STAT 545A
Class meeting #8
Monday, October 1, 2012

Dr. Jennifer (Jenny) Bryan

Department of Statistics and Michael Smith Laboratories

Review of last class

Base R graphics: to get great results, you will have to micro-manage every last detail

par() is useful function to query, set, and simply learn about all sorts of graphical parameters

Typical workflow: "null" plot to set up co-ordinate system, then slowly add graphical elements

Important details: proper way to log an axis, keep axis labels in readable orientation

Review of last class

Use professionally designed color palettes 'as is', when possible (e.g. RColorBrewer, dichromat).

colorRamp() and colorRampPalette() are handy for interpolating colors, i.e. expanding a palette that is too small or preparing a discrete palette for use with continuous data

Store color scheme as a data.frame; one character variable for the colors plus one or more factors that mimic the factors in your main data.frame.

Use match() or merge() to map factor variables into colors.

The color scheme data.frame is handy for making legends and keeping colors consistent across a series of scripts/figures.

Code you see in this lecture can be found in these files:

bryan-a01-15-latticeStepByStep.R

bryan-a01-16-latticePlotGapminderOneYear.R

bryan-a01-17-latticeSoln.R

bryan-a01-18-latticeDemos.R

in this directory:
http://www.stat.ubc.ca/~jenny/notOcto/STAT545A/examples/gapminder/code/

Sources I reference heavily in this class

Paul Murrell's book "R Graphics". I own the 2005 edition but note there is a new August 2011 2nd edition, which is obviously more up-to-date and looks substantially expanded.
Home page for 1st ed and 2nd ed (gives R code for figures)
STATSnetBASE (read book online -- I can't 2nd ed there yet)
Google books search

Deepayan Sarkar's book "Lattice"
webpage w/ all code in book
SpringerLink (read book online)
Google books search
For today, focus especially on Ch 5 Scatter Plots and Extensions.

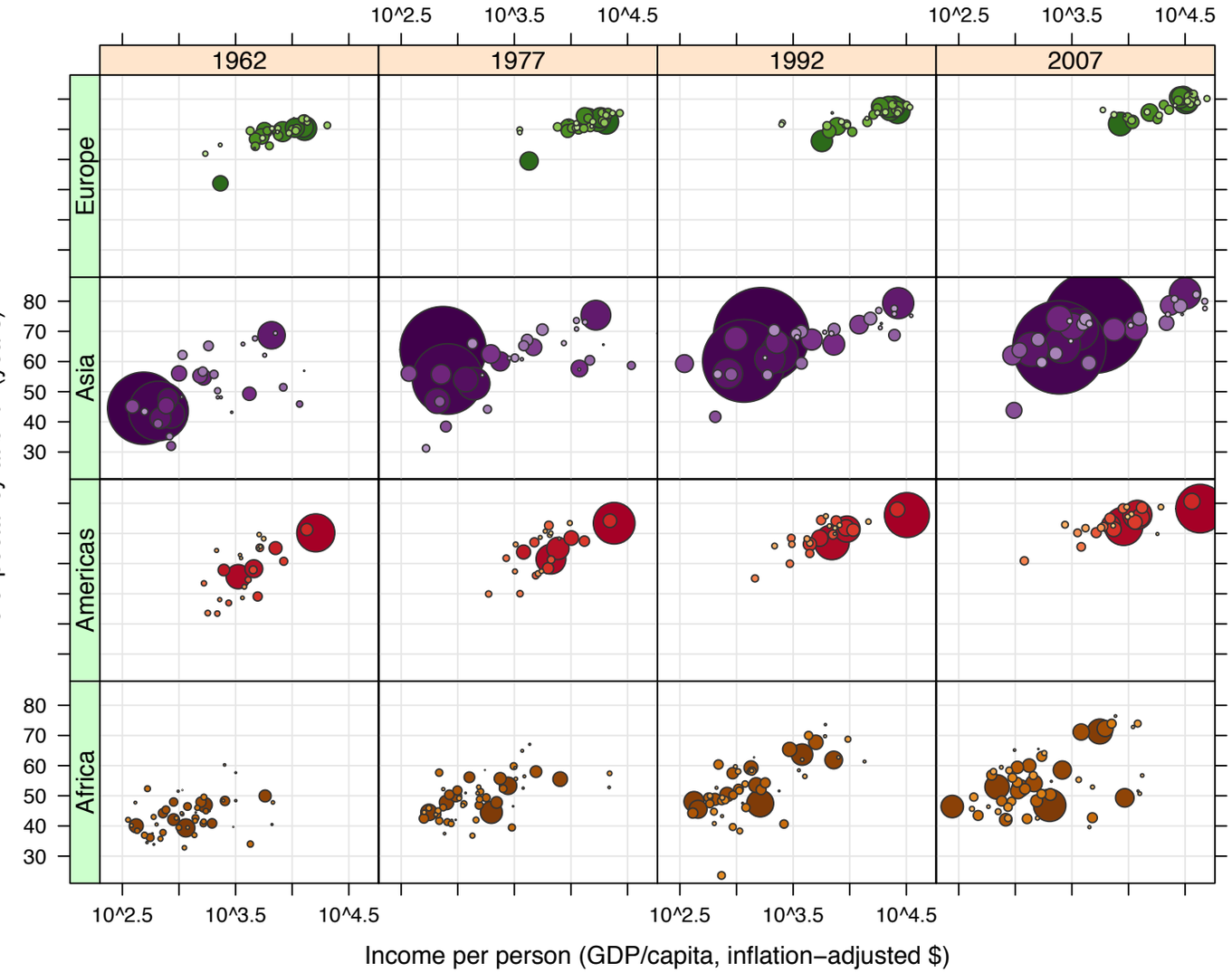
Please recall .....

Two main goals for statistical graphics

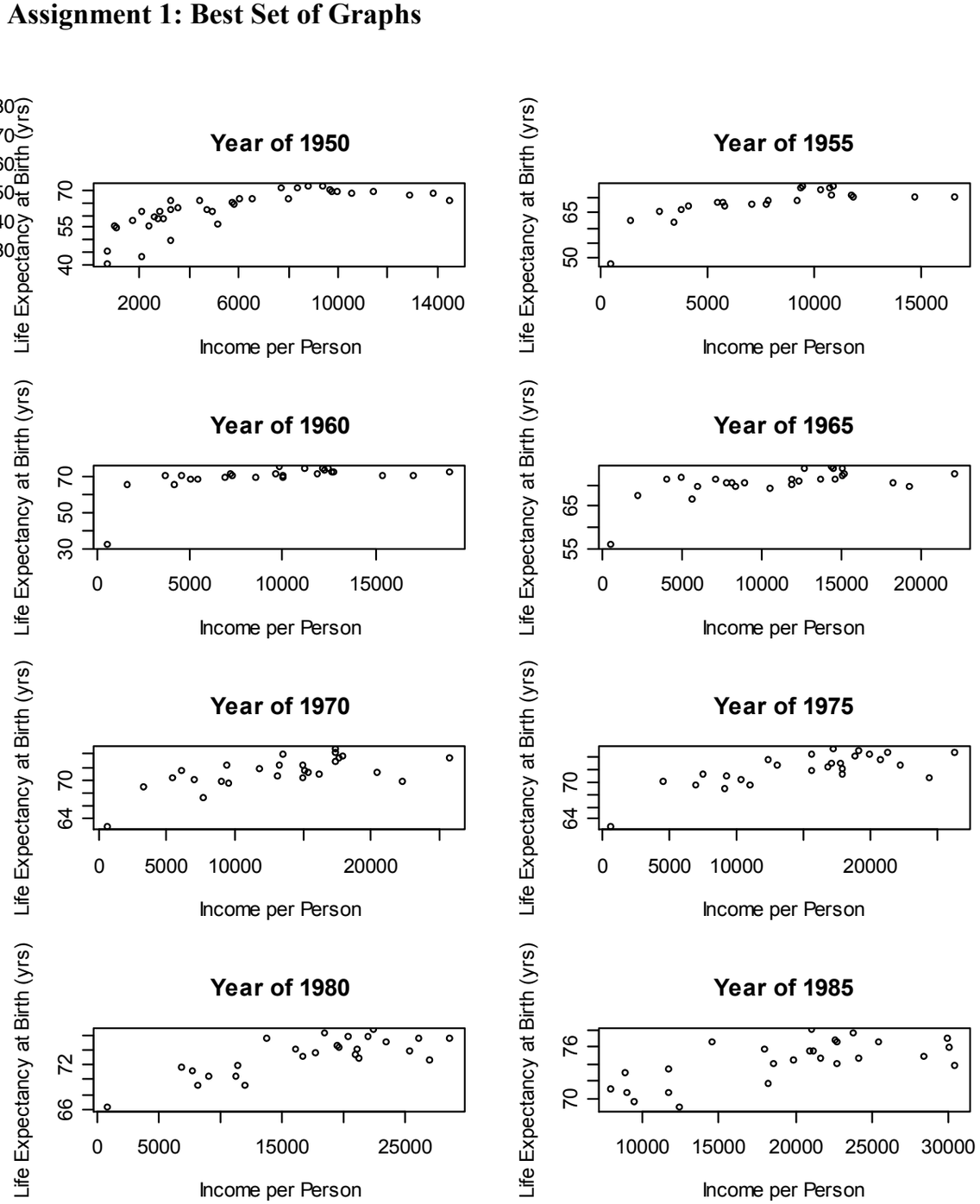
- To facilitate comparisons.
- To identify trends.

lattice graphics are simply better than traditional graphics for achieving these goals

lattice



traditional



Trellis suite from S/S-PLUS, multi-panel conditioning

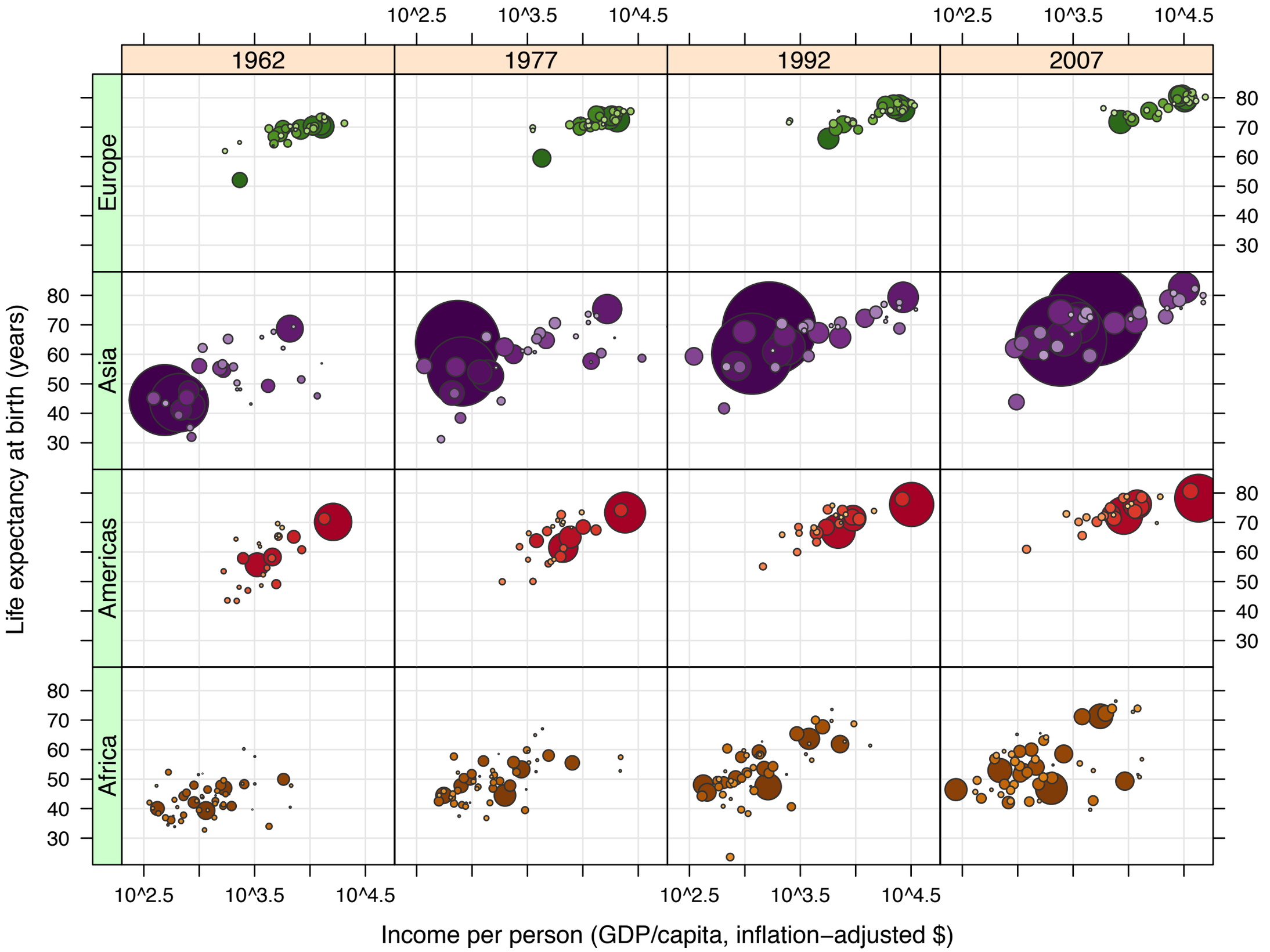
lattice: self-contained independent implementation of Trellis in R, using the low-levels tools of grid (vs. "traditional graphics")

good news: less need to exert fine control

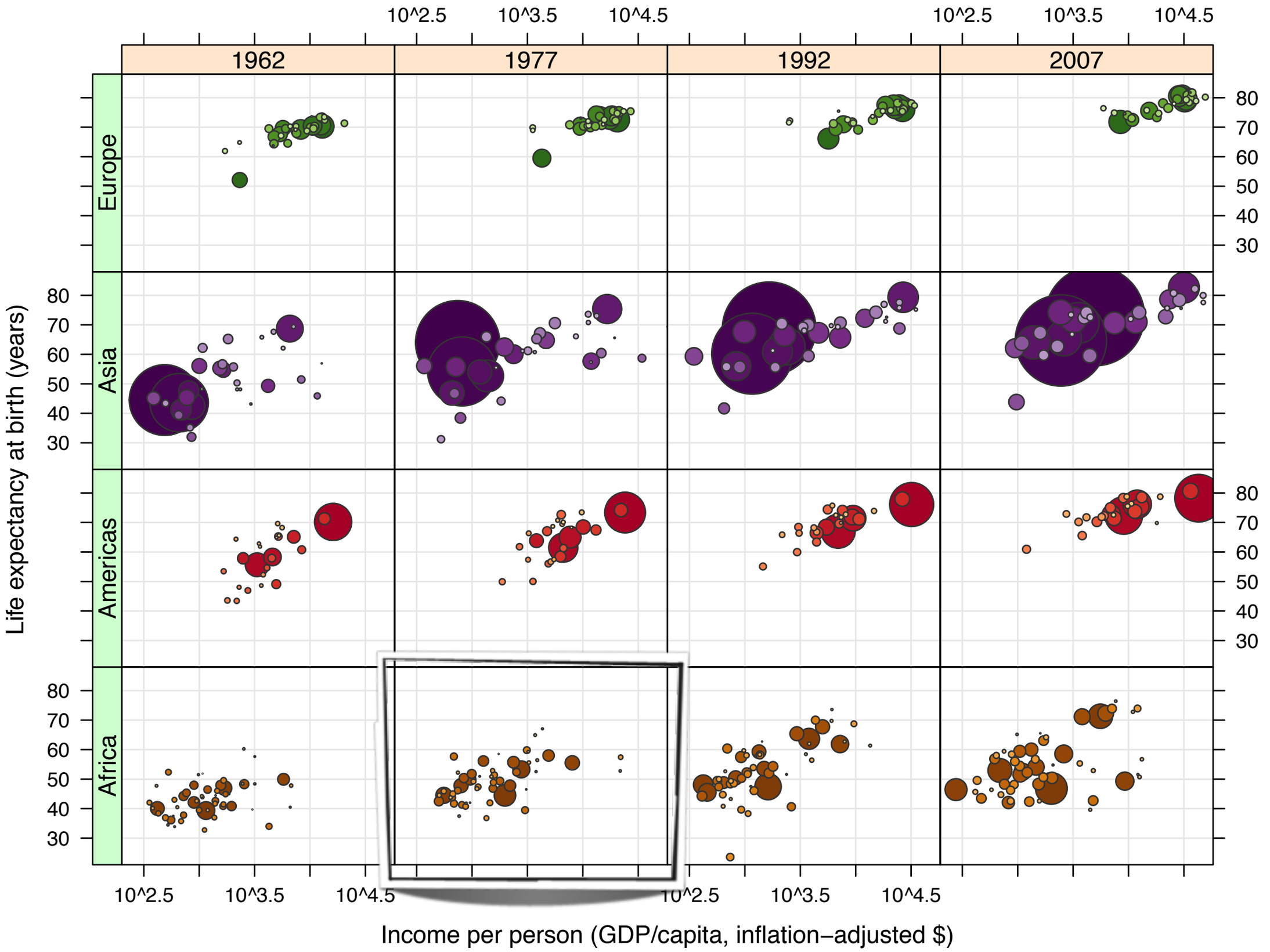
bad news: more difficult (?) and/or more unfamiliar (?) to exert fine control ... arises from the design of the system ("one calls says it all") and from grid (which most of us don't know much about)

bottom line: lattice is still worth mastering, no question

*ggplot2 -- another interesting option that came on the scene later than lattice

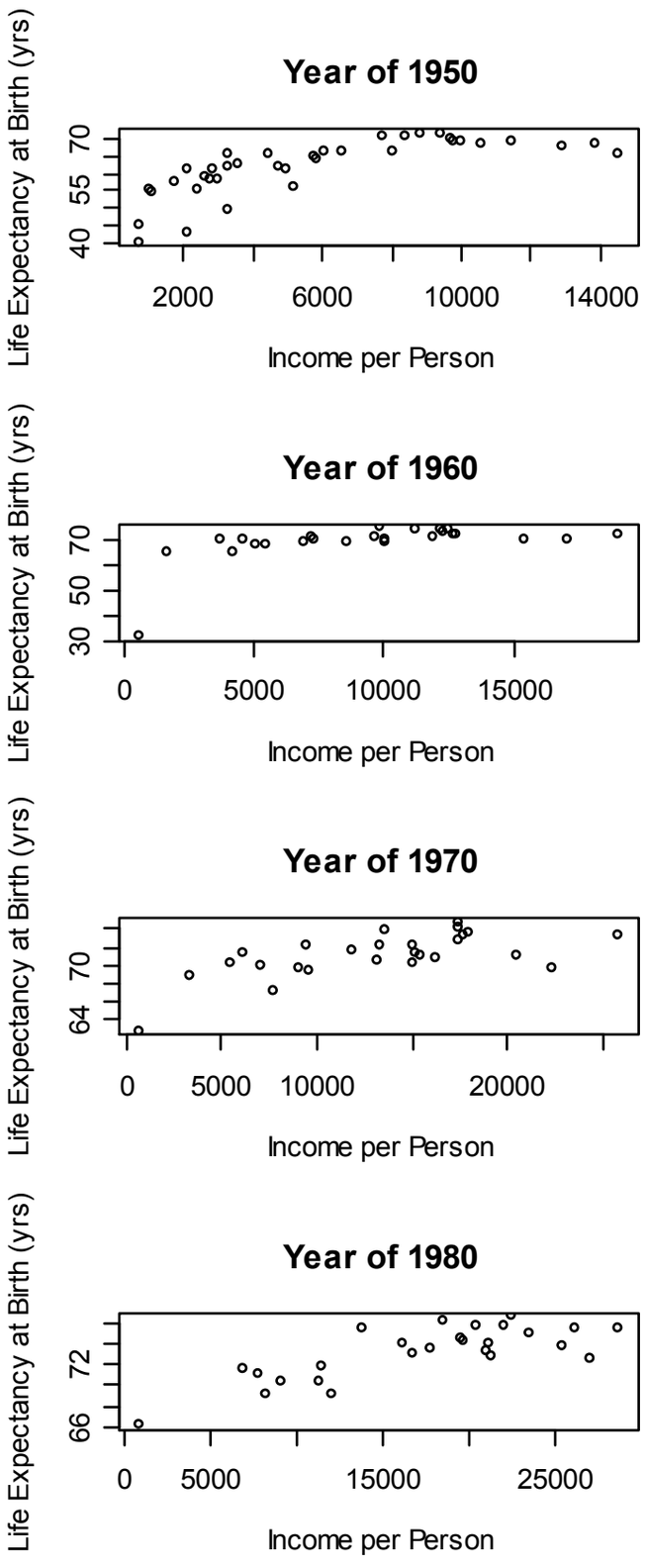


This is what is meant by multi-panel conditioning.



This is a panel.

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

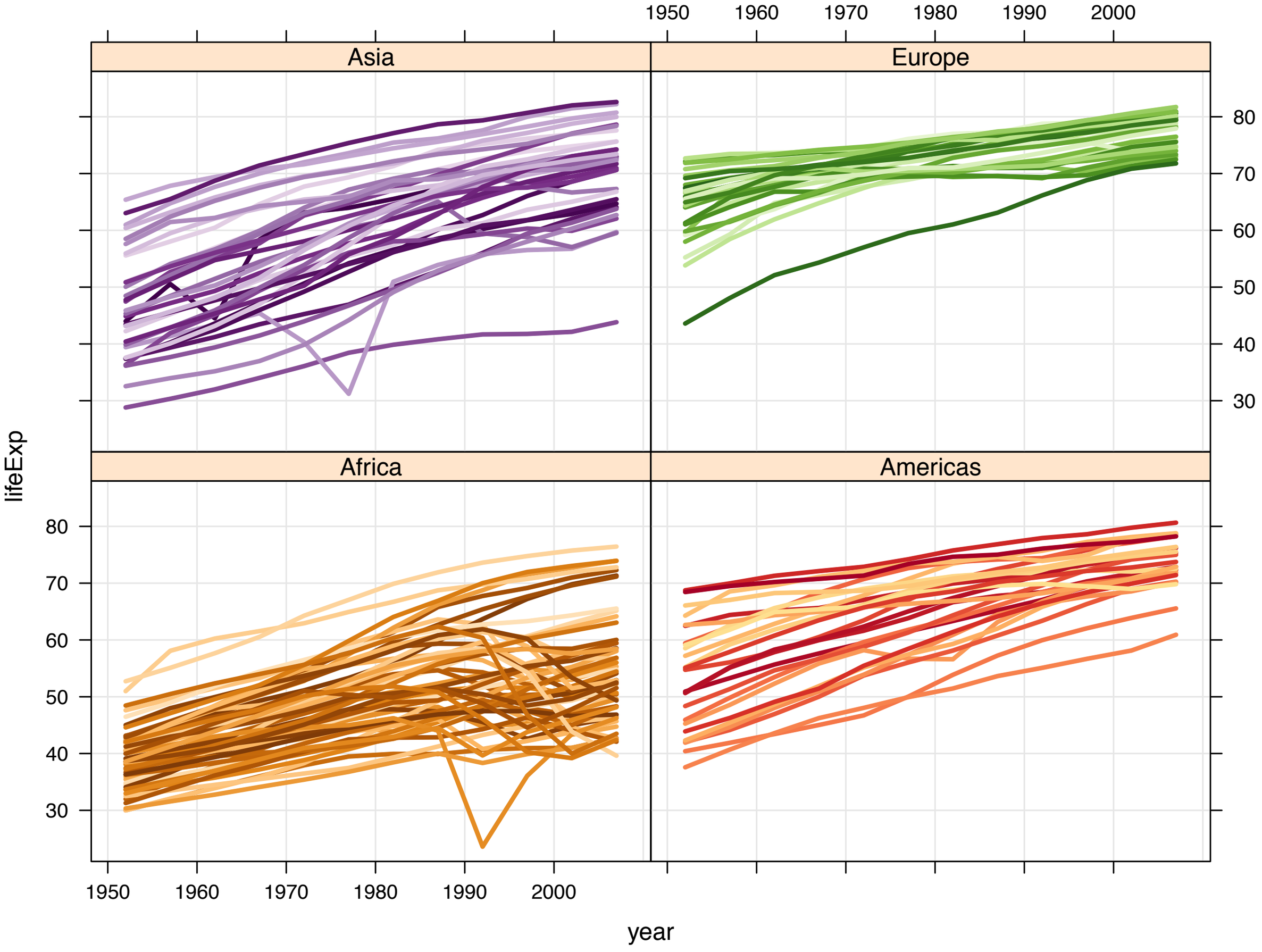
misallocation of space, too much spent on annotation, not enough on *data*

lattice: good for making cohesive groups of plots

- Main reason for lattice (and its inspiration, Trellis) was to combine plots on one page in a sensible manner
- (In real life, multi-panel conditioning and other visual features are more used and more useful than animation)
- lattice is a HUGE improvement over base R graphics for this task
- lattice often produces better plots, at default settings, even for single plots
- It is harder to customize and adorn lattice plots, compared to base R graphics (but the need for this is also greatly reduced).

1992
2007
1962
1977
Life expectancy at birth (years)
Income per person (GDP/capita, inflation–adjusted $)
1000
10000
30
40
50
60
70
80
Africa
Americas
Asia
Europe
Oceania

Asia
Europe
Africa
Americas
1000
10000
80
70
60
50
40
30
Life expectancy at birth (years)
Income per person (GDP/capita, inflation–adjusted $)
Africa
Americas
Asia
Europe
Oceania

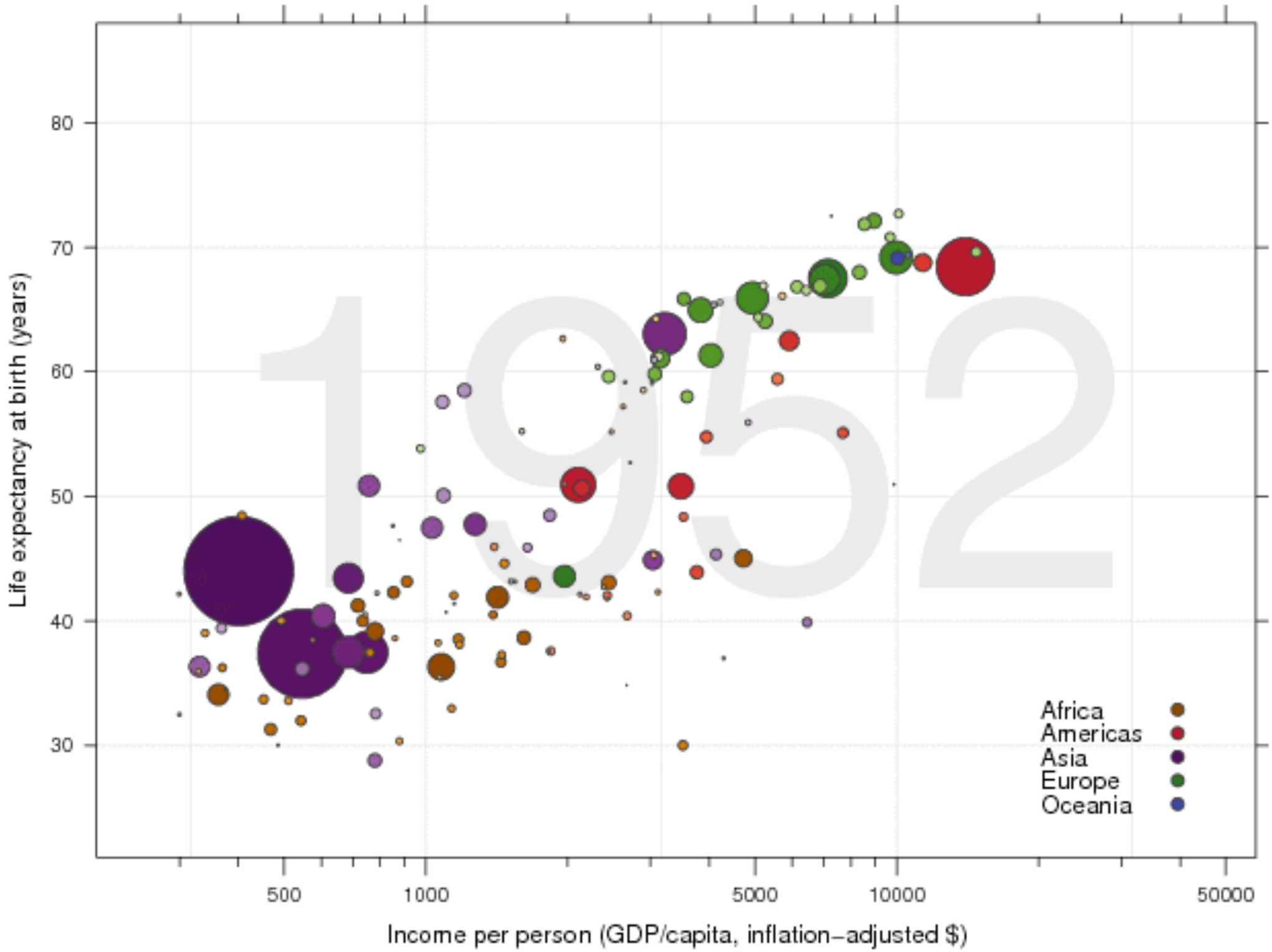

Asia
Europe
Africa
Americas
1950
1960
1970
1980
1990
2000
80
70
60
50
40
30
lifeExp
year

lattice vs. base R graphics

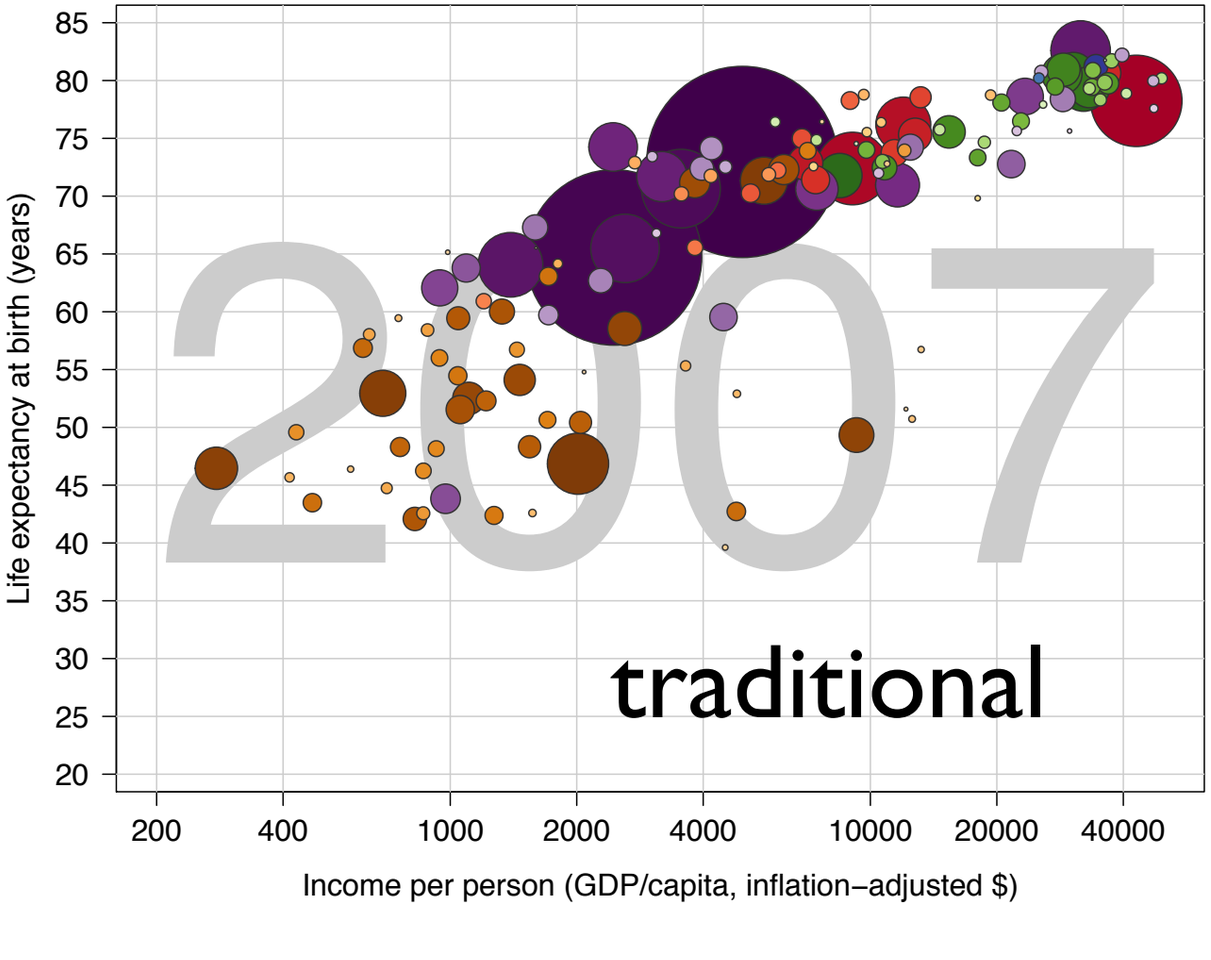
- Know which tool to use when.
- Good rule of thumb:
  - Want a decent plot in 10 mins? Use lattice. (Presumes you learn enough about lattice to get sthg decent in 10 mins.)
  - Want a highly customized plot in ~1 hr and you don't need multi-panel conditioning? Try base graphics. (Presumes you learn enough about adding elements to basic plots to achieve this.)
  - Want a customized plot in 1 day? Do you need multi-panel conditioning? Will you make similar plots again? Or will this be featured in a paper or talk? Use lattice and do what needs to be done, e.g. write a custom panel function.

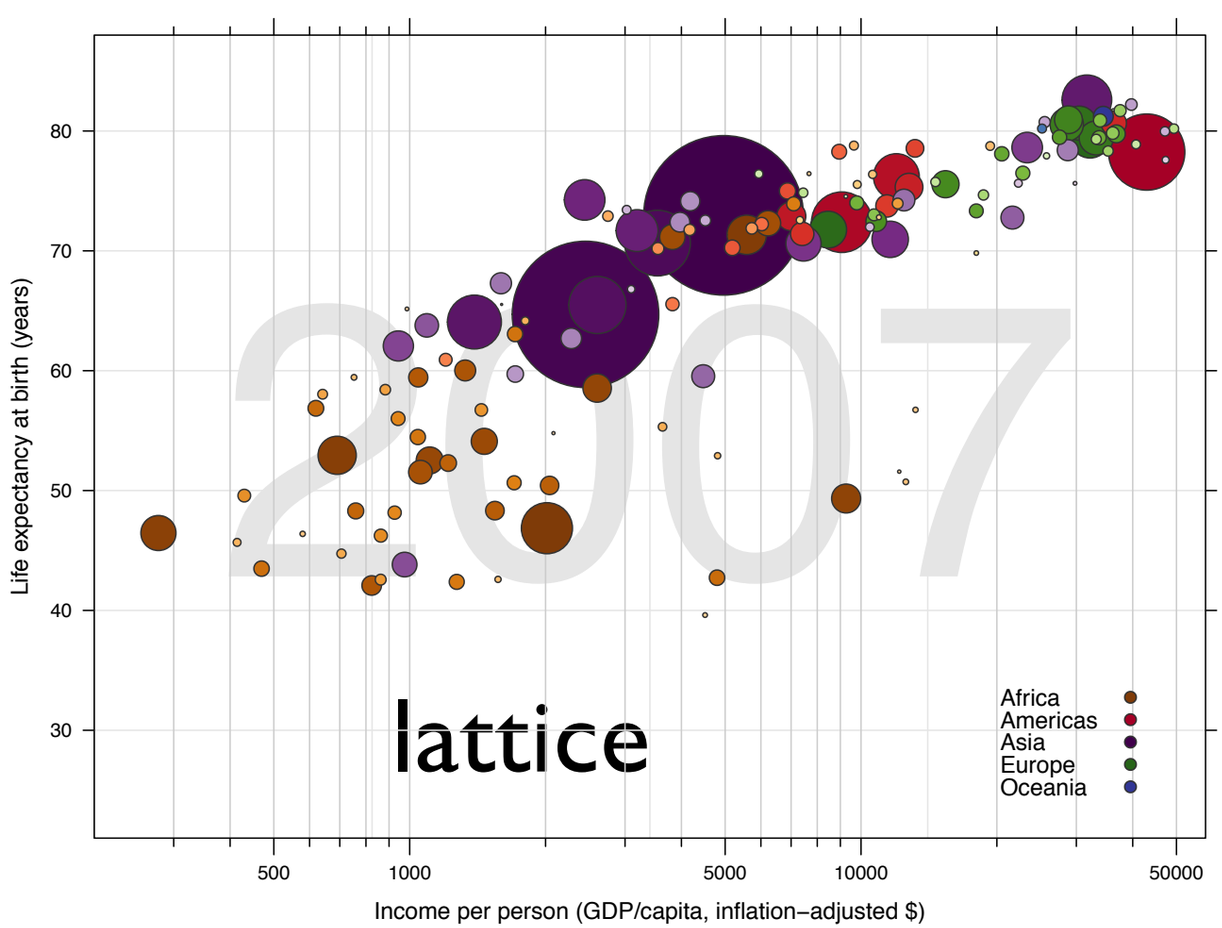
lattice vs. base R graphics

- Know what to expect, how to work.
- Traditional graphics: Call plot() to set up a coordinate system (and maybe print nothing else) and slowly add your elements.
- Lattice, 80% of the time: Call xyplot(), for example, and use oodles of arguments to specify what you want.
- Lattice, other 20% of the time: Re-define some of the standard functions or lists lattice uses to control certain elements, i.e. modify or extend the default to suit your purposes, and then call xyplot(), probably still using oodles of arguments.



