CSSS 569: Visualizing Data

Graphical Programming:
R Graphics from the Ground Up

Christopher Adolph

Department of Political Science
and
Center for Statistics and the Social Sciences
University of Washington, Seattle

`R` from the Ground Up: Outline

Coordinate systems

Line & color

The `grid` graphics system

Using `lattice`

Approach

What I'm giving you today:

More readings from the dictionary. . .

Lots of sample code

Random bits of advice I wish someone had told me

Knowledge I consider most useful for graphical programming

I may gloss over something important

Stop me with questions

Initial minimalism

Always start with a blank screen.

```
filename <- "example.pdf"   # Name of output file
width <- 4                  # width of output
height <- 4.5               # height of output
pdf(filename=filename,
    width=width,
    height=height
    )
    # Other pdf options to consider:
    #  family, fontsize, bg, fg

plot.new()                  # Start the plot

# Do some graphics

dev.off()                   # Save the plot to disk and end
```

Initial minimalism

A good motto is to add nothing without thinking about why it needs to be added

This approach

- eliminates chartjunk
- casts aside convention for creativity
- gives you complete control

Before we ask

What to put on that screen?

we should ask:

Where to put it?

Coordinate systems

Computer graphics can *always* be though of as occurring on a 2D plane.

Convenient to treat the bottom left of screen as 0,0 and the top-right as 1,1.

Let's us put objects on screen w/ easy reference to relative position.

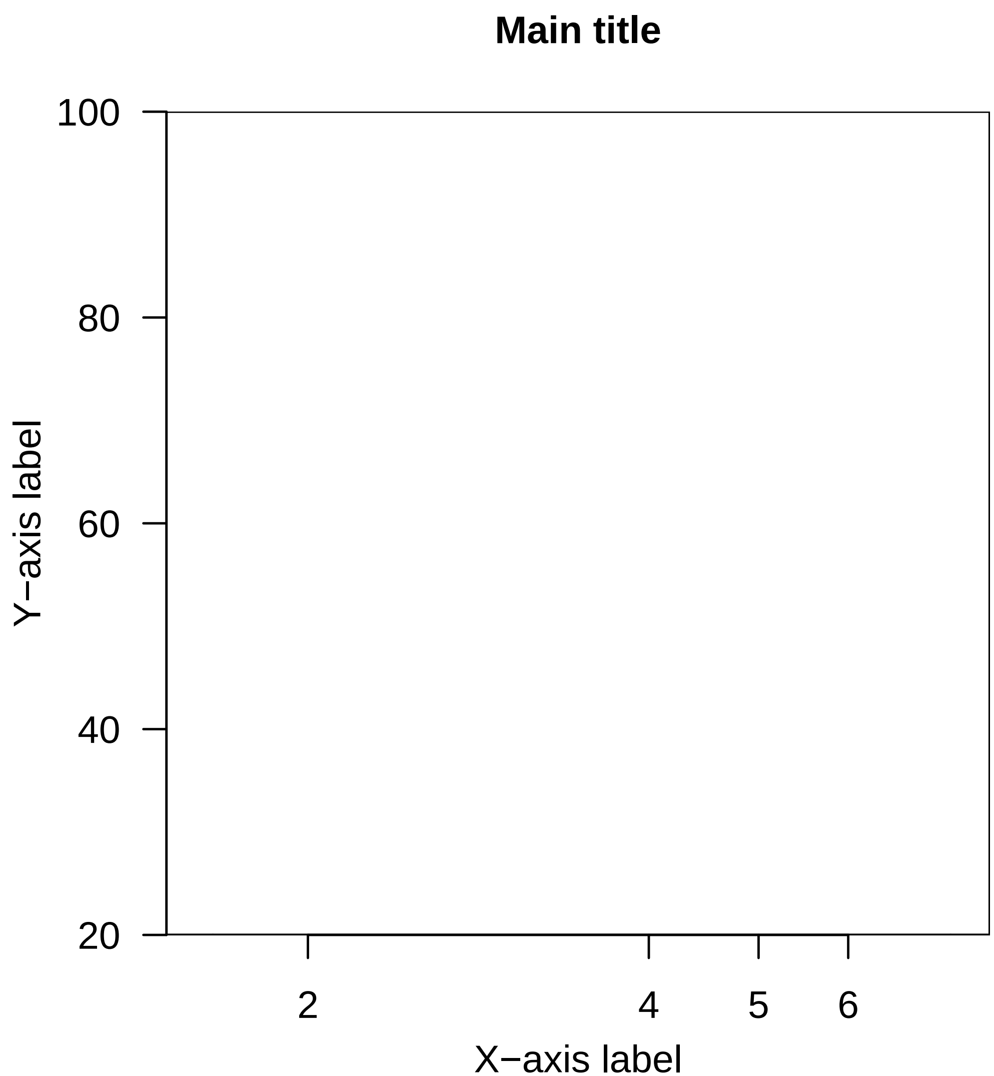
Note this is not an "axis system".

