STAT 545A
Class meeting #7
Monday, September 30, 2013

Dr. Jennifer (Jenny) Bryan

Department of Statistics and Michael Smith Laboratories

```
xyplot(y ~ x, data, ...)
```

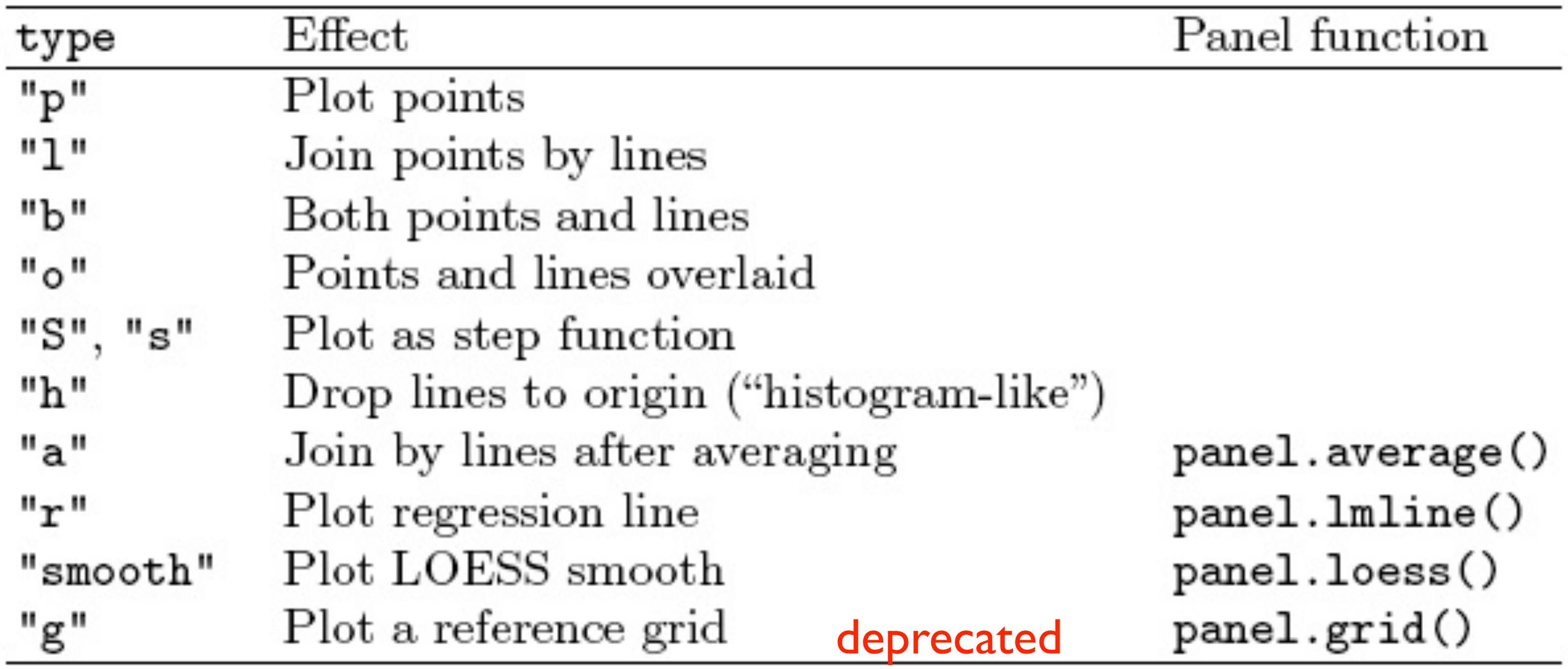
See the full xyplot() tour here:
http://www.stat.ubc.ca/~jenny/STAT545A/block09_xyplotLattice.html

```
xyplot(y ~ x, data,
       subset = ?, ...)
```

```
, type = ....,
```

easy to add a regression line, a loess smooth, connect the dots, etc. via type

- The `type` argument is incredibly useful. Use it!
- It is a character vector consisting of *one or more* of <read the help file for the values>. If `type` has more than one element, an attempt is made to combine the effect of each of the components.

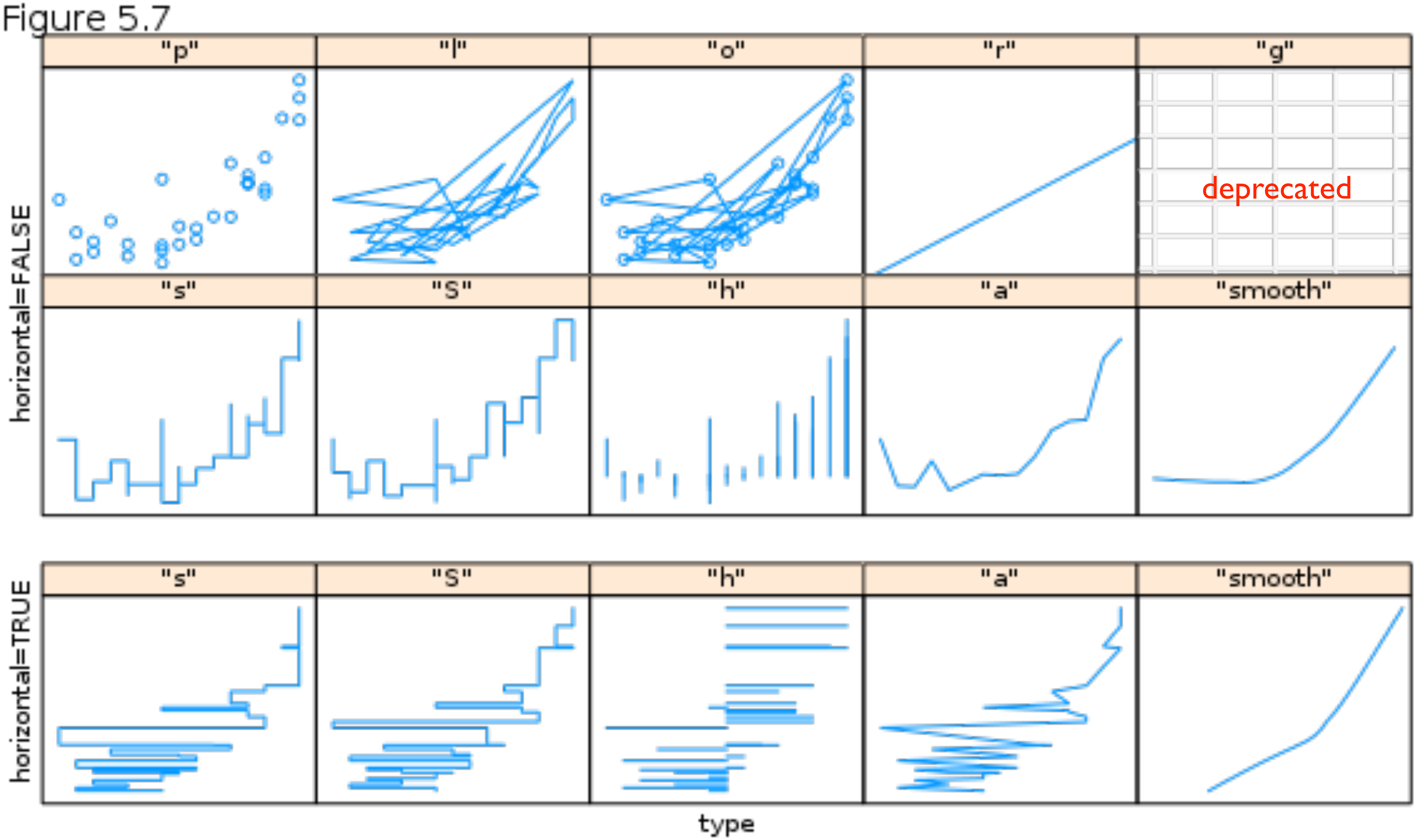


type	Effect	Panel function
"p"	Plot points	
"l"	Join points by lines	
"b"	Both points and lines	
"o"	Points and lines overlaid	
"S", "s"	Plot as step function	
"h"	Drop lines to origin ("histogram-like")	
"a"	Join by lines after averaging	panel.average()
"r"	Plot regression line	panel.lmline()
"smooth"	Plot LOESS smooth	panel.loess()
"g"	Plot a reference grid deprecated	panel.grid()

Table 5.1. The effect of various values of the **type** argument in **panel.xyplot()**.

From Ch. 5 in Sarkar (2008).