Produced with lattice


2007
traditional
Life expectancy at birth (years)
85
80
75
70
65
60
55
50
45
40
35
30
25
20
200
400
1000
2000
4000
10000
20000
40000
Income per person (GDP/capita, inflation–adjusted $)


2007
lattice
Life expectancy at birth (years)
80
70
60
50
40
30
500
1000
5000
10000
50000
Income per person (GDP/capita, inflation–adjusted $)
Africa
Americas
Asia
Europe
Oceania

Scatterplot

See Ch. 5 in Sarkar (2008).

- Graph of two quantitative variables against each other in the plane.
- Single most important graphic in your statistical life.
  - Always plot the data BEFORE doing anything else.
- Things to consider and control:
  - join points? convey extra information via color or symbols or size?
  - juxtapose several scatterplots of Y vs X for different values of Z, a categorical variable?
  - add a regression line or scatterplot smooth?
  - use generalized scatterplot to avoid overplotting?

xyplot()

Read R documentation for xyplot() and panel.xyplot()

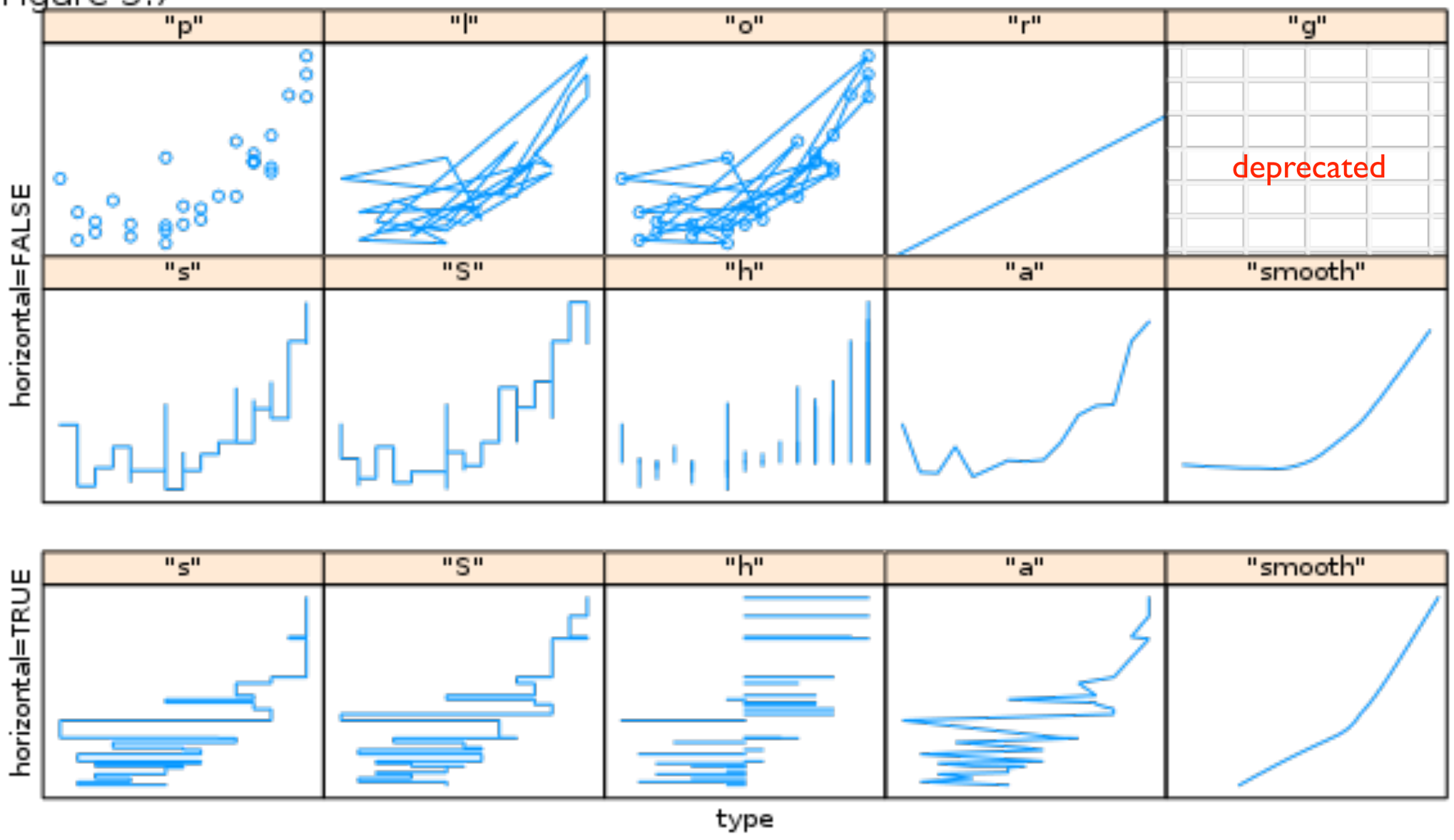
- The first lattice function to learn and master.
- The `type` argument is incredibly useful. Use it!
- It is a character vector consisting of *one or more* of <read the help file for the values>. If `type` has more than one element, an attempt is made to combine the effect of each of the components.

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()

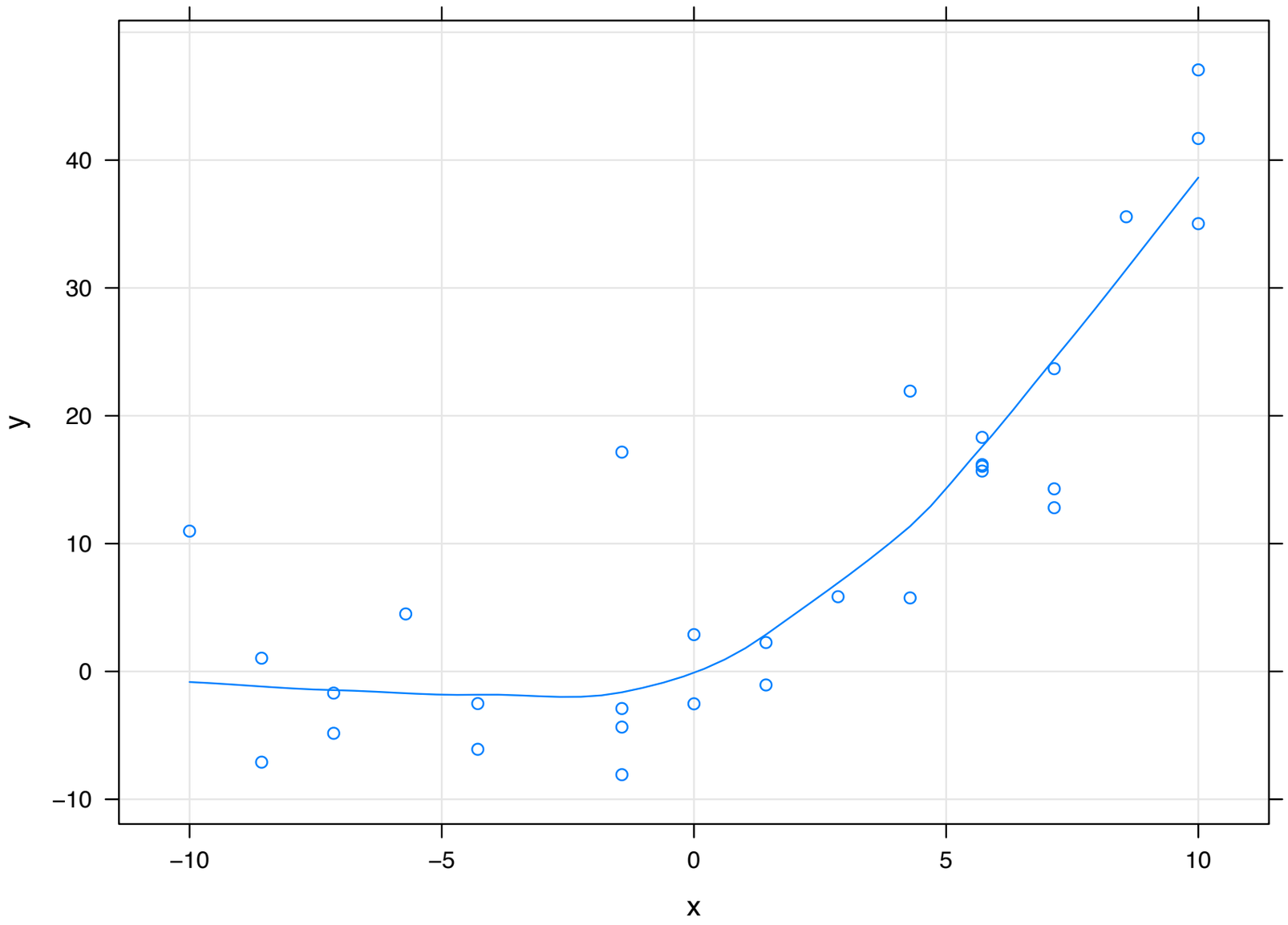
Table 5.1. The effect of various values of the type argument in panel.xyplot().

From Ch. 5 in Sarkar (2008).

Figure 5.7



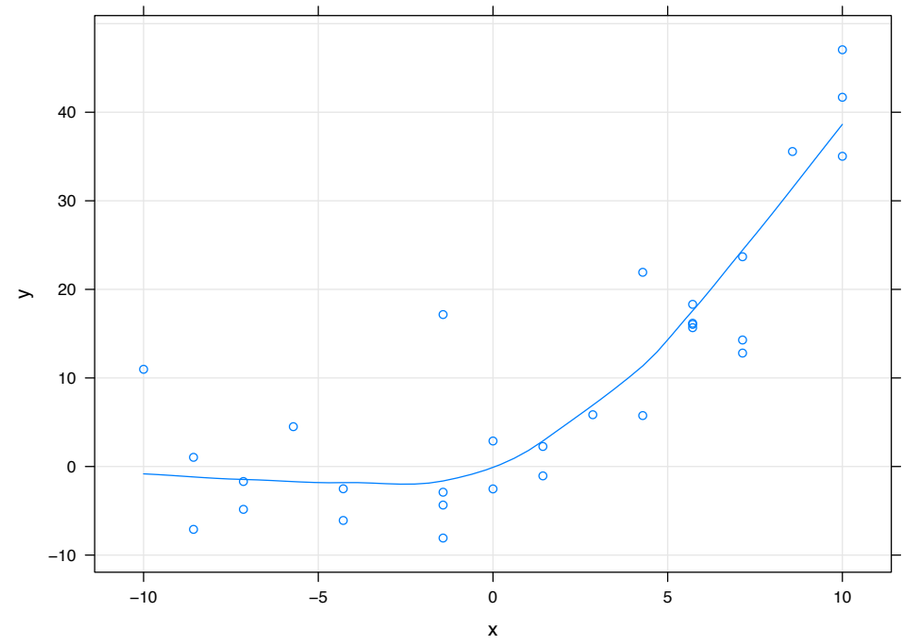
From Ch. 5 in Sarkar (2008).



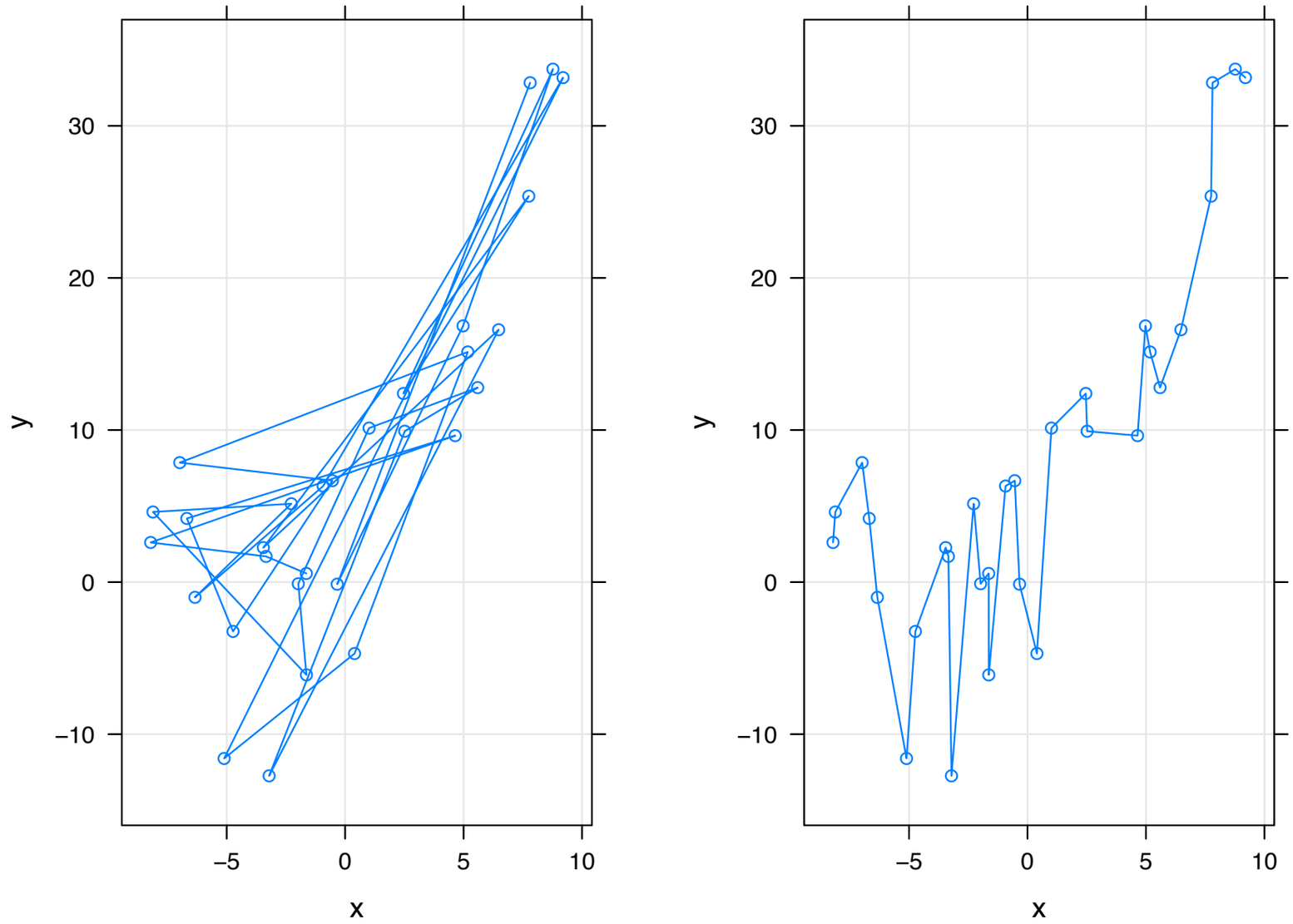

```
xyplot(y ~ x, type = c("g","p","smooth"))
```

Reference grids and other common elements greatly facilitate the interpretation and comparison of multi-panel displays.

If you want to get some quick info on Y ~ X, prior to full-blown model selection, the loess smooth is handy.




```
... type = c("g","p","smooth"))
```




```
x <- runif(n = 30, min = -10, max = 10)
y <- x + 0.25 * (x + 1)^2 + rnorm(length(x), sd = 5)
uglyPlot <- xyplot(y ~ x, type = c("g","b"))
xOrd <- order(x); x <- x[xOrd]; y <- y[xOrd]
prettyPlot <- xyplot(y ~ x, type = c("g","b"))
print(uglyPlot, pos = c(0, 0, 0.5, 1), more = TRUE)
print(prettyPlot, pos = c(0.5, 0, 1, 1), more = FALSE)
```

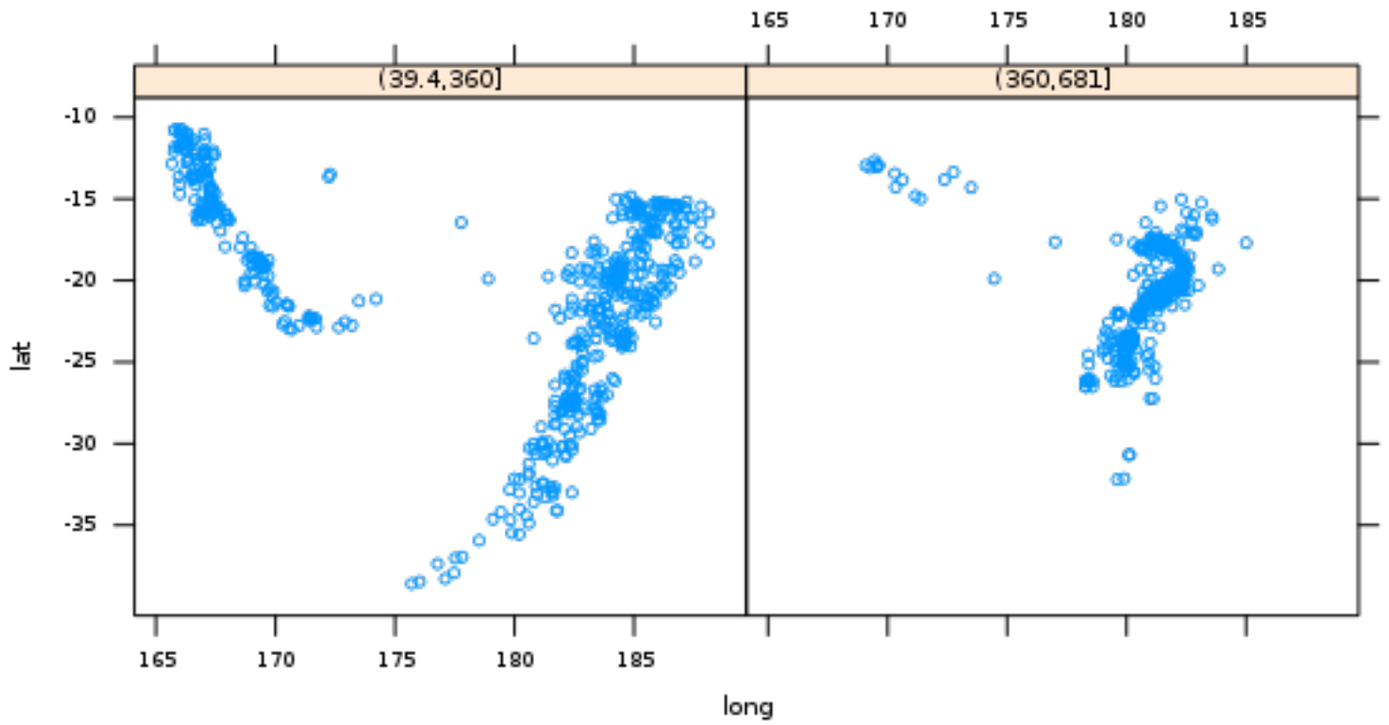
If you connect the dots, make sure to sort first!

quakes data

See Sarkar (2008).

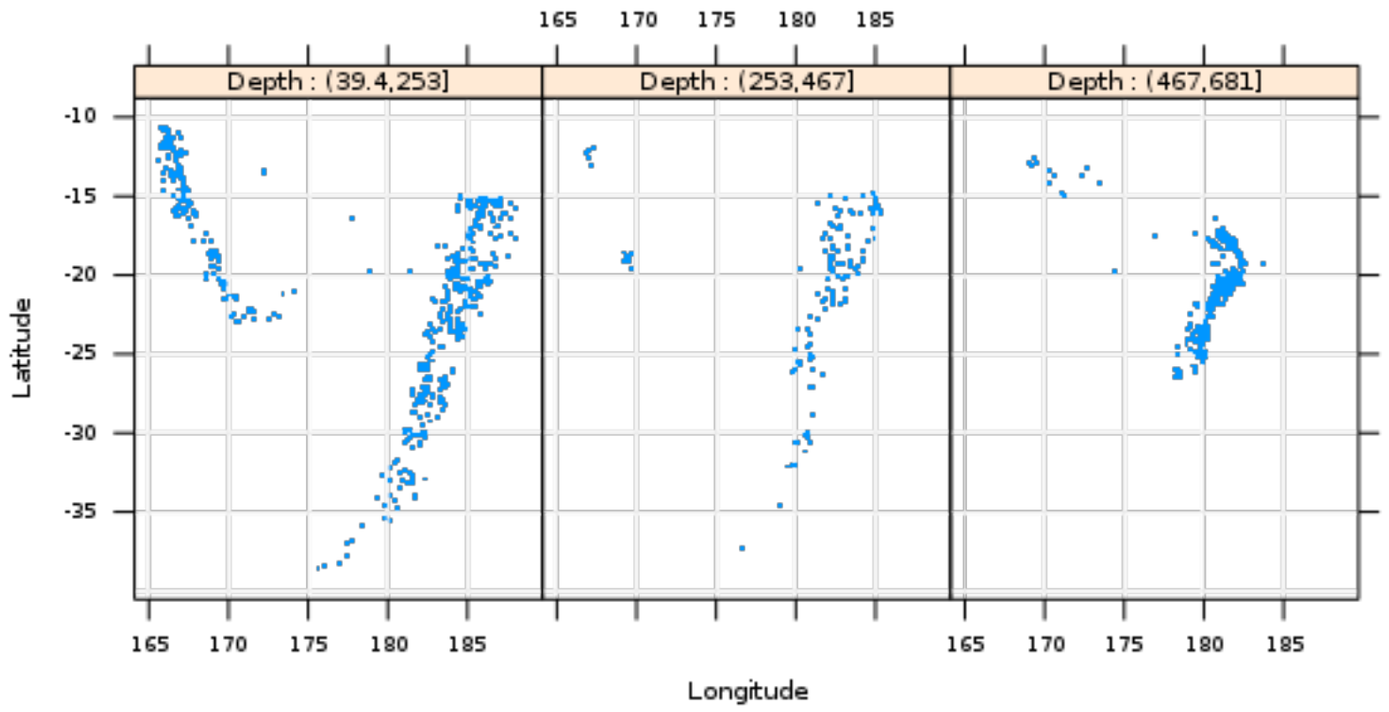
- Records location (latitude, longitude, depth) and magnitude of earthquakes near Fiji since 1964
- Latitude and longitude are (X, Y)
- The third quantitative variable, depth, can be turned into a 'shingle', for graphing purposes. See Sarkar 2.1.4.
- A 'shingle' is similar to converting a quantitative variable into a categorical one, but we potentially allow the intervals to overlap. Shingles are useful as the conditioning variable in lattice plots, i.e. the variable whose levels correspond to distinct panels.
- Functions to create shingles: cut(), equal.count()
- Code for following figures available here

Figure 5.1



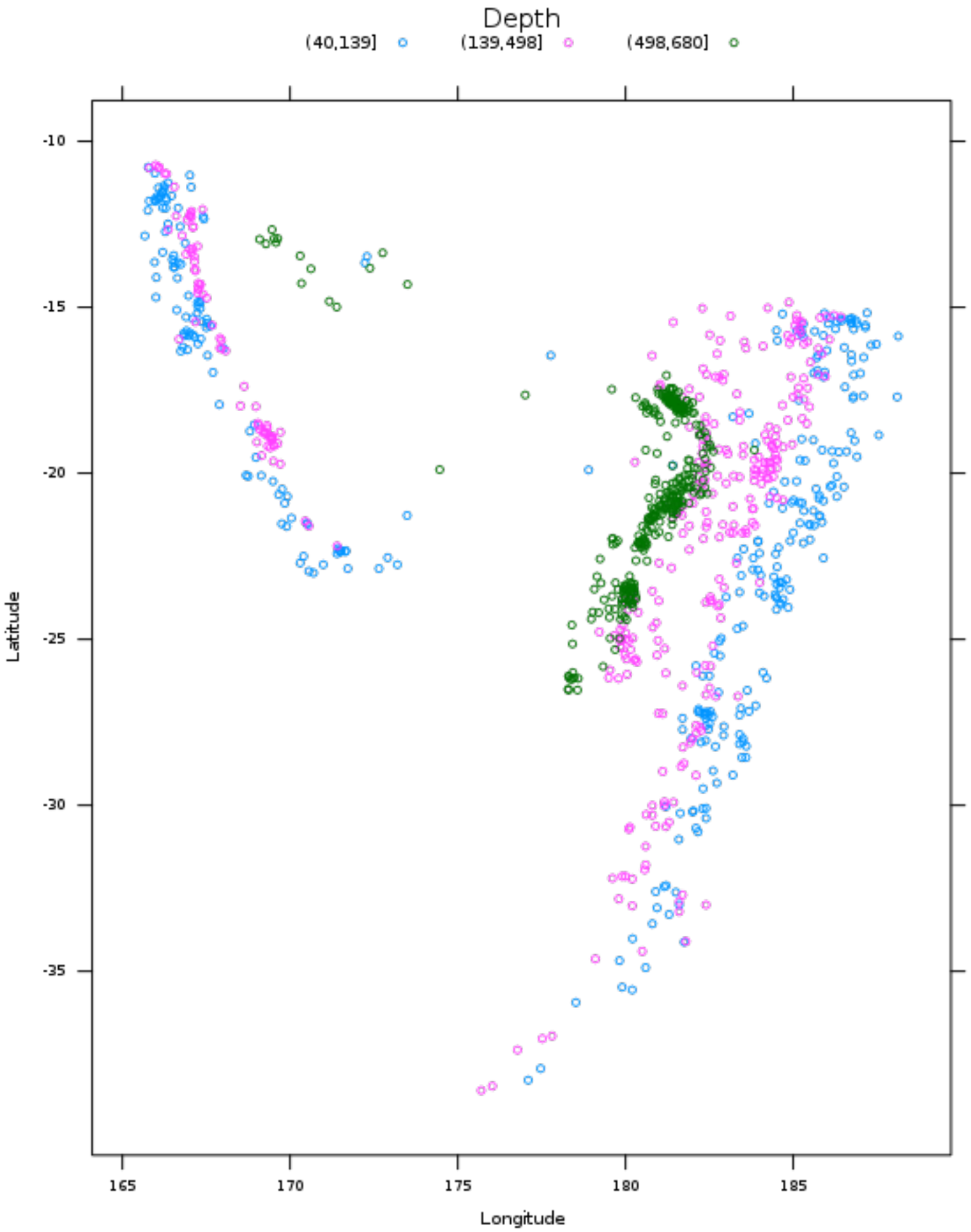
Depth turned into a shingle w/ non-overlapping intervals, i.e. a factor.

Figure 5.2



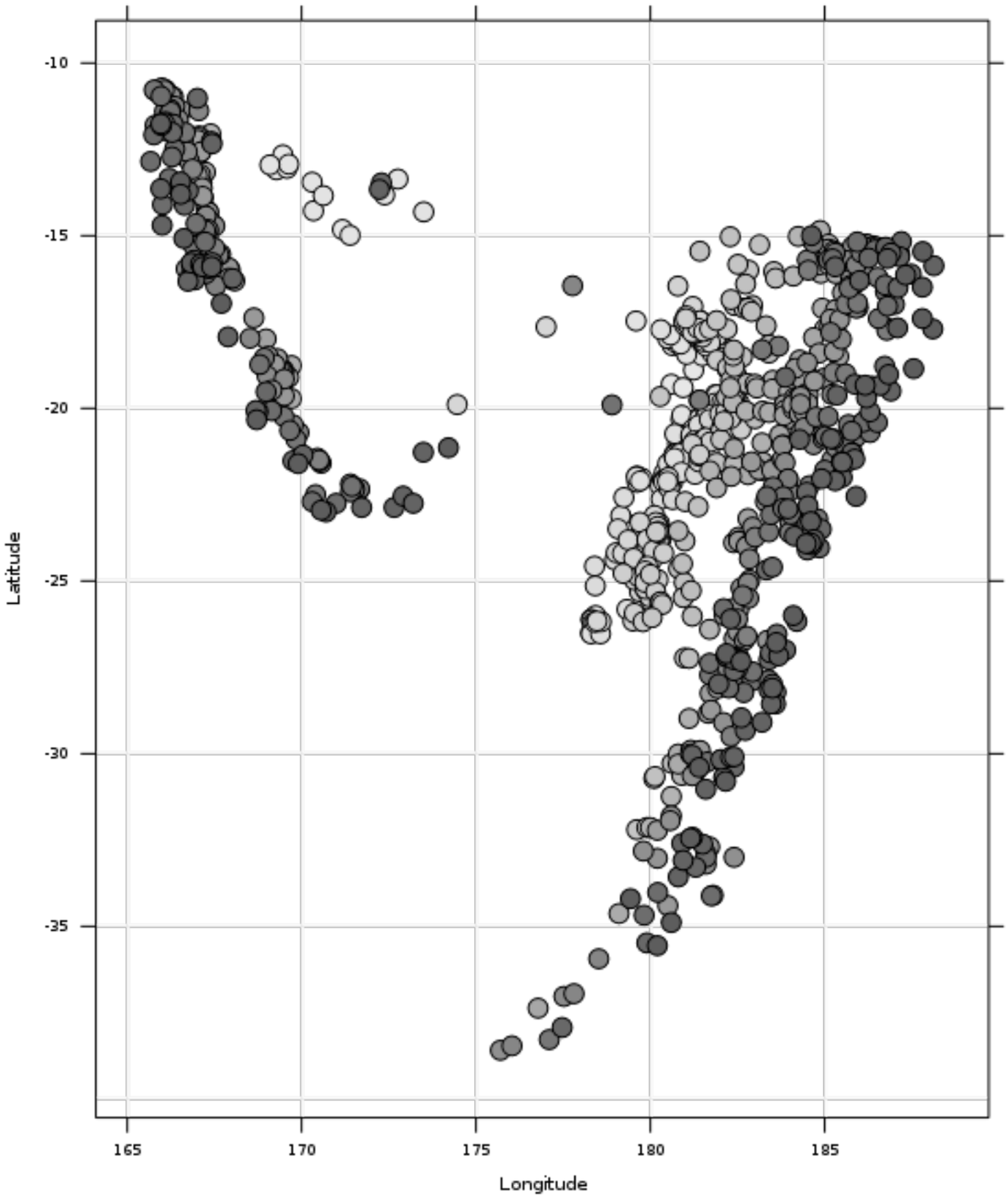
Depth shingle has 3 levels
Strip more informative
Plot symbol is a dot
Isometric scales
Reference grid

Figure 5.3



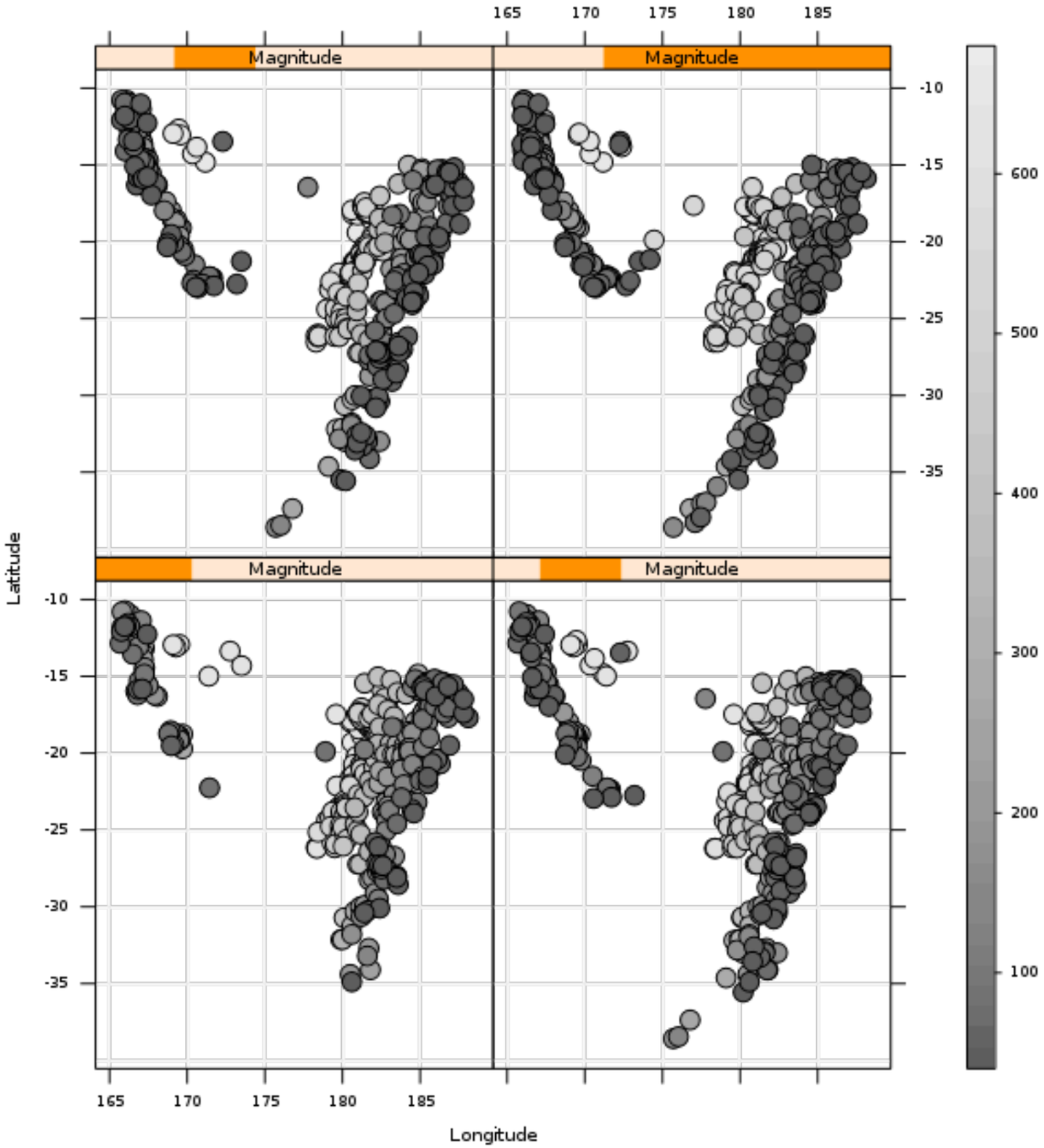
Depth encoded via superposition, via the groups argument

Figure 5.4



Depth encoded via color of shaded circle

Figure 5.6

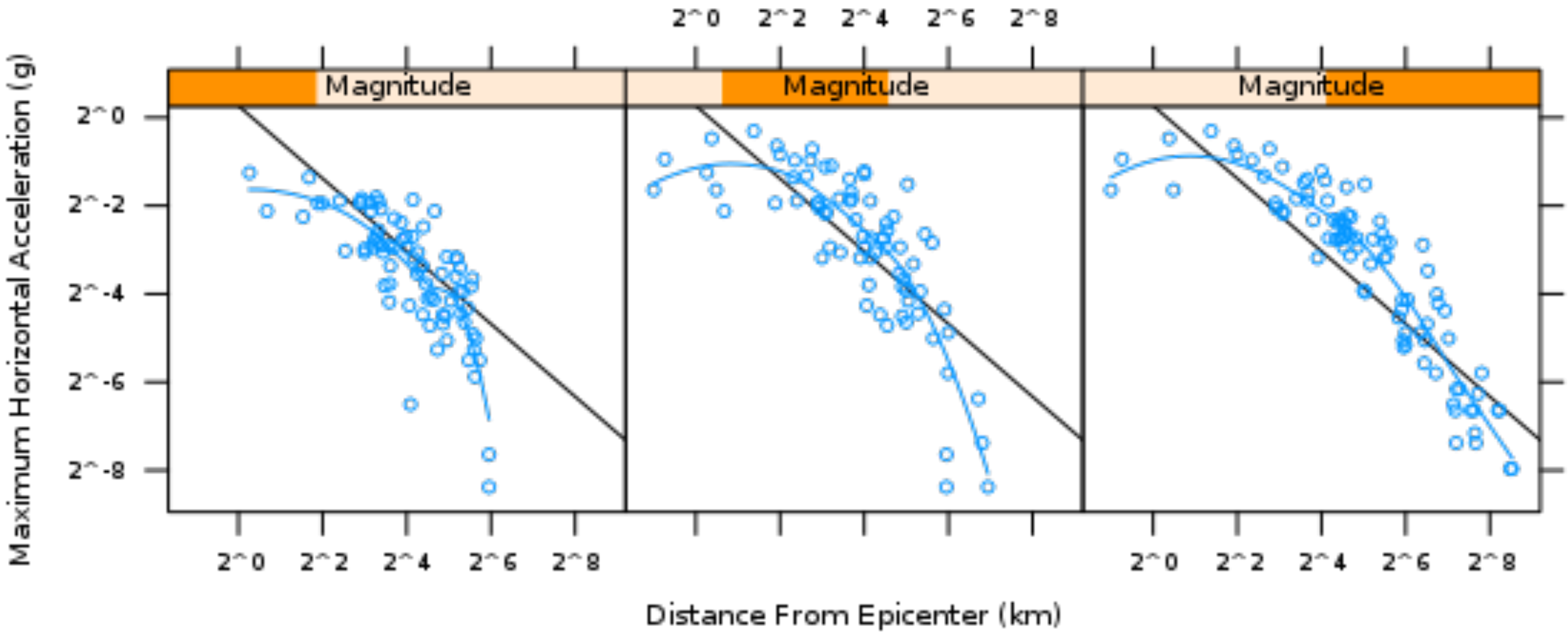


Depth encoded via color of shaded circle

AND

Magnitude treated as a shingle, i.e. the conditioning variable

Figure 5.10




```
library("locfit")
Earthquake$Magnitude <-
    equal.count(Earthquake$Richter, 3, overlap = 0.1)
coef <- coef(lm(log2(accel) ~ log2(distance), data = Earthquake))

## Figure 5.10
xyplot(accel ~ distance | Magnitude, data = Earthquake,
       scales = list(log = 2), col.line = "grey", lwd = 2,
       panel = function(...) {
           panel.abline(reg = coef)
           panel.locfit(...)
       },
       xlab = "Distance From Epicenter (km)",
       ylab = "Maximum Horizontal Acceleration (g)")
```

Magnitude used as a shingle w/ overlapping intervals.