We have drawn no axes.

If we wanted to draw axes denoting this coordinate space, they would lie off the screen by definition, because the coordinate system is the screen

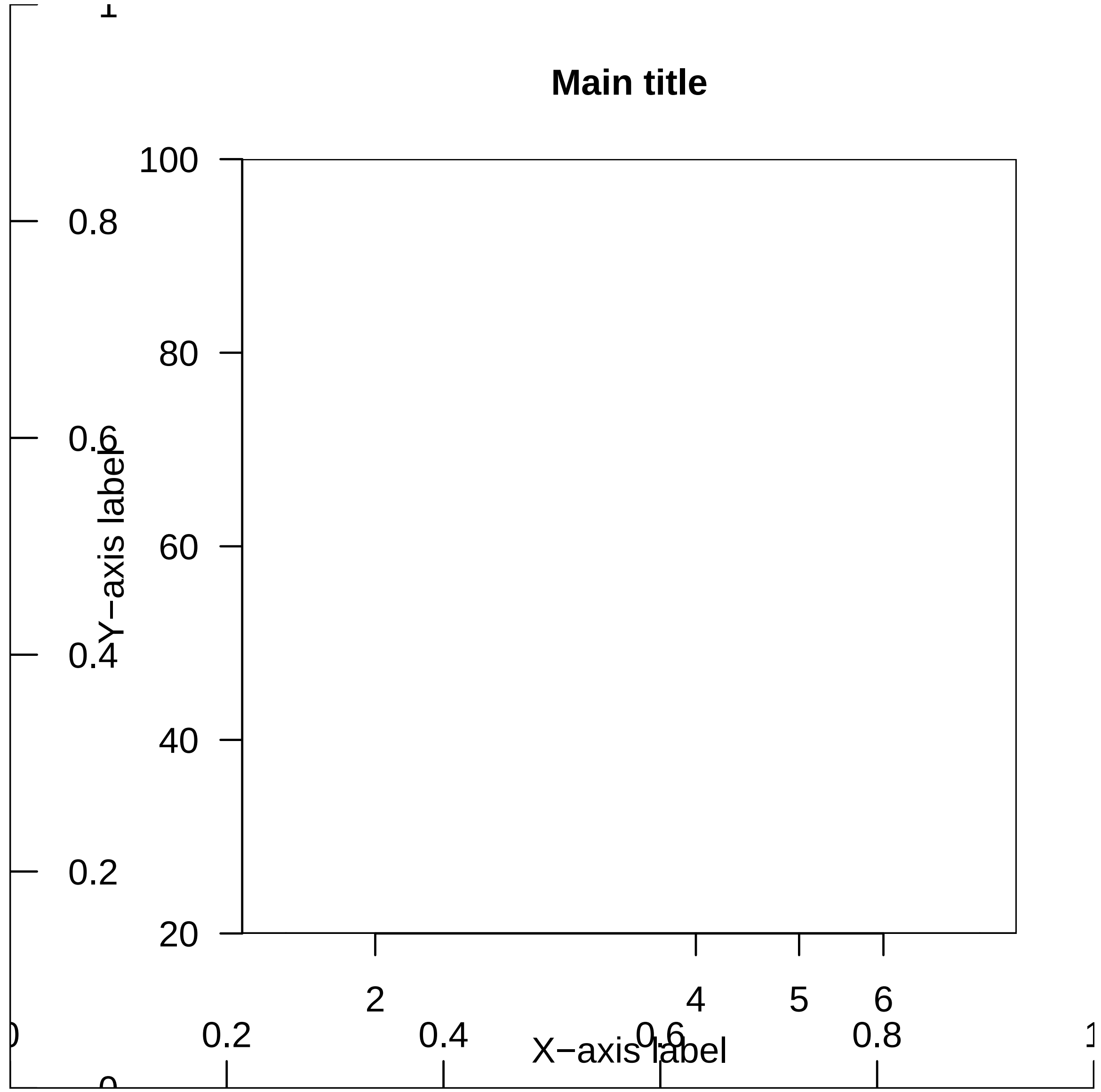
This coordinate system complete not just for 2D images, but for representing "3D" images (ie 2D with false perspective), or for showing movies, or for interactive displays.