Figure 5.7



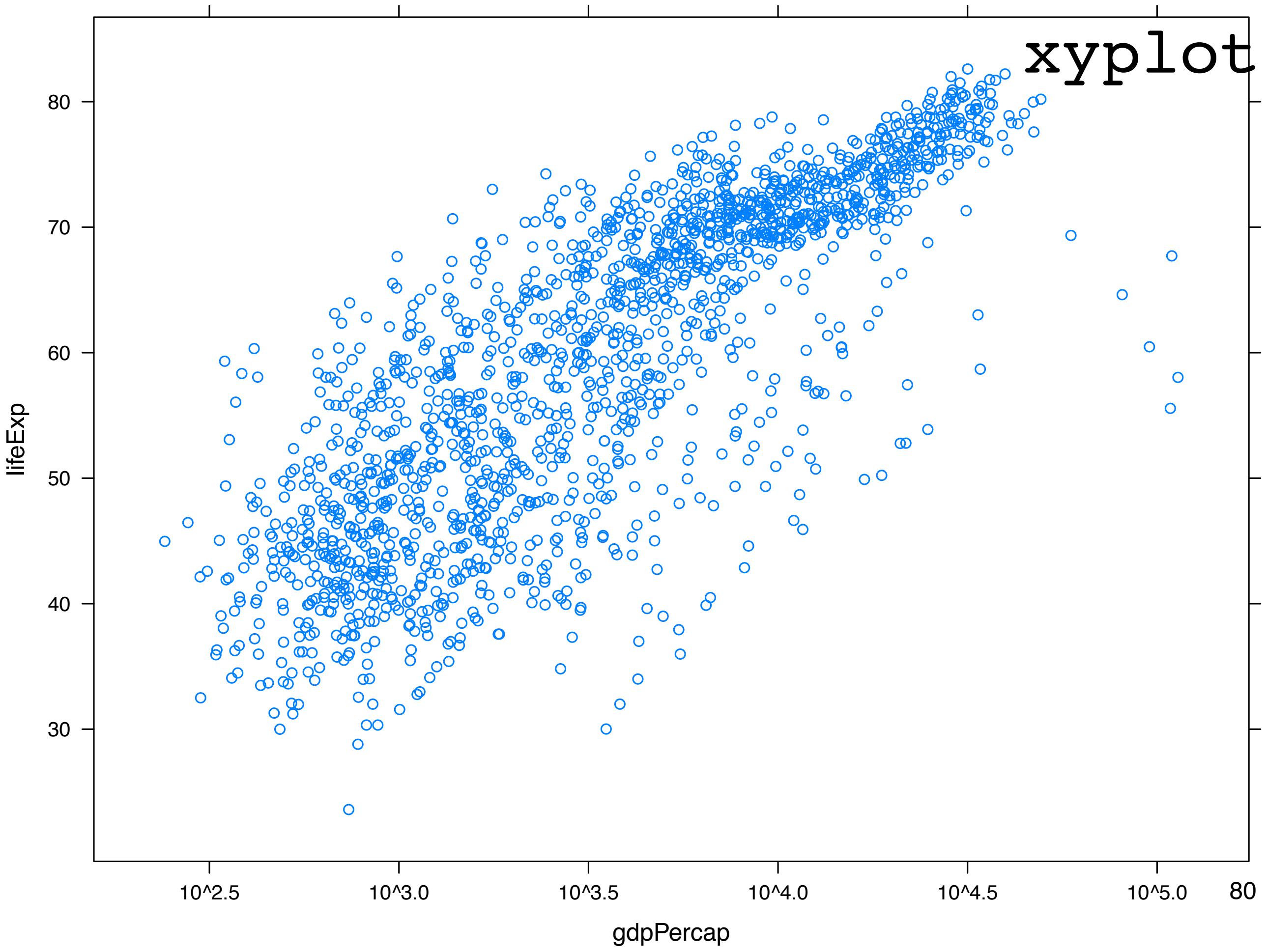
deprecated

From Ch. 5 in Sarkar (2008).

```
, group = ....,
```

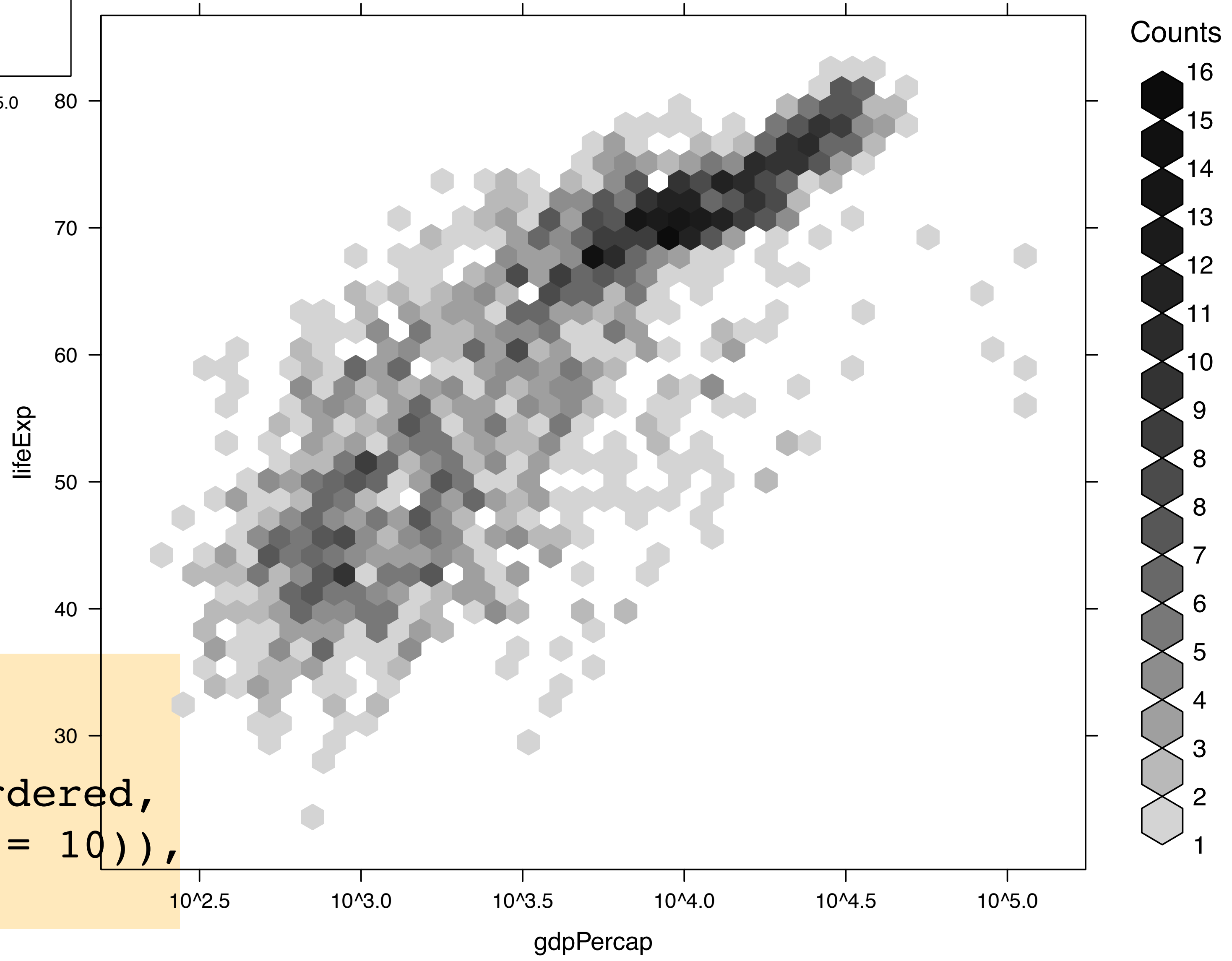
```
, auto.key = TRUE,
xyplot(y ~ x | z, ...)
```

combatting overplotting when you are blessed with lots of data



```
xyplot(lifeExp ~ gdpPercap, gDatOrdered,
         scales = list(x = list(log = 10)))
```

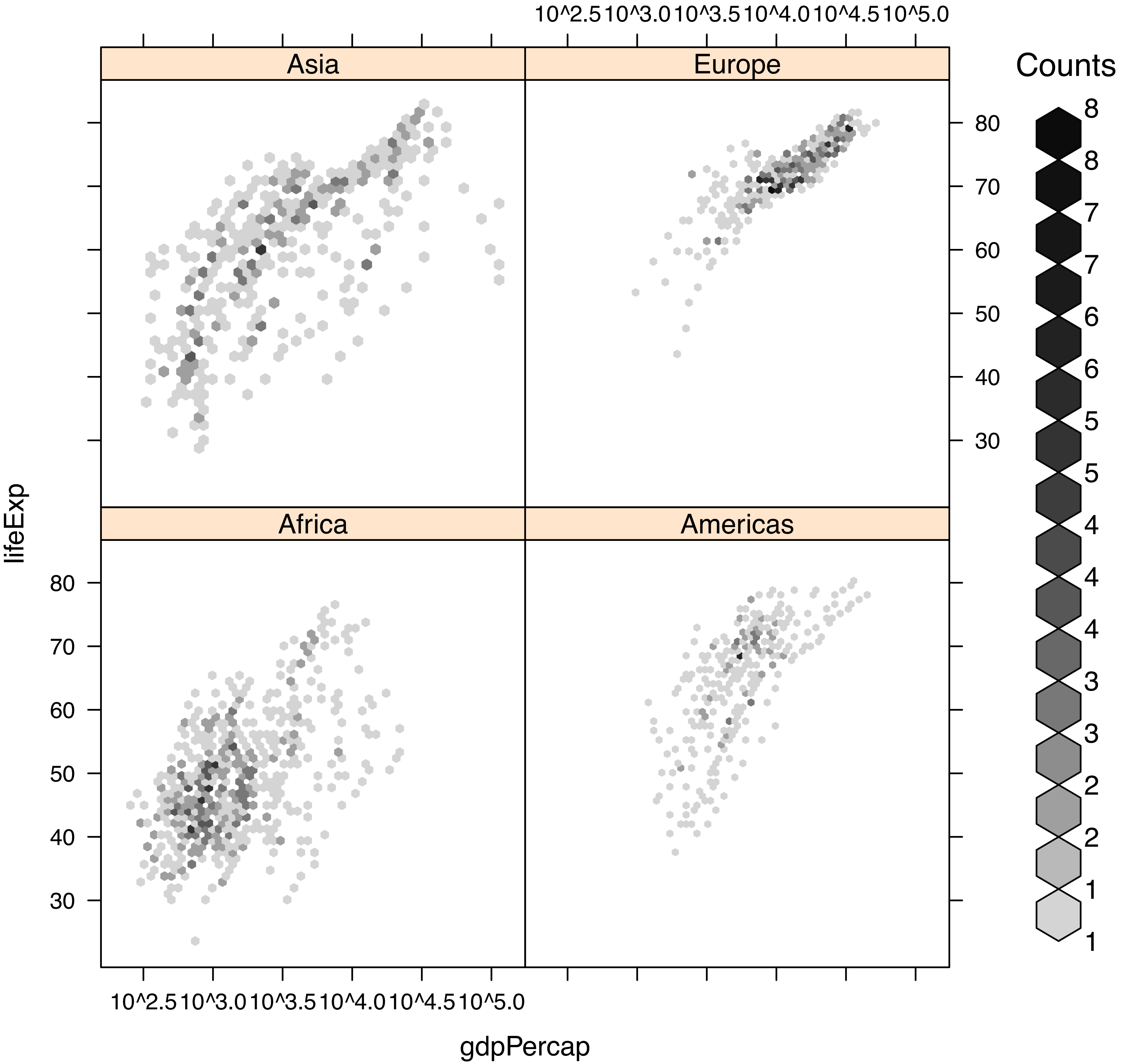
For high volume scatterplots, consider hexagonal binning.



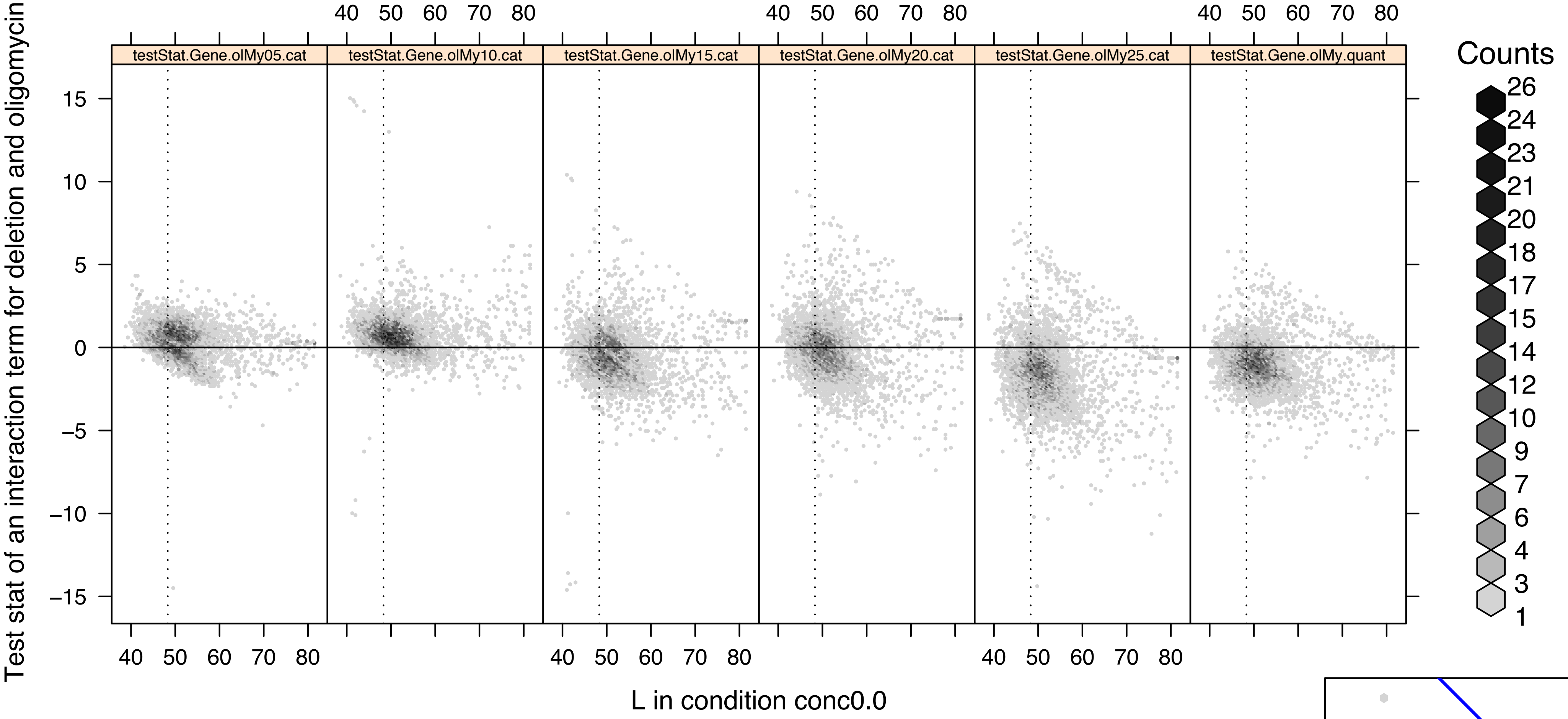
```
library(hexbin)

hexbinplot(lifeExp ~ gdpPercap, gDatOrdered,
           scales = list(x = list(log = 10)),
           xbins = 40)
```

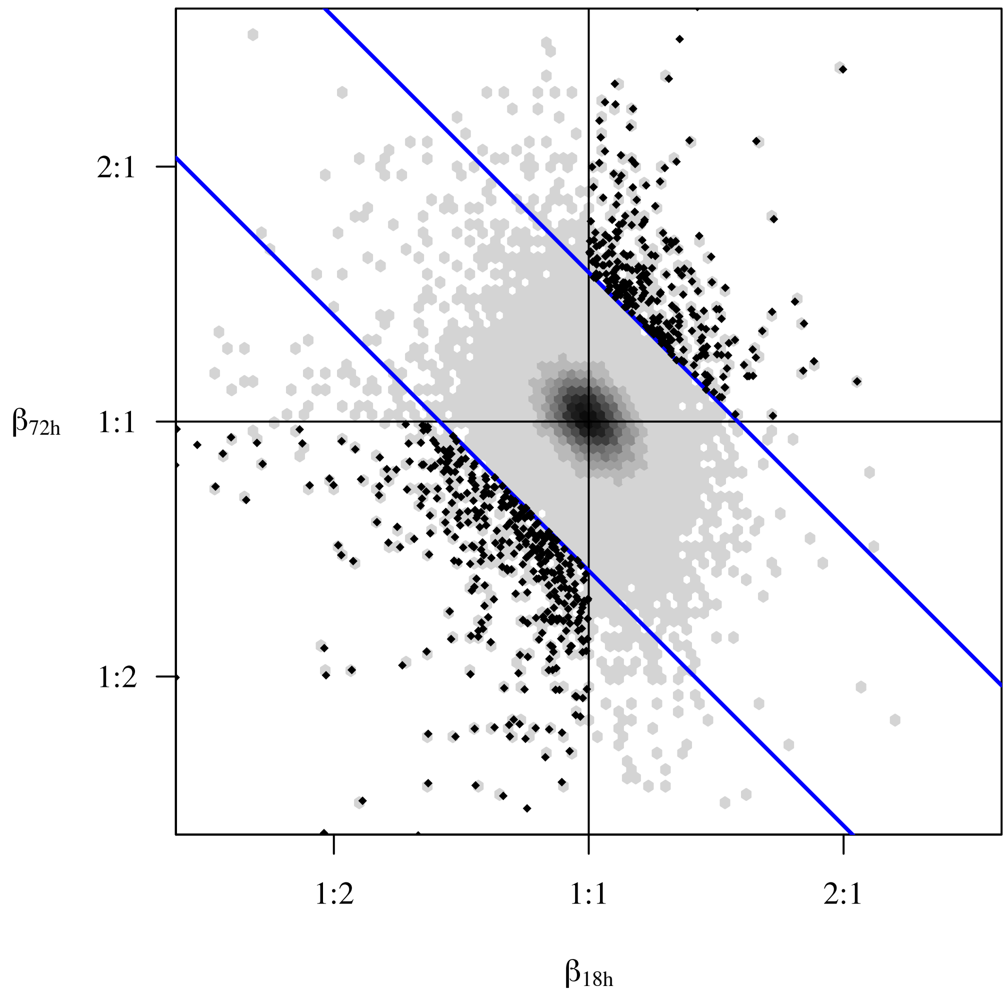
“Hexagonal binning”
Superior to scatterplot, when number of points creates overplotting problem.