Common regression line added as a visual reference

Alternative scatterplot smoother from locfit package.

Lessons from xyplot() tour

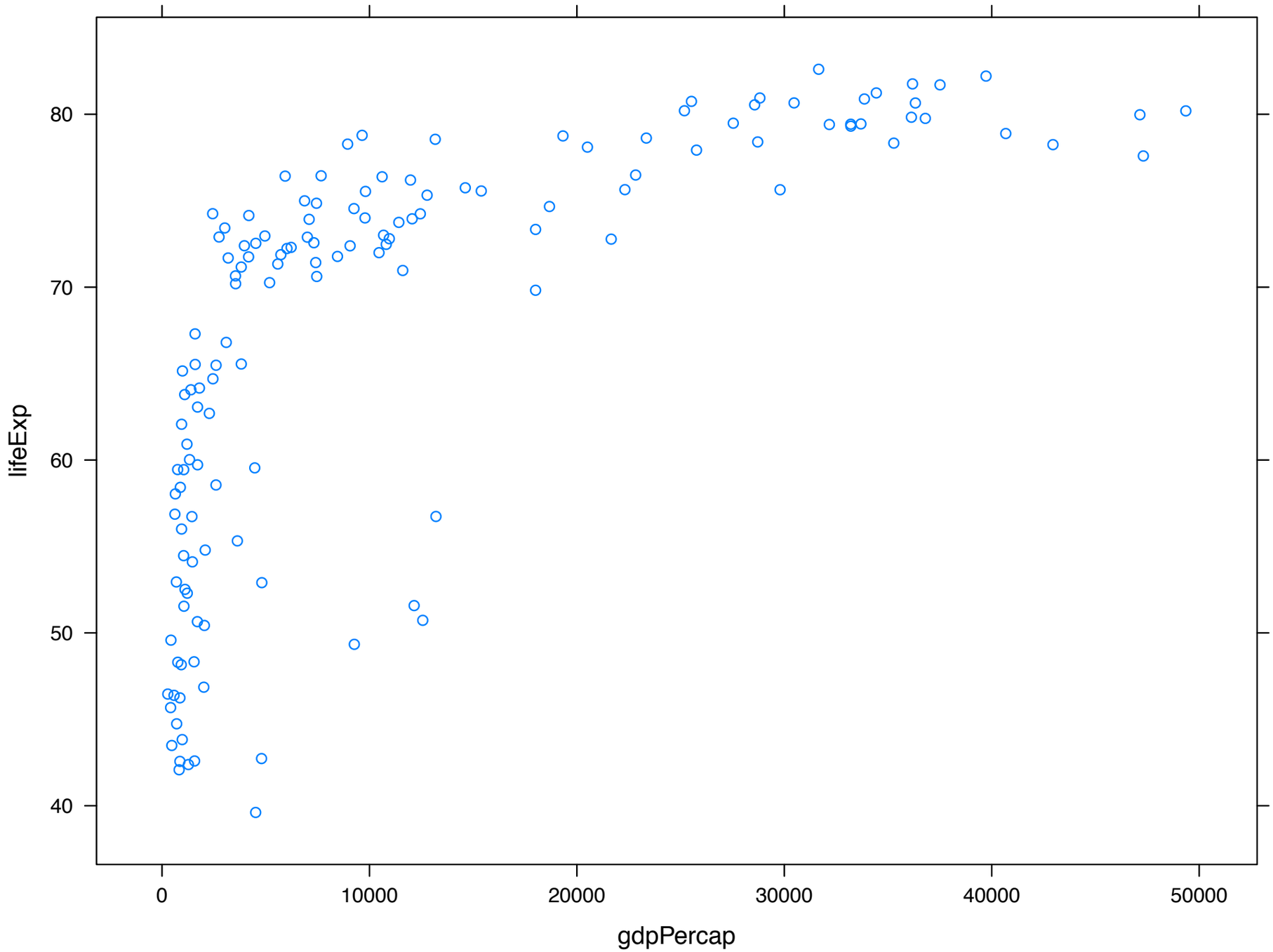
- Scatterplots are conceptually simple yet powerful.
- xyplot() produces nice scatterplots, even basic ones.
- Reference lines and, where appropriate, other common elements facilitate data interpretation.
- Various transformations and regression approaches can be enacted 'on the fly' by xyplot, such as the log transform and loess smoothing.
- The `type` argument is very useful for the above.

Lessons from xyplot() tour (cont'd)

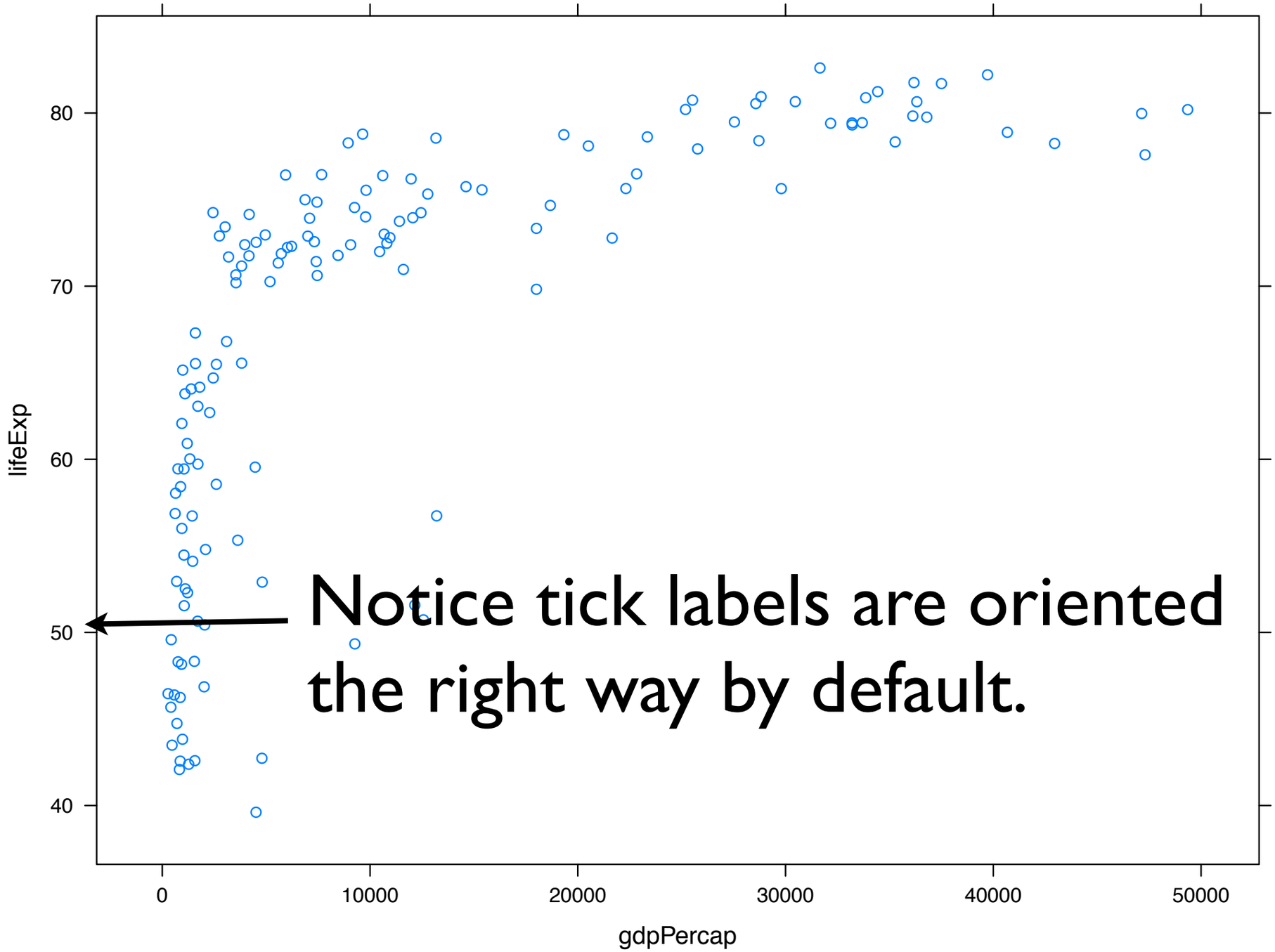
- The relationship between X and Y can be explored simultaneously for different values of a third variable via
  - conditioning into different panels
  - superposition with visual cues for the groups
- Consider treating an ancillary quantitative variable as a shingle.
- It is possible to depict four quantitative variables at once in a scatterplot. X and Y cover the two primary variables, the third can be conveyed (crudely) via a shingle, and the fourth encoded by color and/or size of plotting symbol.

step by step development of the Gapminder figure using lattice graphics

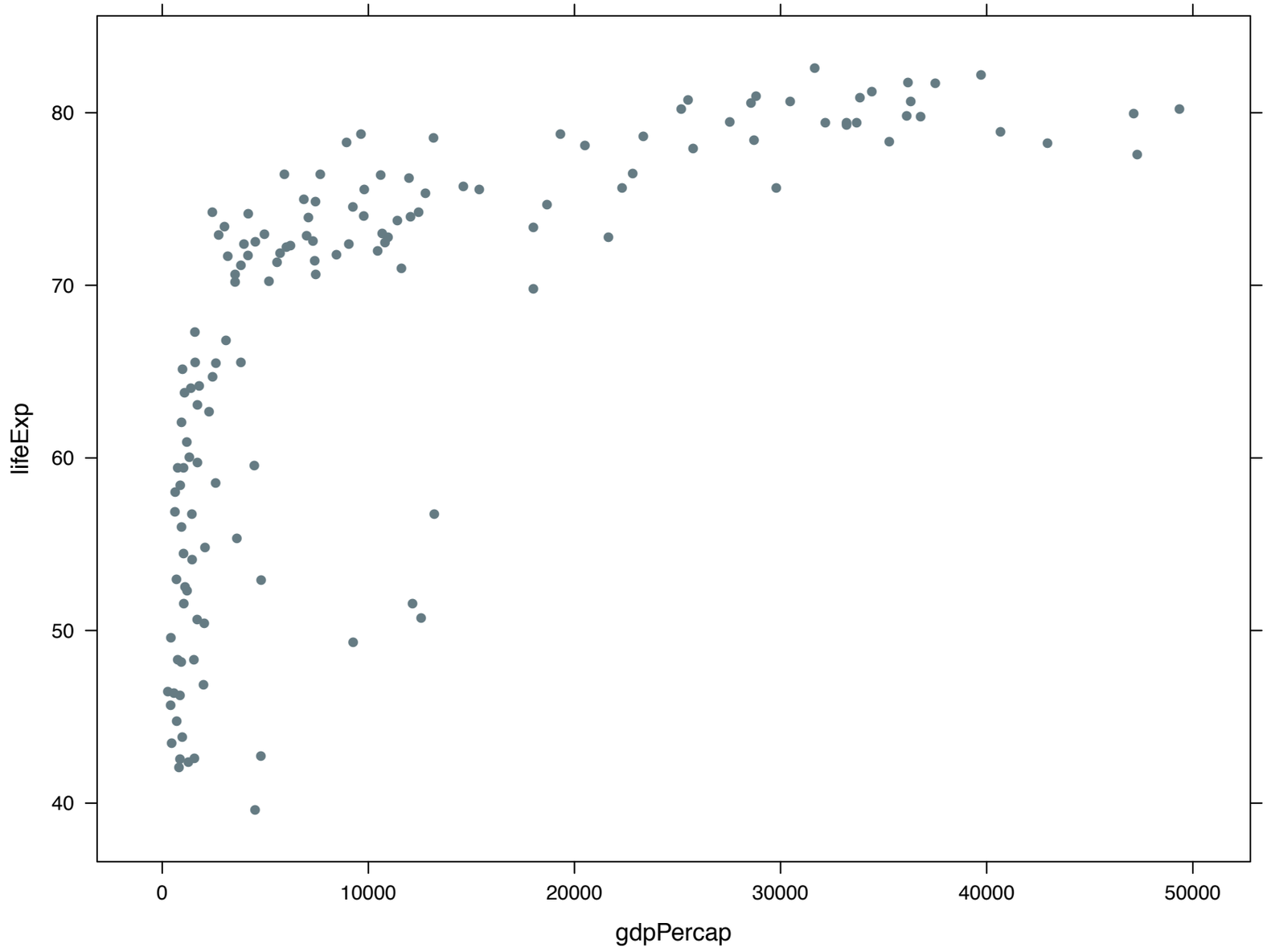
Note: most figures now here in two versions. One made under a default lattice theme and the other made under JB's lattice theme. Once the customizations gets profound enough, the difference goes away ....



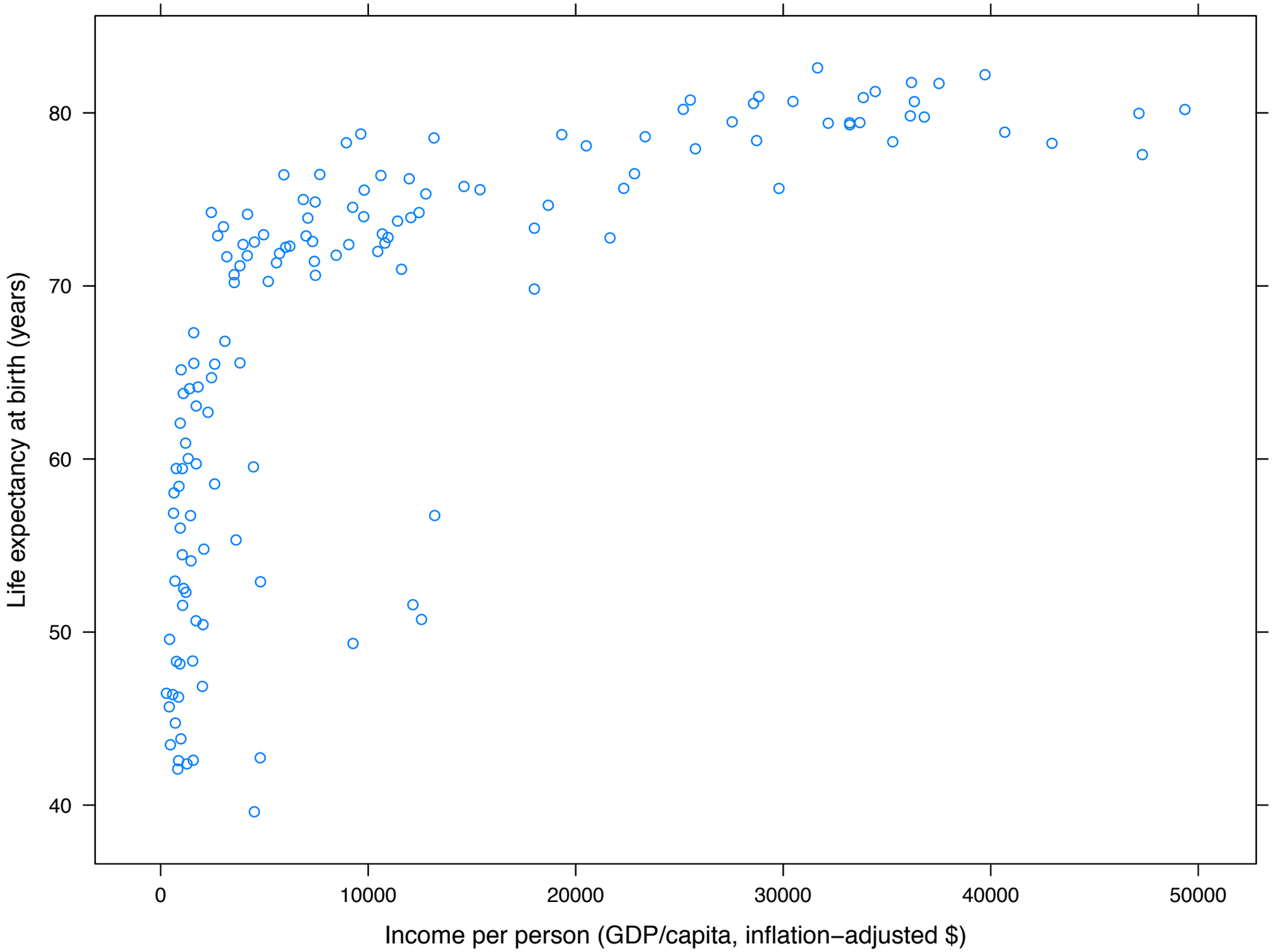

```
xyplot(lifeExp ~ gdpPercap, gDatOrdered, subset = year == jYear)
```



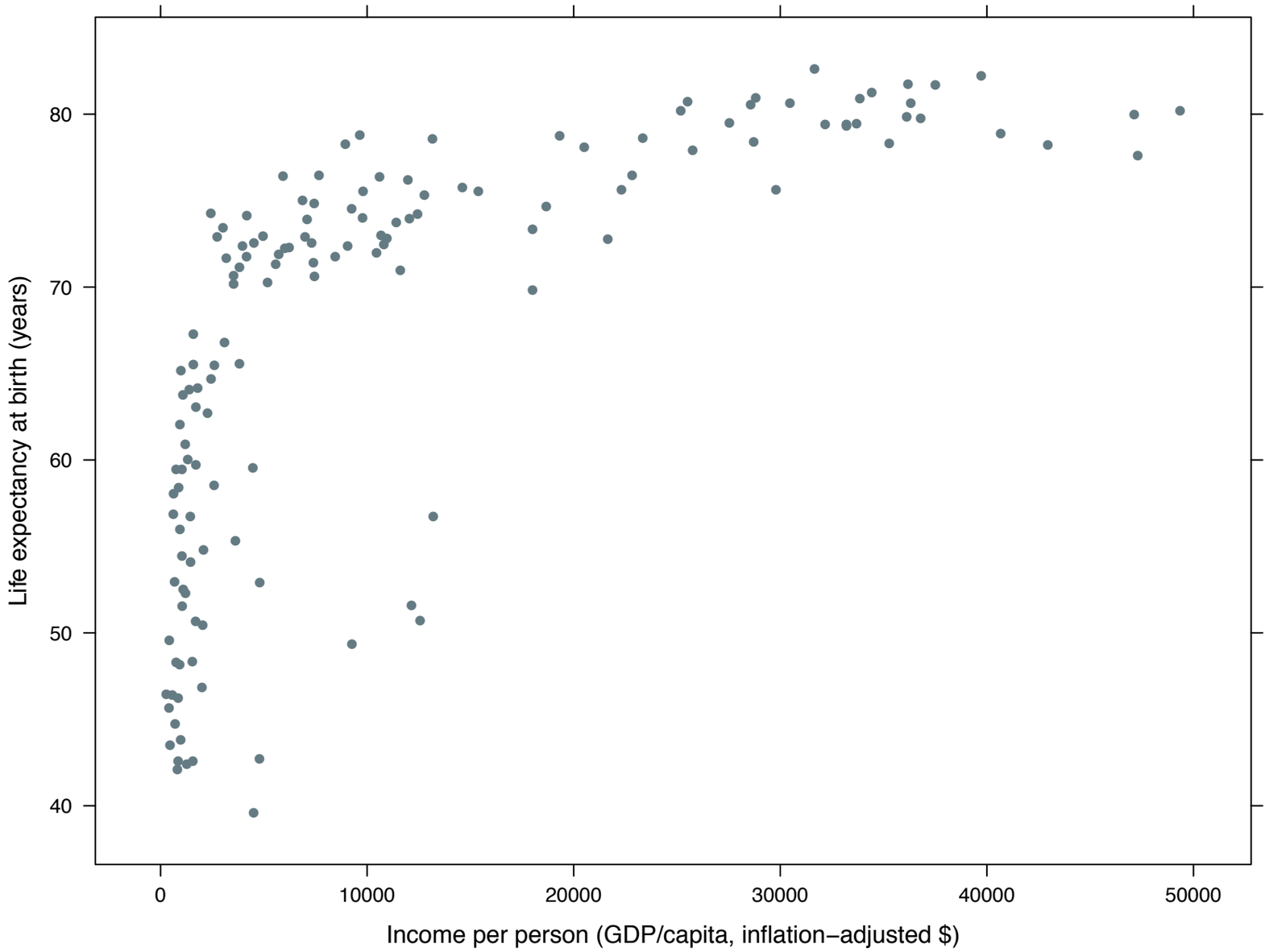

```
xyplot(lifeExp ~ gdpPercap, gDatOrdered, subset = year == jYear)
```



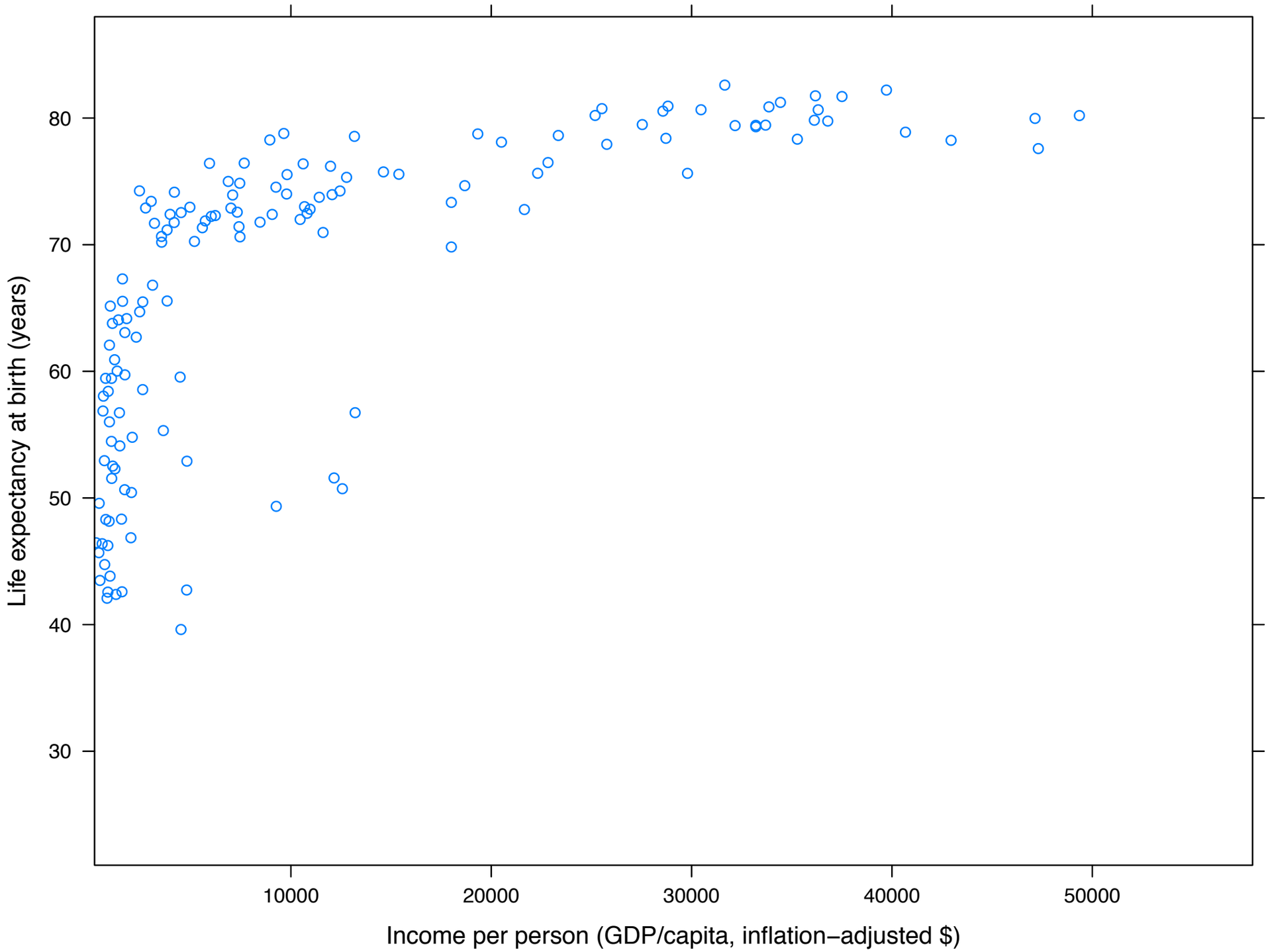

```
xyplot(lifeExp ~ gdpPercap, gDatOrdered, subset = year == jYear)
```



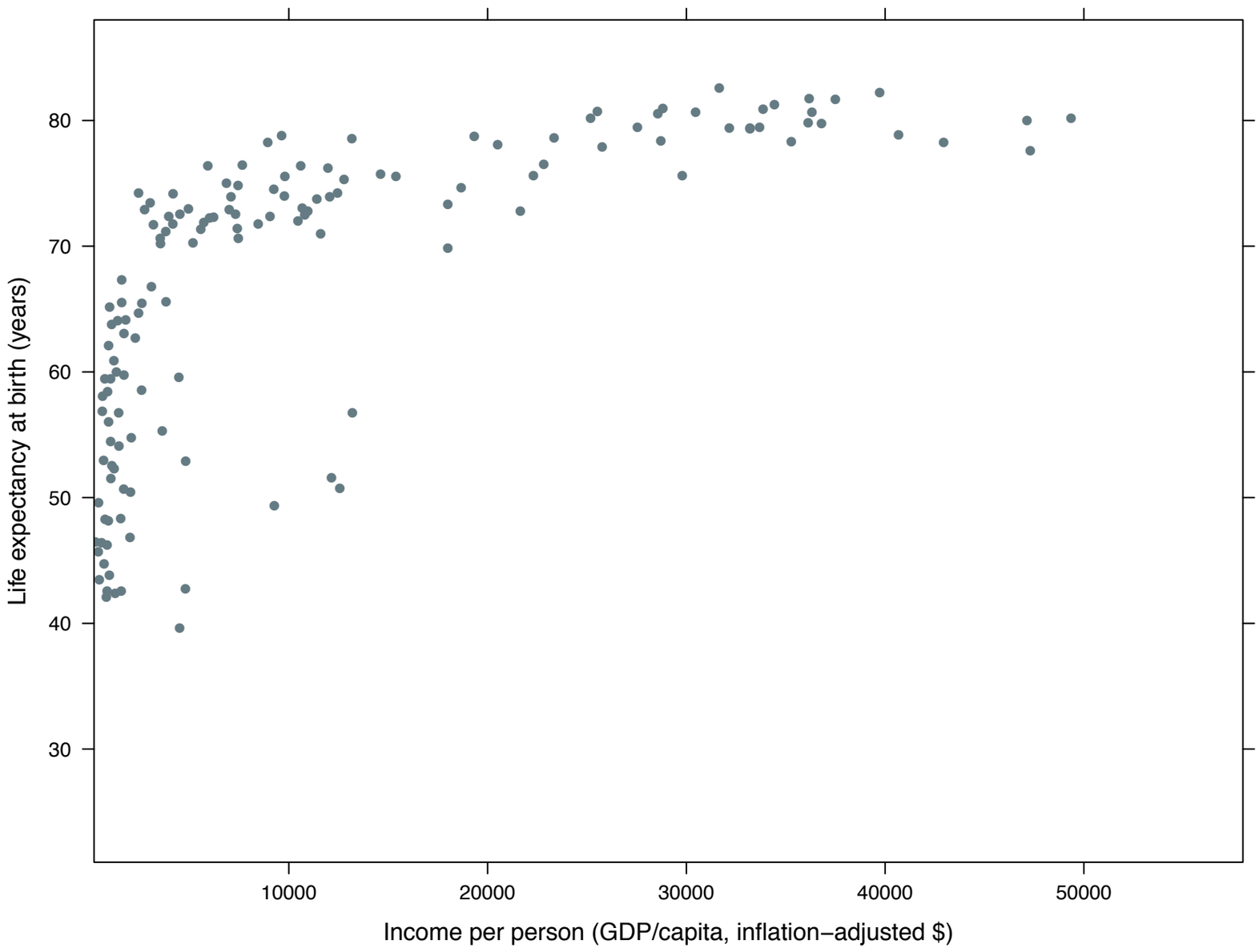

```
jXlab <- "Income per person (GDP/capita, inflation-adjusted $)"
jYlab <- "Life expectancy at birth (years)"
xyplot(lifeExp ~ gdpPercap, gDatOrdered, subset = year == jYear,
       xlab = jXlab, ylab = jYlab)
```



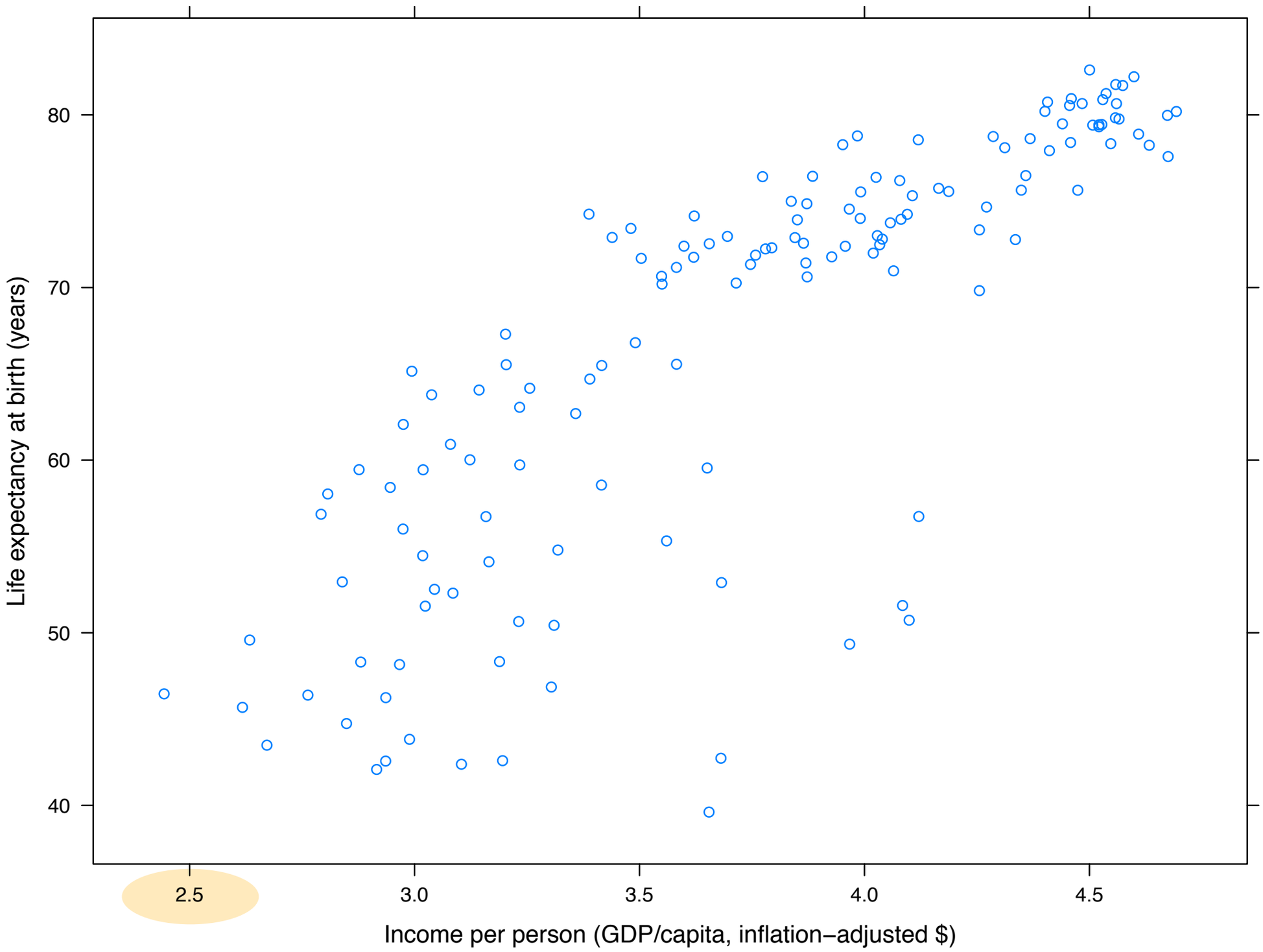

```
jXlab <- "Income per person (GDP/capita, inflation-adjusted $)"
jYlab <- "Life expectancy at birth (years)"
xyplot(lifeExp ~ gdpPercap, gDatOrdered, subset = year == jYear,
       xlab = jXlab, ylab = jYlab)
```




```
jXlim <- c(200, 58000)
jYlim <- c(21, 88)
xyplot(lifeExp ~ gdpPercap, gDatOrdered, subset = year == jYear,
       xlab = jXlab, ylab = jYlab,
       xlim = jXlim, ylim = jYlim)
```

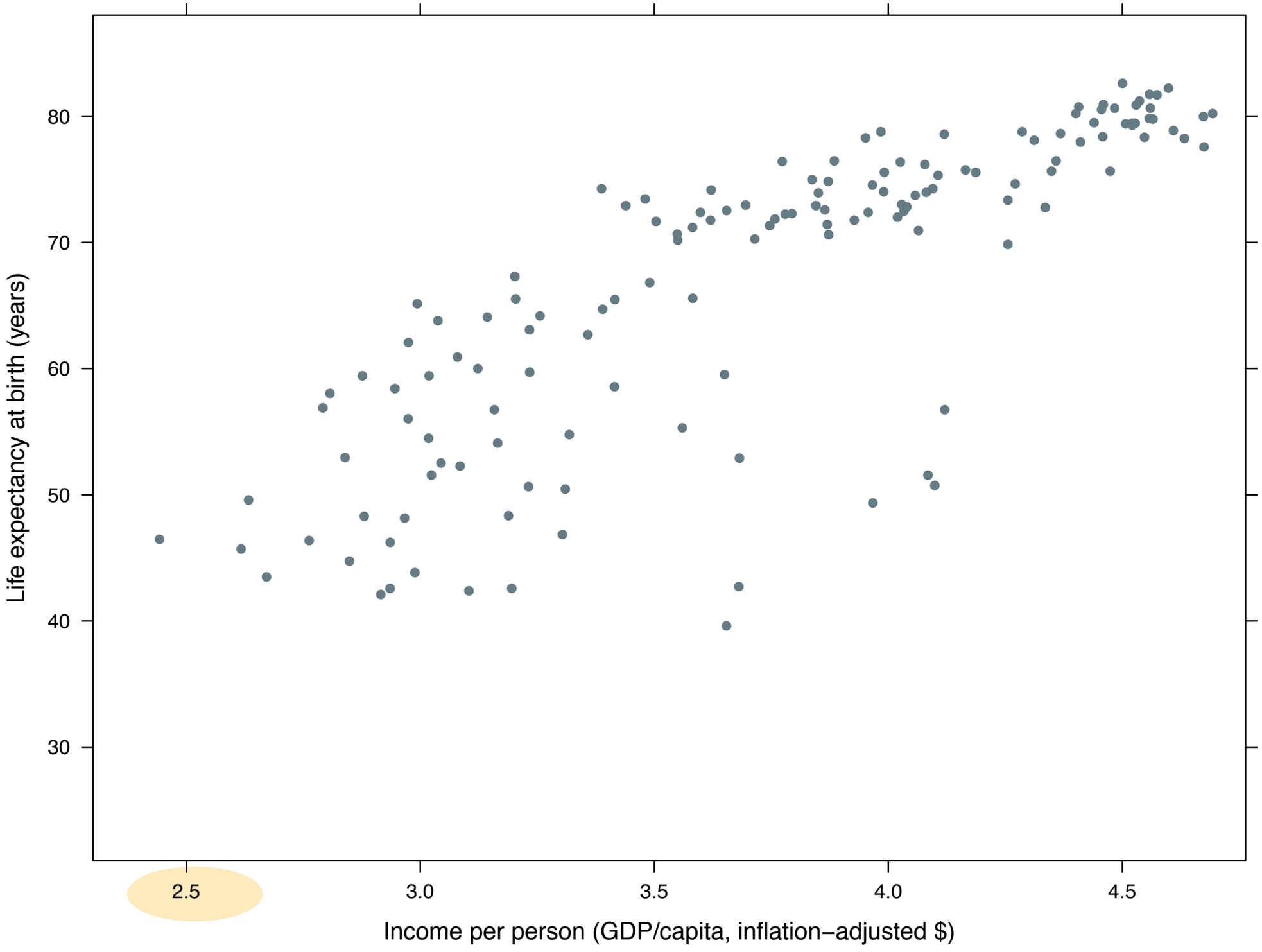



```
jXlim <- c(200, 58000)
jYlim <- c(21, 88)
xyplot(lifeExp ~ gdpPercap, gDatOrdered, subset = year == jYear,
       xlab = jXlab, ylab = jYlab,
       xlim = jXlim, ylim = jYlim)
```



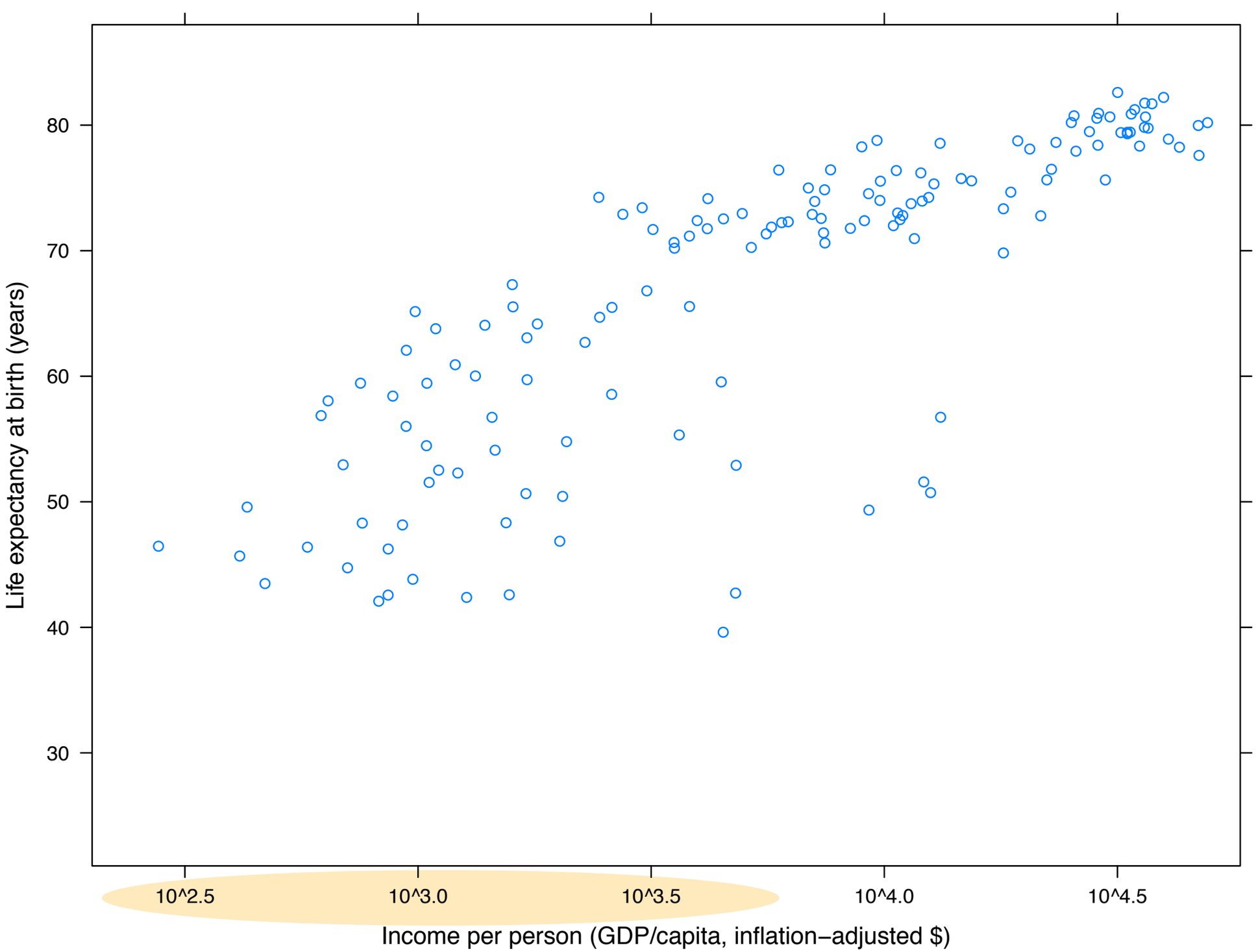

```
xyplot(lifeExp ~ log10(gdpPercap), gDatOrdered, subset = year == jYear,
       xlab = jXlab, ylab = jYlab,
       xlim = log10(jXlim), ylim = jYlim))
```

It's still a bad idea to log 'by hand'.