Coordinate systems



Are the axes controlling the plot? Or just added ornamentation?

Coordinate systems



Axes don't control anything. Like everything else, axes are drawn on the canvas

Everything is line and color

What can we to plot? Last time we saw dozens of options

But really, there are just two: line and color

We can build anything from these elements

- Drawing lines:

  `lines(x,y,...)`

  Note `...` can include `col`, `lty`, `lwd`, etc.

  Can even alter the style of the line endings

Everything is line and color

- Drawing filled shapes: `polygon(x,y,col="red",border=NA,...)`

  This draws a red polygon with vertices at (x, y).

  Need to set the `col` and `border` as above to get a plain shape

  If we were hardcore, that would be enough. we could draw anything, even letters and glyphs from `lines()` and `polygon()`

  But that would be a real pain.

Add two more primitive commands

- Drawing glyphs:

  `points(x,y,...)`

  Note that `...` can include `col`, `pch`, etc.

- Drawing text: `text(x,y,labels,...)`

  Note `...` can include `col`, `xpd`, `srt`, etc.

  Useful: `offset` moves the label a set amount (to position under a glyph)

Programming tips

The best programs are:

- stand-alone functions
- use clear, consistent variable names
- generalized. variables should be allowed to vary

Justify to yourself any numerical constants or strings hard-coded.

Programming tips

```
#Don't do this:
example <- function(x,
                    y) {
 points(x=x,
        y=y,
        col="blue"
       )
}

#Do this:
example <- function(x,
                    y,
                    col="blue") {
 points(x=x,
        y=y,
        col=col
       )
}
```

Programming tips

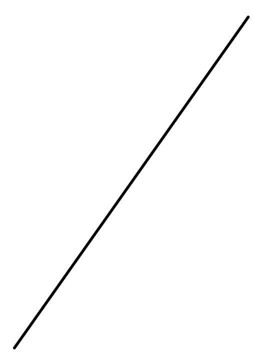
More advice:

- Comment on blocks or lines of code
- Think about making your code extensible (hard)
- Think about how your code will interact with other code (hard)
- Be realistic:
  do *just* enough programming to make yourself most efficient as a scientist

The `base` system: Example

```
# Building traditional R graphics from primitives
x11()          # Opens a graphics window (technically, a device)
plot.new()     # Clears the graphics screen

# Let's draw a line
lines( x=c(0,0.25), y=c(0,0.5) )

# We connected the points (0,0) and (0.25,0.5)
```