Idea: tile the plane with hexagons, shade according to relative frequency of points.



```
hexbinplot(lifeExp ~ gdpPercap | continent, gDatOrdered,
           subset = continent != "Oceania",
           scales = list(x = list(log = 10)),
           xbins = 40)
```


testStat.Gene.olMy05.cat
testStat.Gene.olMy10.cat
testStat.Gene.olMy15.cat
testStat.Gene.olMy20.cat
testStat.Gene.olMy25.cat
testStat.Gene.olMy.quant
Test stat of an interaction term for deletion and oligomycin
15
10
5
0
−5
−10
−15
40 50 60 70 80
L in condition conc0.0
Counts
26
24
23
21
20
18
17
15
14
12
10
9
7
6
4
3
1


2:1
1:1
1:2
β_{72h}
1:2
1:1
2:1
β_{18h}

Relevant R functions:

hexbinplot() and hexplom() from hexbin package

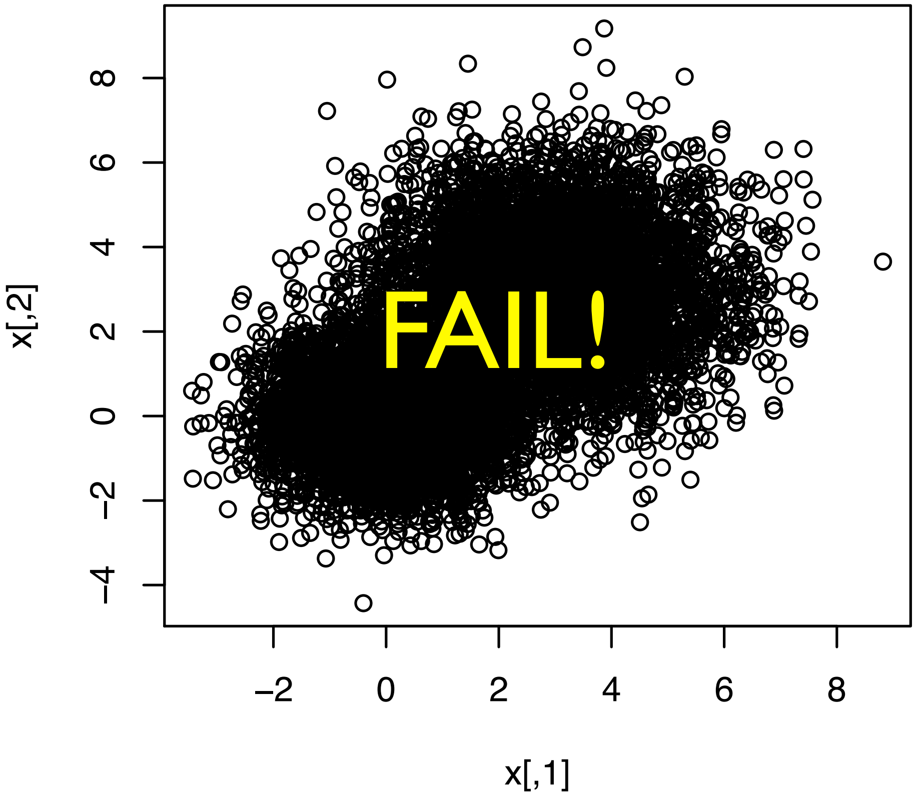
smoothScatter() -- also works well with pairs()

try running this:

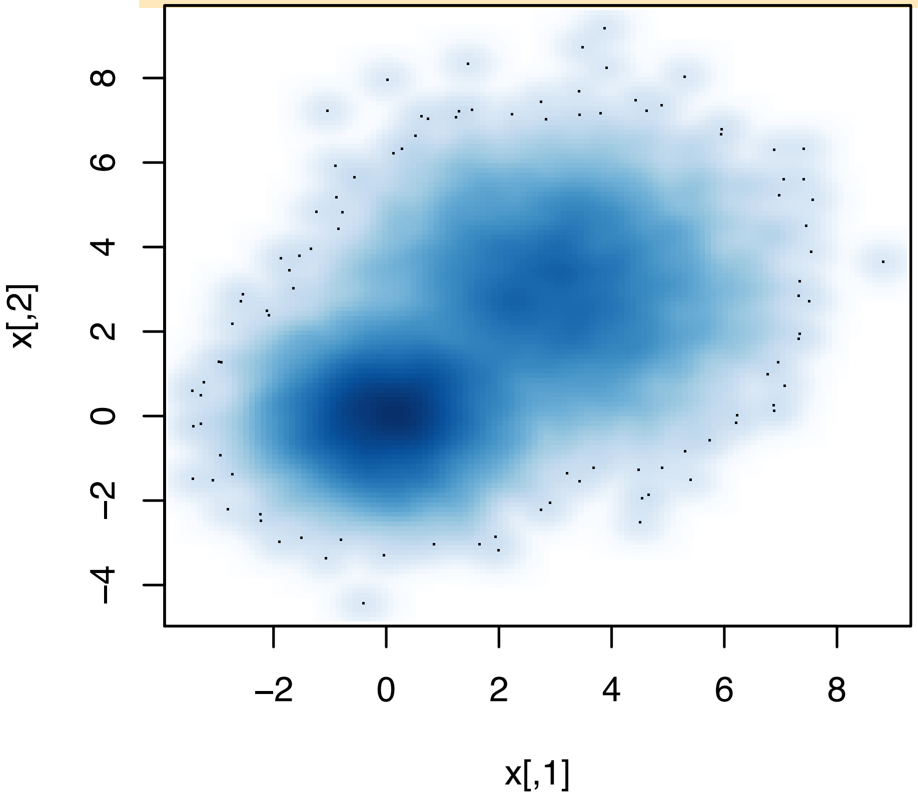
```
example(smoothScatter)
```

```
library(hexbin)
example(hexbin)
```