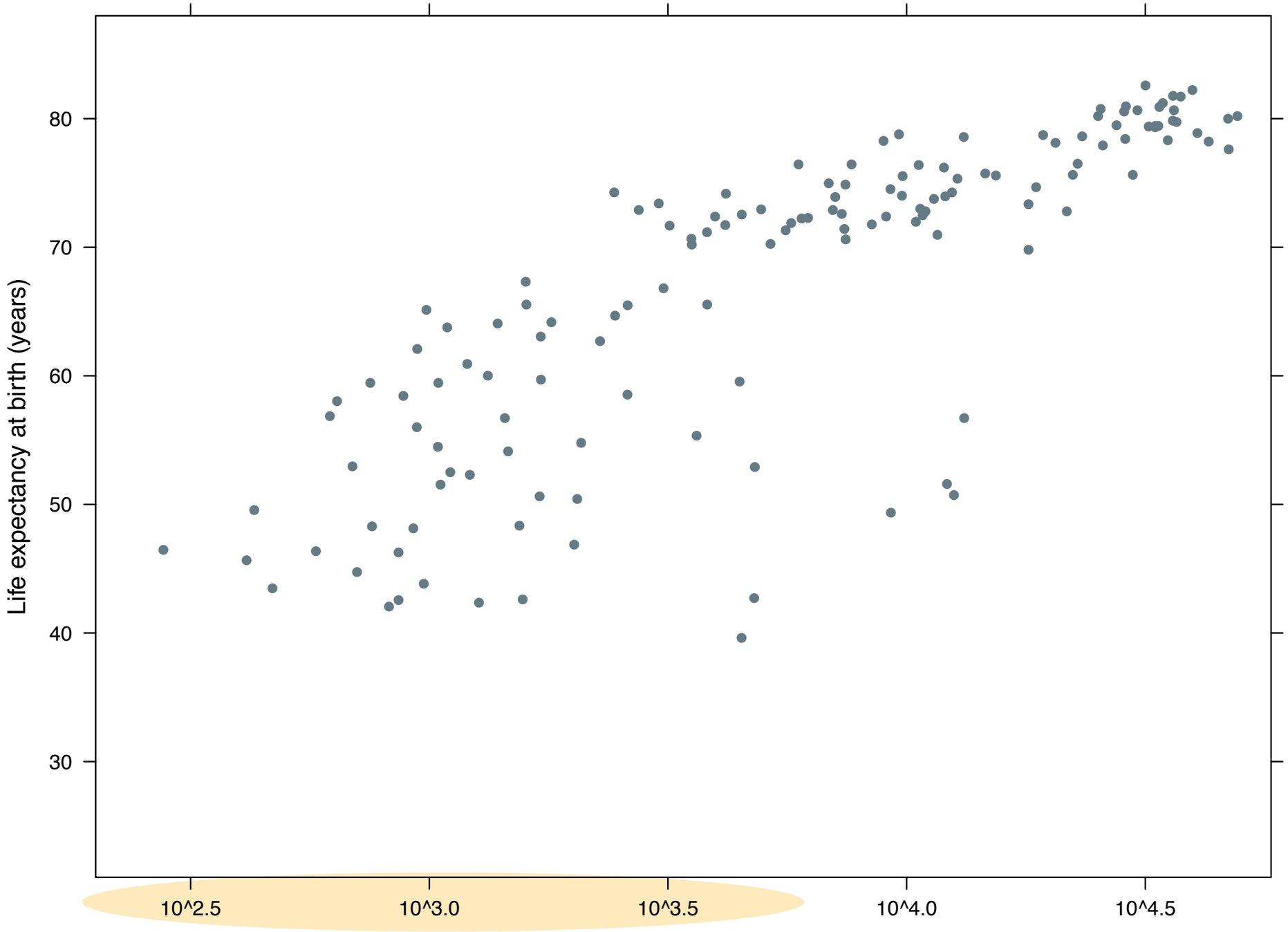

```
xyplot(lifeExp ~ log10(gdpPercap), gDatOrdered, subset = year == jYear,
       xlab = jXlab, ylab = jYlab,
       xlim = log10(jXlim), ylim = jYlim))
```

It's still a bad idea to log 'by hand'.



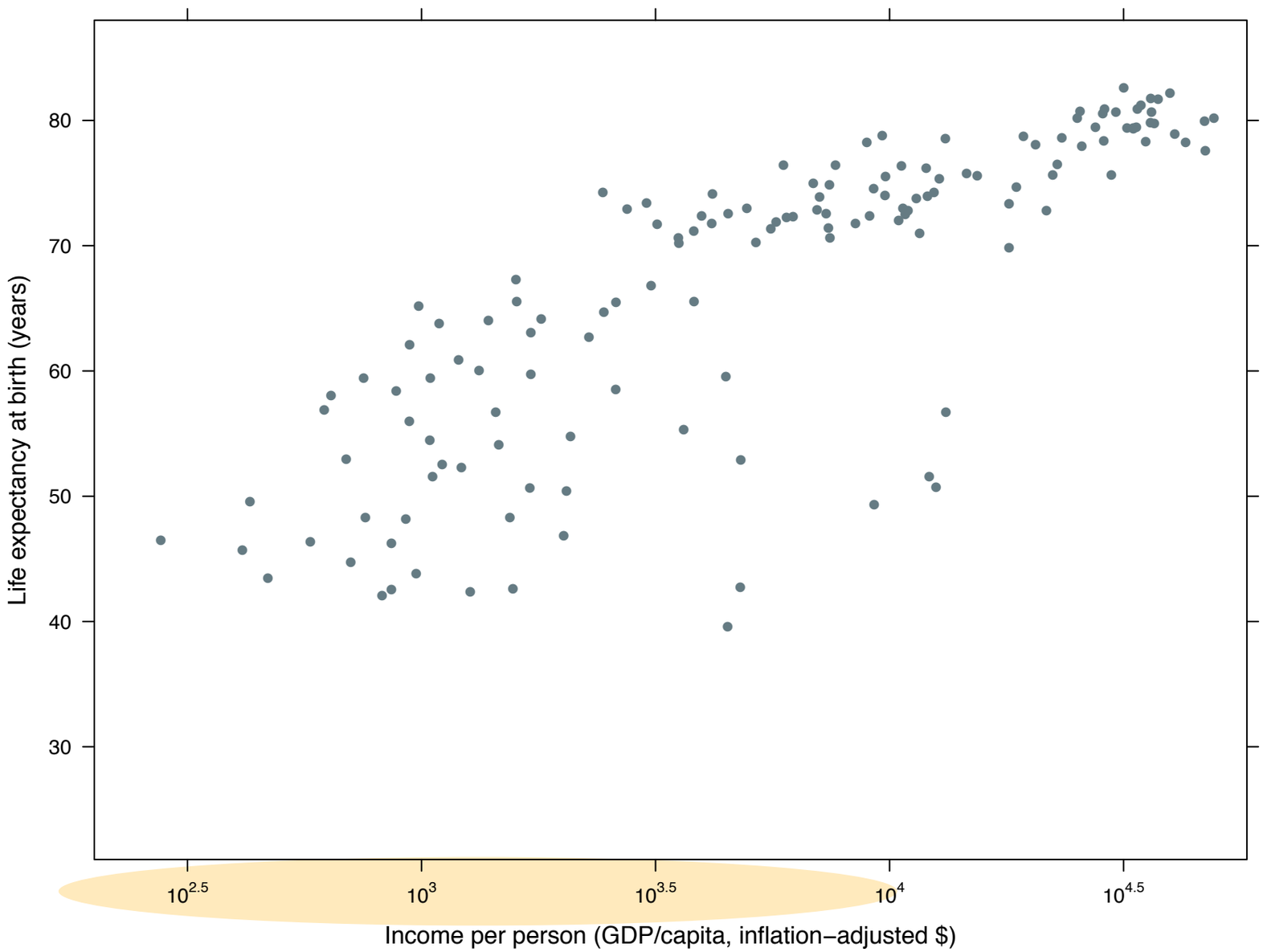

```
xyplot(lifeExp ~ gdpPercap, gDatOrdered, subset = year == jYear,
       xlab = jXlab, ylab = jYlab,
       xlim = jXlim, ylim = jYlim,
       scales = list(x = list(log = 10)))
```

Use the scales argument to log transform an axis (among many many other things).



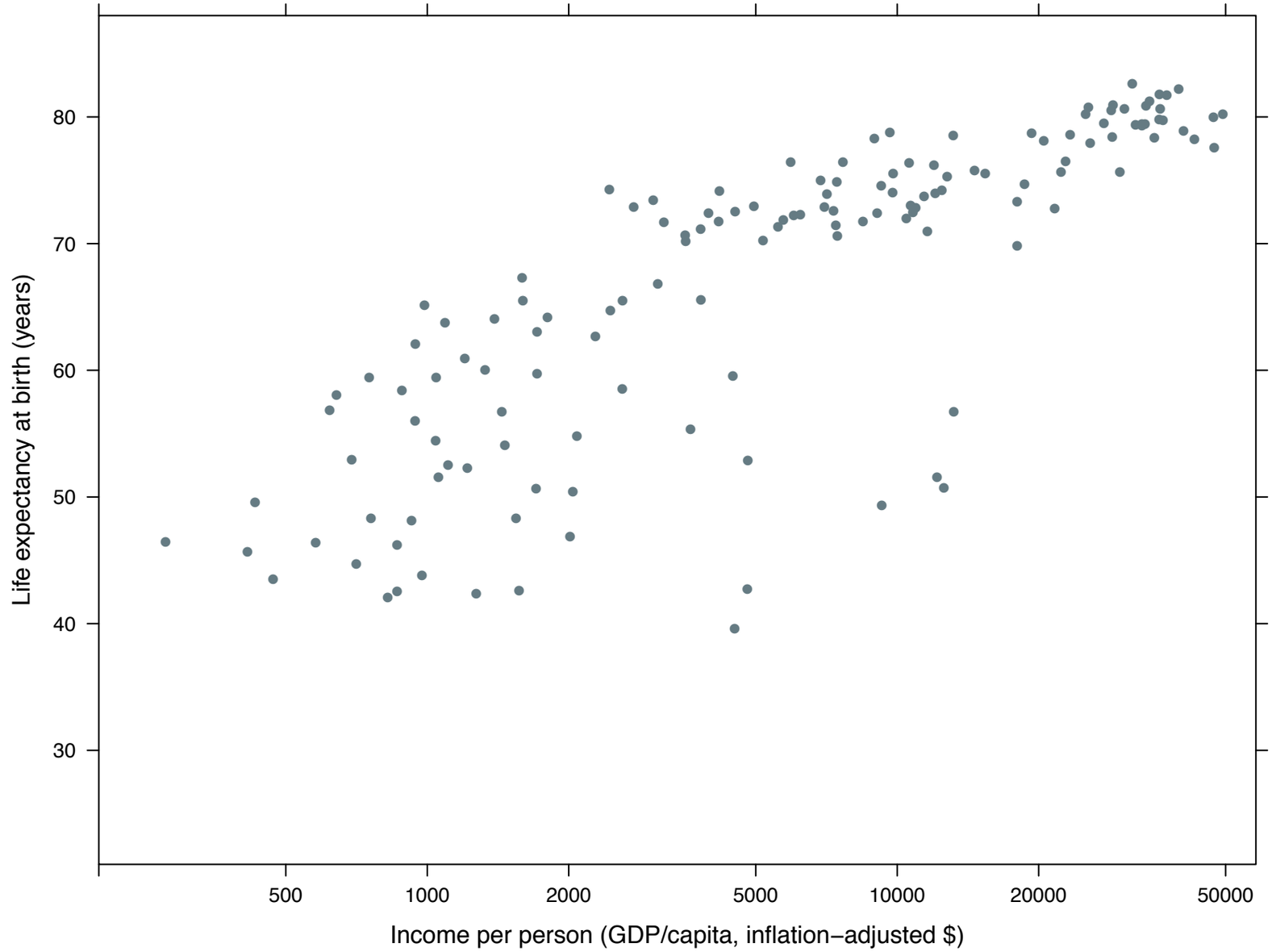

```
xyplot(lifeExp ~ gdpPercap, gDatOrdered, subset = year == jYear,
       xlab = jXlab, ylab = jYlab,
       xlim = jXlim, ylim = jYlim,
       scales = list(x = list(log = 10)))
```

Use the scales argument to log transform an axis (among many many other things).



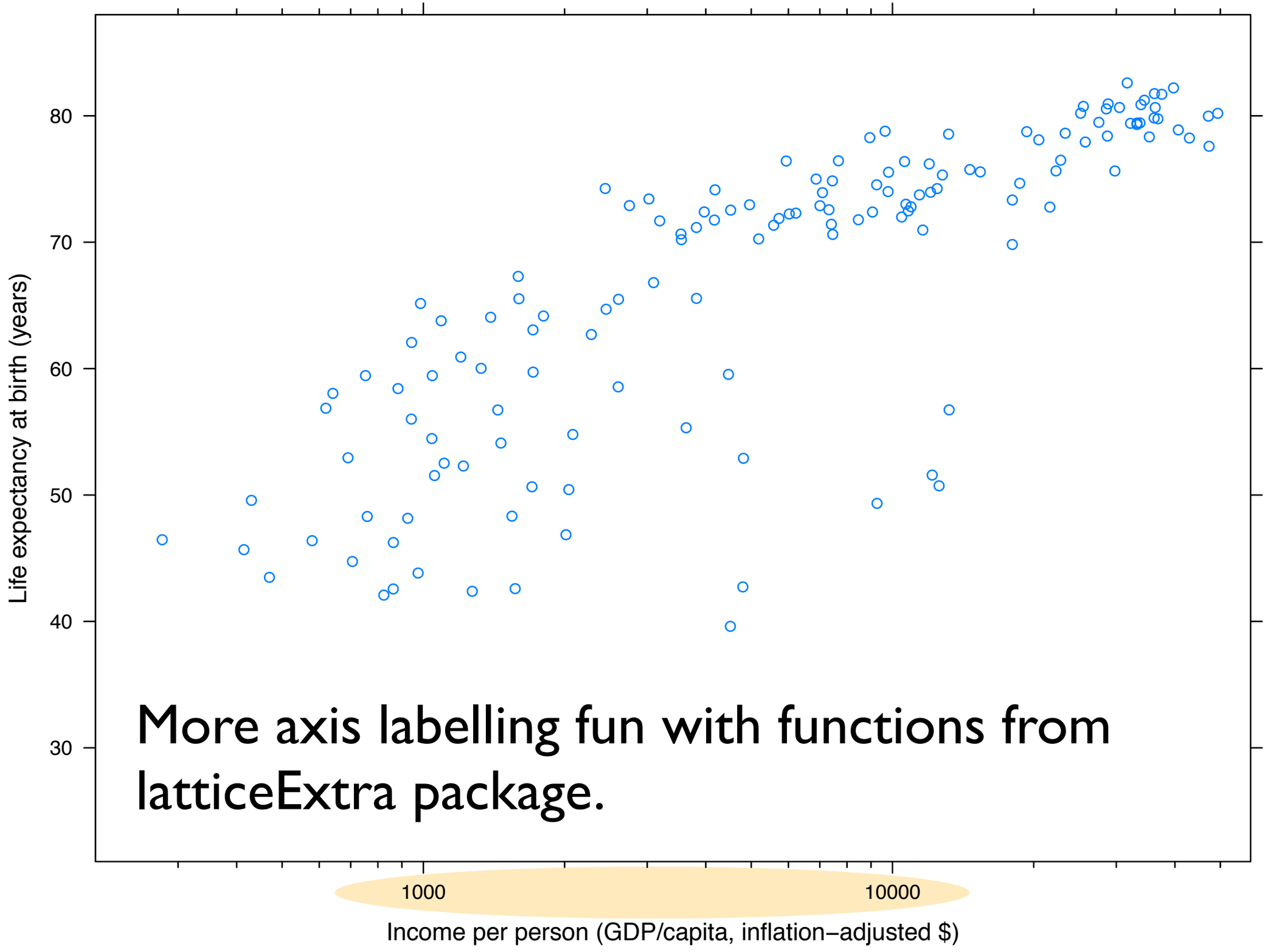

```
xyplot(lifeExp ~ gdpPercap, gDatOrdered, subset = year == jYear,
       xlab = jXlab, ylab = jYlab,
       scales = list(x = list(log = 10)),
       xlim = jXlim, ylim = jYlim,
       xscale.components = xscale.components.logpower)
```

latticeExtra package offers facilities for better log axis labelling; here an improved axis drawing function, xscale.components.logpower.



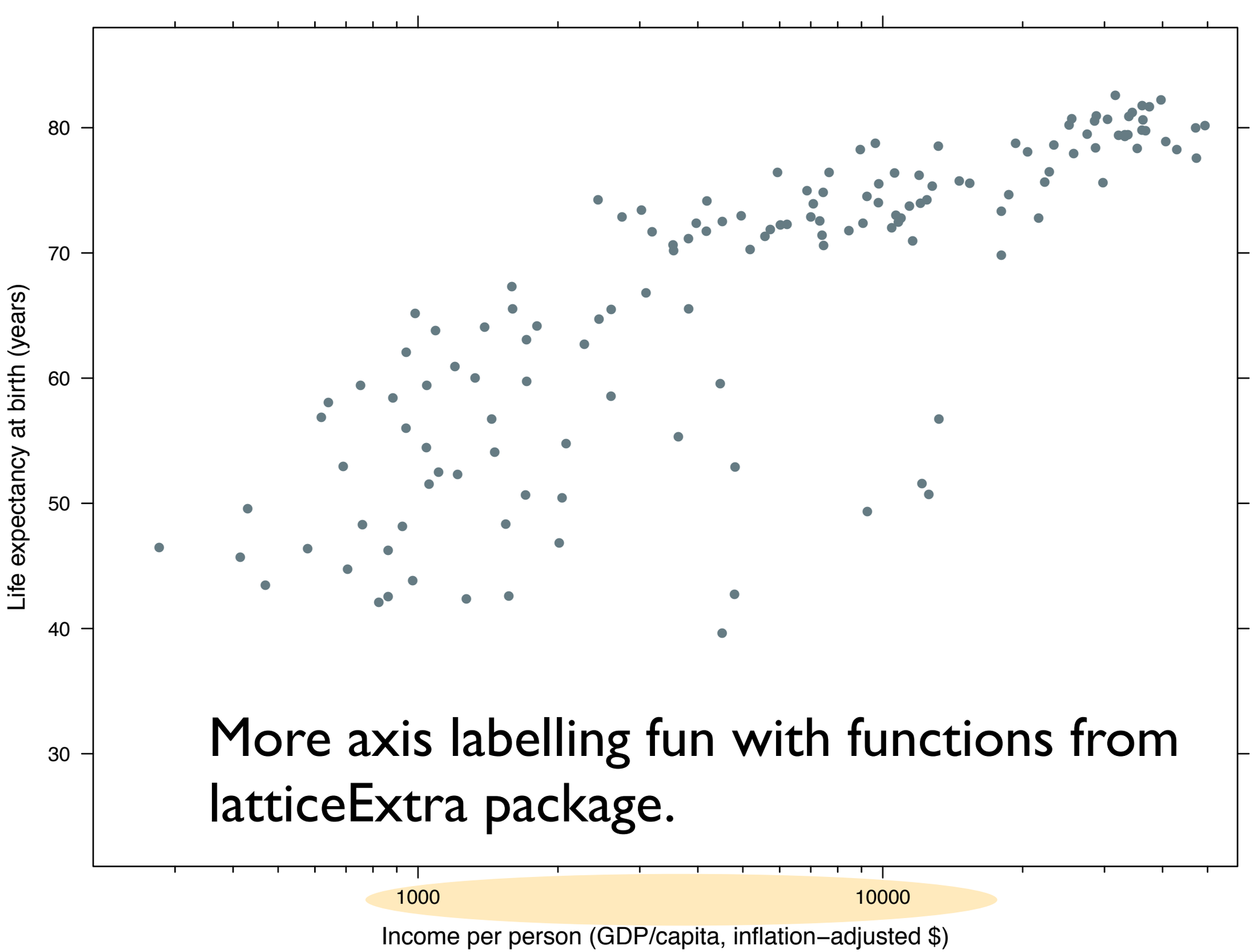

```
xyplot(lifeExp ~ gdpPercap, gDatOrdered, subset = year == jYear,
       xlab = jXlab, ylab = jYlab,
       xlim = jXlim, ylim = jYlim,
       scales = list(x = list(log = 10), equispaced.log = FALSE))
```

New argument for scales emulates log axis tick marking in traditional graphics.



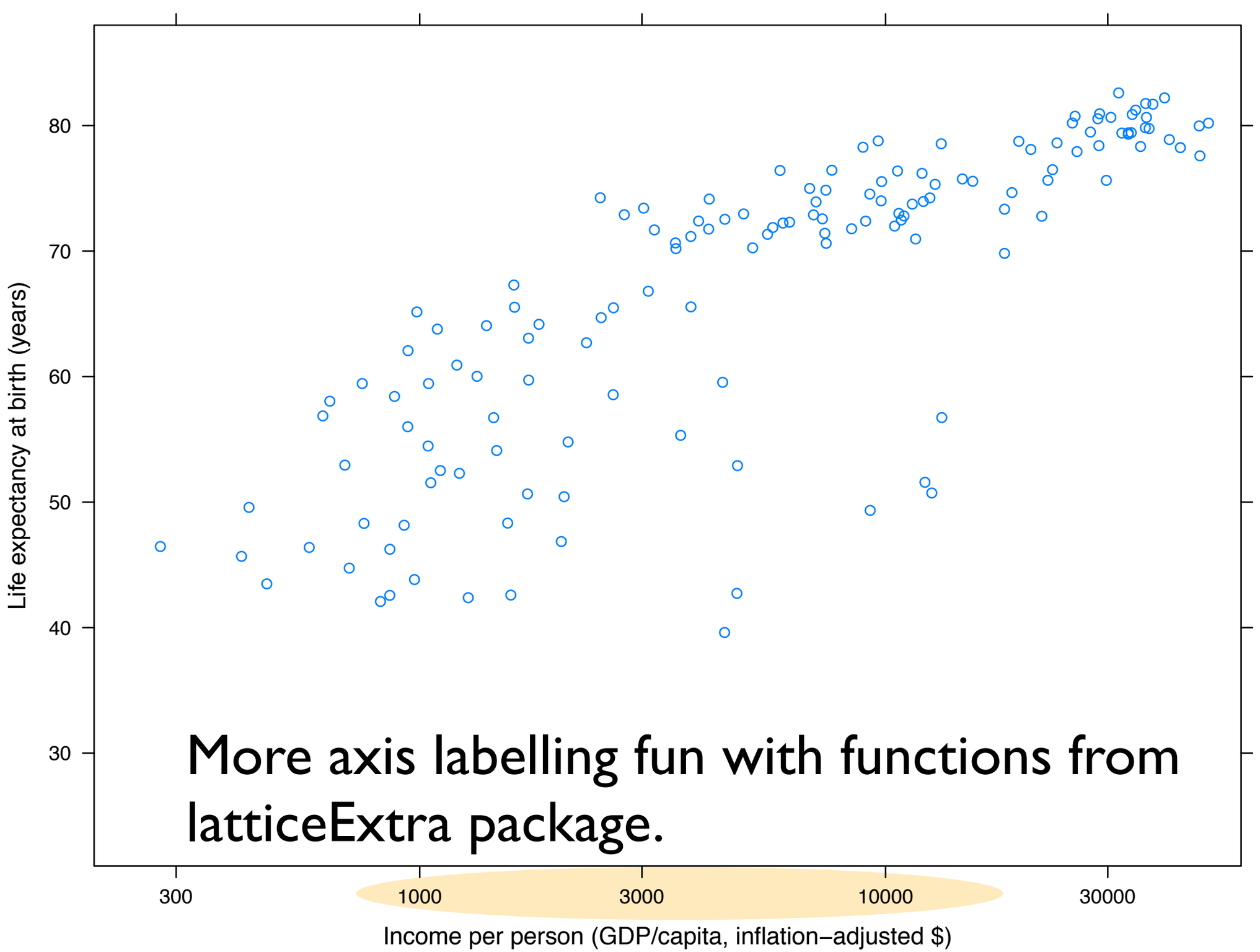
More axis labelling fun with functions from latticeExtra package.

```
xyplot(lifeExp ~ gdpPercap, <snip, snip>,
       xscale.components = xscale.components.log10ticks)
```

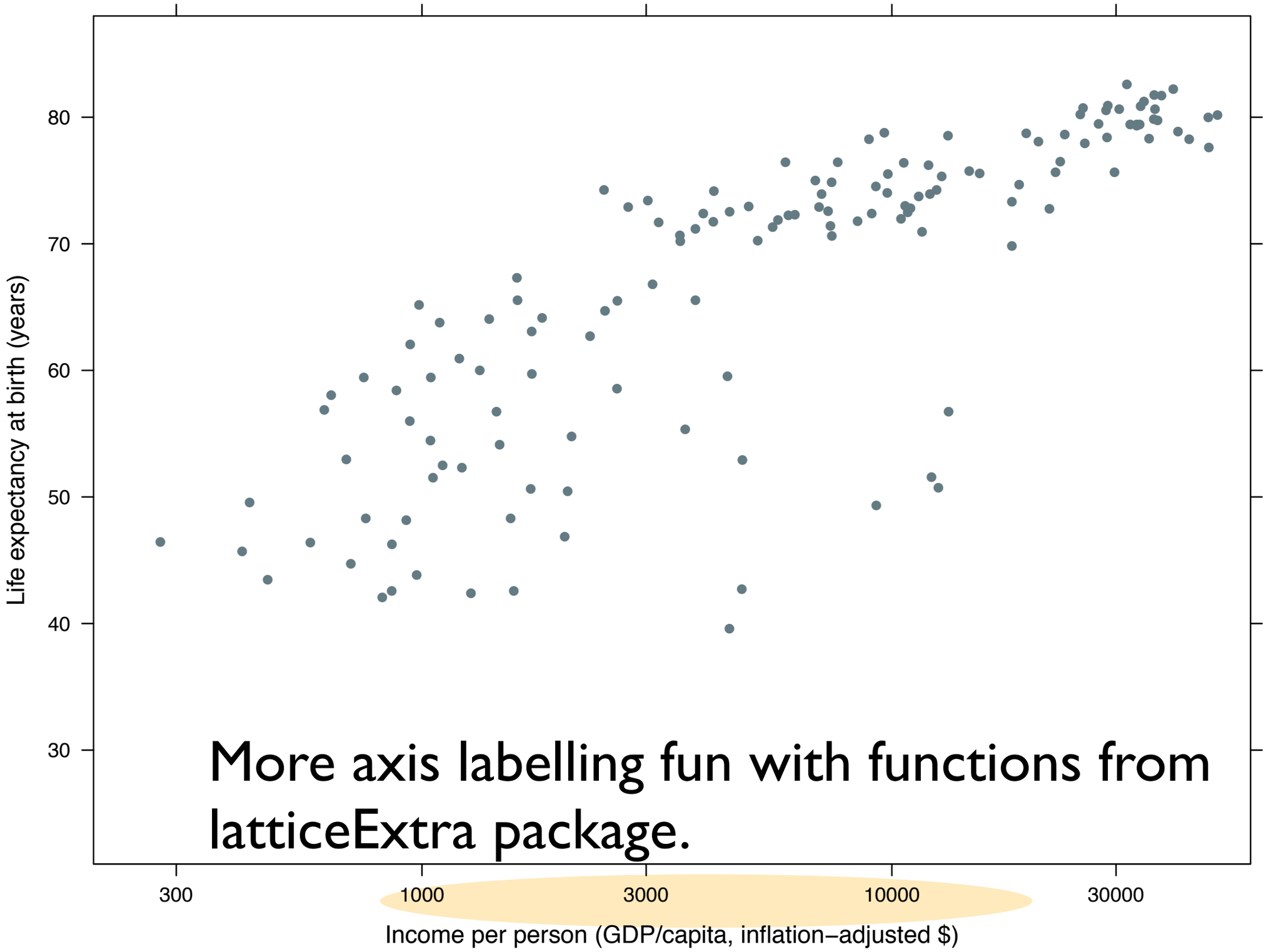


More axis labelling fun with functions from latticeExtra package.