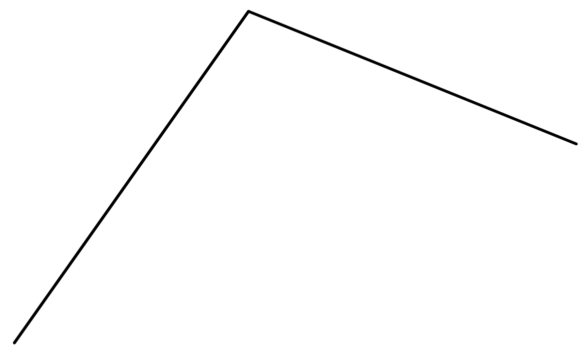
The plot so far



The `base` system: Example

```
# Okay, now let's draw a kinky line
lines( x=c(0,0.25,0.6), y=c(0,0.5,0.3) )
```

The plot so far

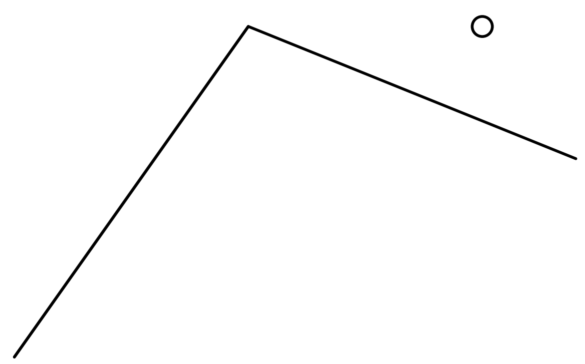


We connected the points (0,0), (0.25,0.5), and (0.6,0.3)
Using lines() we can draw any shape

The `base` system: Example

```
# What if we want a point?
points(x=0.5, y =0.5)
```

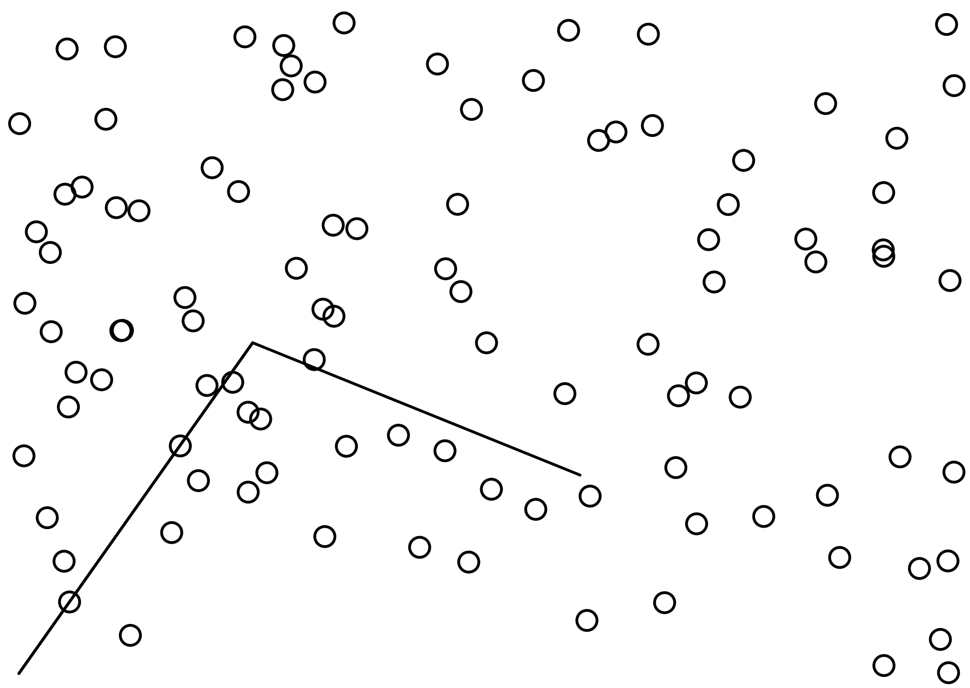
The plot so far



The `base` system: Example

```
# Or a lot of points?
points(x = runif(100), y = runif(100) )
```