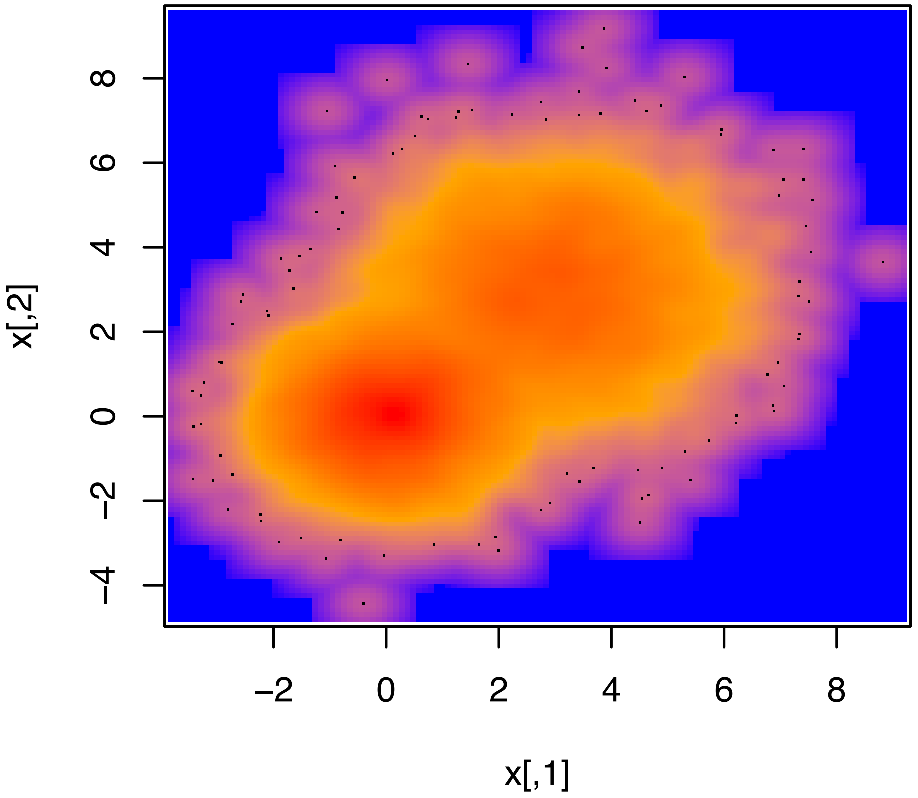
```
plot(x)
```



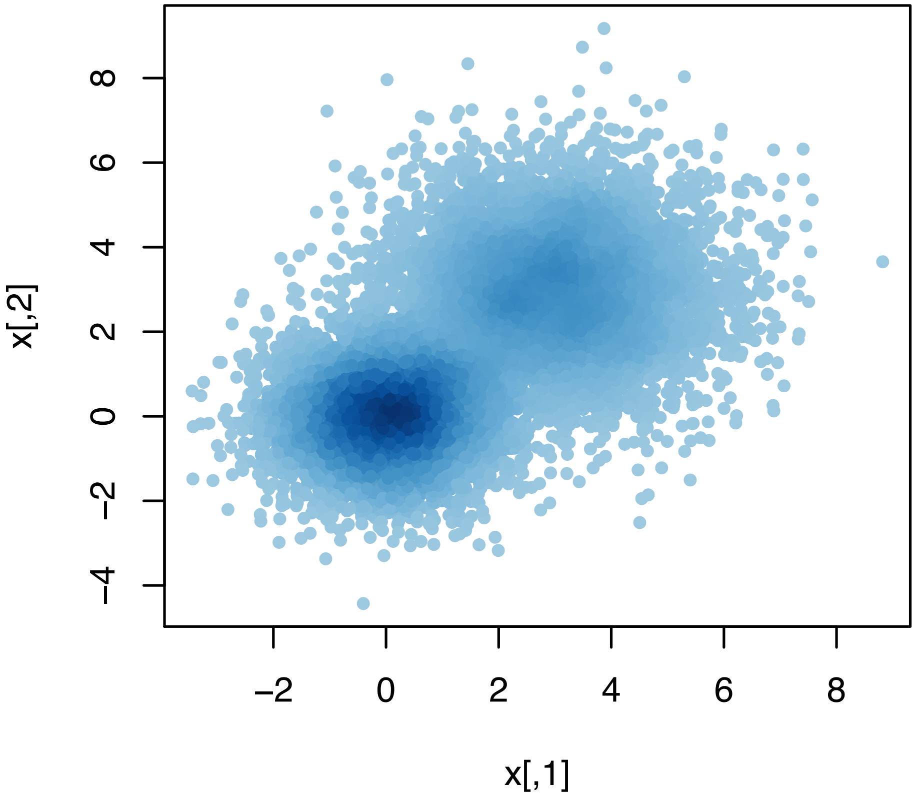
```
smoothScatter(x)
```

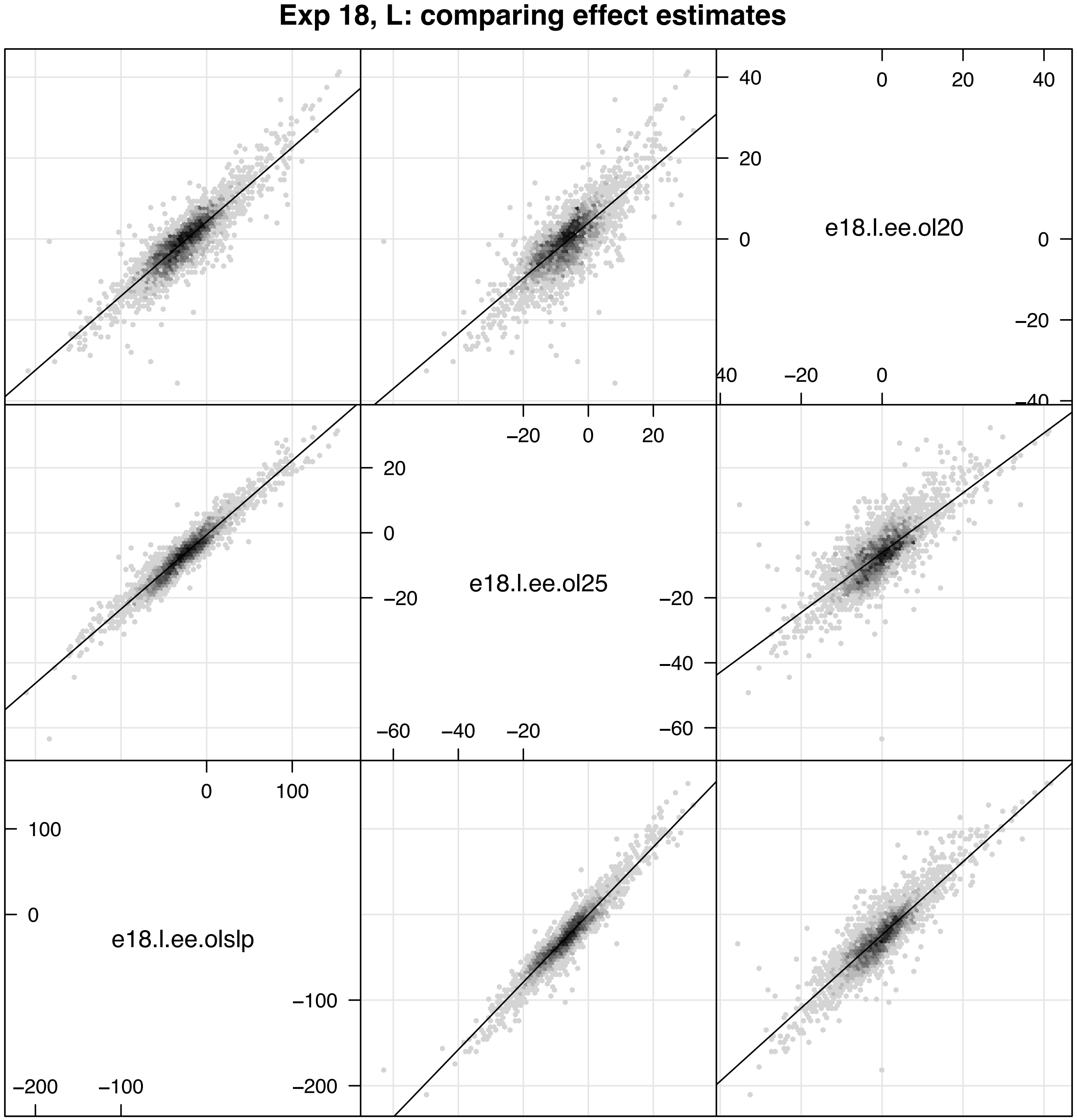


```
Lab.palette <- colorRampPalette(c("blue", "orange", "red"), space = "Lab")
smoothScatter(x, colramp = Lab.palette)
```



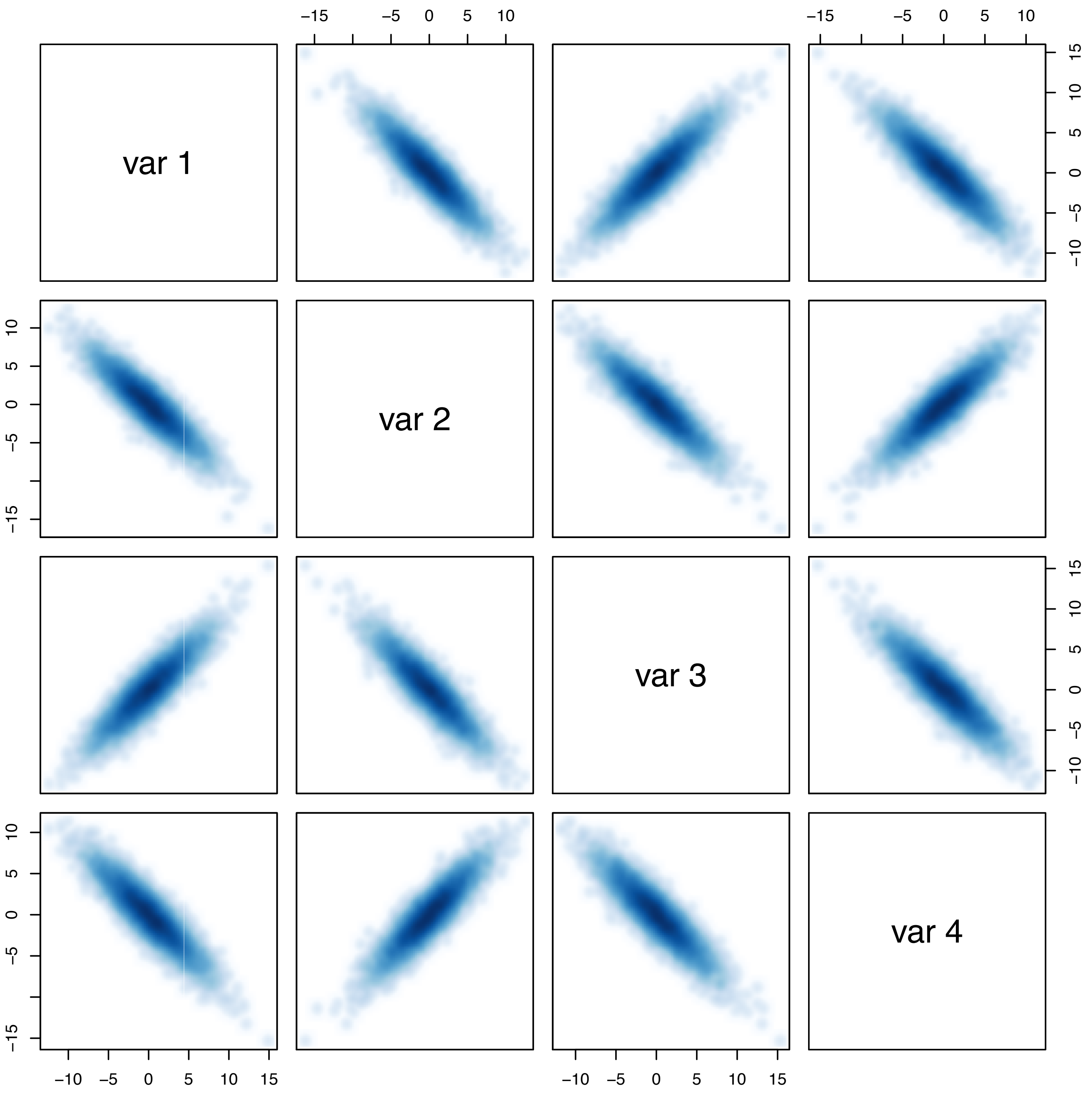
```
plot(x, col = densCols(x), pch=20)
```





Hexagonal binning, again, but in context of a scatterplot matrix. Read about pairs(), splom() and hexplom().

```
pairs(y, panel = function(...) smoothScatter(..., nrpoints=0, add=TRUE))
```



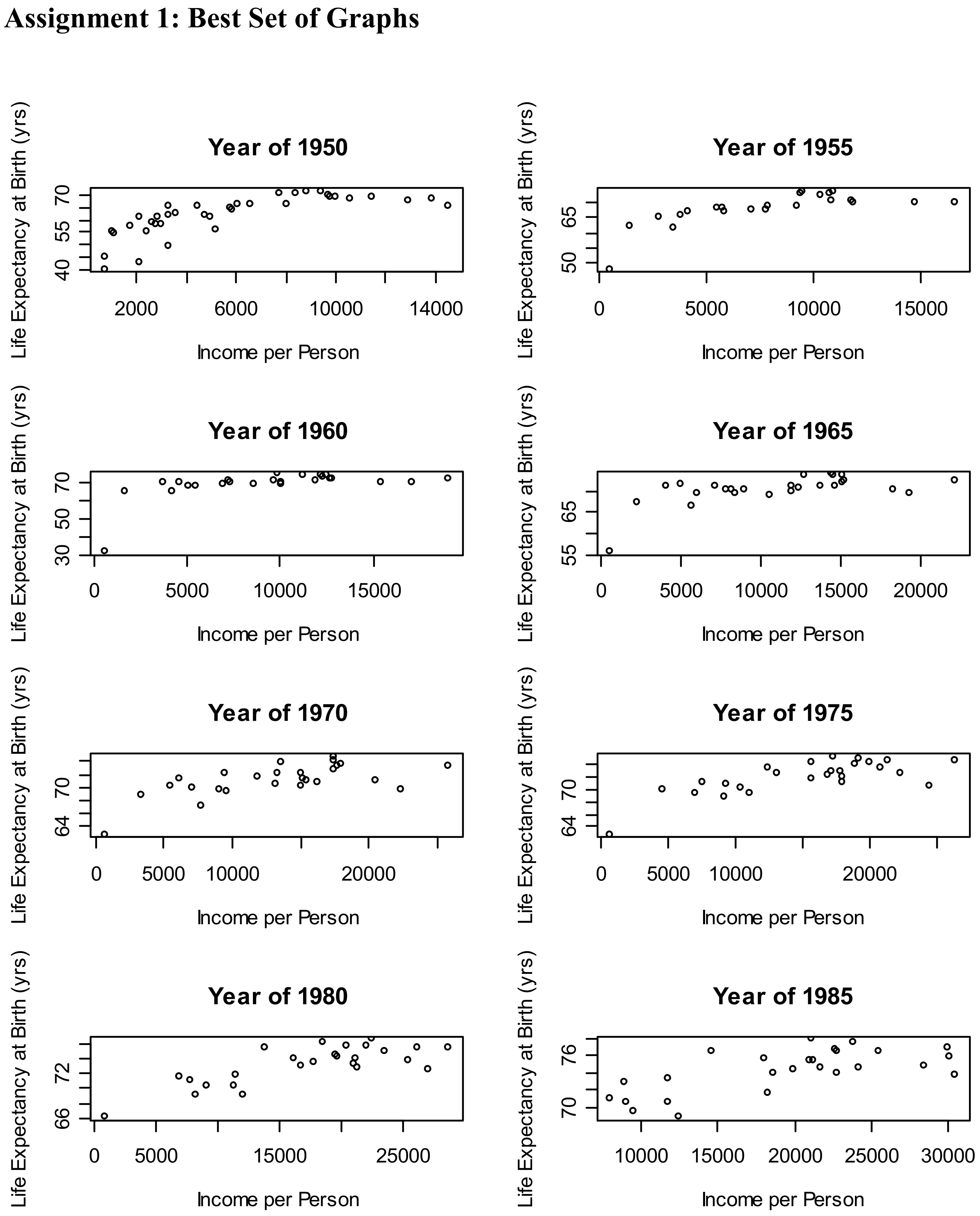
three different ways of making graphics in R these days

* Base or traditional R graphics
* `lattice` add-on package
* `ggplot2` add-on package

The CRAN Task View: Graphic Displays & Dynamic Graphics & Graphic Devices & Visualization provides a survey of the whole R graphics landscape.

base graphics <<< lattice
base graphics <<< ggplot2

more on this topic can be found here



Assignment 1: Best Set of Graphs

An excellent demonstration of how traditional graphics fails at such tasks.