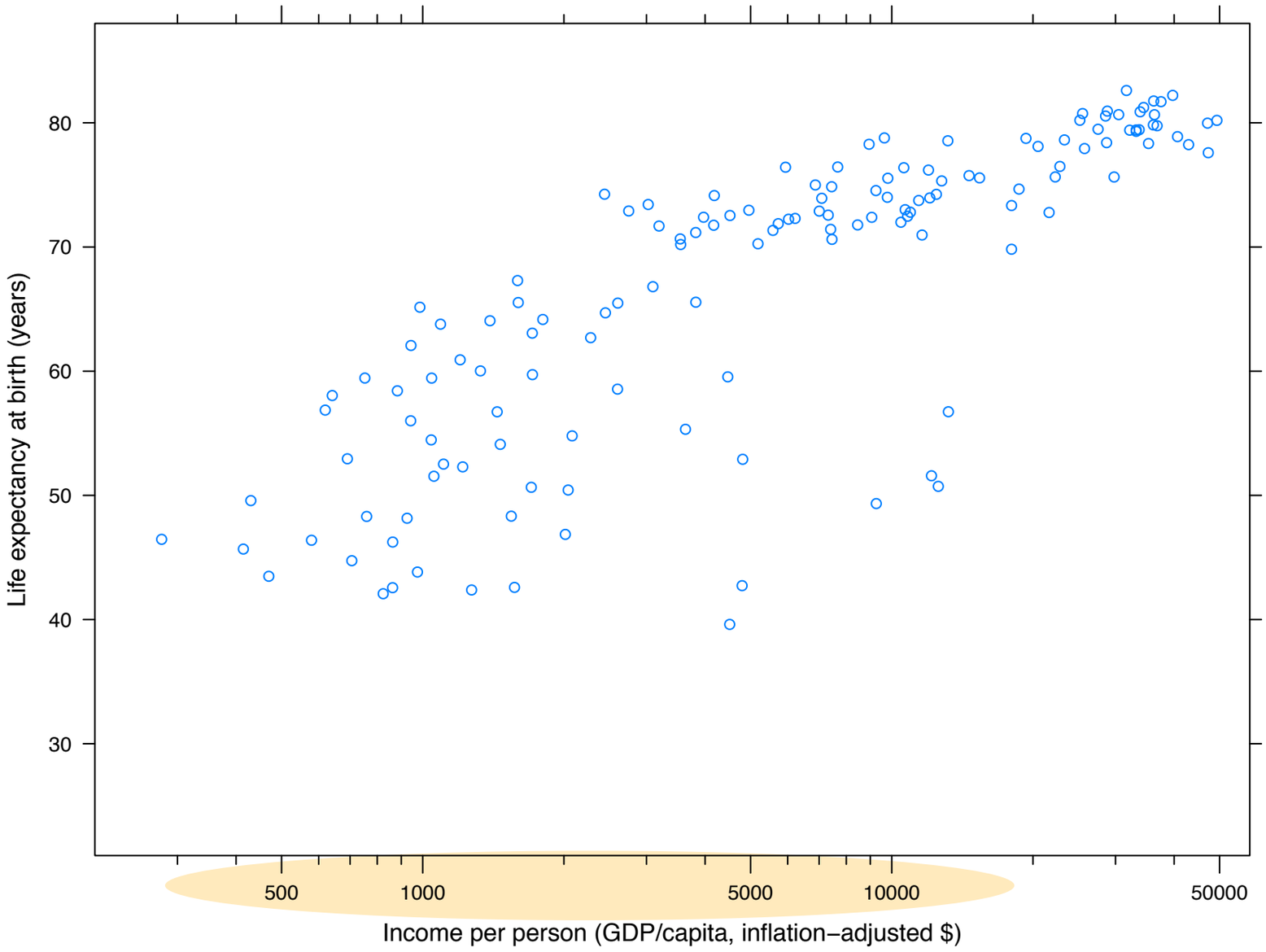
```
xyplot(lifeExp ~ gdpPercap, <snip, snip>,
       xscale.components = xscale.components.log10ticks)
```




```
xyplot(lifeExp ~ gdpPercap, <snip, snip>,
            xscale.components = xscale.components.log10.3)
```

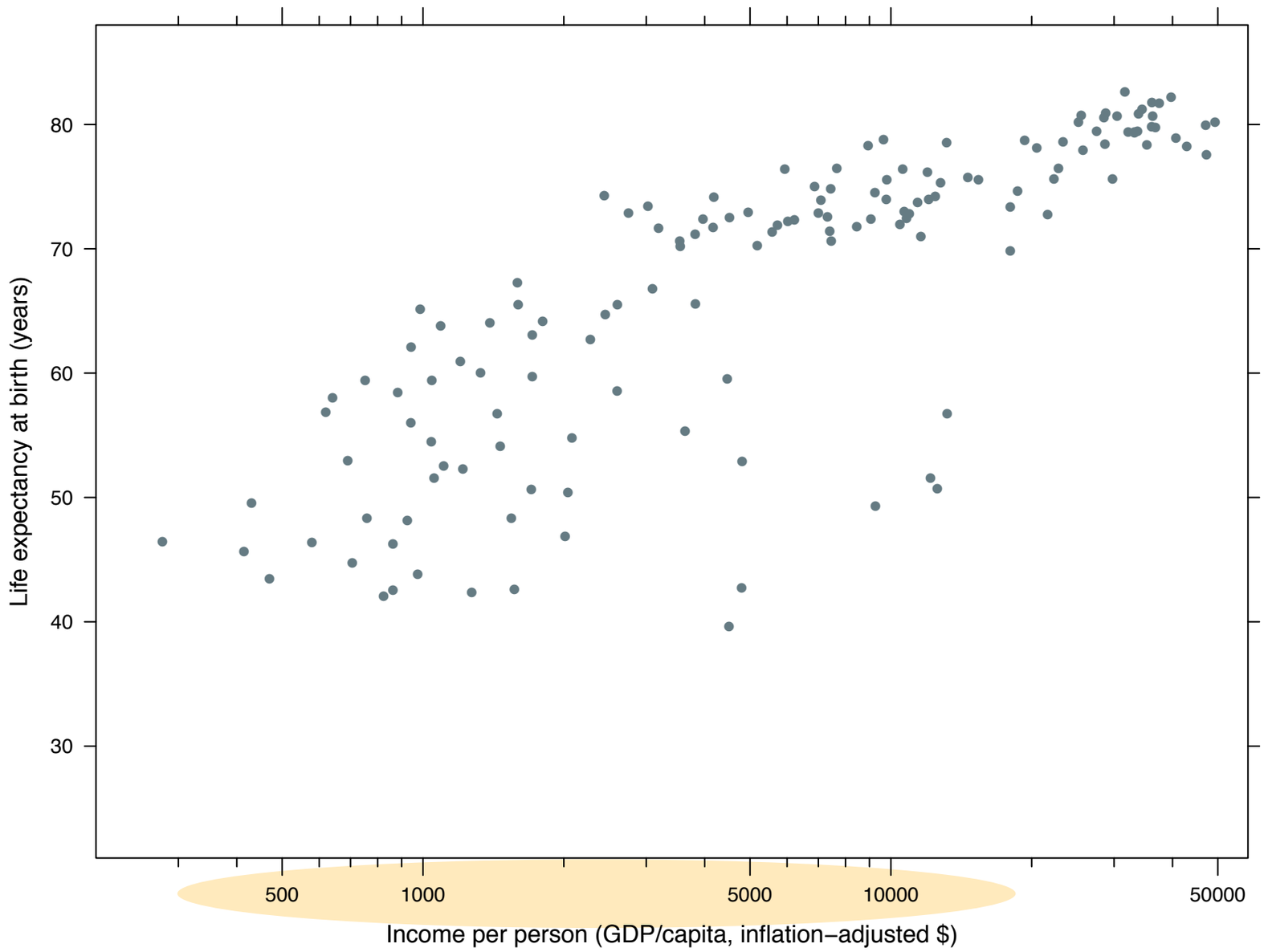



```
xyplot(lifeExp ~ gdpPercap, <snip, snip>,
            xscale.components = xscale.components.log10.3)
```



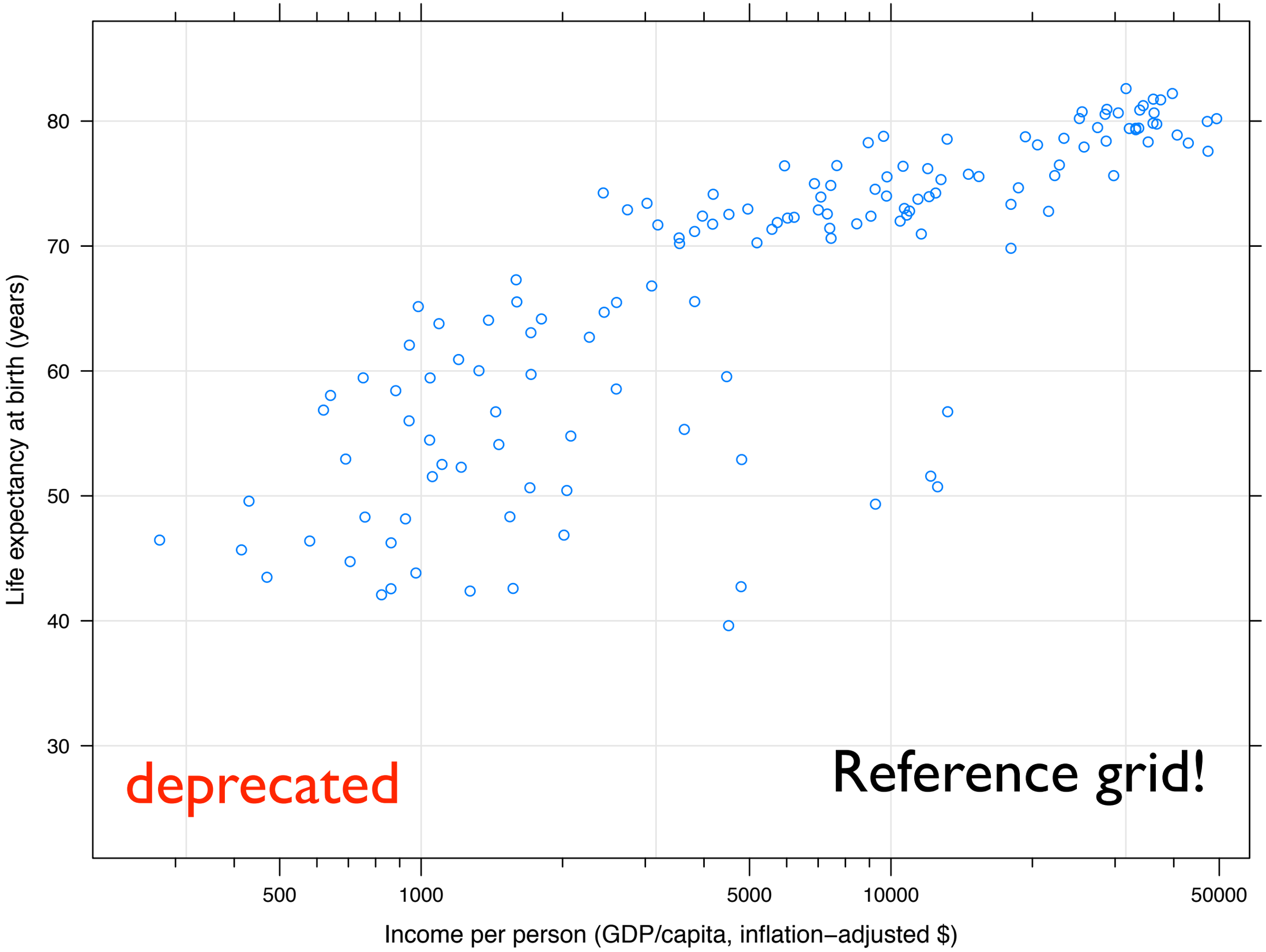

```
xyplot(lifeExp ~ gdpPercap, gDatOrdered, subset = year == jYear,
       xlab = jXlab, ylab = jYlab,
       scales = list(x = list(log = 10)),
       xlim = jXlim, ylim = jYlim,
       xscale.components = xscale.components.log10)
```

For even more control, can write a custom function for this. Kind of advanced / obscure. See the code.

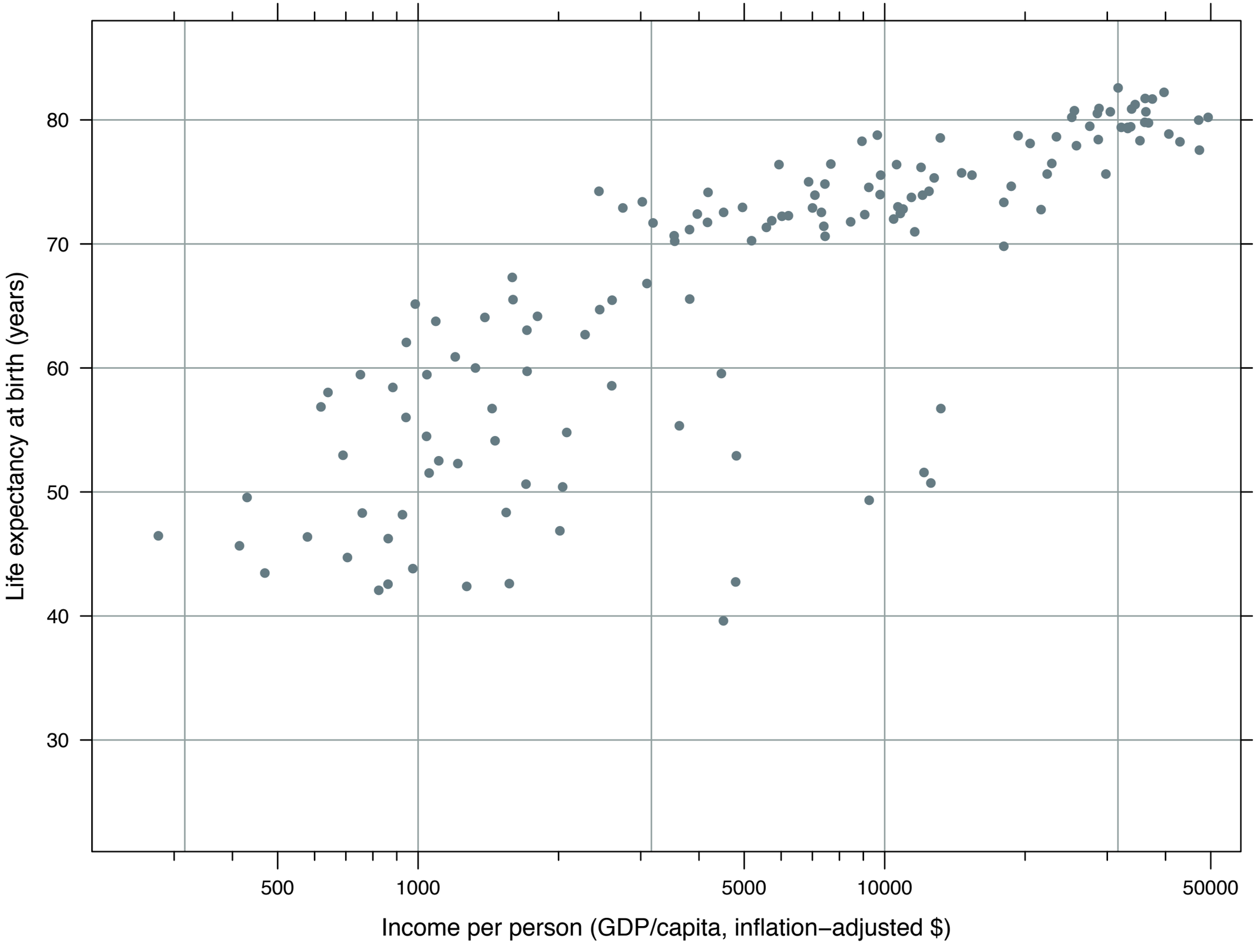



```
xyplot(lifeExp ~ gdpPercap, gDatOrdered, subset = year == jYear,
       xlab = jXlab, ylab = jYlab,
       scales = list(x = list(log = 10)),
       xlim = jXlim, ylim = jYlim,
       xscale.components = xscale.components.log10)
```

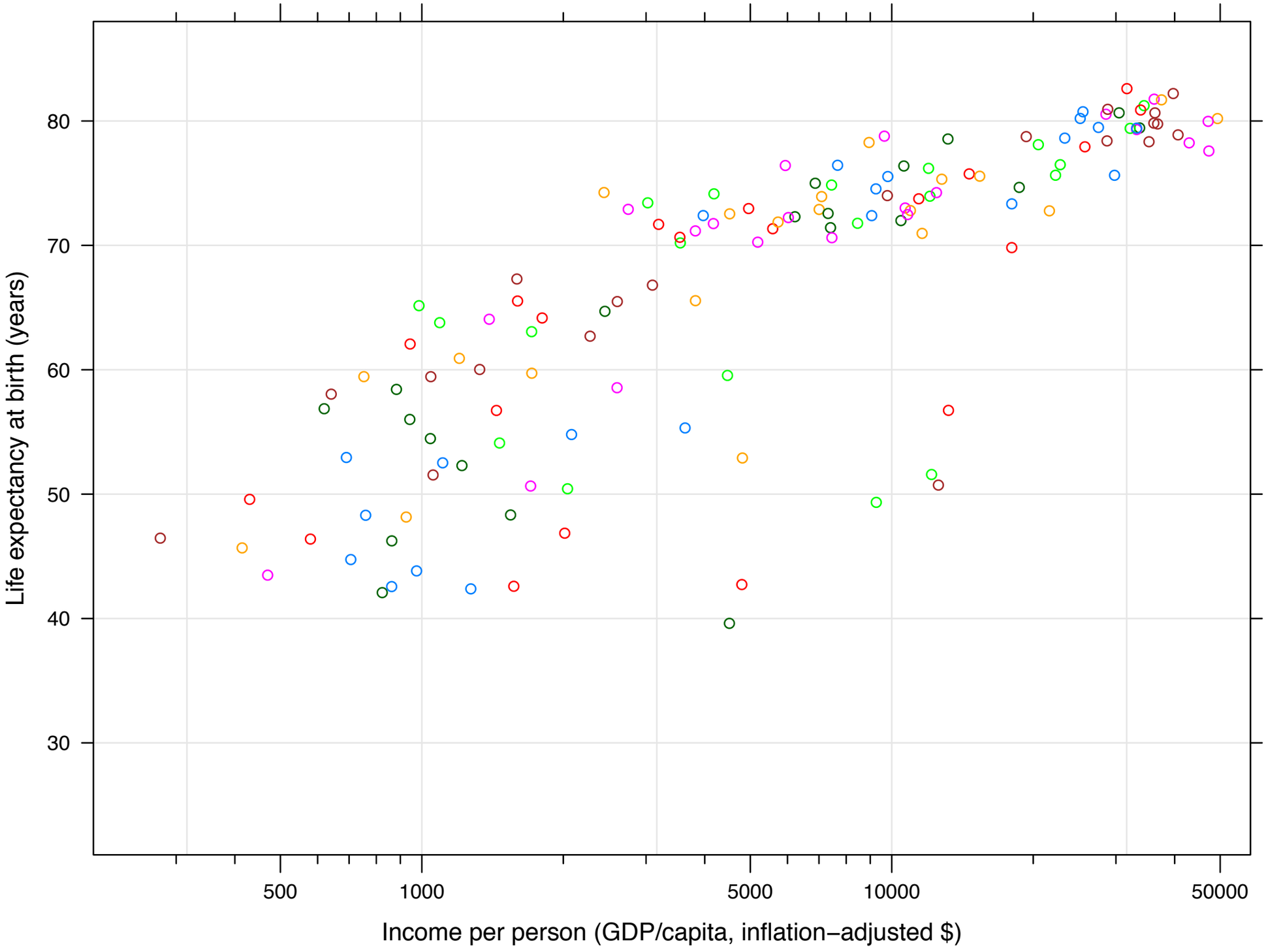
For even more control, can write a custom function for this. Kind of advanced / obscure. See the code.



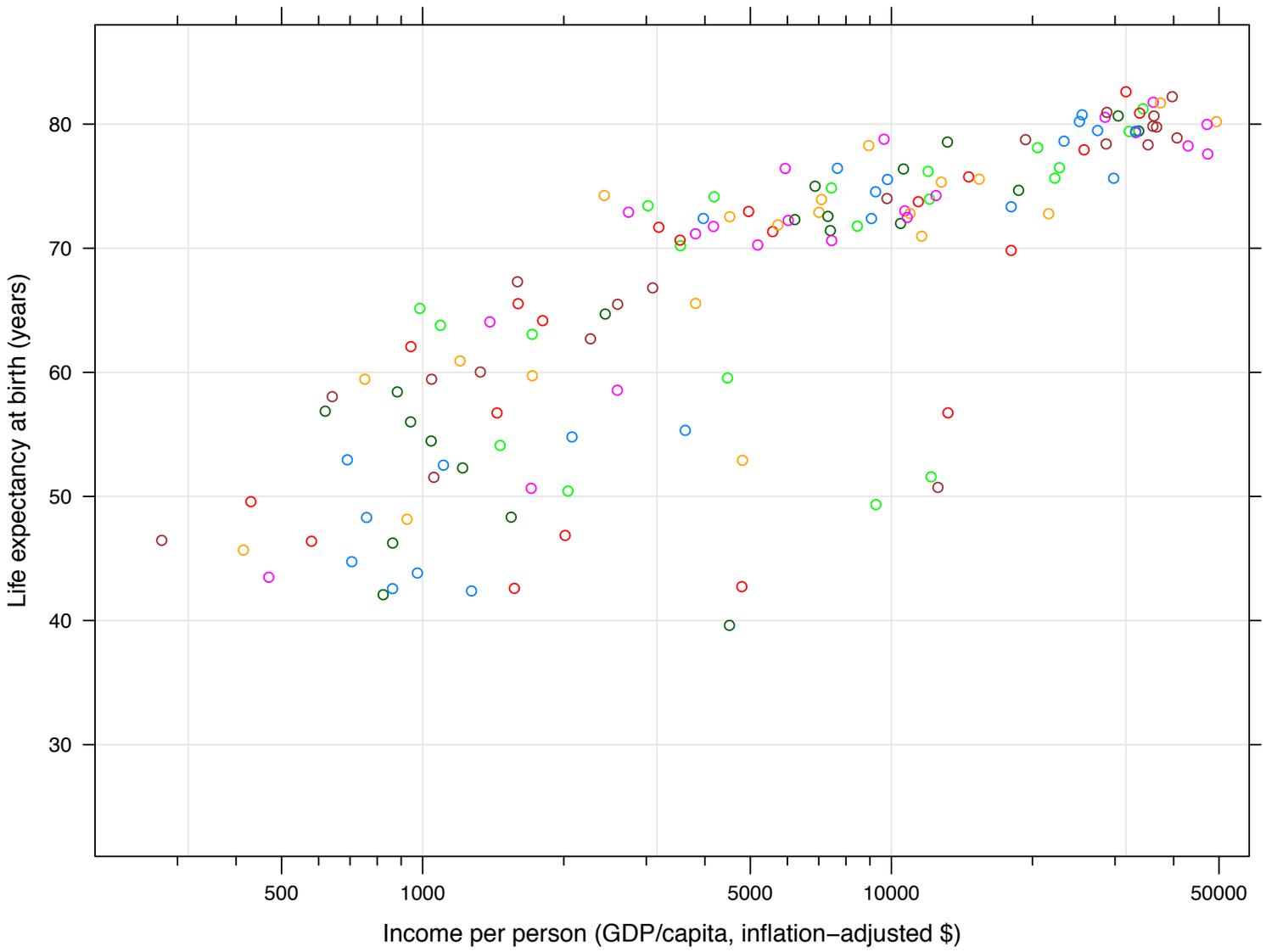

```
xyplot(lifeExp ~ gdpPercap, <snip, snip>,
       type = c("p", "g"))
```




```
xyplot(lifeExp ~ gdpPercap, <snip, snip>,
       grid = TRUE)
```

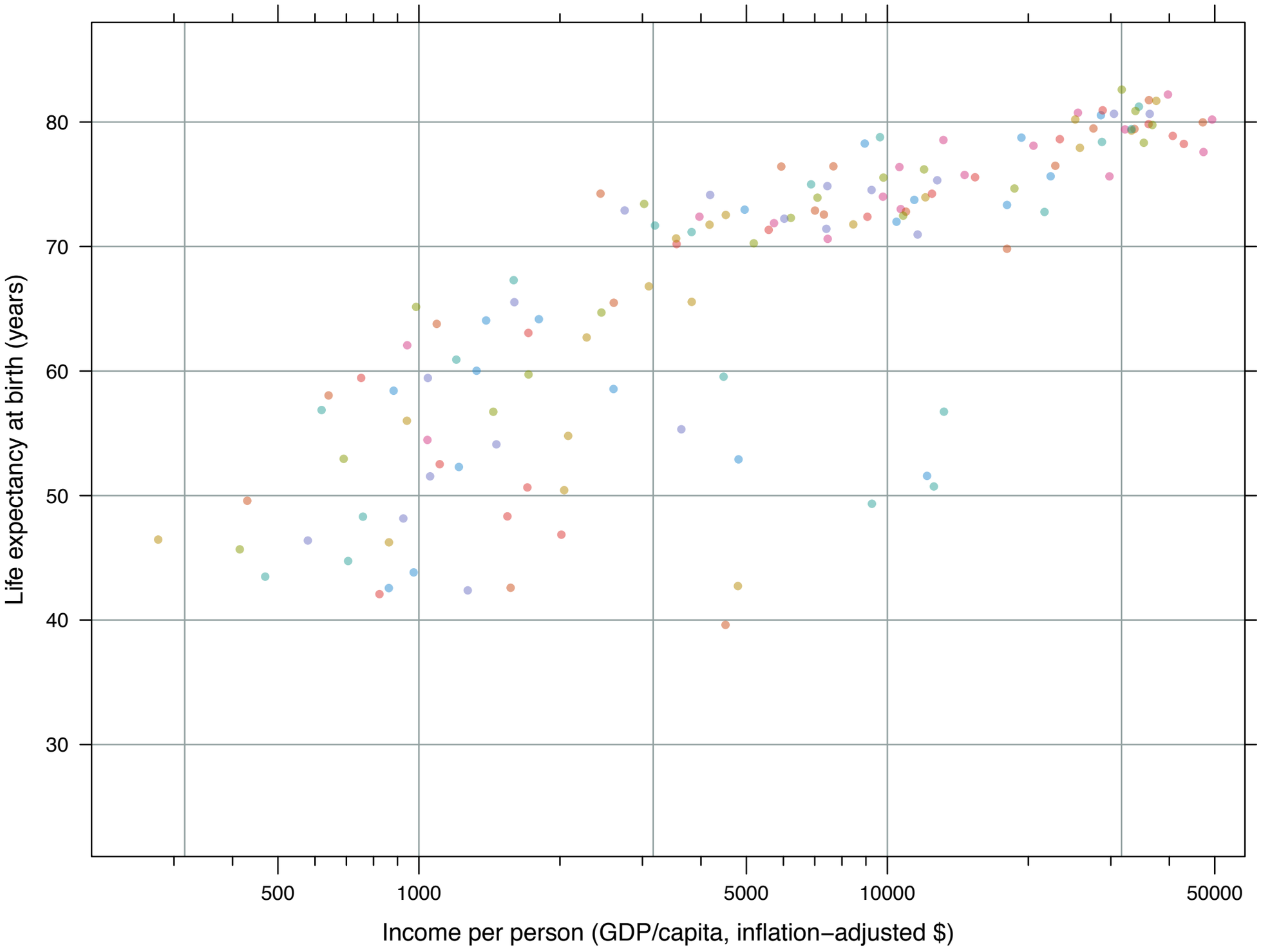



```
xyplot(lifeExp ~ gdpPercap, ..., group = country)
```

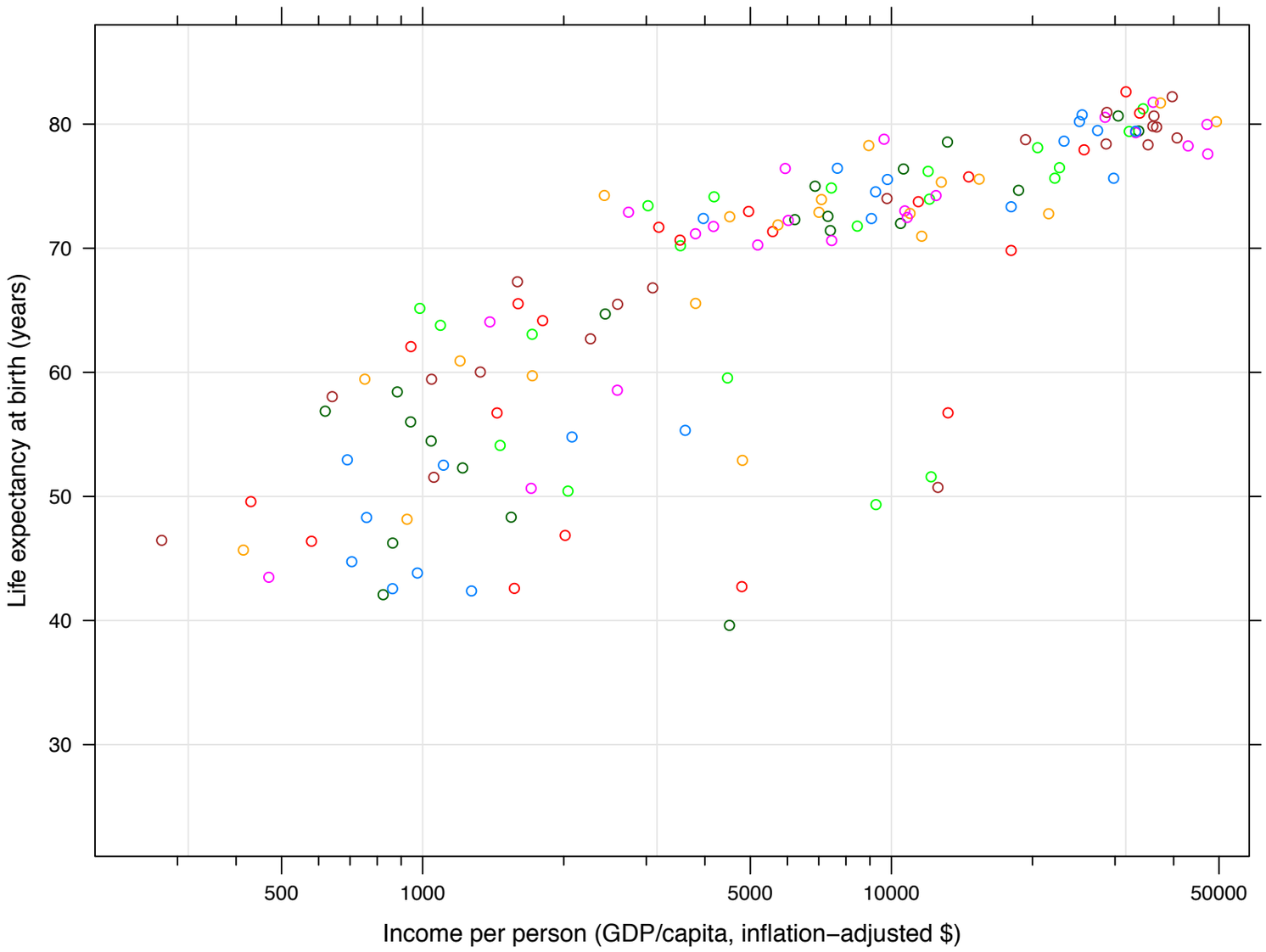



```
xyplot(lifeExp ~ gdpPercap, ..., group = country)
```

The group argument specifies a factor to be "superposed" in the plot, generally via color. Each level of the factor will be visually distinguished.

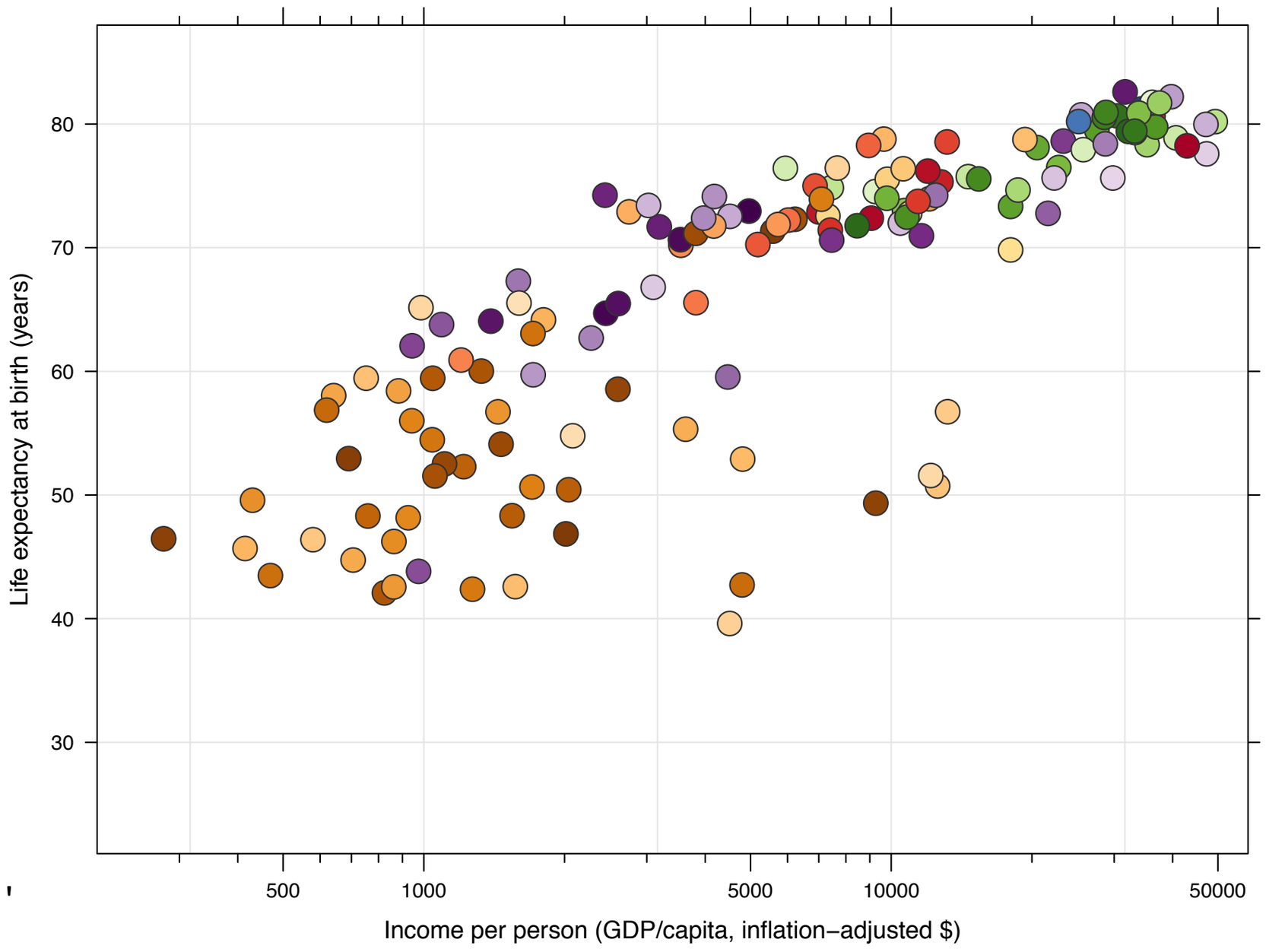



```
xyplot(lifeExp ~ gdpPercap, ..., group = country)
```




```
xyplot(lifeExp ~ gdpPercap, ..., group = country)
```

Here, we have two problems to solve: the default color palette is ugly and it is not large enough for our country factor.




```
jDarkGray <- 'grey20'
jPch <- 21
jGapminderPars <-
  list(superpose.symbol = list(pch = jPch, col = jDarkGray, cex = 2,
         fill = countryColors$color[match(levels(gDatOrdered$country),
           countryColors$country)]))

xyplot(lifeExp ~ gdpPercap, ...
       group = country,
       par.settings = jGapminderPars)
```

You can specify your own color palette for a plot.

B_00_xyplot package:lattice R Documentation

Common Bivariate Trellis Plots

Description:

This help page documents several commonly used high-level Lattice functions. 'xyplot' produces bivariate scatterplots or time-series plots.....

Usage:

xyplot(x, data, ...)

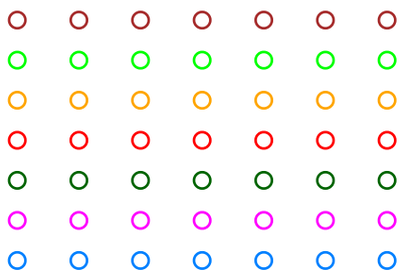
<snip, snip>

'par.settings': A list that could be supplied to 'trellis.par.set'. When the resulting object is plotted, these options are applied temporarily for the duration of the plotting, after which the settings revert back to what they were before. This enables the user to attach some display settings to the trellis object itself rather than change the settings globally. See also the 'lattice.options' argument described above for a similar treatment of non-graphical options.

Here's where one sets graphing symbols, line types, colors, alpha transparency, etc etc etc.

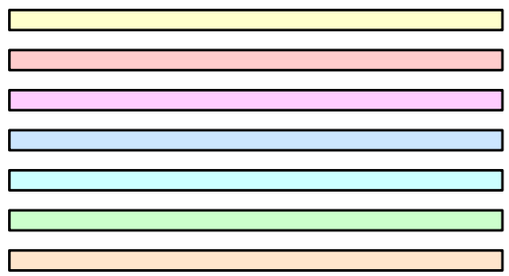
`show.settings()`

These are the settings in the default theme.