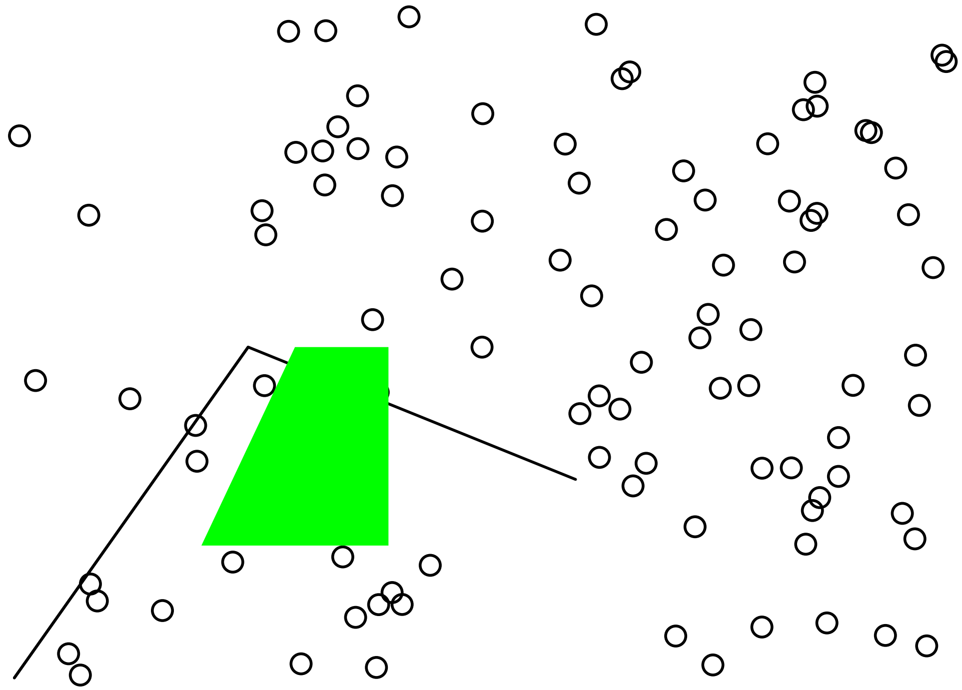
The plot so far



The `base` system: Example

```
# Or a polygon?

xpoly <- c(0.2,0.4,0.4,0.3)
ypoly <- c(0.2,0.2,0.5,0.5)

polygon(x=xpoly,
        y=ypoly,
        col="green",
        border=NA
        )
```