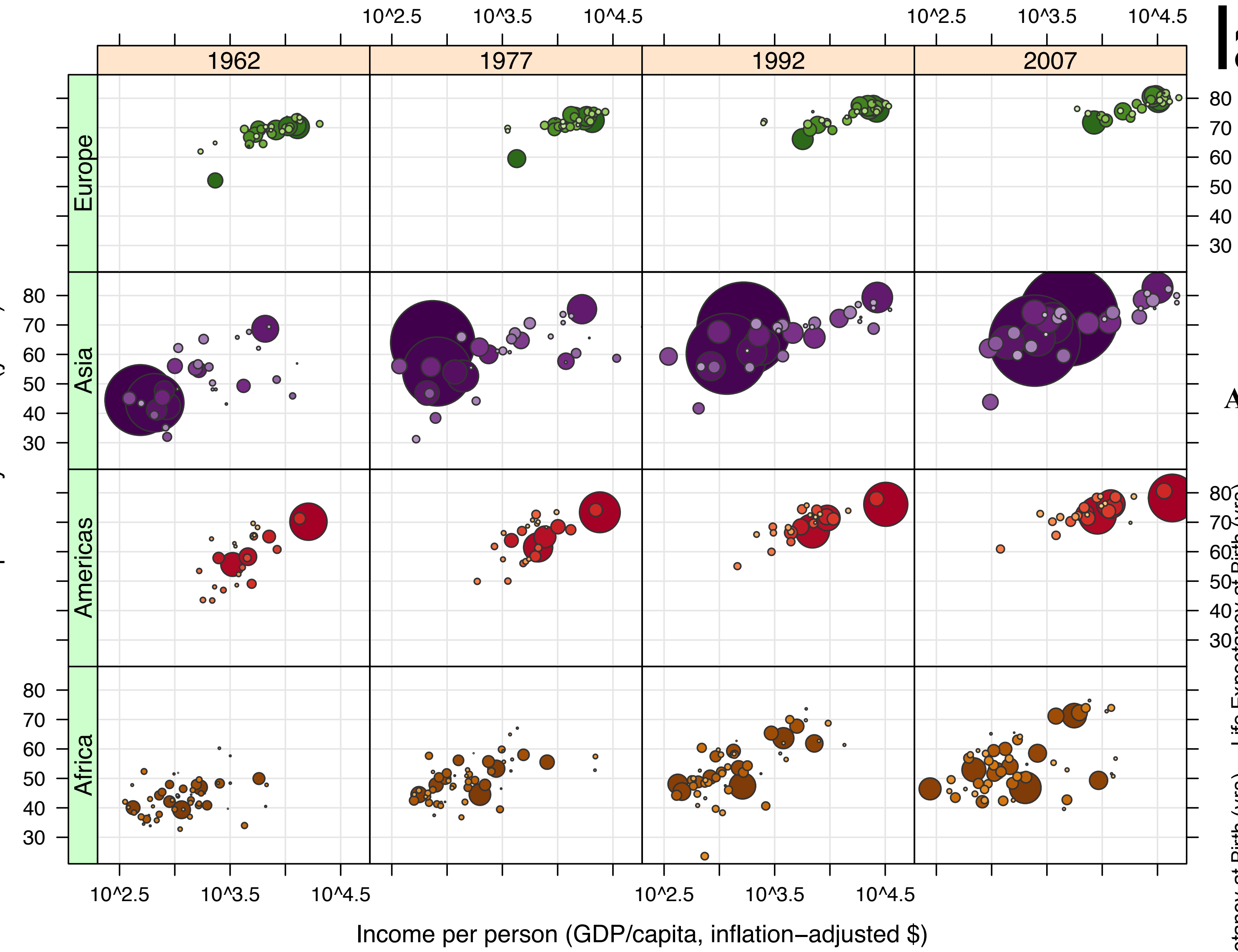
hello whitespace!

non-common axes

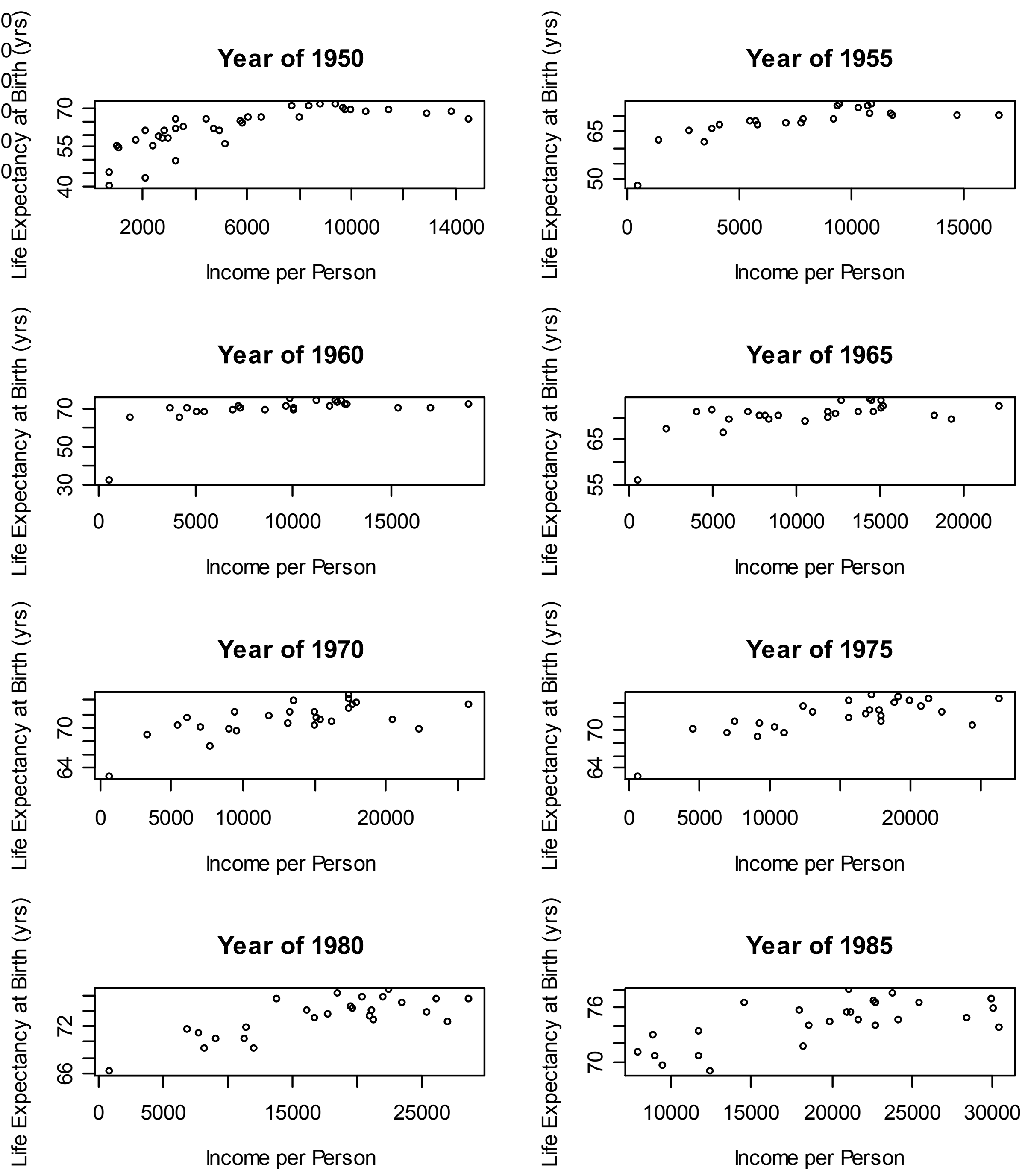
repeated elements, e.g. axis labels

misallocation of space, too much spent on annotation, not enough on *data*

lattice

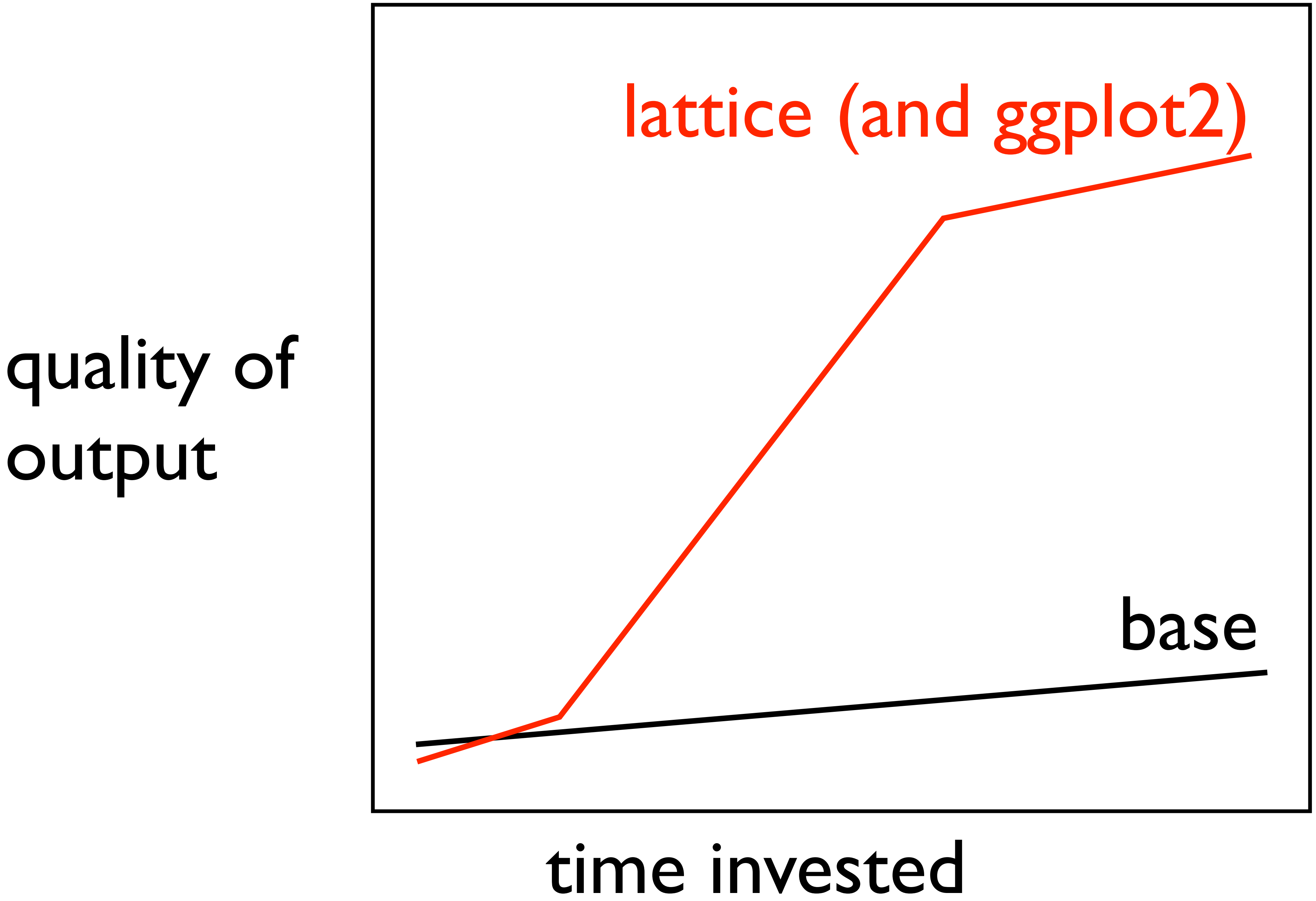


Assignment 1: Best Set of Graphs



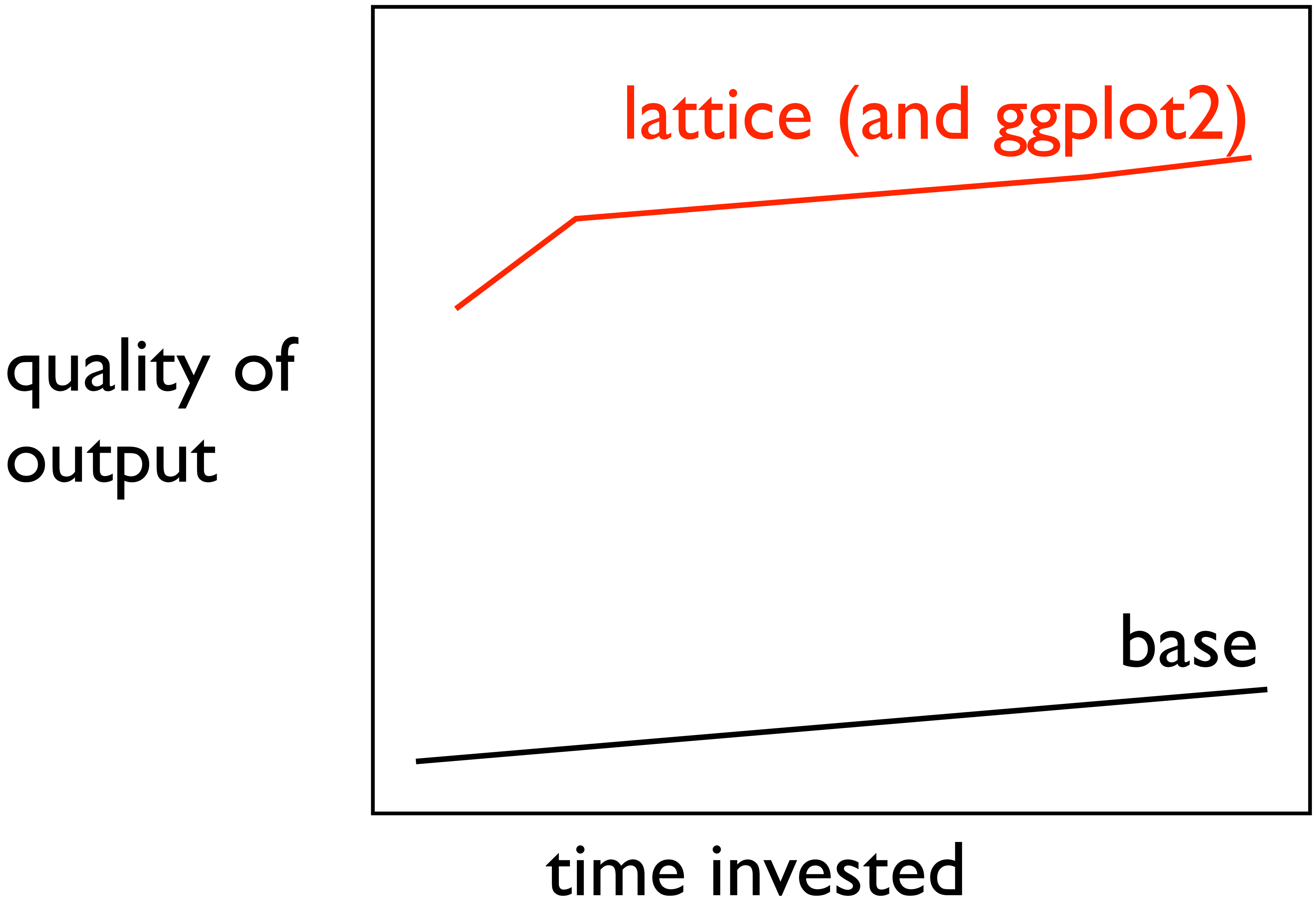
base

week one ....