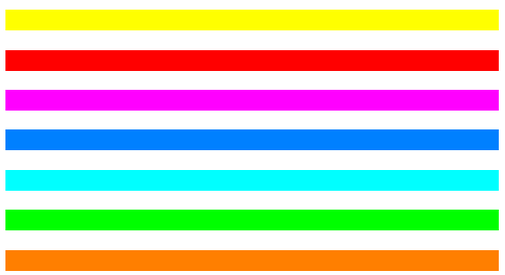


superpose.symbol

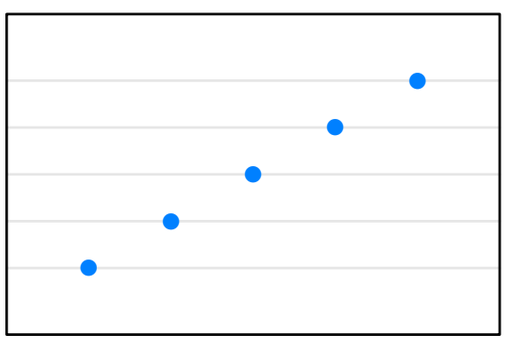
superpose.line



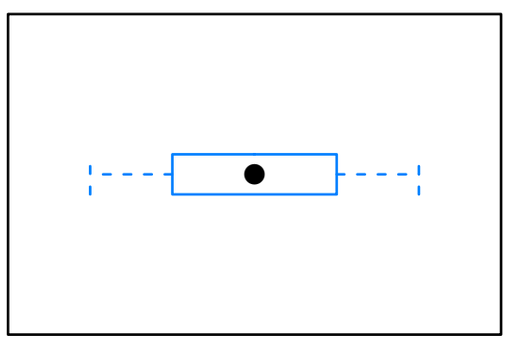
strip.background



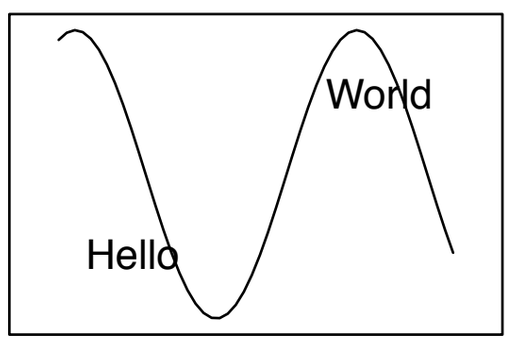
strip.shingle



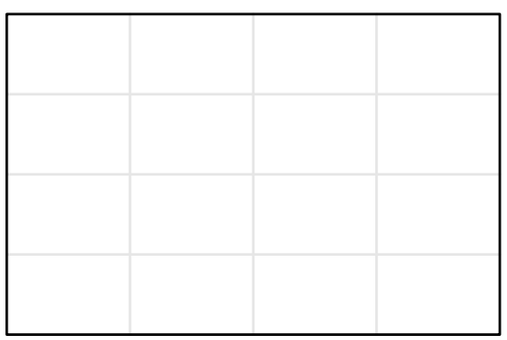
dot.[symbol, line]



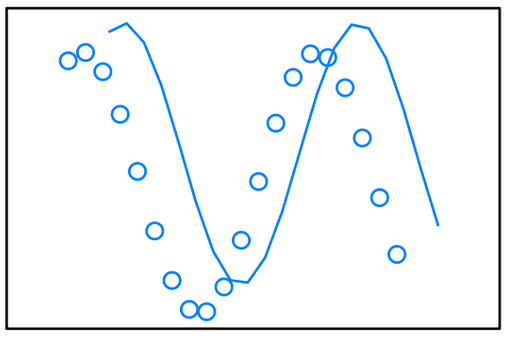
box.[dot, rectangle, umbrella]



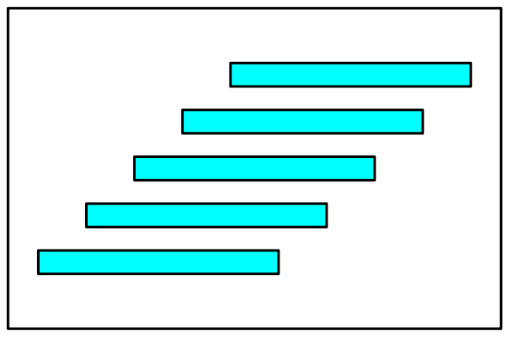
add.[line, text]



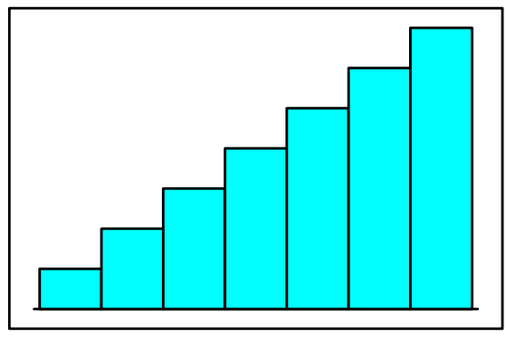
reference.line



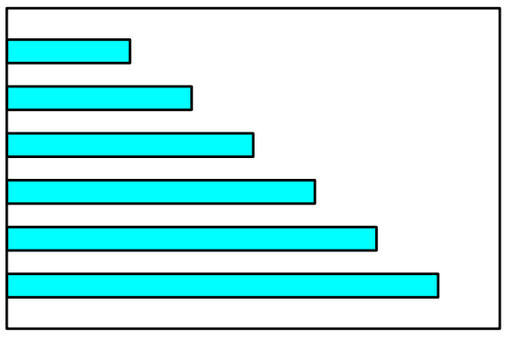
plot.[symbol, line]



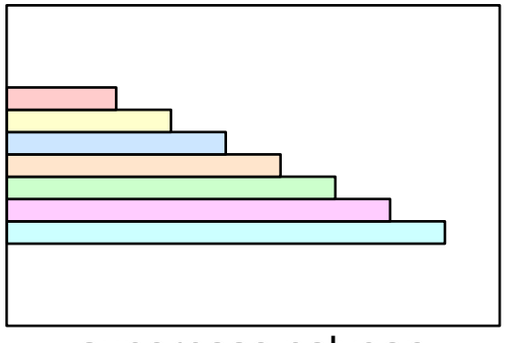
plot.shingle[plot.polygon]



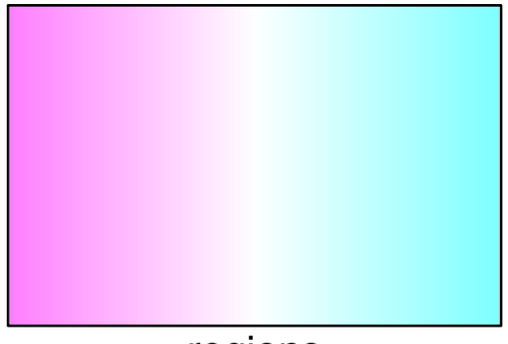
histogram[plot.polygon]



barchart[plot.polygon]



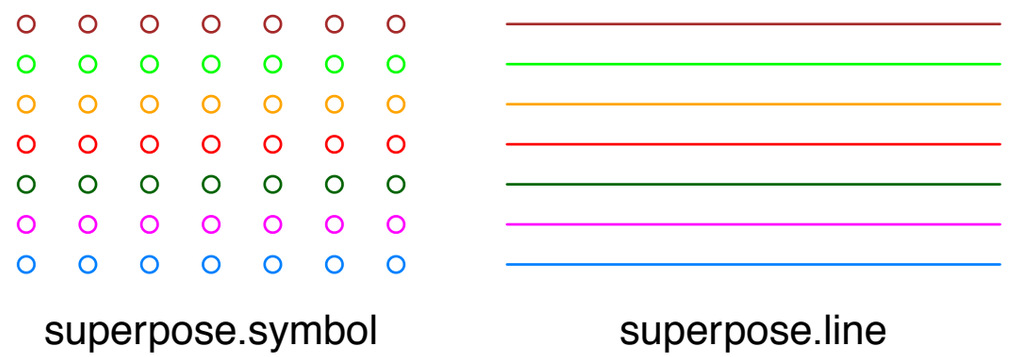
superpose.polygon

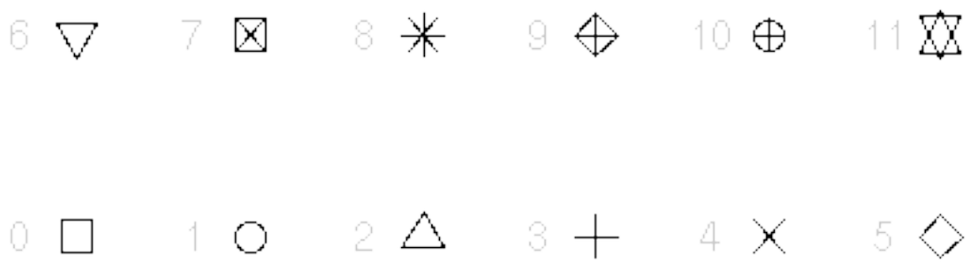


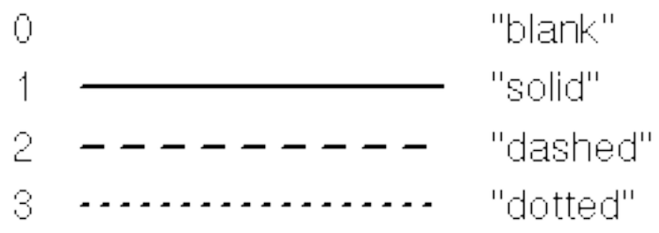
regions

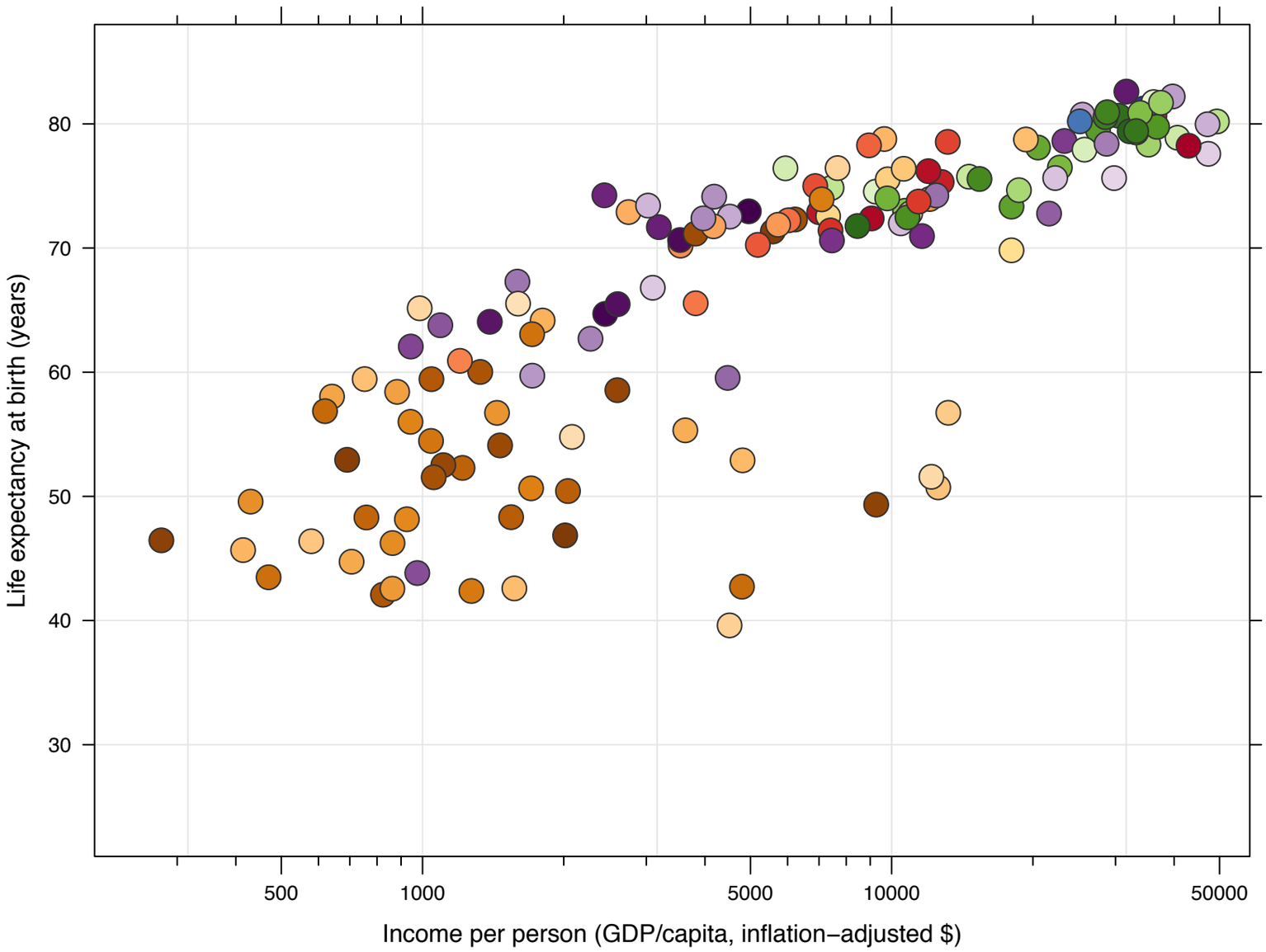
These are the settings in the default theme.

```
> trellis.par.get()

<snip, snip>

$superpose.line
$superpose.line$alpha
[1] 1

$superpose.line$col
[1] "#0080ff"   "#ff00ff"   "darkgreen" "#ff0000"   "orange"    "#00ff00"
[7] "brown"

$superpose.line$lty
[1] 1 1 1 1 1 1 1

$superpose.line$lwd
[1] 1 1 1 1 1 1 1


$superpose.symbol
$superpose.symbol$alpha
[1] 1 1 1 1 1 1 1

$superpose.symbol$cex
[1] 0.8 0.8 0.8 0.8 0.8 0.8 0.8

$superpose.symbol$col
[1] "#0080ff"   "#ff00ff"   "darkgreen" "#ff0000"   "orange"    "#00ff00"
[7] "brown"

$superpose.symbol$fill
[1] "#CCFFFF" "#FFCCFF" "#CCFFCC" "#FFE5CC" "#CCE6FF" "#FFFFCC" "#FFCCCC"

$superpose.symbol$font
[1] 1 1 1 1 1 1 1

$superpose.symbol$pch
[1] 1 1 1 1 1 1 1

<snip, snip>
```



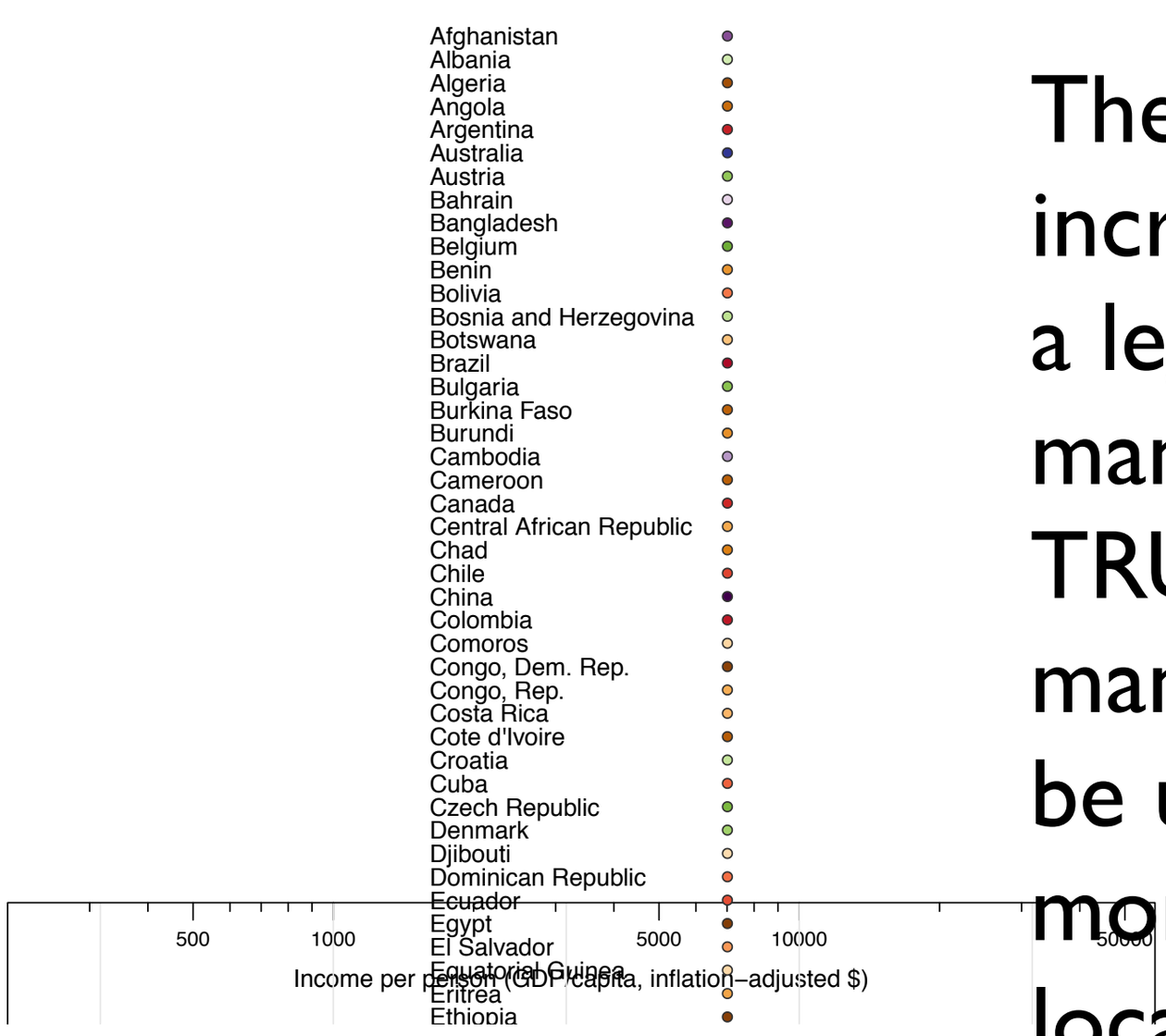






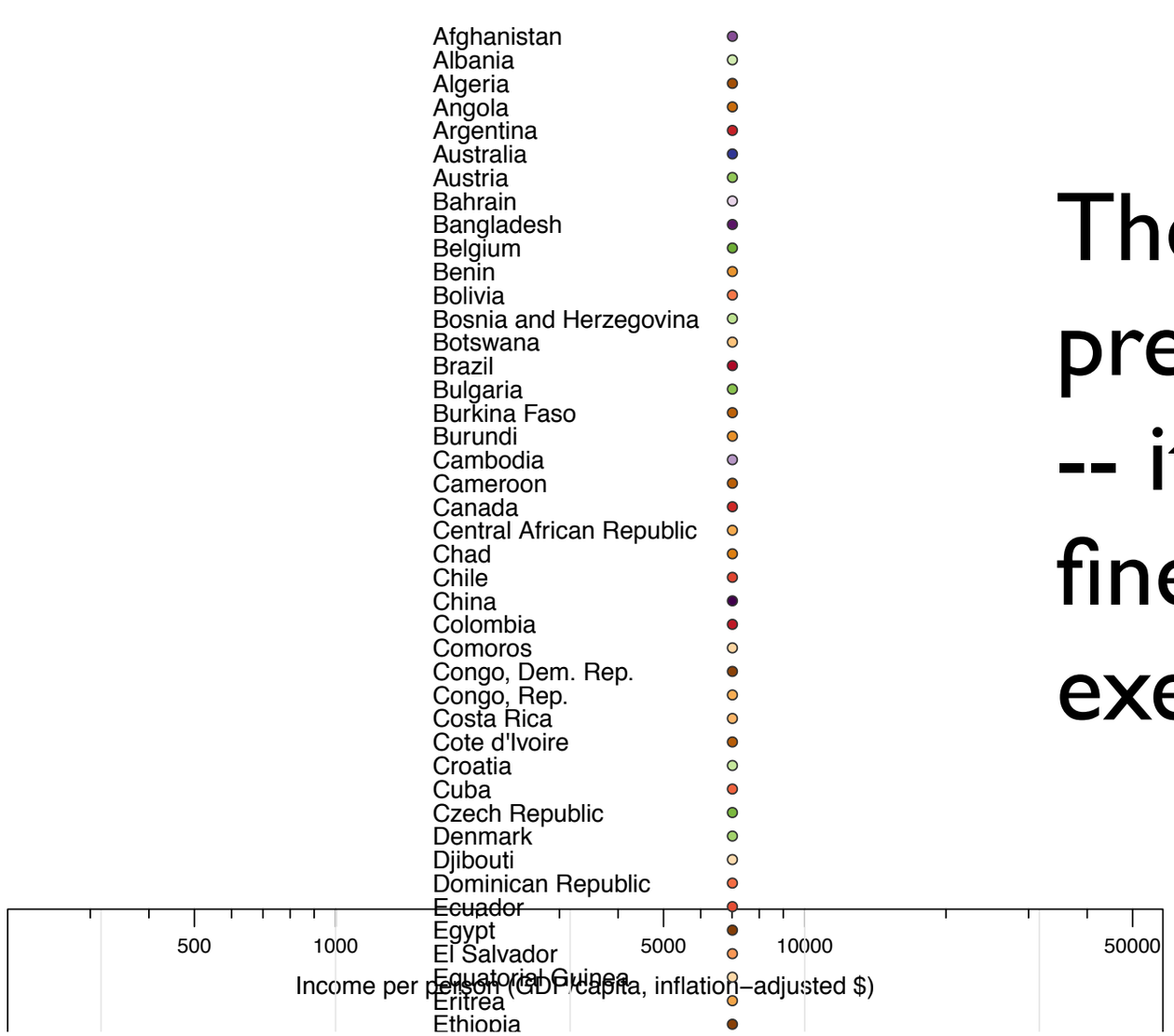

```
jGapminderPars <-
  list(superpose.symbol = list(pch = jPch, col = jDarkGray, cex = 2,
         fill = countryColors$color[match(levels(gDatOrdered$country),
           countryColors$country)]))

xyplot(lifeExp ~ gdpPercap, ...
       group = country,
       par.settings = jGapminderPars)
```

I redefined the graphics settings controlling plot symbol, fill color, and size of superposed points.



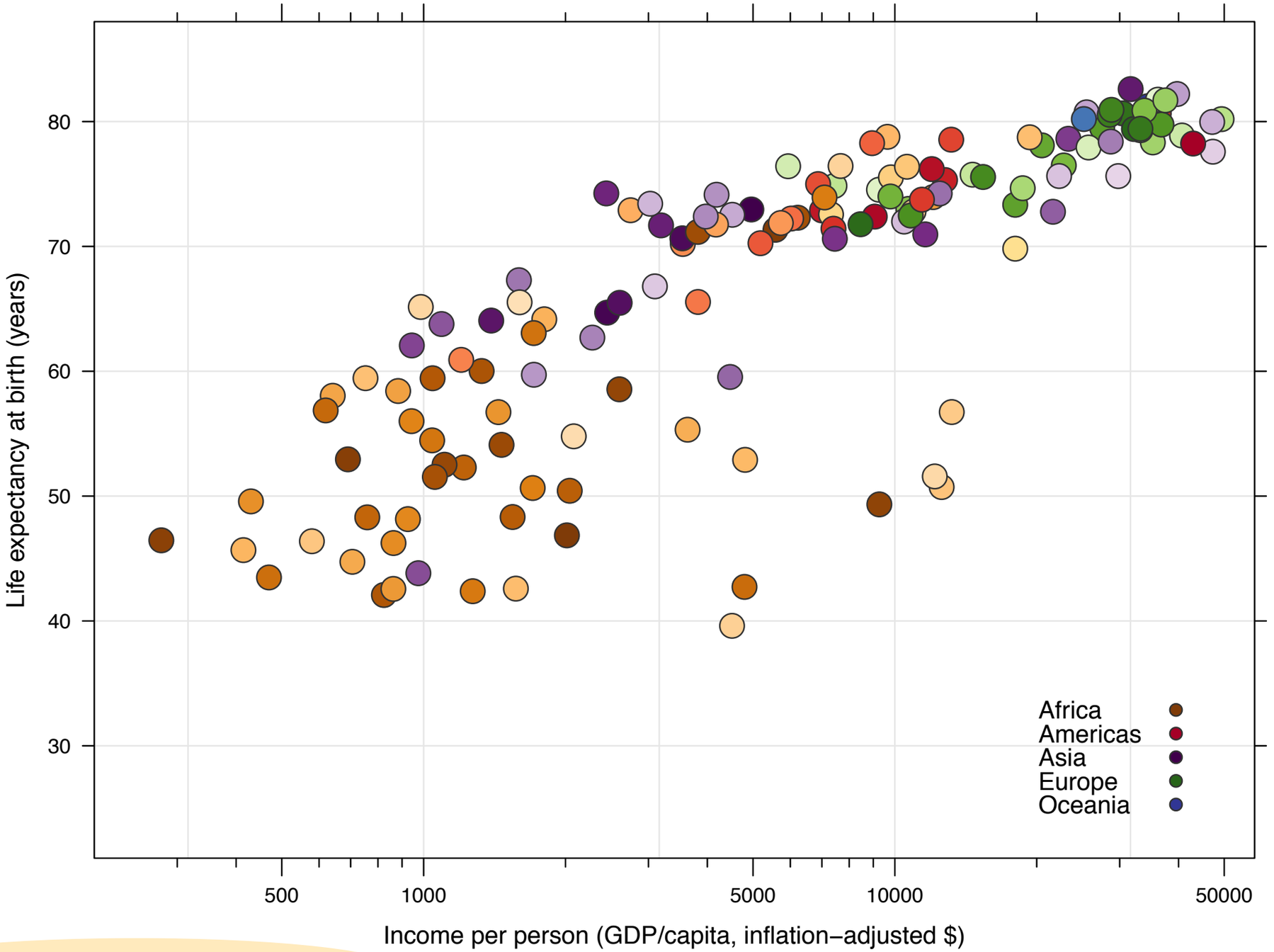
The auto.key argument is incredibly useful for adding a legend automatically. In many situations, auto.key = TRUE works perfectly. In many others, auto.key can be used to provide a bit more info, e.g. specifying the location of the legend, while letting everything else happen automatically.

```
xyplot(lifeExp ~ gdpPercap, ...
       group = country,
       par.settings = jGapminderPars,
       auto.key = TRUE)
```

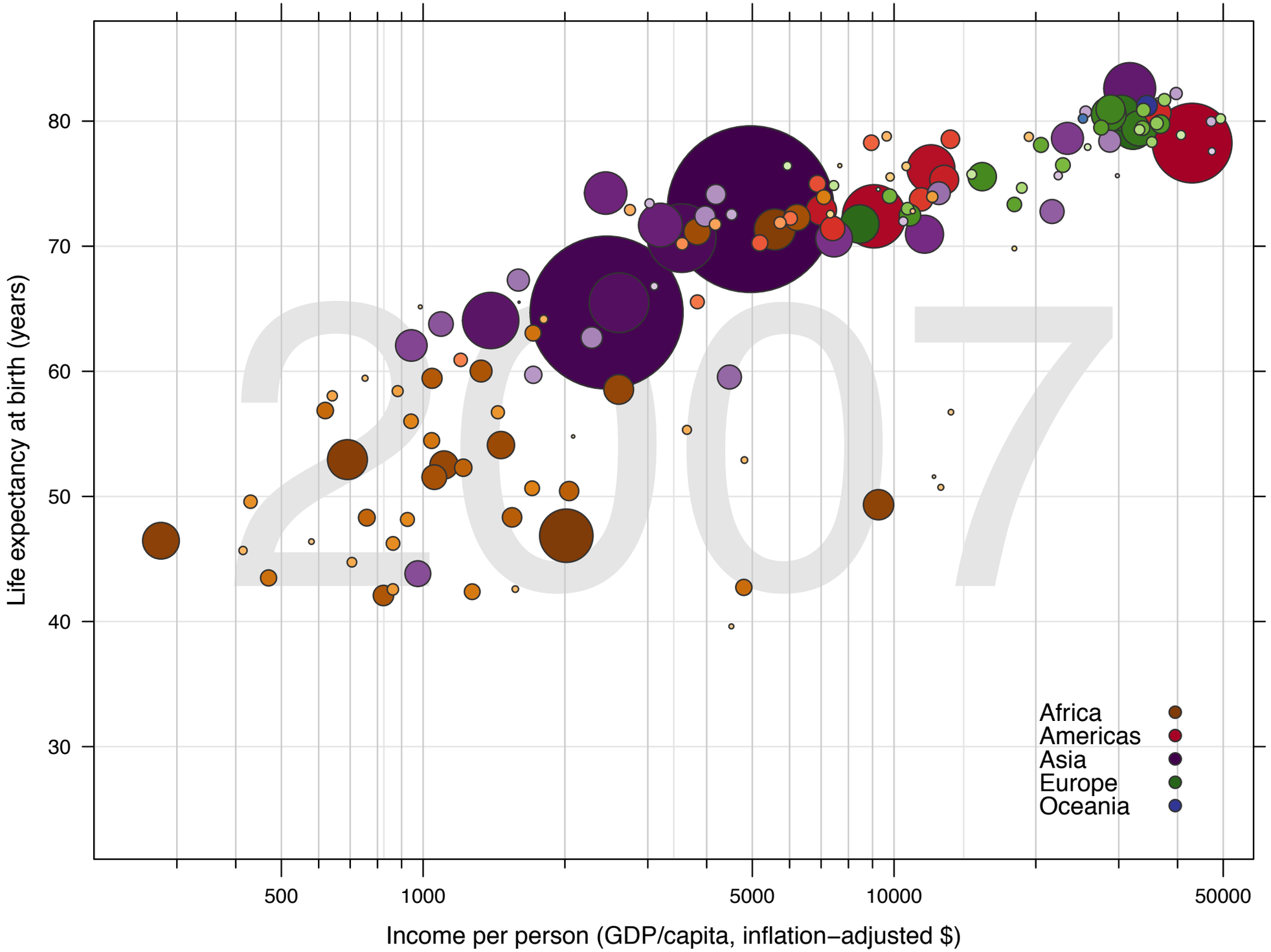


The Gapminder data presents some tricky issues -- it's a case where even finer control must be exerted.

```
xyplot(lifeExp ~ gdpPercap, ...
       group = country,
       par.settings = jGapminderPars,
       auto.key = TRUE)
```




```
continentKey <-
  with(continentColors,
       list(x = 0.95, y = 0.05, corner = c(1, 0),
            text = list(as.character(continent)),
            points = list(pch = jPch, col = jDarkGray, fill = color)))

xyplot(lifeExp ~ gdpPercap, ...,
       key = continentKey)
```



Finally ... sizing the circles s.t. area = pop. Again, it's a bit tricky -- not a problem encountered in a typical figure, so don't be discouraged. How to do?

```
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 = function(x, y, ...) {
           grid.text(jYear,
                     gp = gpar(fontsize = jFontsize, col = jLightestGray))
           panel.grid(h = -1, v = 0, col.line = jLightGray)
           panel.abline(v = log10(logTicks(c(10, 50000), loc = 1:9)),
                        col.line = jLightGray)
           panel.points(x, y, pch = jPch,
                        cex = jPopRadFun(yDat$pop)/jCexDivisor,
                        col = jDarkGray, fill = yDat$color)
   })
```

Example of redefining the panel function ‘on the fly’. Though it looks geeky, this is in fact a very common way to exert control over lattice figures.

If the control you need cannot be exerted by clever use of arguments or (temporarily) changing some graphics settings, then you may need to rewrite the panel function.

This is not as awful as it sounds.

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()

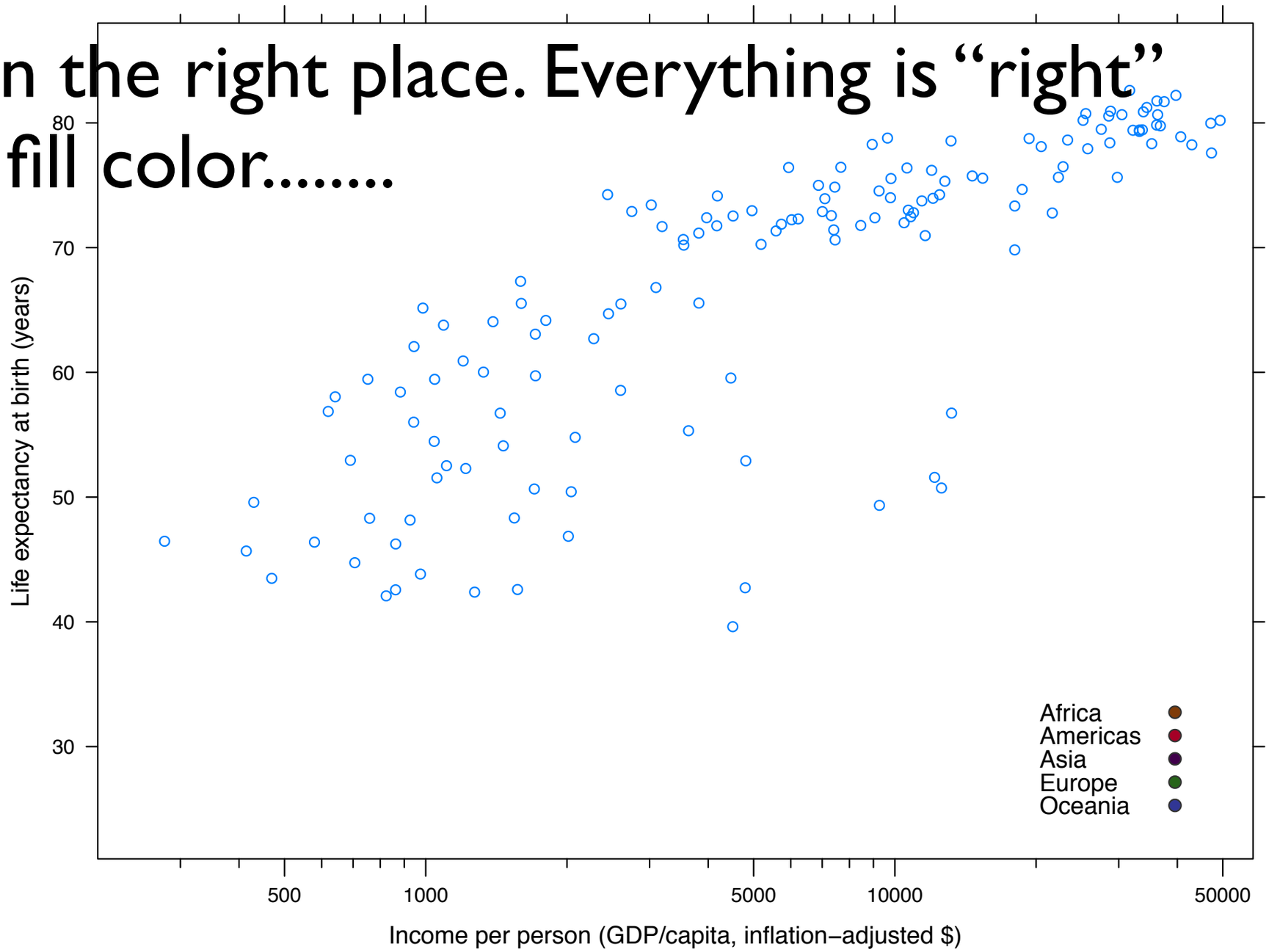
Workflow for rewriting (e.g. extending) a panel function:

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.....

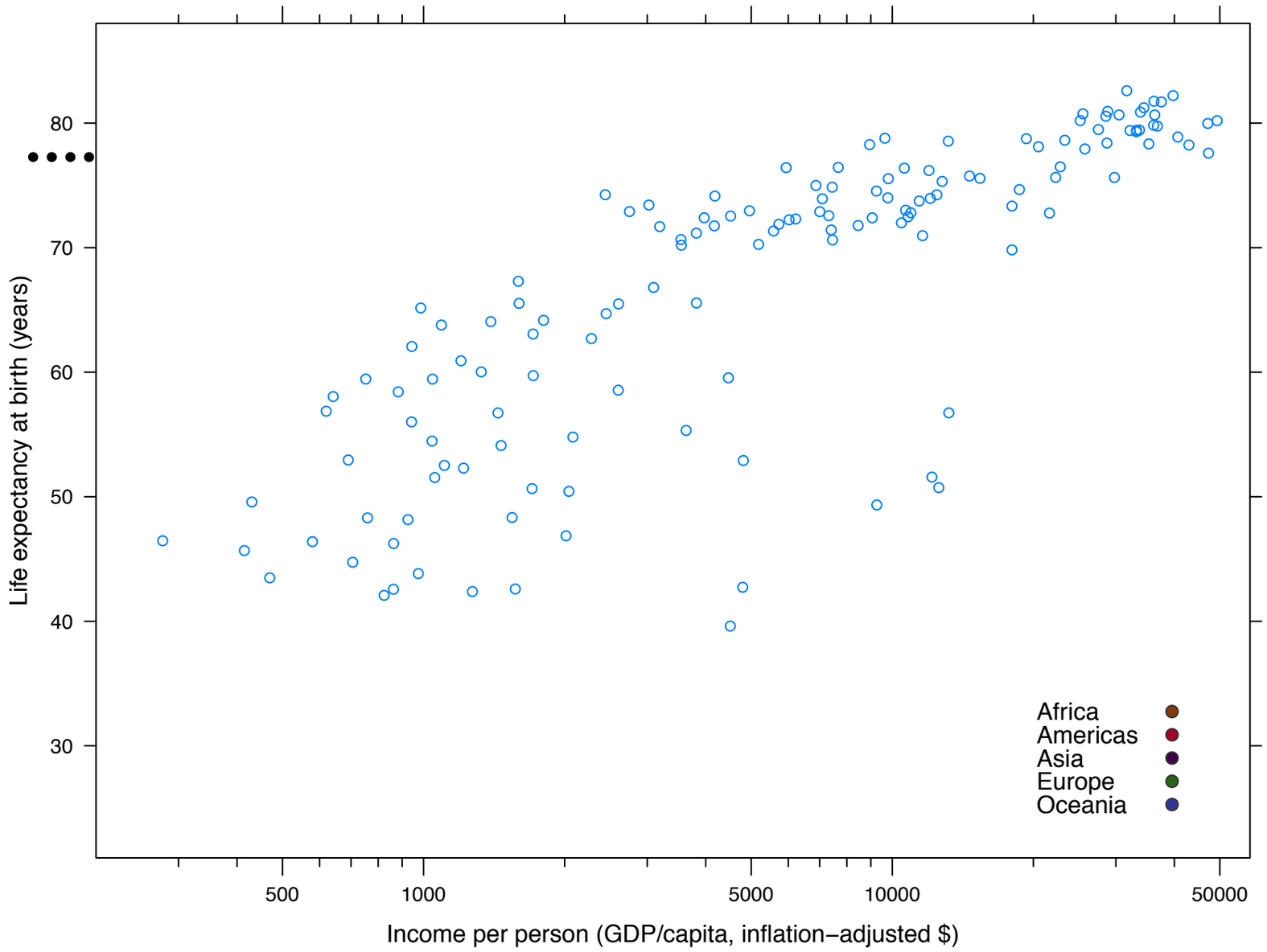



```
yDat <- subset(gDatOrdered, year == jYear)
xyplot(lifeExp ~ gdpPercap, yDat,
       xlab = jXlab, ylab = jYlab,
       scales = list(x = list(log = 10)),
       xlim = jXlim, ylim = jYlim,
       xscale.components = xscale.components.log10,
       key = continentKey,
       panel = panel.xyplot)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

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

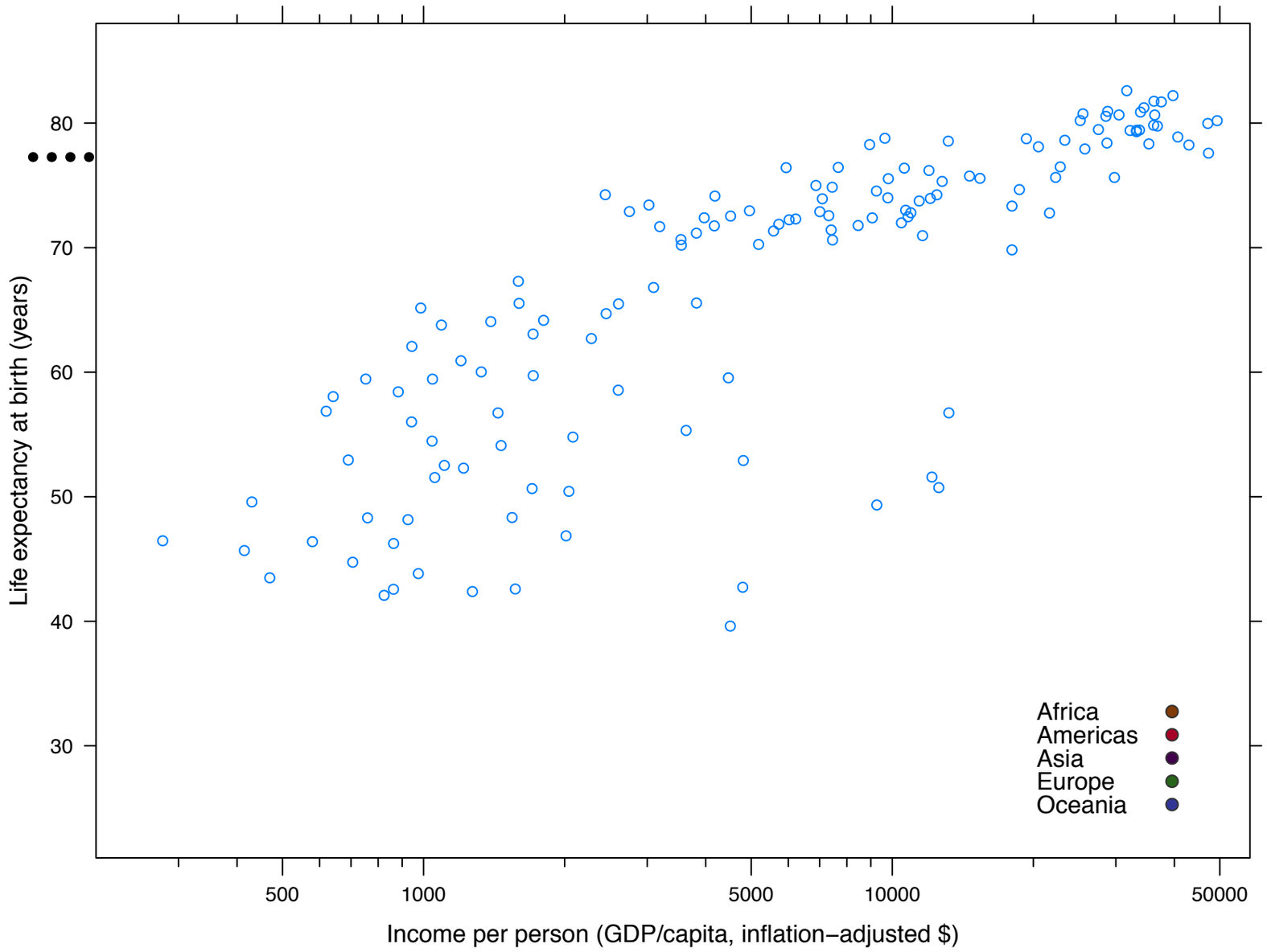

```
yDat <- subset(gDatOrdered, year == jYear)
xyplot(lifeExp ~ gdpPercap, yDat,
       xlab = jXlab, ylab = jYlab,
       scales = list(x = list(log = 10)),
       xlim = jXlim, ylim = jYlim,
       xscale.components = xscale.components.log10,
       key = continentKey,
       panel = panel.xyplot)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

Same plot as before ....