The plot so far

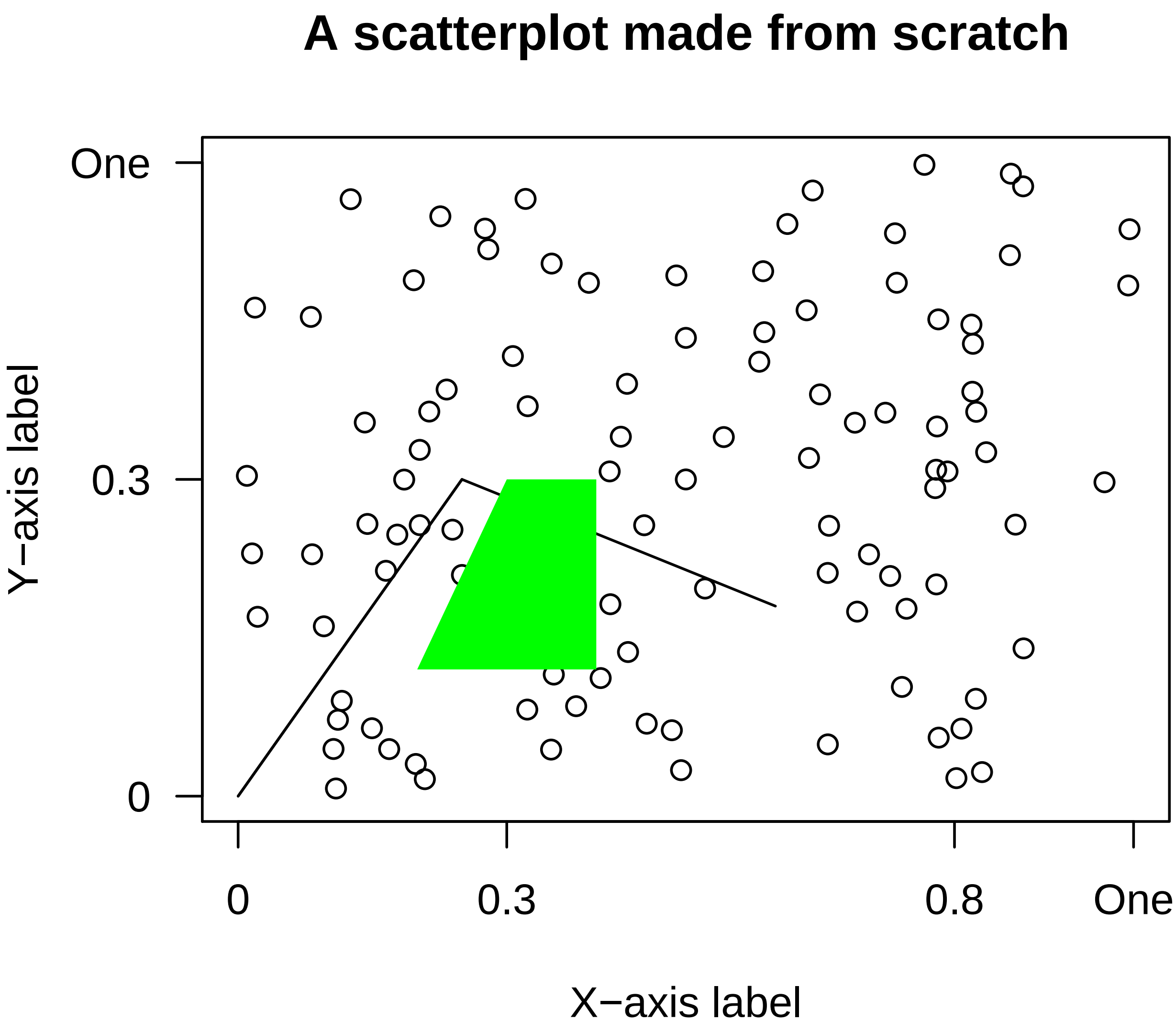


Notice that it covers everything else. R is "Painter-style"
Plot polygons first. May use alpha transparency (pdf only)

The `base` system: Example

```
# We can draw the axes at any time
axis(side=1,                         # 1 = x.  Lovely
     at=c(0,0.3,0.8,1),              # Where the ticks are
     labels=c(0,0.3,0.8,"One")   # What the ticks say
     )

axis(side=2,                         # 2 = y.  Obviously
     at=c(0,0.5,1),
     labels=c(0,0.3,"One"),
     las=1                           # rotate labels
     )

title(main = "A scatterplot made from scratch",
      xlab = "X-axis label",
      ylab = "Y-axis label"
      )

box()
```

The final plot



The `grid` system

"Traditional" `R` Graphics are fairly powerful

. . . As long as you only want to make one graphic, with a single coordinate system

Plotting multiple graphs, or plotting "in the margin" is difficult

Workarounds exist, but a package with powerful low level control of multiple plotting regions would be better

If you are planning to develop new graphical software in R, I recommend using `grid` as your toolkit

The `grid` system

3 things to remember:

- You can create a "plotting region" (with implicit coordinates & axes) anywhere on the canvas
- You can nest these plotting regions, producing a hierarchical graphical object
- You can reference (and plot to) points with respect to any plotting region using any system of measurement

Viewports

A grid plotting region is called a viewport

Some key commands:

`pushViewport()`, `upViewport()`, `downViewport`

What can you do with viewports?