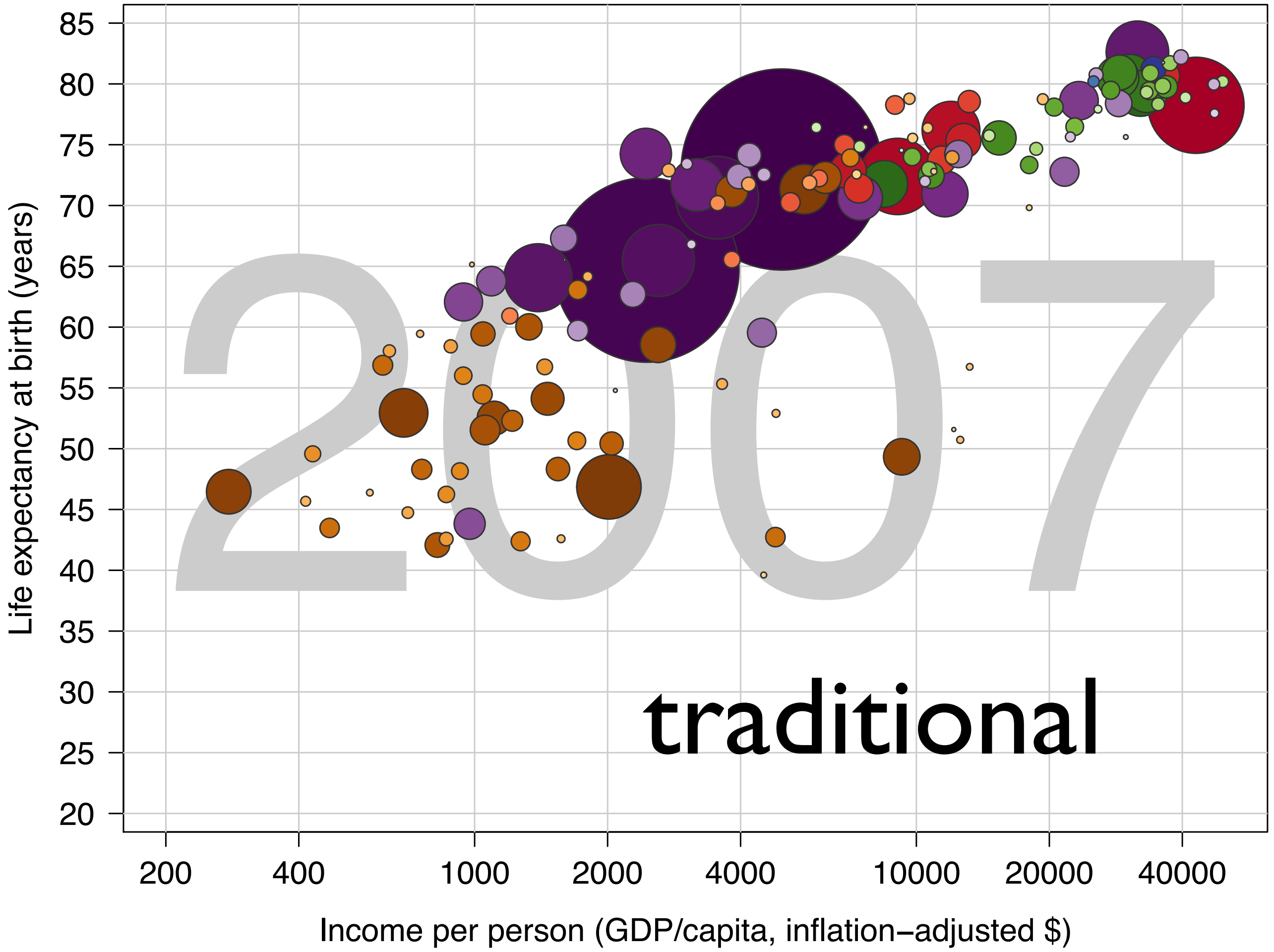


* figure is totally fabricated but, I claim, still true

week two and thereafter



* figure is totally fabricated but, I claim, still true



traditional

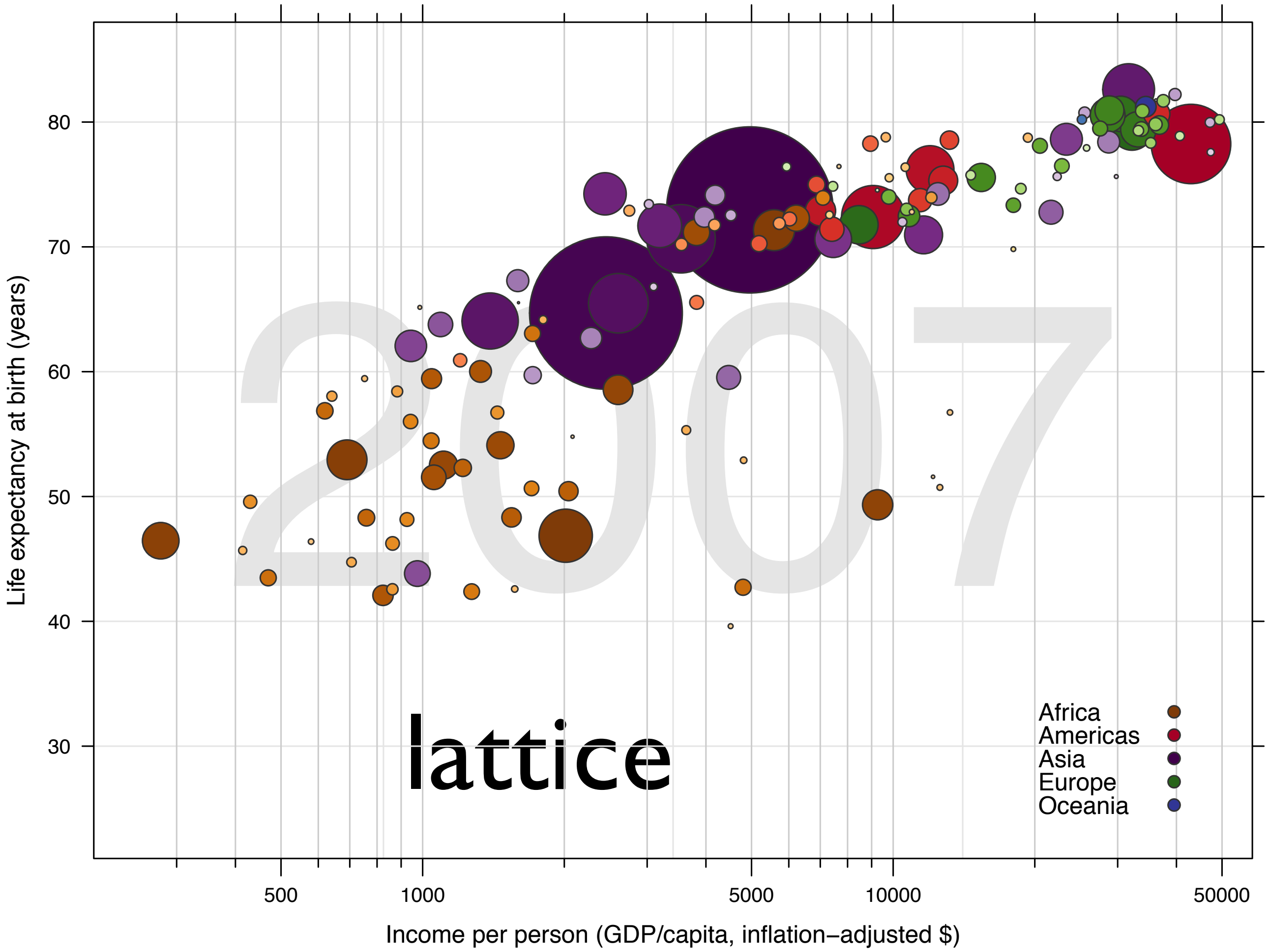
you should be making TONS of figures

about 1% of them are very fancy and highly specific

these will take lots of time period.

you need to focus on the other 99%

get better results with far less time by using lattice or ggplot2



lattice

high level lattice function
return objects of class “trellis”

Get more technical info on lattice here:
http://www.stat.ubc.ca/~jenny/STAT545A/block10_latticeNittyGritty.html

you can make an assignment to store the return value from, e.g. xyplot()

in that case, the trellis `print()` method does not get called and no figure appears on your screen

call print(myFig) explicitly to see it

you will need to this, e.g., if making a figure inside a function, inside a loop, etc.

plot() is almost equivalent to print()

storage + explicit print() call allows you to access optional arguments

main application is putting 2 or more lattice plots on one “page”

there is also an update() method which I rarely use reshaping

it’s really important

I assume you’re beginning to understand that from personal experience

tall or long data generally best for plotting and modelling

short or wide data often best for generating human-facing tables in lattice, some reshaping -- specifically from wide to tall -- can be avoided via the “extended formula interface”

high level lattice functions
panel functions

Function	**Default Display**
`histogram()`	Histogram
`densityplot()`	Kernel Density Plot
`qqmath()`	Theoretical Quantile Plot
`qq()`	Two-sample Quantile Plot
`stripplot()`	Stripchart (Comparative 1-D Scatter Plots)
`bwplot()`	Comparative Box-and-Whisker Plots
`dotplot()`	Cleveland Dot Plot
`barchart()`	Bar Plot
`xyplot()`	Scatter Plot
`splom()`	Scatter-Plot Matrix
`contourplot()`	Contour Plot of Surfaces
`levelplot()`	False Color Level Plot of Surfaces
`wireframe()`	Three-dimensional Perspective Plot of Surfaces
`cloud()`	Three-dimensional Scatter Plot
`parallel()`	Parallel Coordinates Plot

Table 1.1. High-level functions in the lattice package and their default displays.

how to customize a lattice figure:

- use those arguments! READ THE HELP FILE. I know they're long and boring . Sorry.
  - tip: read help on the default panel function (I'll elaborate)

- change graphical parameters, such as the vector of colors used by a grouping factor (I'll elaborate)

- customize the panel function

Borrowing heavily from Sarkar’s talk “Lattice Tricks for the Power UseR” from UseR! 2007 conference ......

Things to know about panel functions:

Panel functions are functions (!)

