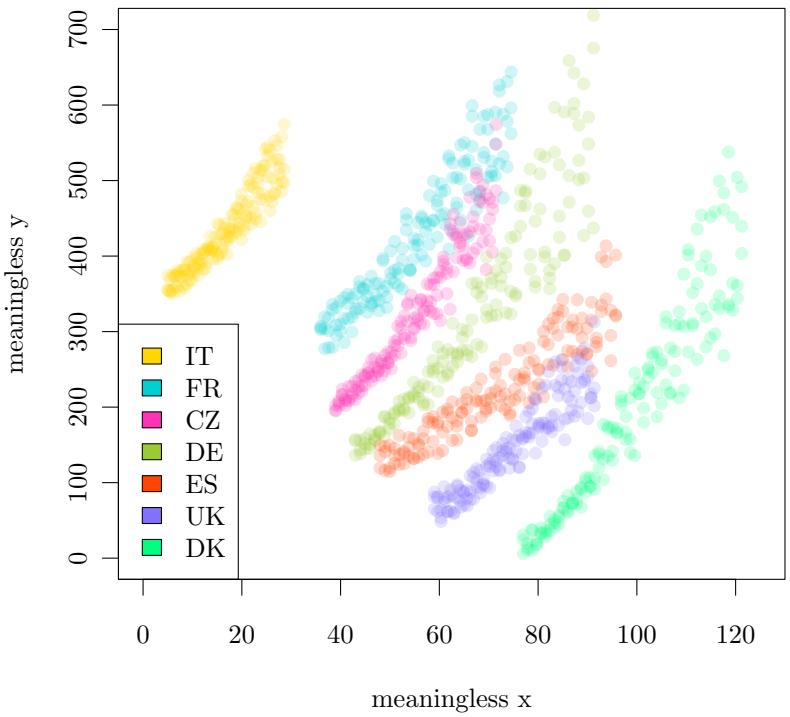



There is some mixing in there that is difficult to separate visually, though it's clear that points are somehow grouped, probably by country. A quick and ugly diagnostic, and a plotting function you should frequently use when getting to know your data is `pairs()`, which plots a matrix of bivariate plots:

```
> # 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))
```



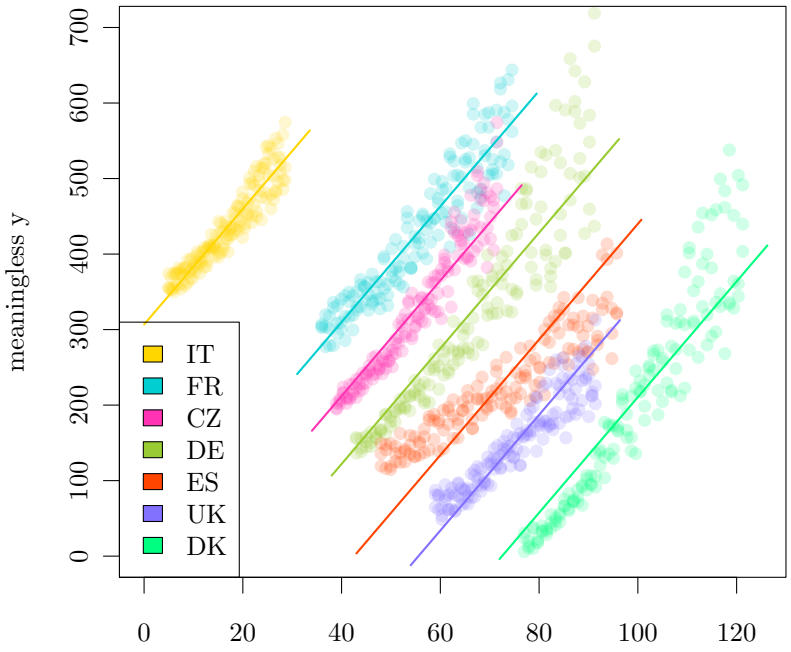

It being clearly the case that each country shows a similar but shifted pat-

tern, 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))
```

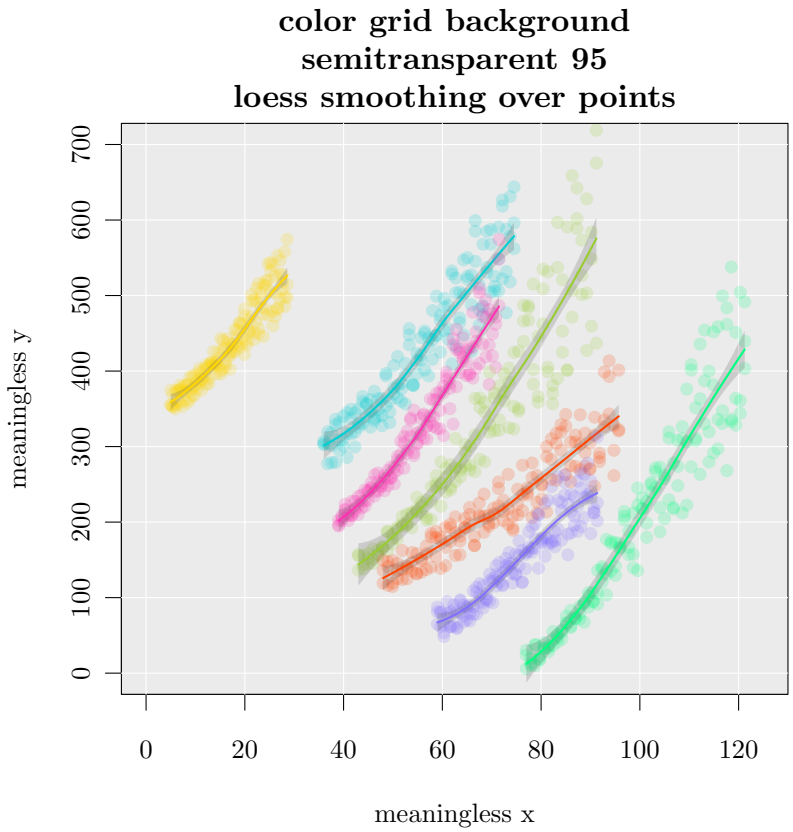



You could without much effort add lines for confidence intervals to these lines, using the same steps as for the naive(r) regression that we started with. For that you'd want to throw the `predict()` steps into a loop as well to be able to calculate separate lines for each country. Even better would 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 averything 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")
> # 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)
 }
```




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[3]. 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 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

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

`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 overplotting

# 4 transparency

# 5 surfaces and animation

# 6 pairs

# 7 confidence intervals