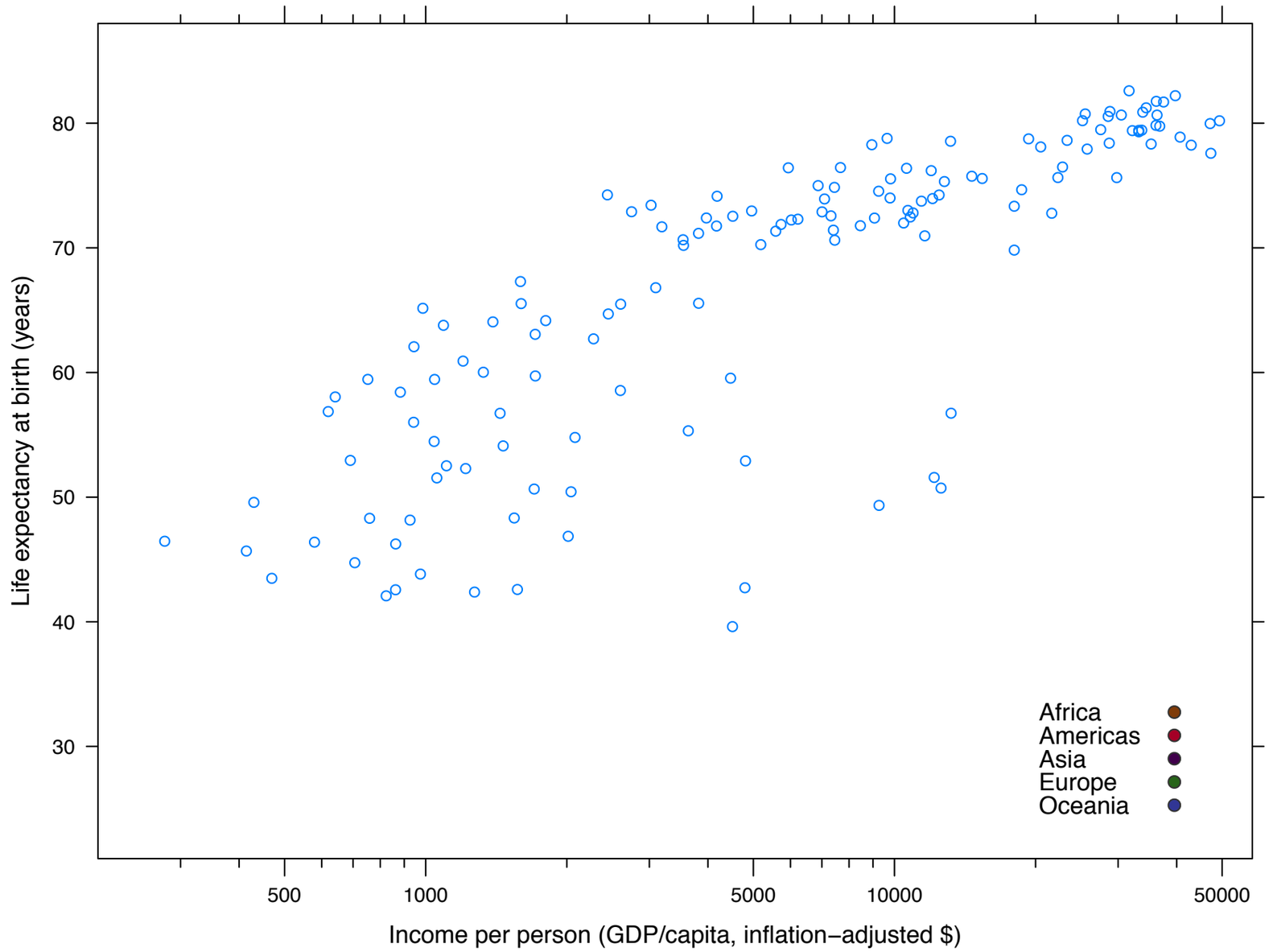

```
xyplot(lifeExp ~ gdpPercap, yDat,
       xlab = jXlab, ylab = jYlab,
       scales = list(x = list(log = 10)),
       xlim = jXlim, ylim = jYlim,
       xscale.components = xscale.components.log10,
       key = continentKey,
       panel = function(...) {
           panel.xyplot(...)
       })
```

Same plot as before ....



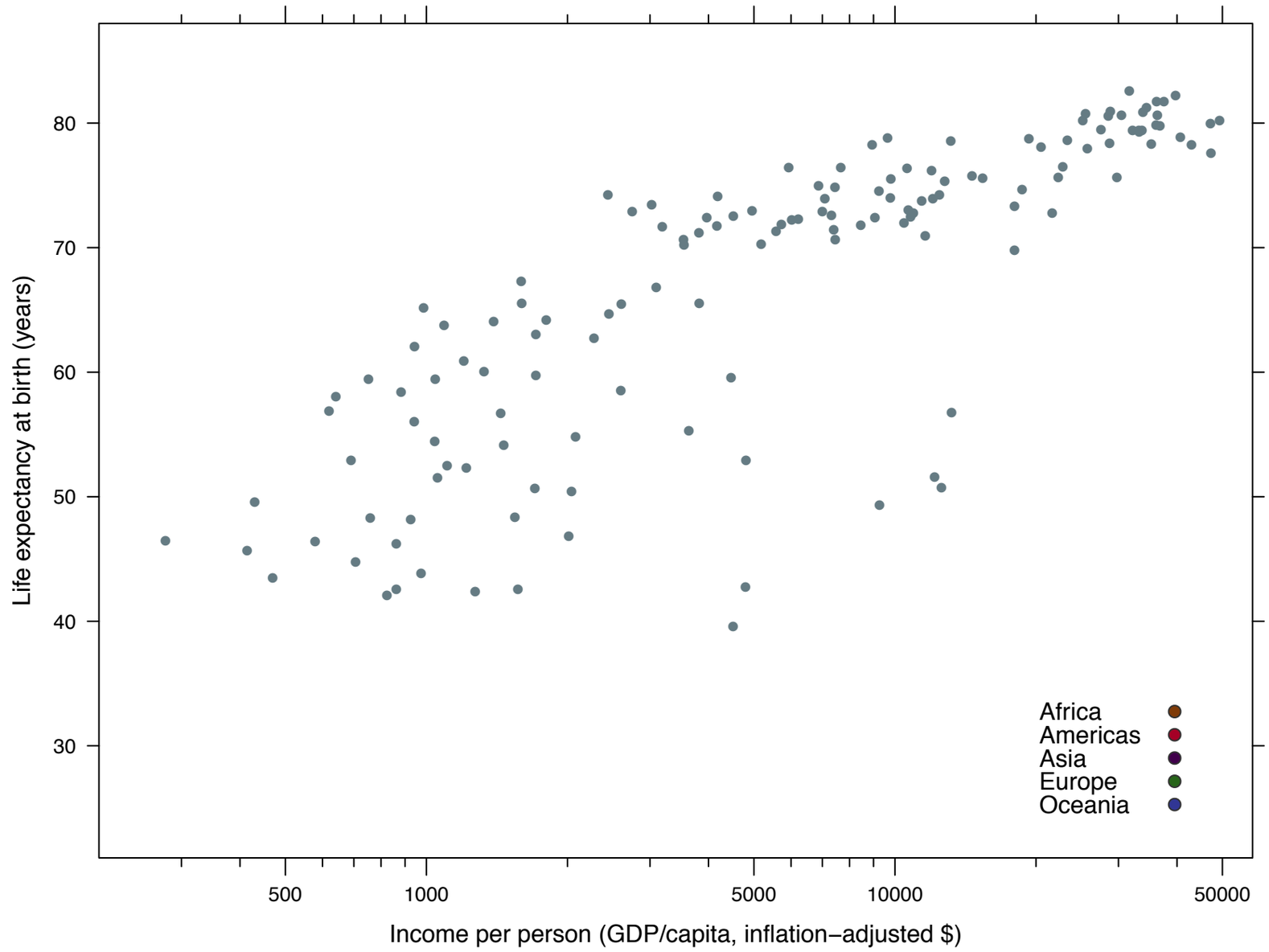

```
xyplot(lifeExp ~ gdpPercap, yDat,
       xlab = jXlab, ylab = jYlab,
       scales = list(x = list(log = 10)),
       xlim = jXlim, ylim = jYlim,
       xscale.components = xscale.components.log10,
       key = continentKey,
       panel = function(x, y, ...) {
           panel.xyplot(x, y, ...)
       })
```

Same plot as before .... but I've drawn the points myself!



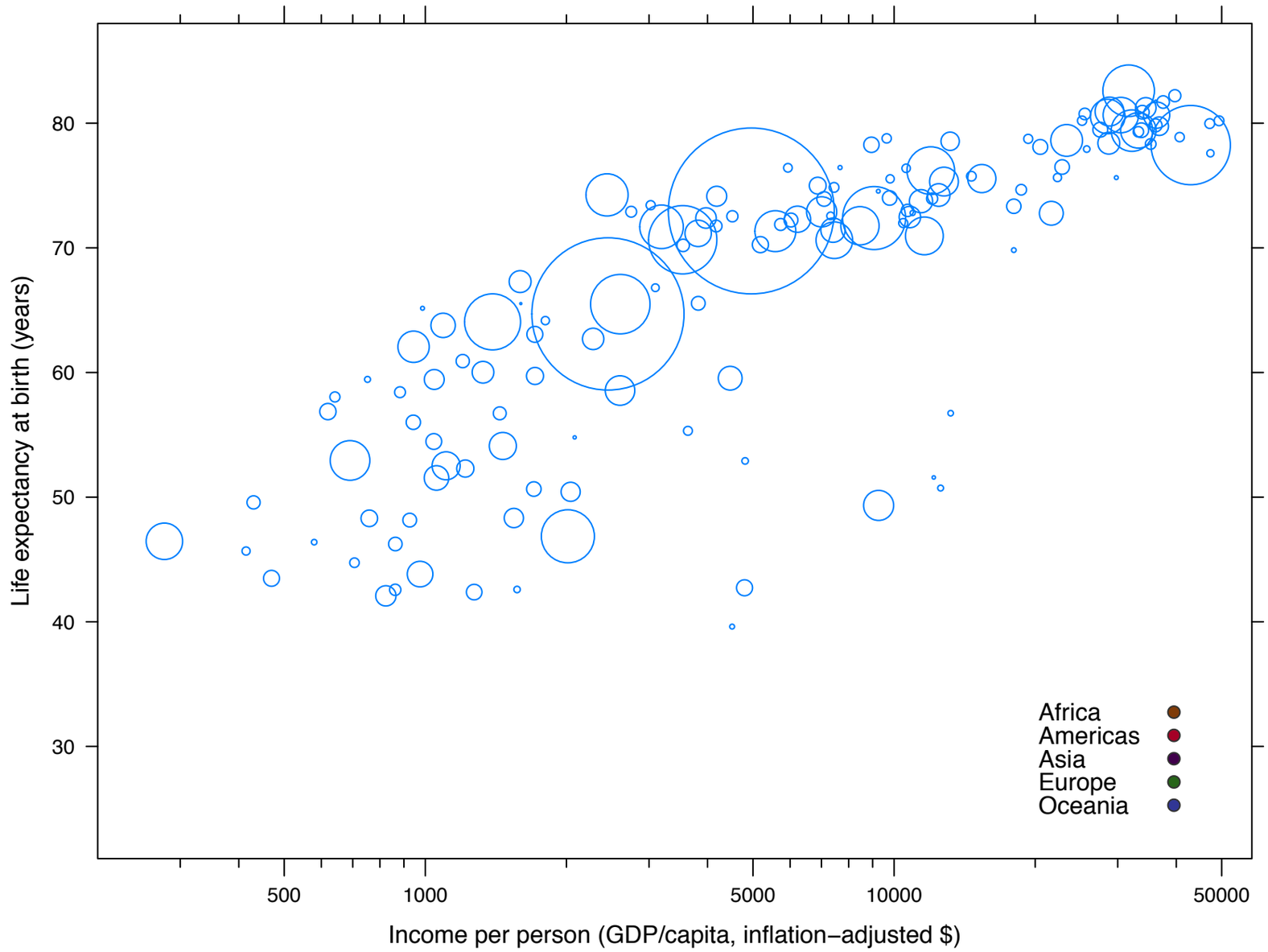

```
xyplot(lifeExp ~ gdpPercap, yDat,
       xlab = jXlab, ylab = jYlab,
       scales = list(x = list(log = 10)),
       xlim = jXlim, ylim = jYlim,
       xscale.components = xscale.components.log10,
       key = continentKey,
       panel = function(x, y, ...) {
           panel.points(x, y)
       })
```

Same plot as before .... but I've drawn the points myself!



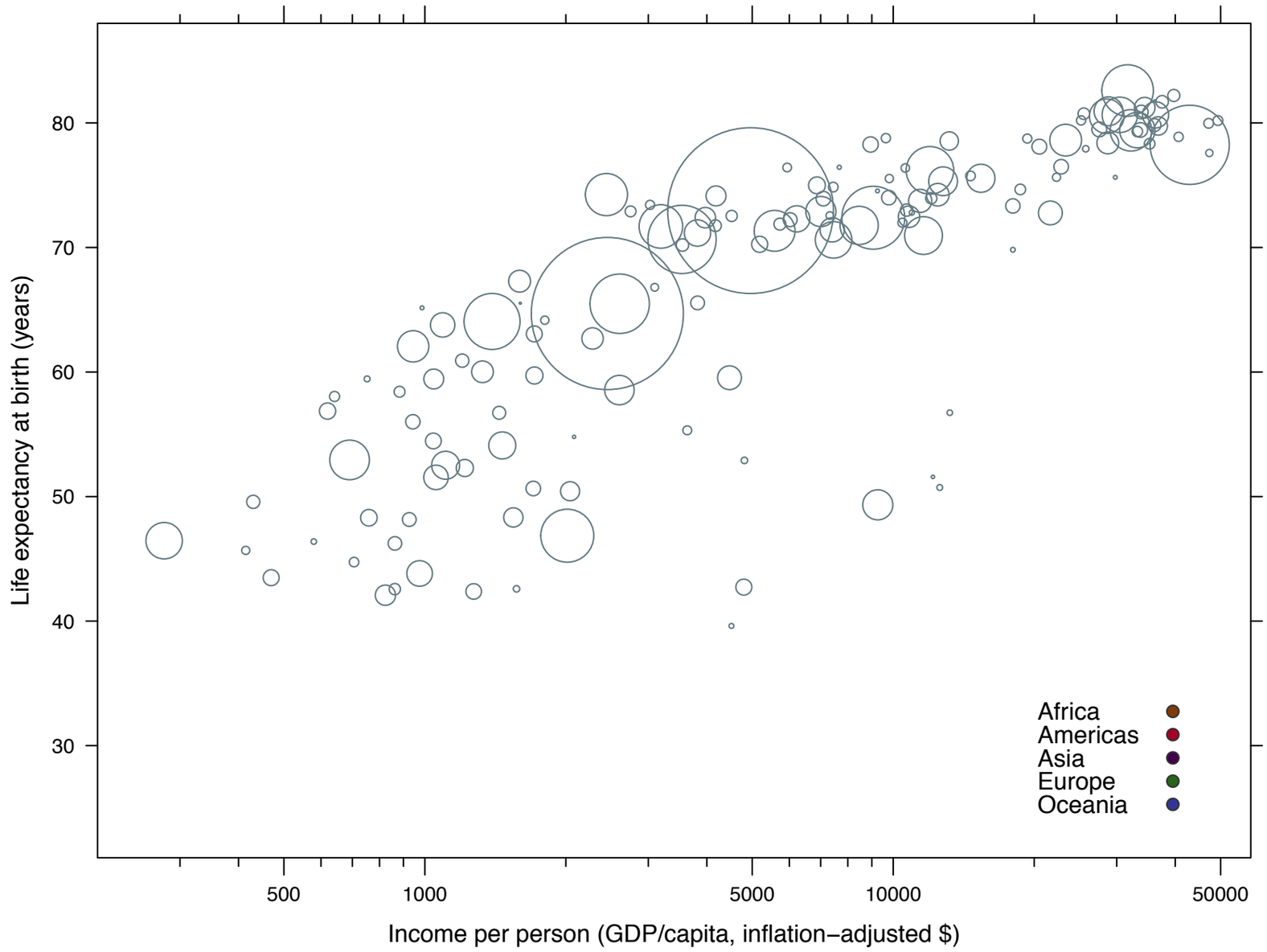

```
xyplot(lifeExp ~ gdpPercap, yDat,
       xlab = jXlab, ylab = jYlab,
       scales = list(x = list(log = 10)),
       xlim = jXlim, ylim = jYlim,
       xscale.components = xscale.components.log10,
       key = continentKey,
       panel = function(x, y, ...) {
           panel.points(x, y)
       })
```

Now I'm controlling the size! Yay!



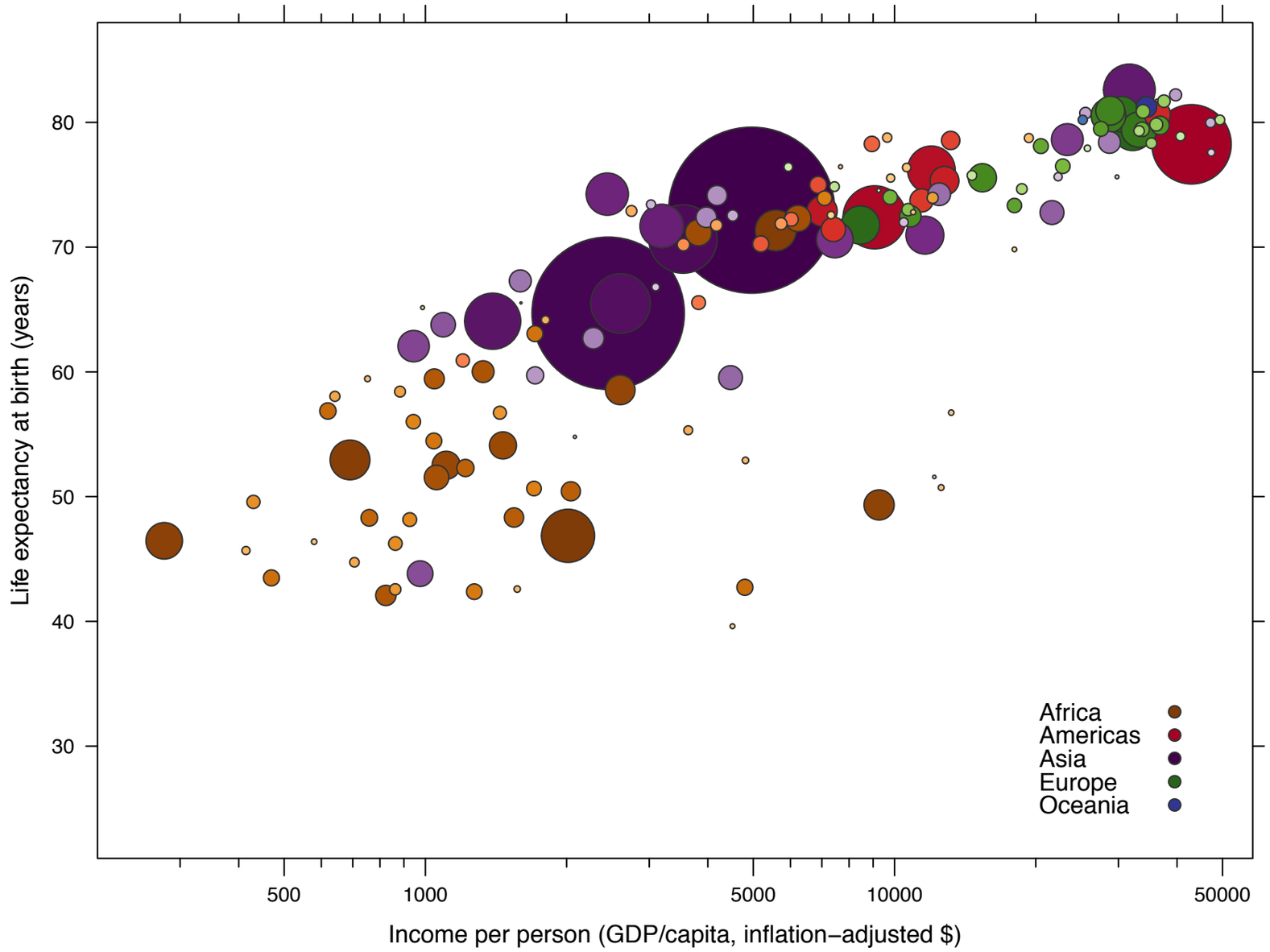

```
xyplot(lifeExp ~ gdpPercap, yDat,
       xlab = jXlab, ylab = jYlab,
       scales = list(x = list(log = 10)),
       xlim = jXlim, ylim = jYlim,
       xscale.components = xscale.components.log10,
       key = continentKey,
       panel = function(x, y, ...) {
           panel.points(x, y, pch = jPch,
                        cex = jPopRadFun(yDat$pop)/jCexDivisor)
       })
```

Now I'm controlling the size! Yay!



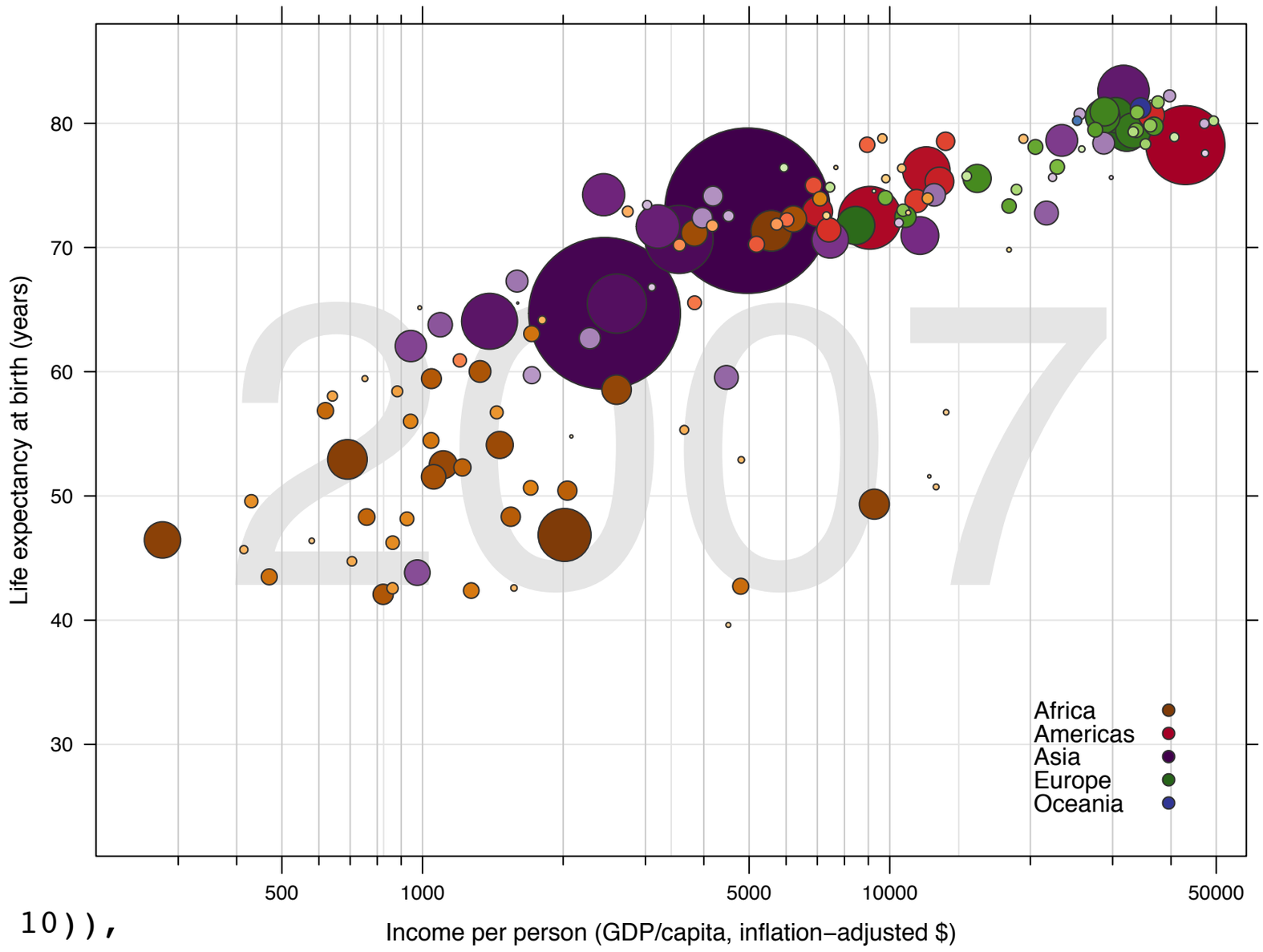

```
xyplot(lifeExp ~ gdpPercap, yDat,
       xlab = jXlab, ylab = jYlab,
       scales = list(x = list(log = 10)),
       xlim = jXlim, ylim = jYlim,
       xscale.components = xscale.components.log10,
       key = continentKey,
       panel = function(x, y, ...) {
           panel.points(x, y, pch = jPch,
                        cex = jPopRadFun(yDat$pop)/jCexDivisor)
       })
```

Now I'm controlling the color! YES!




```
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 = function(x, y, ...) {
           panel.points(x, y, pch = jPch,
                        cex = jPopRadFun(yDat$pop)/jCexDivisor,
                        col = jDarkGray, fill = yDat$color)
       })
```

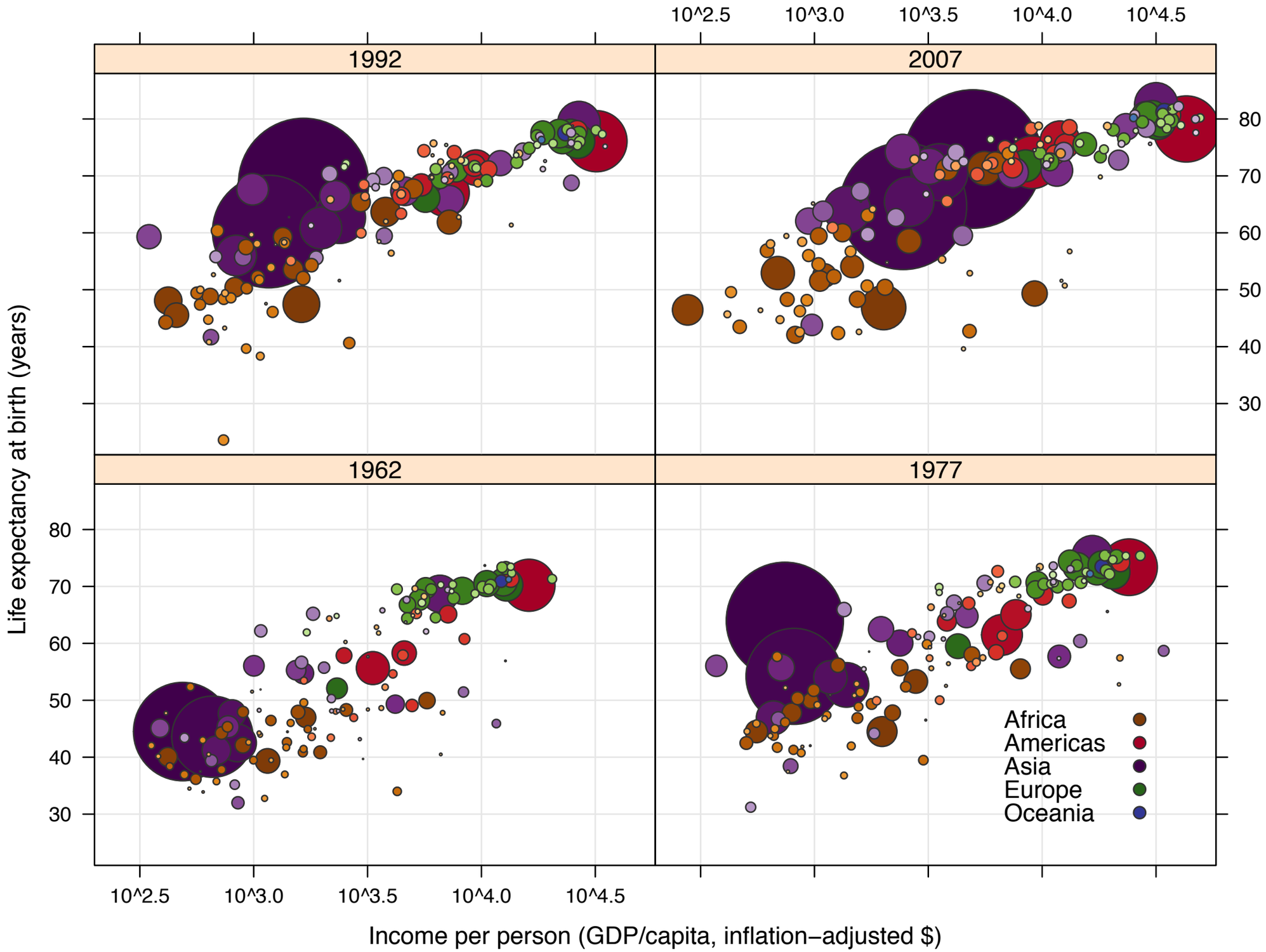
Reference grid!
Year!



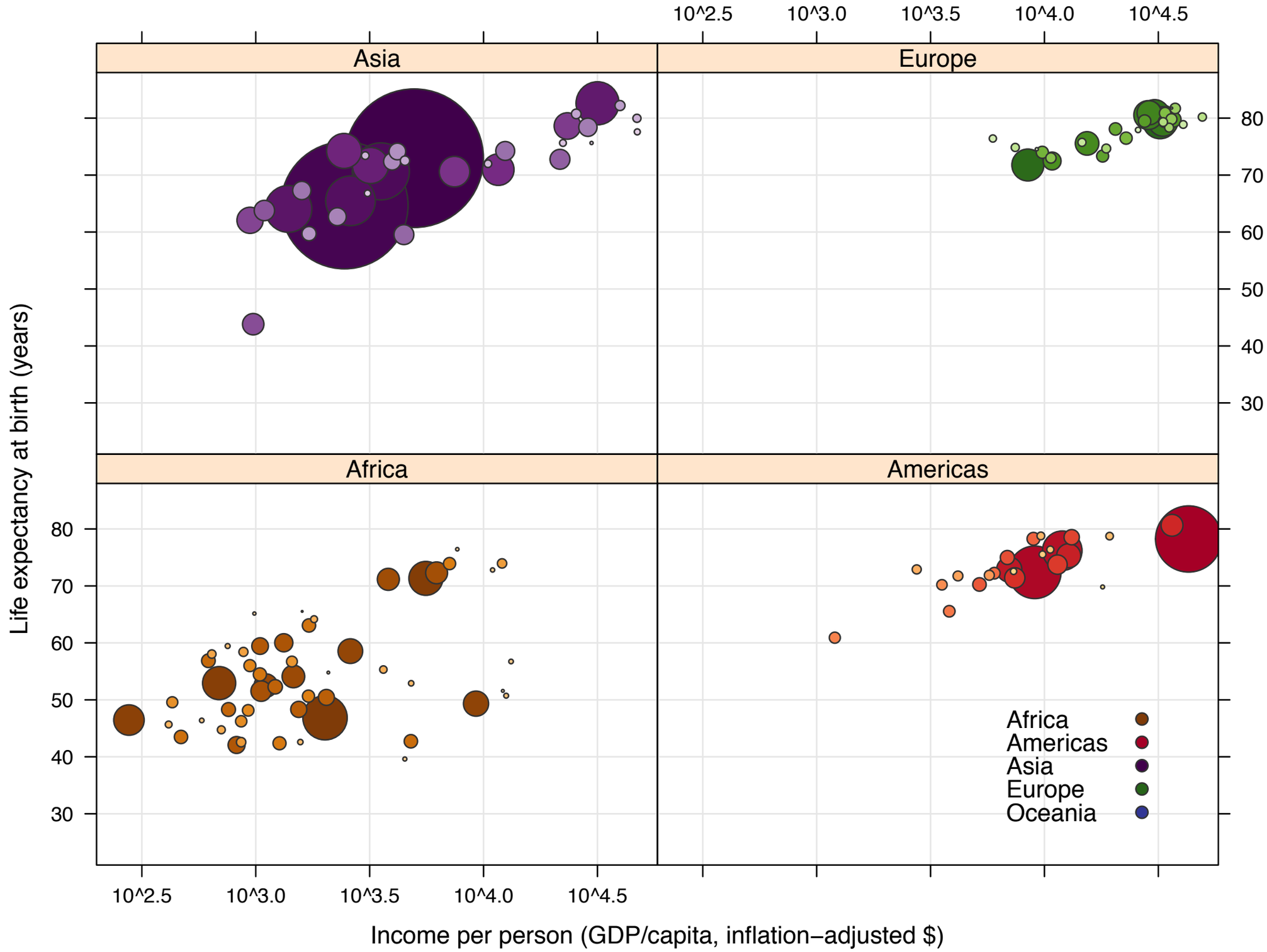

```
xyplot(lifeExp ~ gdpPercap, yDat,
       xlab = jXlab, ylab = jYlab,
       scales = list(x = list(log = 10)),
       xlim = jXlim, ylim = jYlim,
       xscale.components = xscale.components.log10,
       key = continentKey,
       panel = function(x, y, ...) {
           grid.text(jYear,
                     gp = gpar(fontsize = jFontsize, col = jLightestGray))
           panel.grid(h = -1, v = 0, col.line = jLightGray)
           panel.abline(v = log10(logTicks(c(10, 50000), loc = 1:9)),
                        col.line = jLightGray)
           panel.points(x, y, pch = jPch,
                        cex = jPopRadFun(yDat$pop)/jCexDivisor,
                        col = jDarkGray, fill = yDat$color)
       })
```

```
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 = function(x, y, ...) {
           grid.text(jYear,
                     gp = gpar(fontsize = jFontsize, col = jLightestGray))
           panel.grid(h = -1, v = 0, col.line = jLightGray)
           panel.abline(v = log10(logTicks(c(10, 50000), loc = 1:9)),
                        col.line = jLightGray)
           panel.points(x, y, pch = jPch,
                        cex = jPopRadFun(yDat$pop)/jCexDivisor,
                        col = jDarkGray, fill = yDat$color)
       })
```