They are responsible for graphical content inside panels

They get executed once for every panel

Every high level function has a default panel function

e.g., xyplot() has default panel function panel.xyplot()

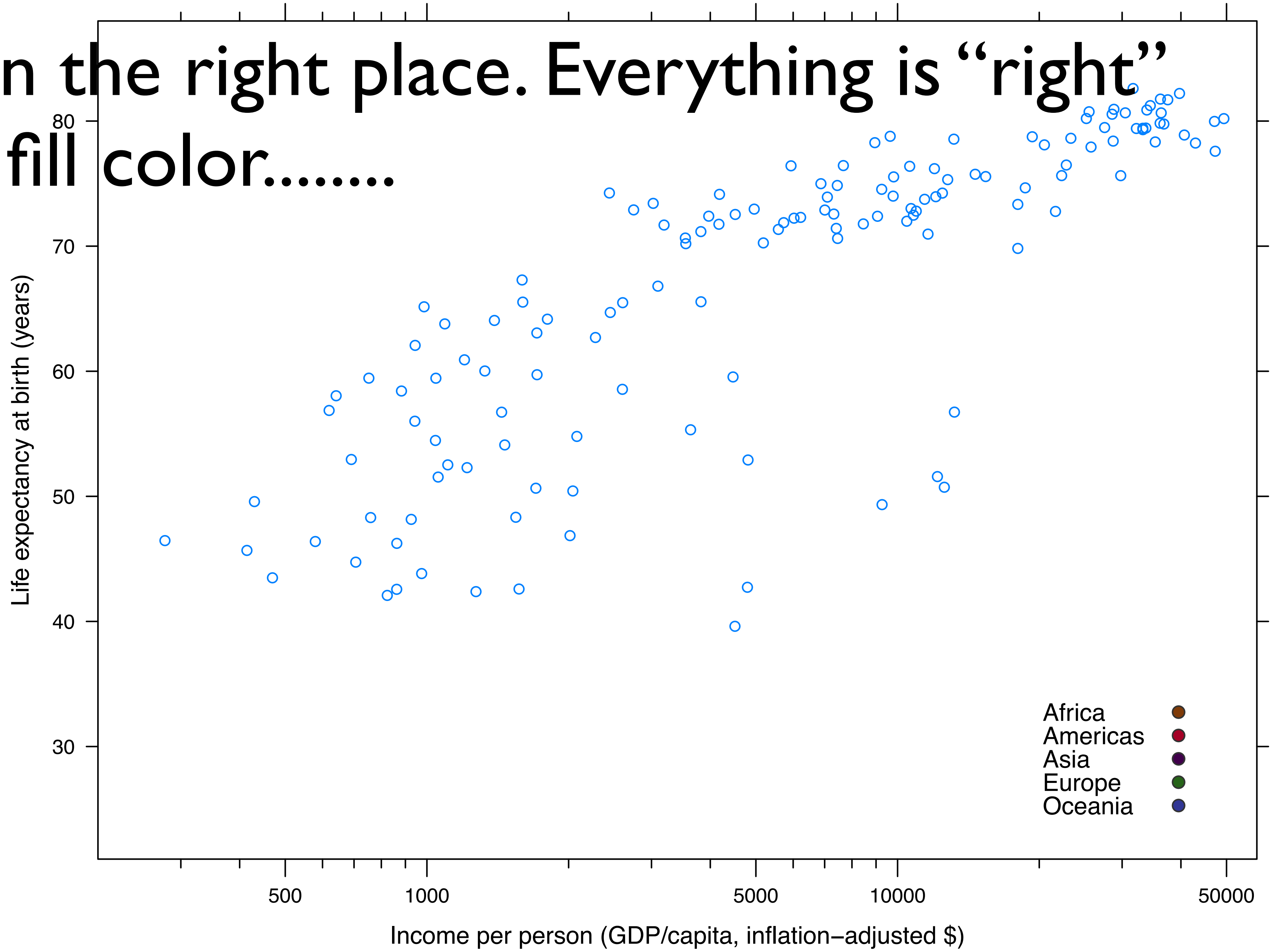
Workflow for rewriting (e.g. extending) a panel function:

Start with a graphing command (e.g. a call to xyplot) that “works”, i.e. it has nothing wrong with it -- it’s just missing some cool features.

Rewrite the panel function but without adding any functionality!

Gradually add the elements you need.

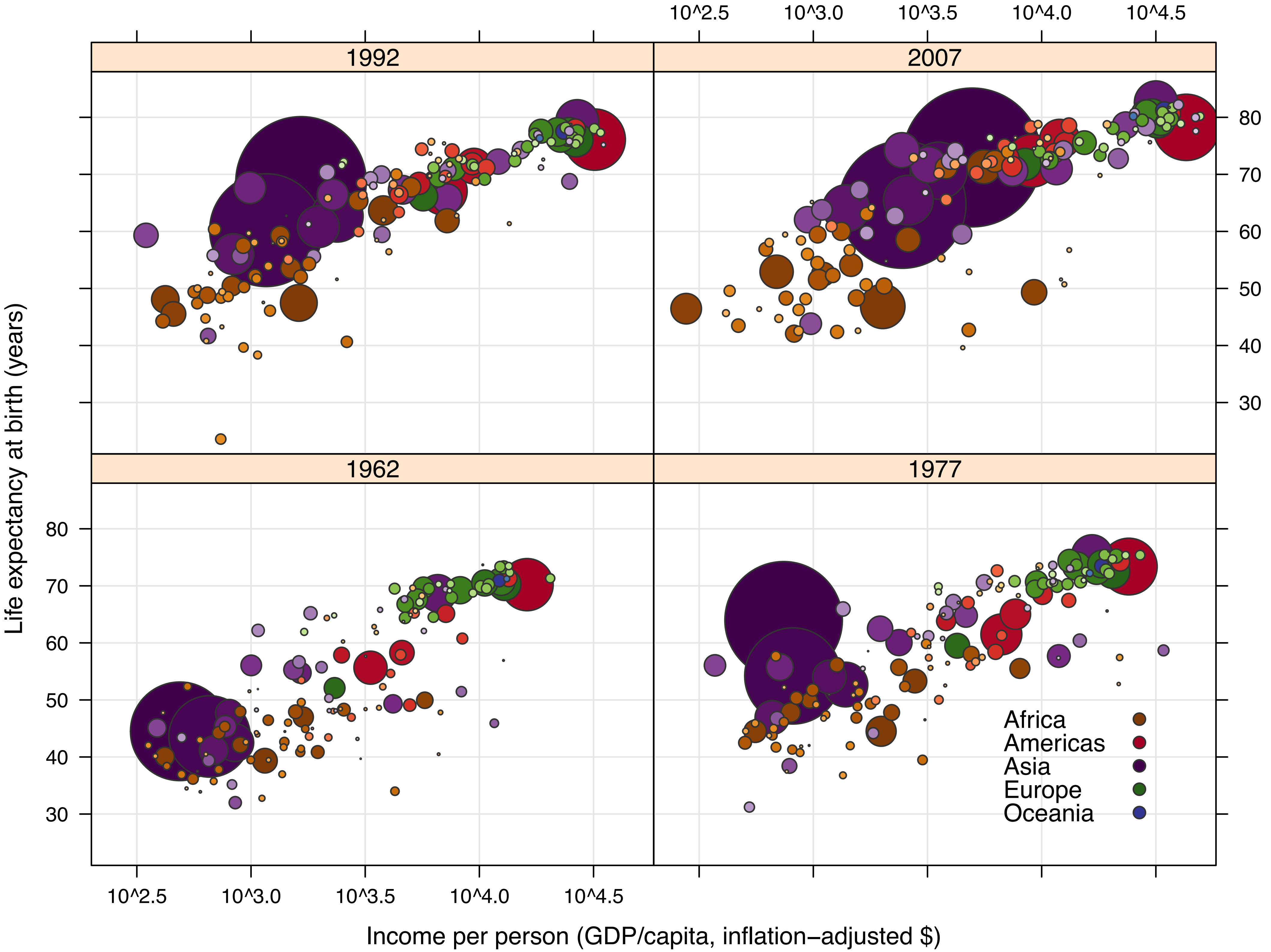
The points are in the right place. Everything is “right” except size and fill color........



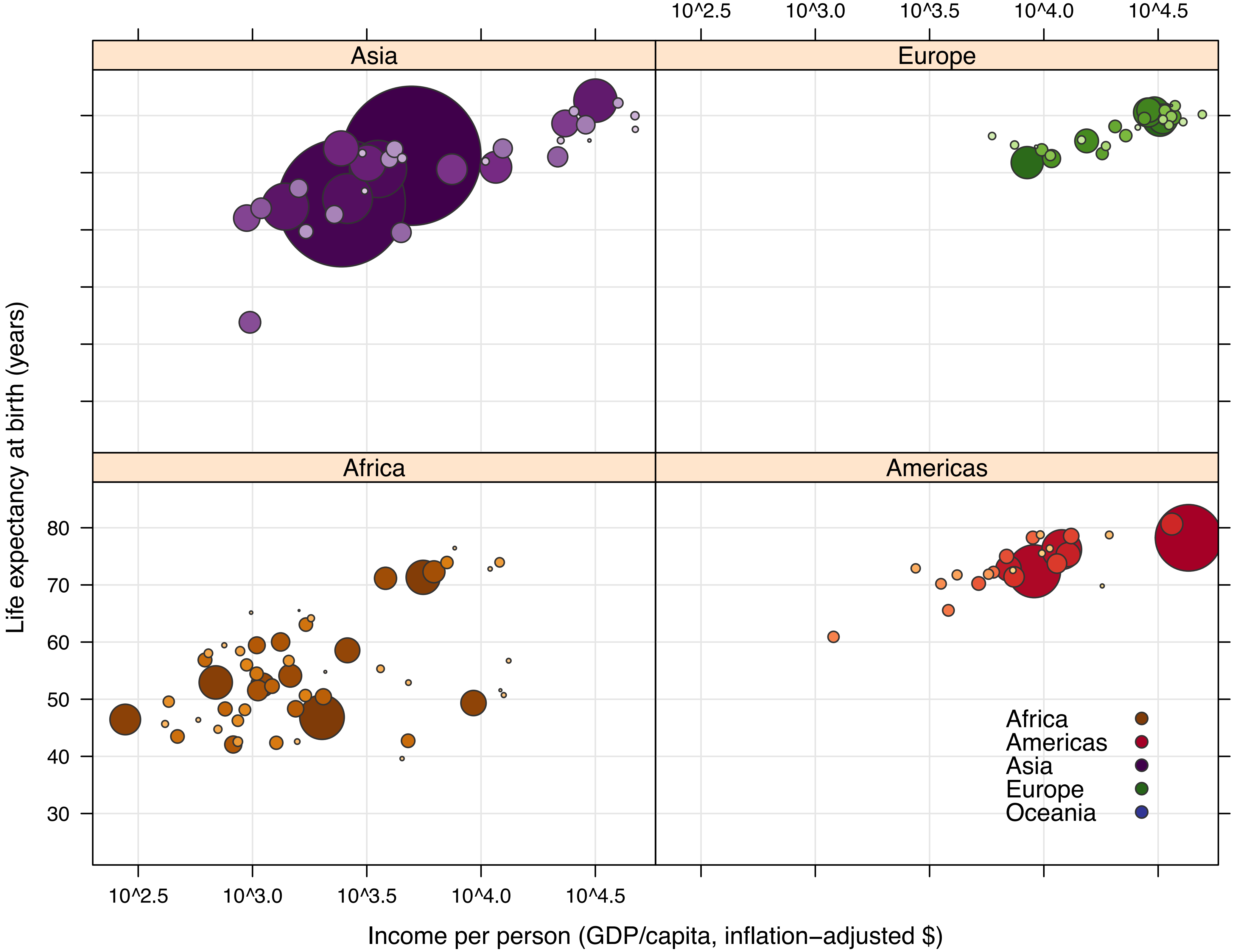
```
yDat <- subset(gDatOrdered, year == jYear)
xyplot(lifeExp ~ gdpPercap, yDat,
       xlab = jXlab, ylab = jYlab,
       scales = list(x = list(log = 10)),
       xlim = jXlim, ylim = jYlim,
       xscale.components = xscale.components.log10,
       key = continentKey,
       panel = panel.xyplot)
```

see step-by-step development towards a non-default panel function in the tutorial:

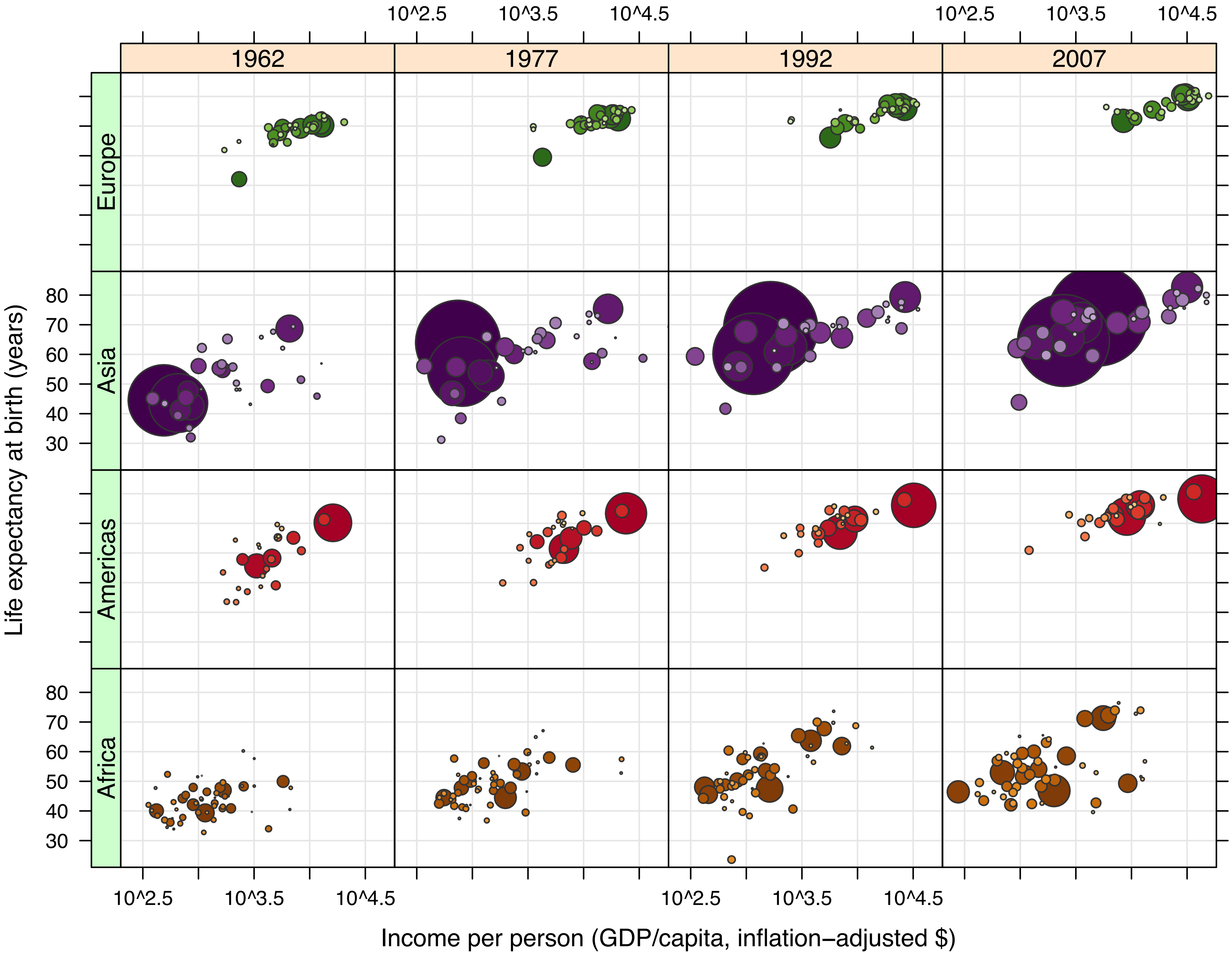
http://www.stat.ubc.ca/~jenny/STAT545A/block10_latticeNittyGritty.html



```
xyplot(lifeExp ~ gdpPercap | factor(year), ...)
```



```
xyplot(lifeExp ~ gdpPercap | continent, ...)
```



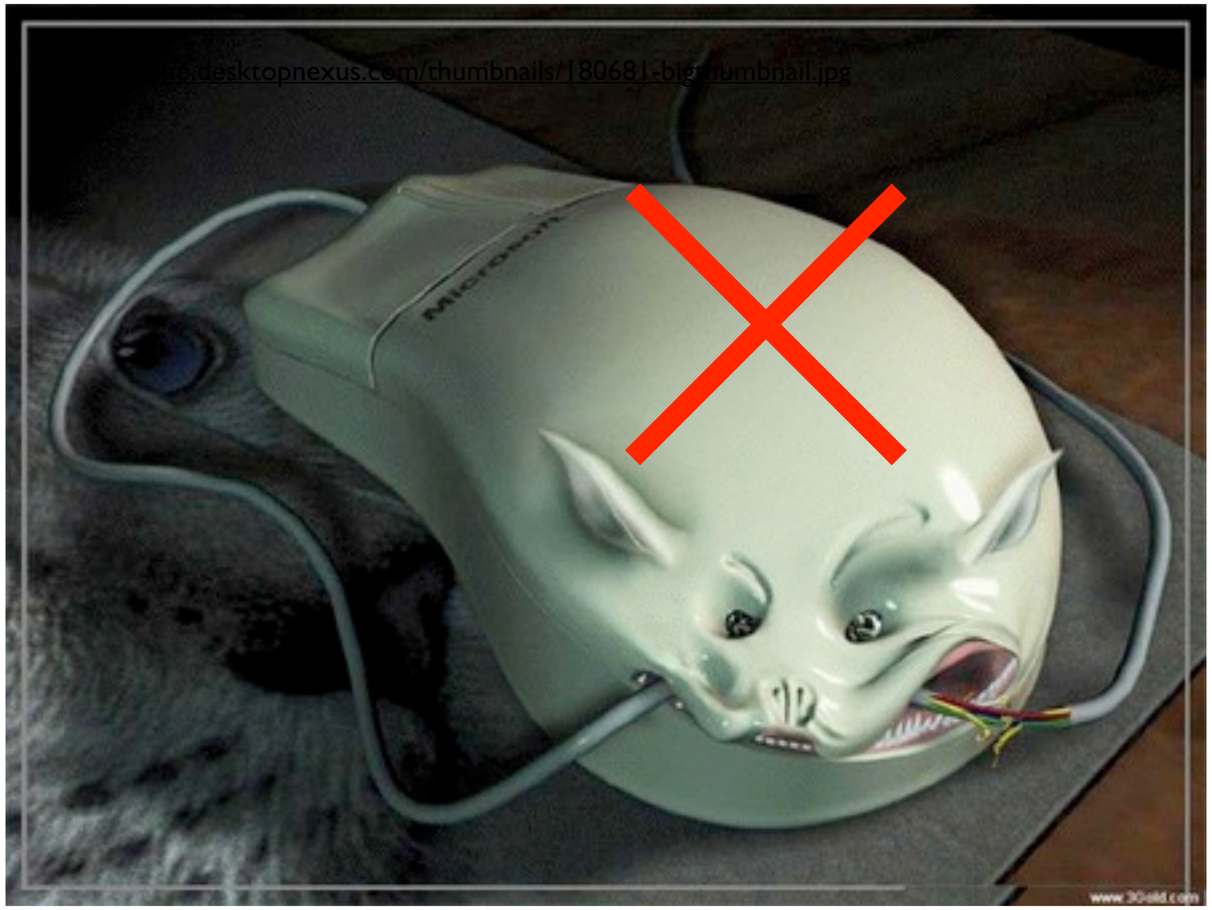
```
library(latticeExtra)                    # useOuterStrips()
jPlot <-
  xyplot(lifeExp ~ gdpPercap | factor(year) * continent, ...)
useOuterStrips(jPlot)
```

saving figures to file

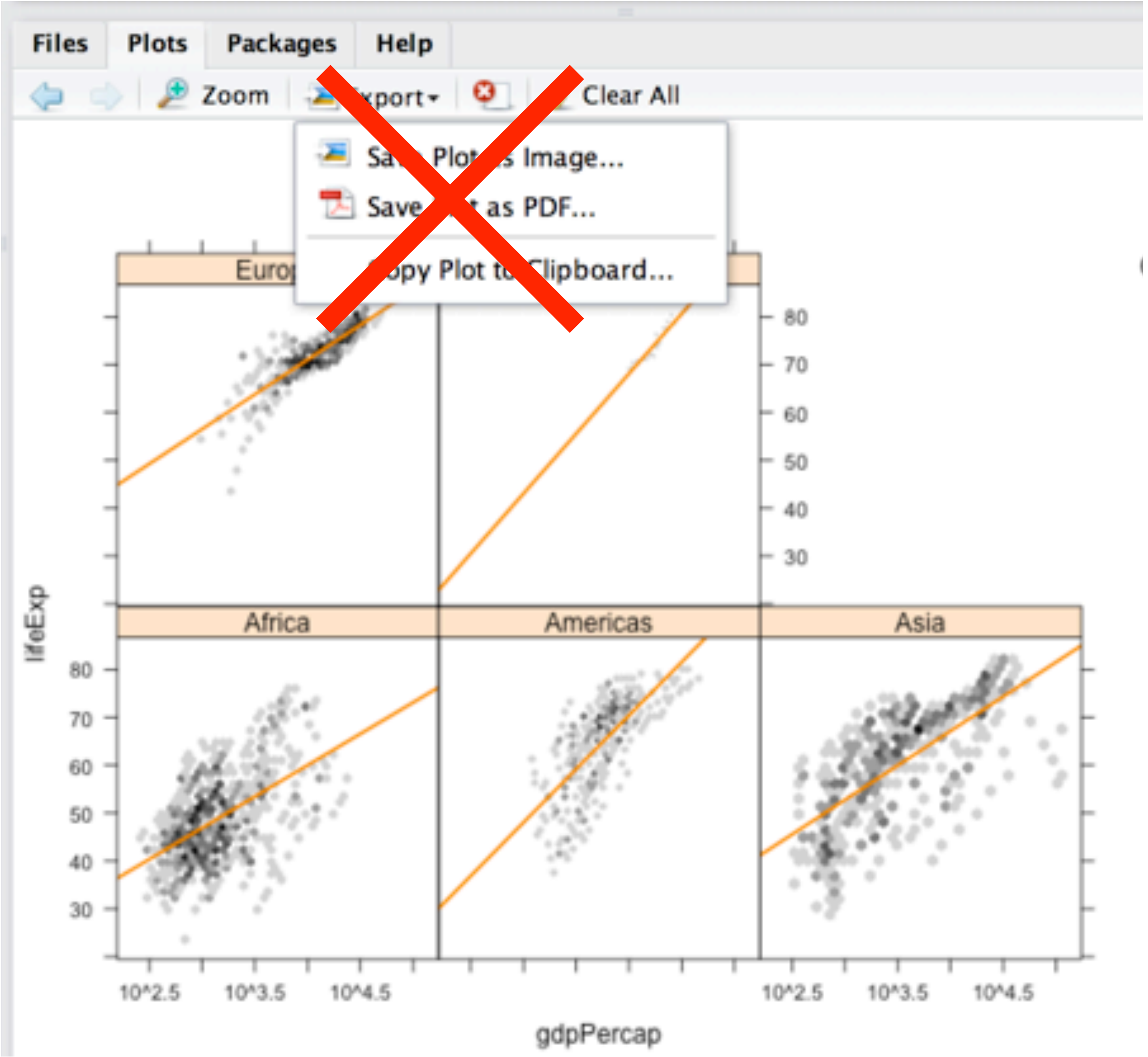
do not save figures mouse-y style

not self-documenting

not reproducible



he.desktopnexus.com/thumbnails/180681-bigthumbnail.jpg



(Getting ready to) make a figure

- First draft (and the next ~100 attempts!) should be displayed in a window; for example, the Rstudio Plots pane
- When ready to preserve for posterity, I advise you default to PDF. Most generally useful format to have around.
  - Educate yourself about vector versus raster graphics here (or Google it yourself). Short version: default to *vector*, which PDF is an example of.
- Save the figure w/ a line of R code in your script, not with the mouse.
- If preparing a figure for the web and/or a figure that presents a very large number of data points -- png() is a good option to consider. The web winds are blowing towards SVG these days.

(Getting ready to) make a figure

- If you rigorously produce your figures from code you save -- versus via typing at the command line or (shudder) with a mouse -- it will always be easy to remake a figure natively on a different device. This will come up!
- Additionally, it will be easy to find the R code that generated any figure by searching for the figure filename on your computer, filtering for *.R files if possible.
- The following methods for writing to PDF should work for other devices.

Two good blog posts on saving R graphics

10 tips for making your R graphics look their best
Revolutions blog post
http://blog.revolutionanalytics.com/2009/01/10-tips-for-making-your-r-graphics-look-their-best.html <-- highly recommended

High-quality R graphics on the Web with SVG
Revolutions blog post
http://blog.revolutionanalytics.com/2011/07/r-svg-graphics.html

Save a figure -- Method 1

- Open a PDF device by calling pdf(), execute all your graphics commands, close the PDF by calling dev.off()
  - Pros: Most ‘correct’ method. Possible to create multi-page files.
  - Cons: pdf() and dev.off() commands clutter up your R file and require repeated (de-)commenting out. It’s annoying to re-submit all those commands, when you’re staring at a beautiful figure on the screen.

```
pdf(paste0(whereAmI,"figs/rawPlotsByORF.pdf"),
    width = 9.5, height = 7)
for(i in seq(along = levels(kDat$ORF)))
   print(xyplot(pheno ~ tm | day, kDat,<blah, blah, ...>)
dev.off()
```

Save a figure -- Method 2

- Make the figure in a screen device, such as X11(). Call dev.print to copy that to a PDF file.
  - Pros: Immediate gratification. PDF-generating code is limited to one command, so easier to (de-)comment out.
  - Cons: Slightly 'incorrect'. Certain aspects of the figure depend on the graphics device (which would be X11, not PDF), therefore, in some cases, you can get different PDF files with Methods 1 and 2. It's a risk I'm often willing to take. Also, won't work for multi-page figures.

```
xyplot(lifeExp ~ gdpPercap | year, gDat)
dev.print(pdf,
          file = paste0(whereAmI,"figs/bryan-a01-latticeMinimal.pdf"),
          width = 9, height = 6)
```

BTW ggplot2 has some special ways to write figures to file