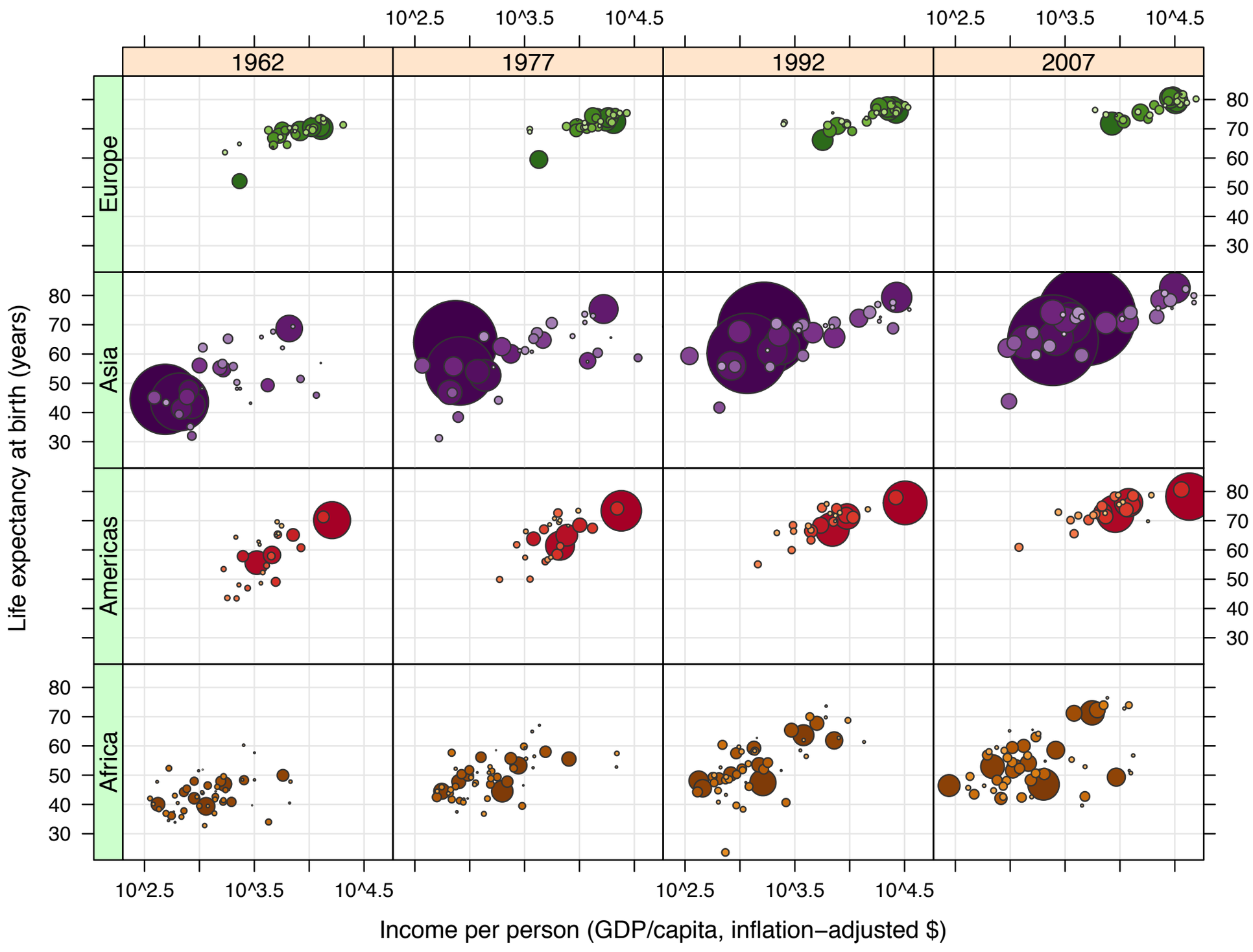
grid.text() puts the year in the background
panel.grid() adds reference lines (used here for horizontal)
panel.abline adds arbitrary lines (used here for custom vertical reflines)
panel.points() draws the filled circles



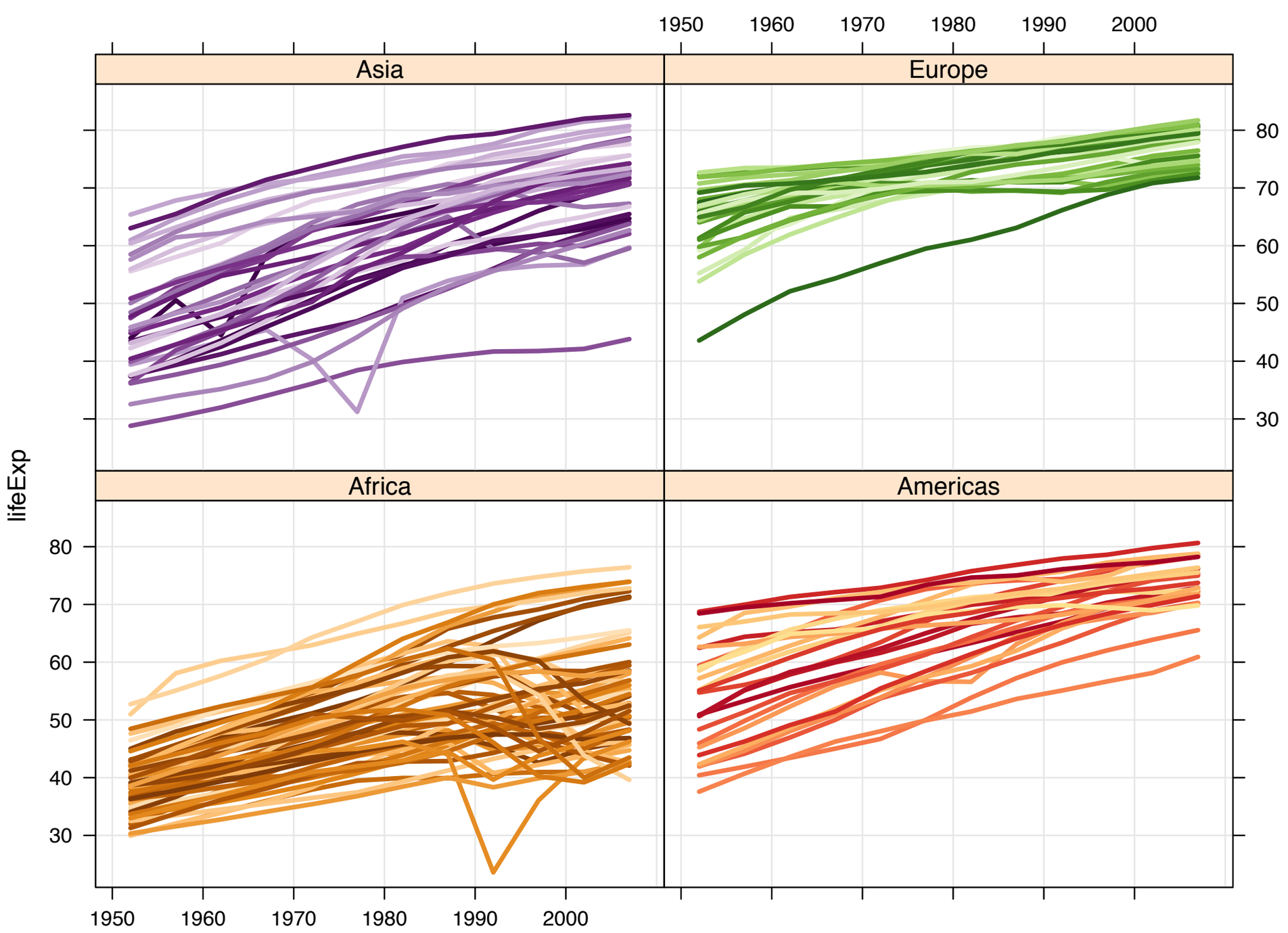

```
xyplot(lifeExp ~ gdpPercap | factor(year), ...)
```




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

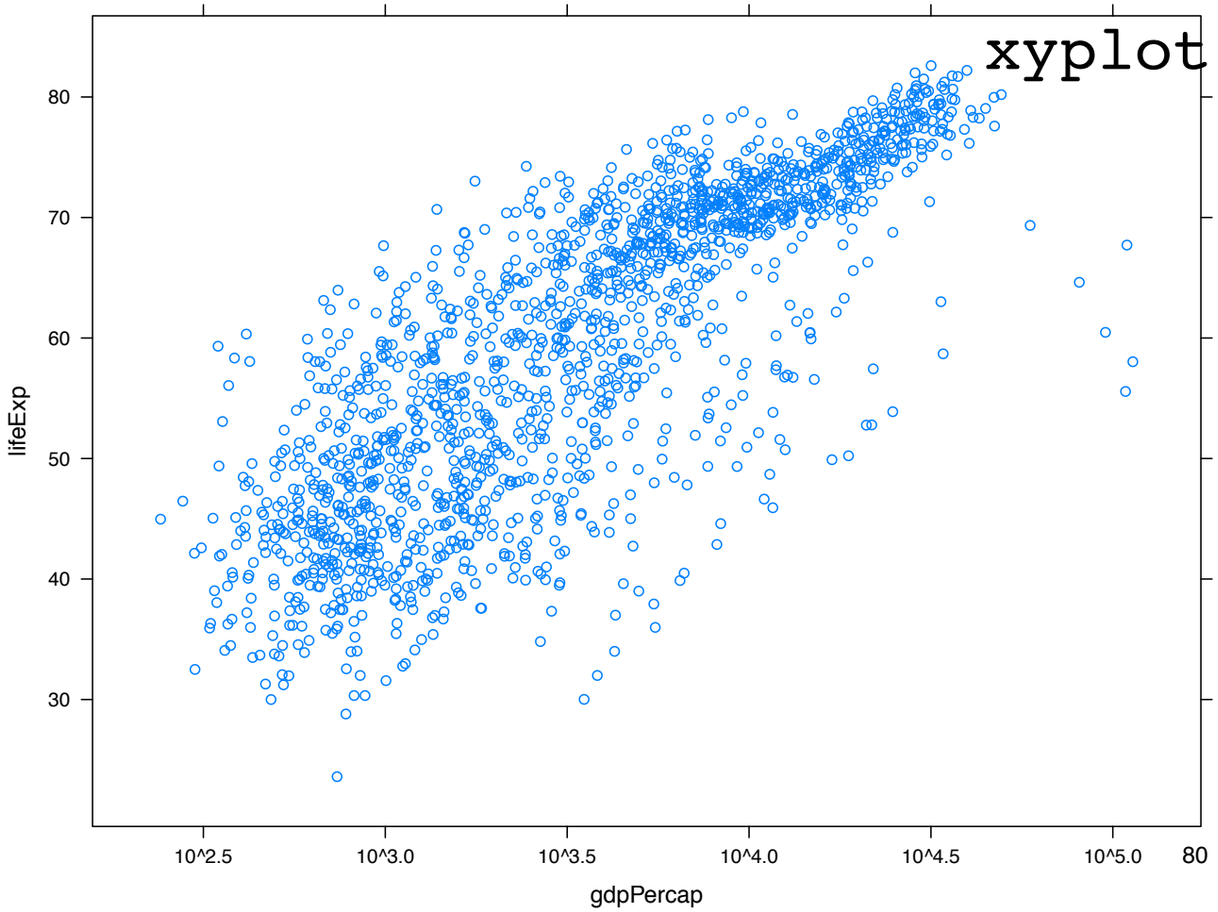



```
library(latticeExtra)                          # useOuterStrips()
jPlot <-
  xyplot(lifeExp ~ gdpPercap | factor(year) * continent, ...)
useOuterStrips(jPlot)
```

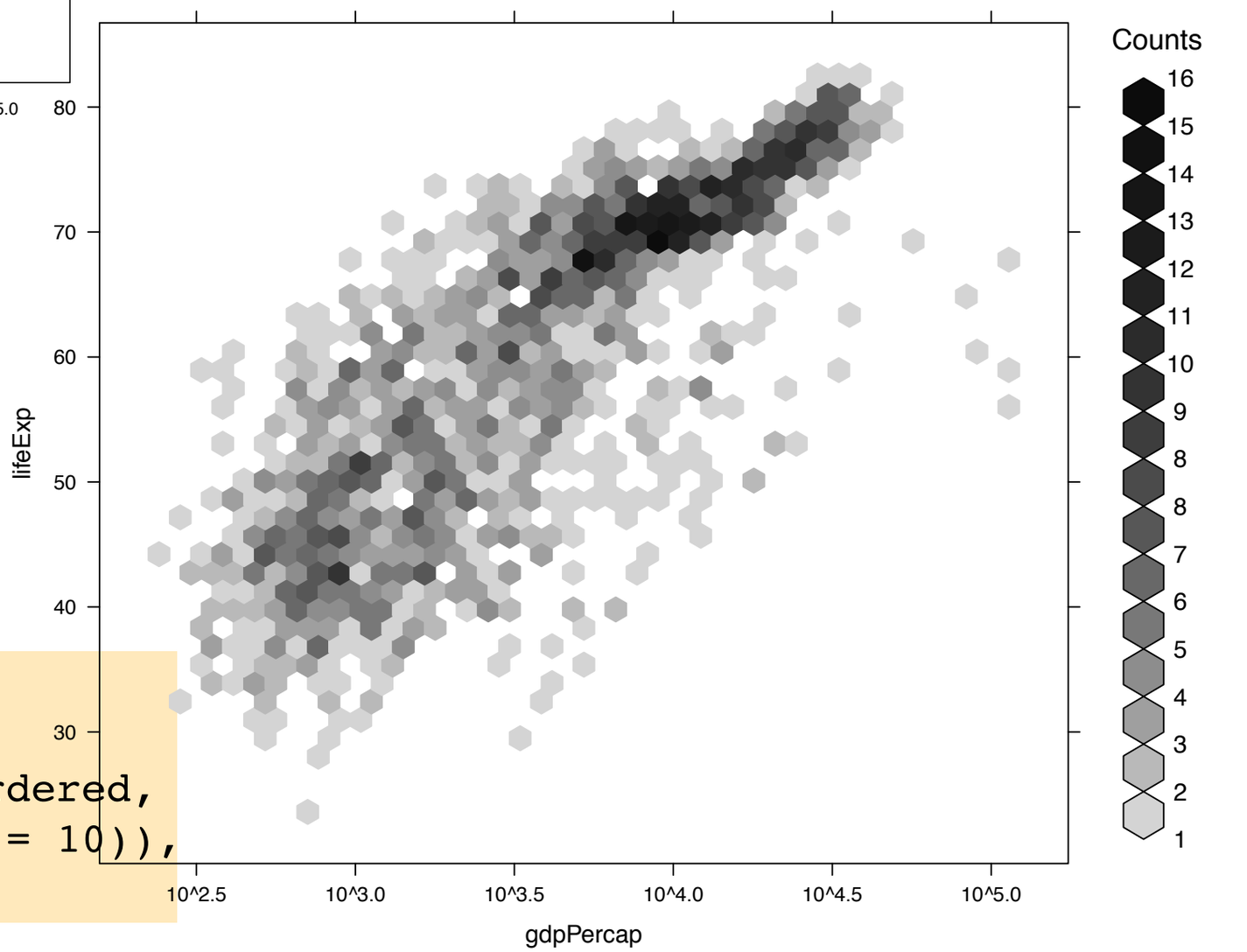



```
countryColorsOrdered <-
  countryColors$color[match(levels(gDatOrdered$country),
                            countryColors$country)]
jGapminderPars <-
  list(superpose.line = list(alpha = 0.1, lwd = 3, col = countryColorsOrdered))

xyplot(lifeExp ~ year | continent, gDatOrdered,
       subset = continent != "Oceania", ylim = jYlim, type = c("l","g"),
       groups = country, par.settings = jGapminderPars)
```

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



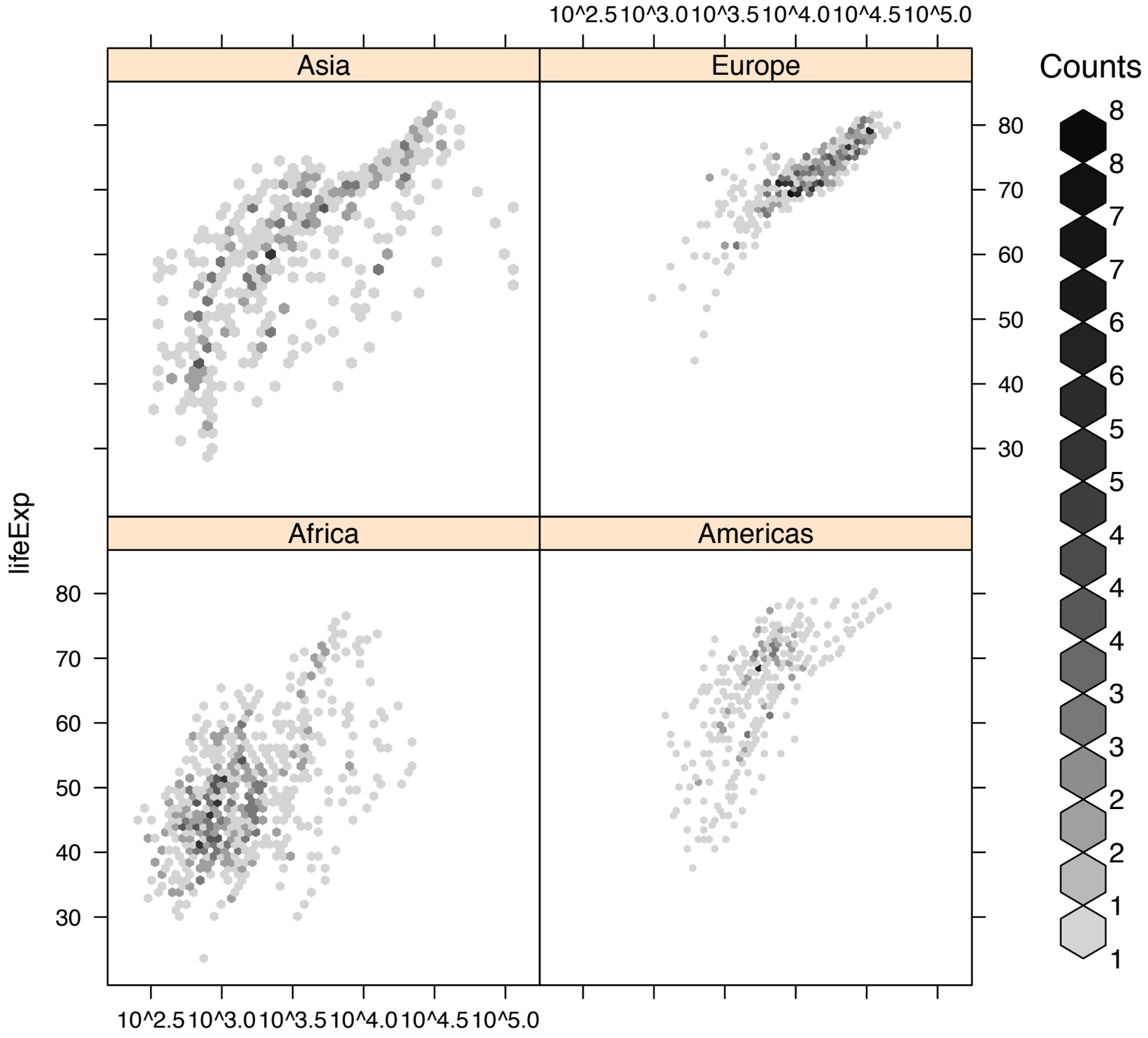
For high volume scatterplots, consider hexagonal binning.



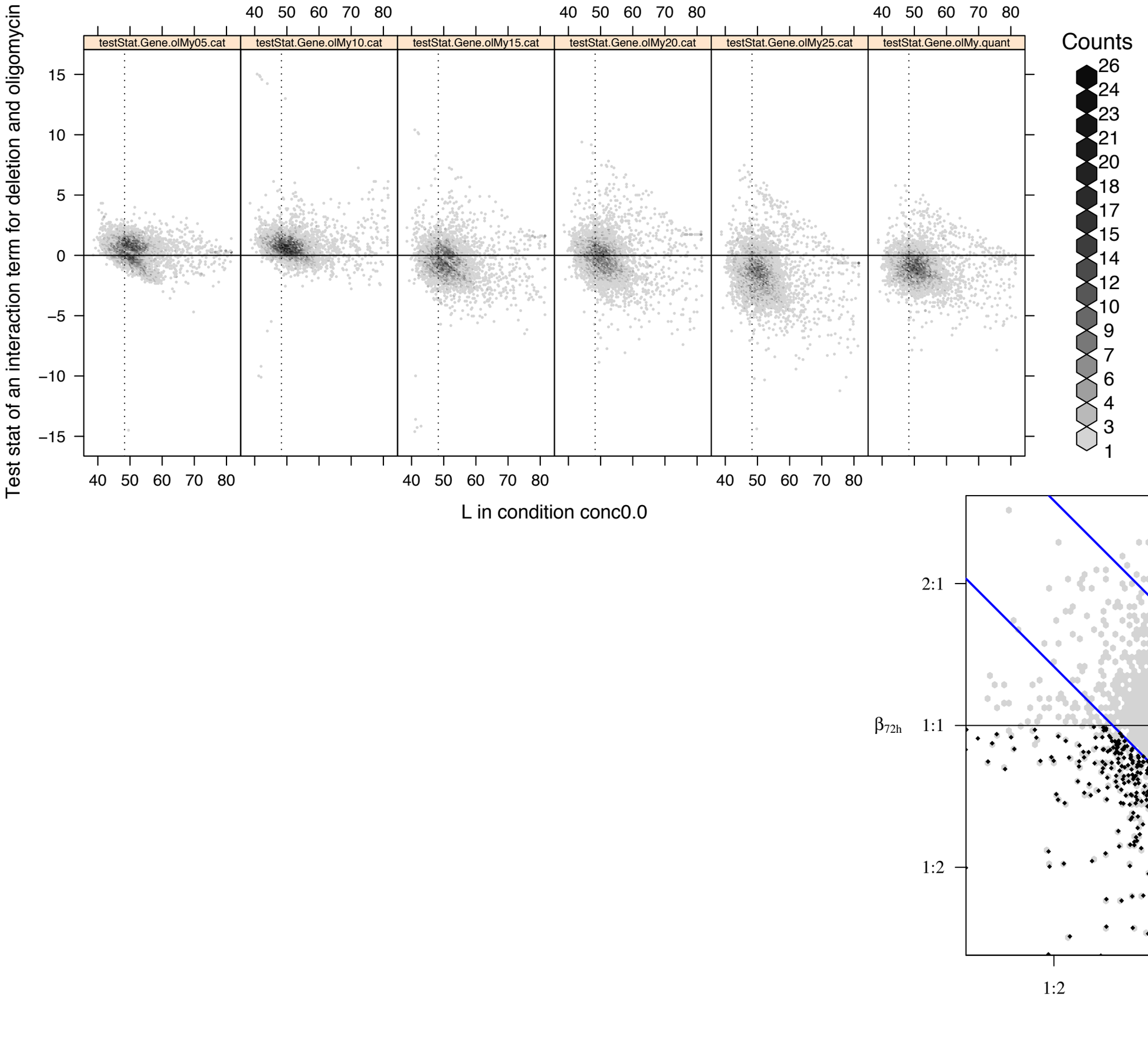

```
library(hexbin)

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

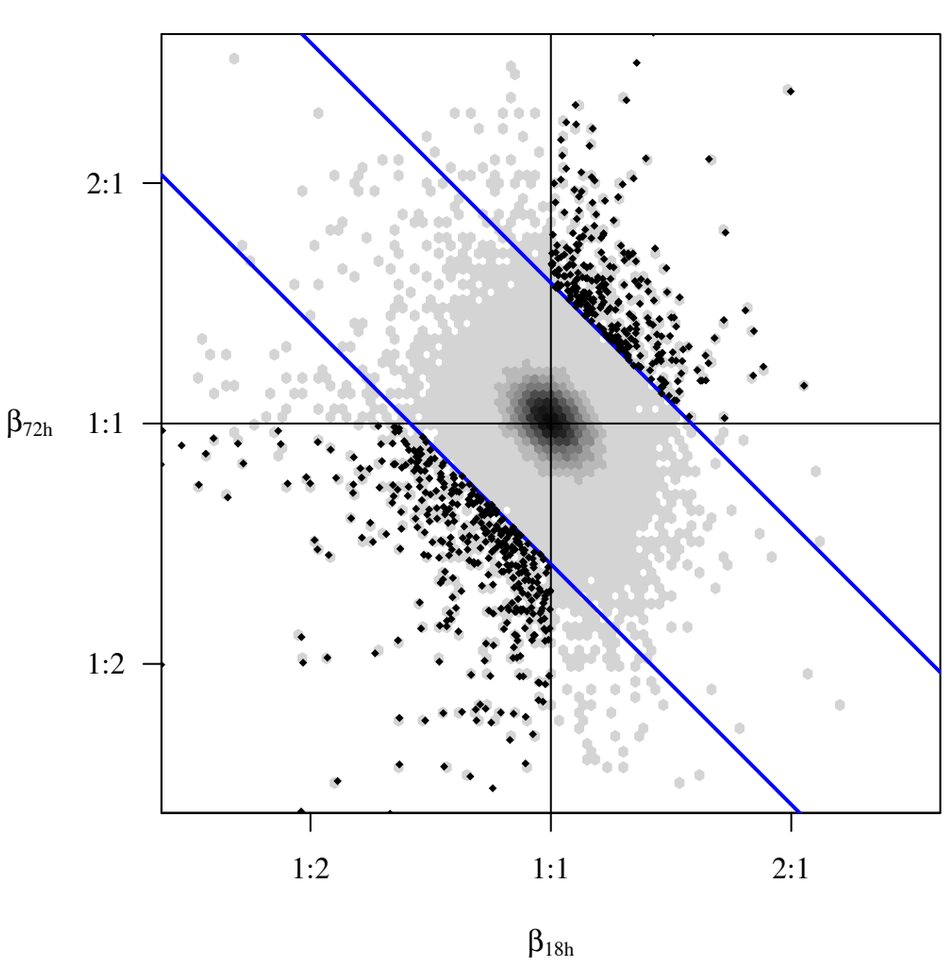
"Hexagonal binning"
Superior to scatterplot, when number of points creates overplotting problem.

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
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
β72h
1:2
1:1
2:1
β18h

Relevant R functions:

hexbinplot() and hexplom() from hexbin package

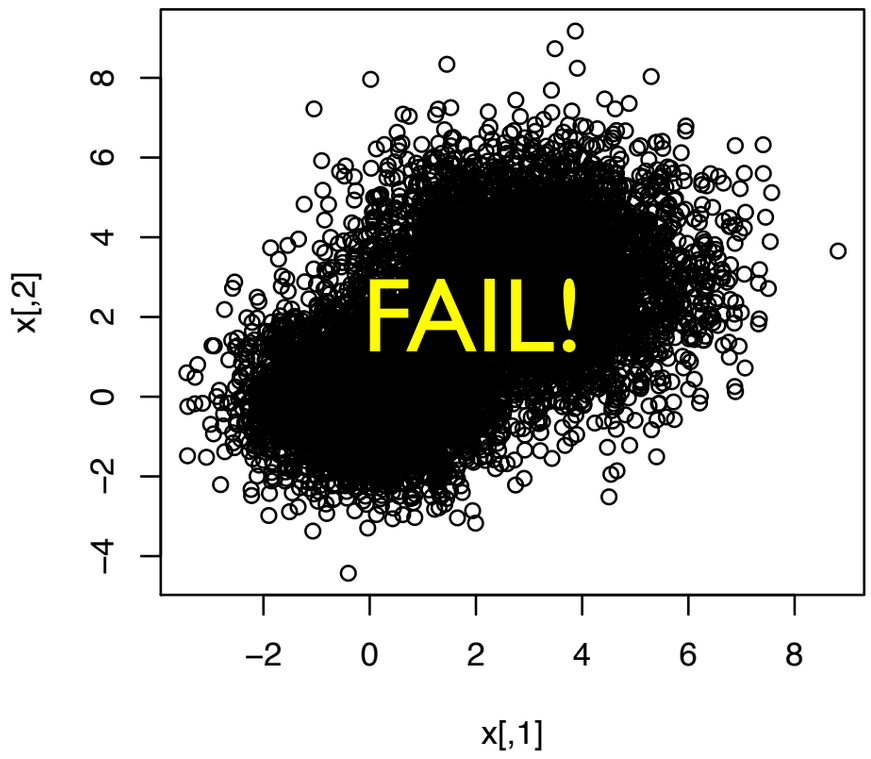
smoothScatter() -- also works well with pairs()

try running this:

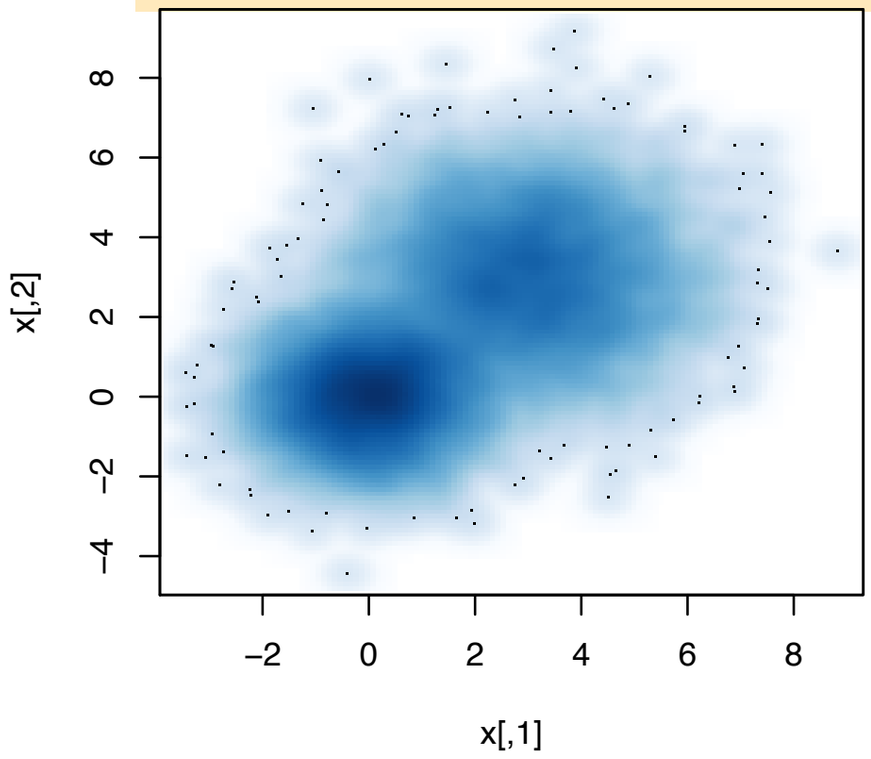
example(smoothScatter)

library(hexbin)
example(hexbin)

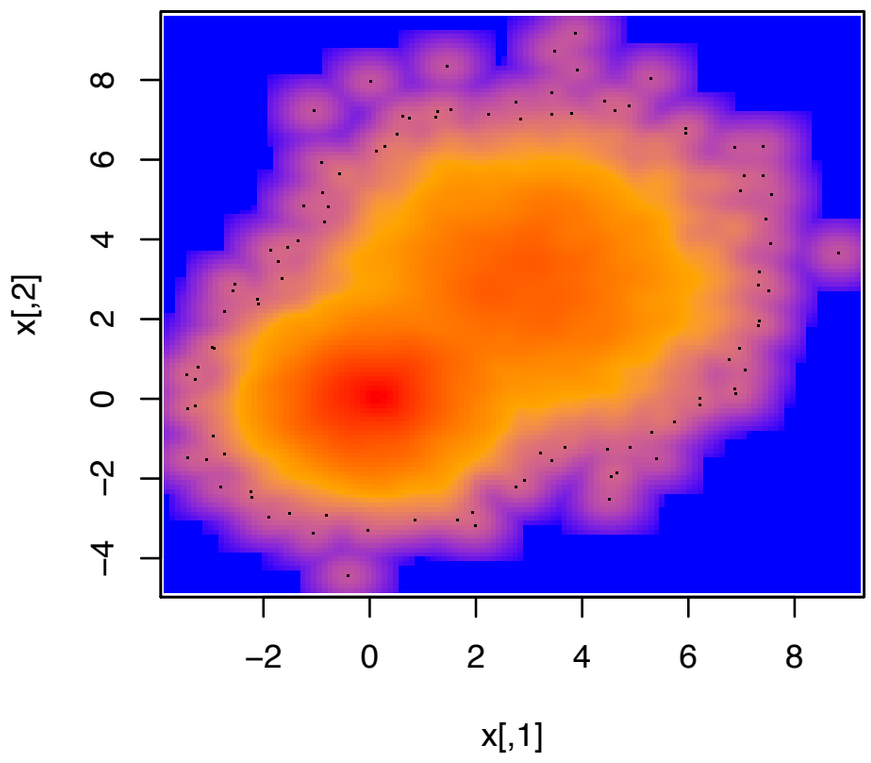
plot(x)



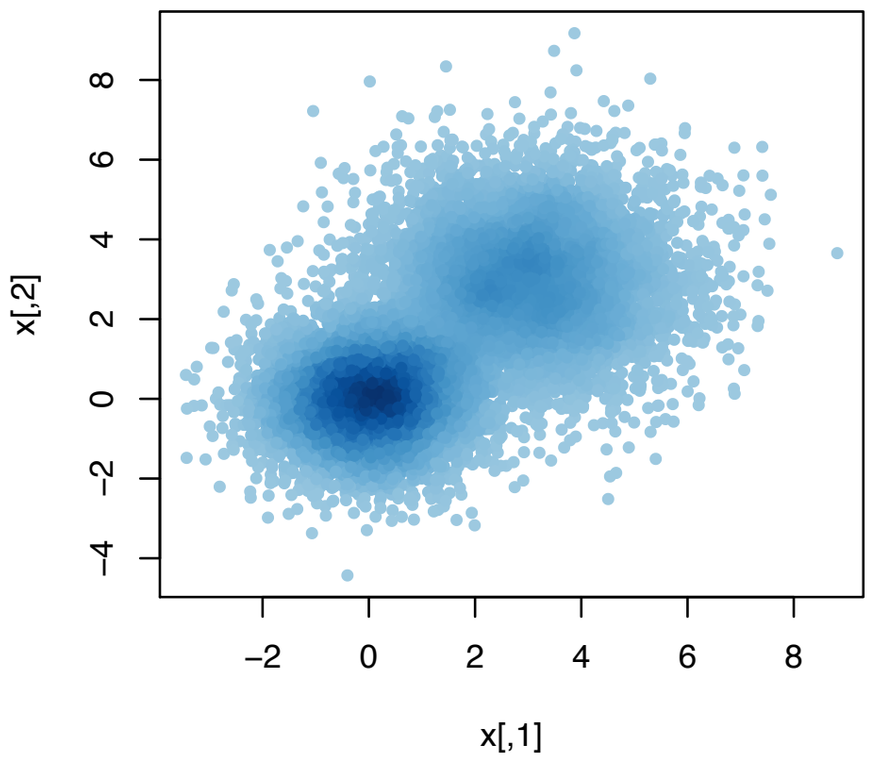
smoothScatter(x)

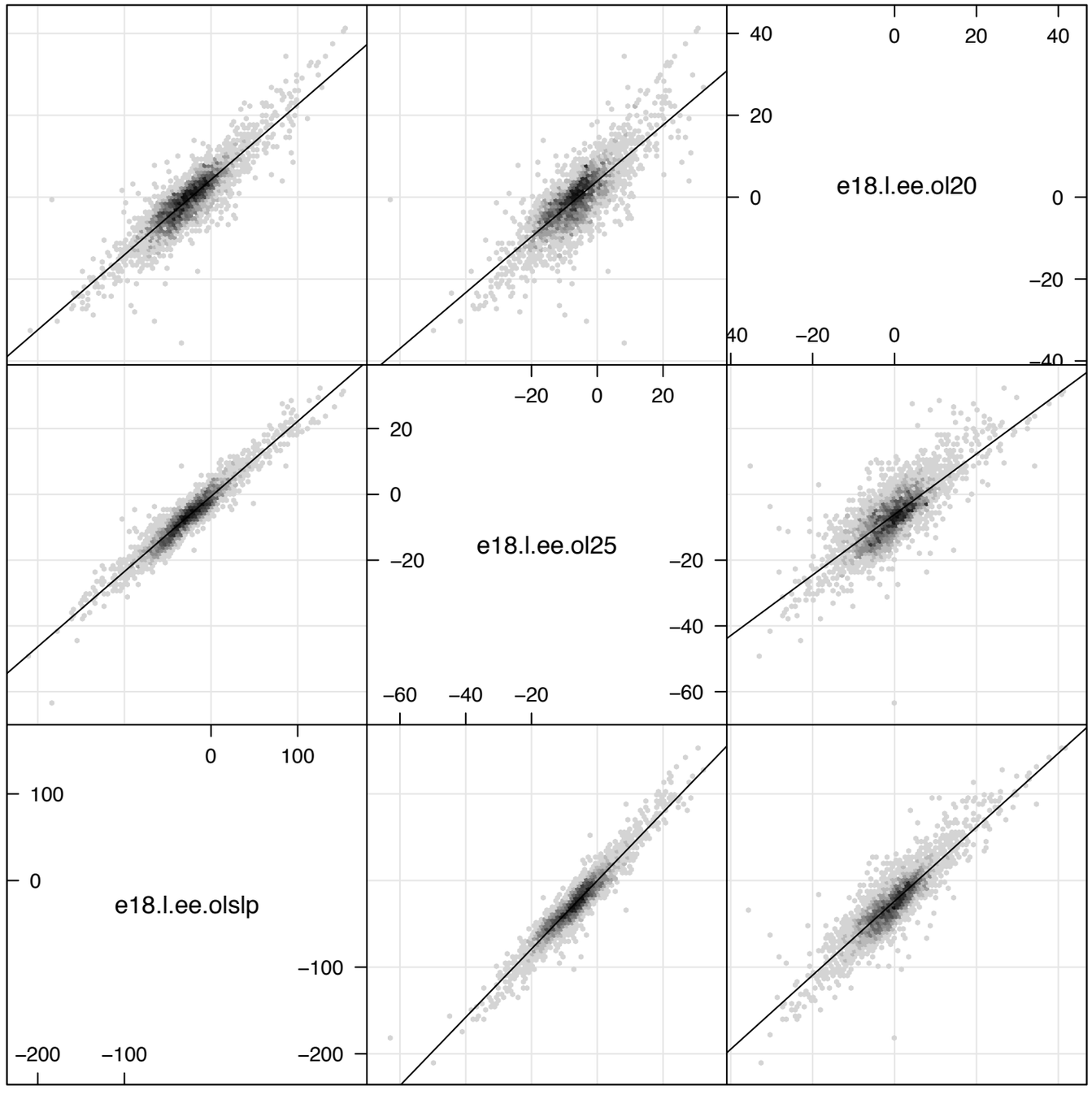



```
Lab.palette <- colorRampPalette(c("blue", "orange", "red"), space = "Lab")
smoothScatter(x, colramp = Lab.palette)
```



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))
```