- Create separate plotting regions: Grids of plots
- Fine control of margins
- Plots inside plots
- Even plots inside of plotting symbols

Units in the `grid` system

`grid` needs to be told the unit of things it plots

Instead of `points(x = 0.5, y = 0.25)` use

```
grid.points( x = unit(0.5, "native"), y = unit(0.25, "native") )
```

Some units available:

`native`	Based on the current x, y scales (e.g., your data)
`npc`	Treats the current viewport as (0,0) to (1,1)
`inches`	This and other physical unit available, given device
`strwidth`	Multiples of the width of a given string
`strheight`	Multiples of the height of a given string
`null`	In layouts, any remaining space is divided among nulls

The last three are very powerful

`unit( 1, "strwidth", "this string" )` creates a unit as wide as the text "this string"

Can't `c()` on `unit()` terms. Use `unit.c()` instead

Primitives in the `grid` system

Plot as usual.

Except you need to use the grid packages commands.

Traditional graphics commands don't work in grid!

Use instead

```
grid.lines()
grid.polygon()
grid.points()
grid.text()
etc
```

Primitives in the `grid` system

Let's look at an example and an alternative:

```
grid.points(x, y,
            pch = 1,
            size = unit(1, "char"),
            default.units = "native",
            name = NULL,
            gp=gpar(),
            draw = TRUE,
            vp = NULL)

x <- pointsGrob(x, y,
                pch = 1,
                size = unit(1, "char"),
                default.units = "native",
                name = NULL,
                gp=gpar(),
                vp = NULL)
```

`grid` graphics parameters

Grid replaces `par` with `gpar`

Near complete list (from `help(gpar)`):

col	Colour for lines and borders.
fill	Colour for filling rectangles, polygons, ...
alpha	Alpha channel for transparency
lty	Line type
lwd	Line width
cex	Multiplier applied to fontsize
lineend	Line end style (round, butt, square)
linejoin	Line join style (round, mitre, bevel)
linemitre	Line mitre limit (number greater than 1)
fontsize	The size of text (in points)
fontfamily	The font family
fontface	The font face (bold, italic, ...)
lineheight	The height of a line as a multiple of the size of text

Other important `grid` commands

layout	Makes a layout of viewports
editGrob	Edits an existing graphical object
unit.length	Returns the length of a unit

A longer `grid` example

Let's plot a regression line and shaded confidence envelope

Grid makes most sense if you're planning to:

- Design an unusual graphic
- Write a function for generic use

This example really isn't either; so we'll use lot of code for a little result

An example `grid` session

Start by loading some data:

```
rm(list=ls())

file <- "iver.csv";
data <- read.csv(file,header=TRUE);
attach(data)

y <- povred
x <- lnenp
```

An example `grid` session

. . . and some helper functions

```
# Here's an effort at a color lightener that could use work
lighten <- function(col,
                    pct=0.75,
                    alpha=1){
  if (abs(pct)>1) {
    print("Warning:  Error in lighten; invalid pct")
    pcol <- col2rgb(col)/255
  } else {
    col <- col2rgb(col)/255
    if (pct>0) {
      pcol <- col + pct*(1-col)
    } else {
      pcol <- col*pct
    }
  }
  pcol <- rgb(pcol[1],pcol[2],pcol[3],alpha)
  pcol
}
```

An example `grid` session

. . . and some more helper functions

```
# Sort a matrix on multiple columns
sortmc <- function(Mat, Sort, decreasing=FALSE)
{
  if (decreasing) direction <- -1 else direction <- 1
  m <- do.call("order",
               as.data.frame(direction*Mat[, Sort,drop=FALSE])
              )
  Mat[m, ,drop=FALSE]
}
```

An example `grid` session

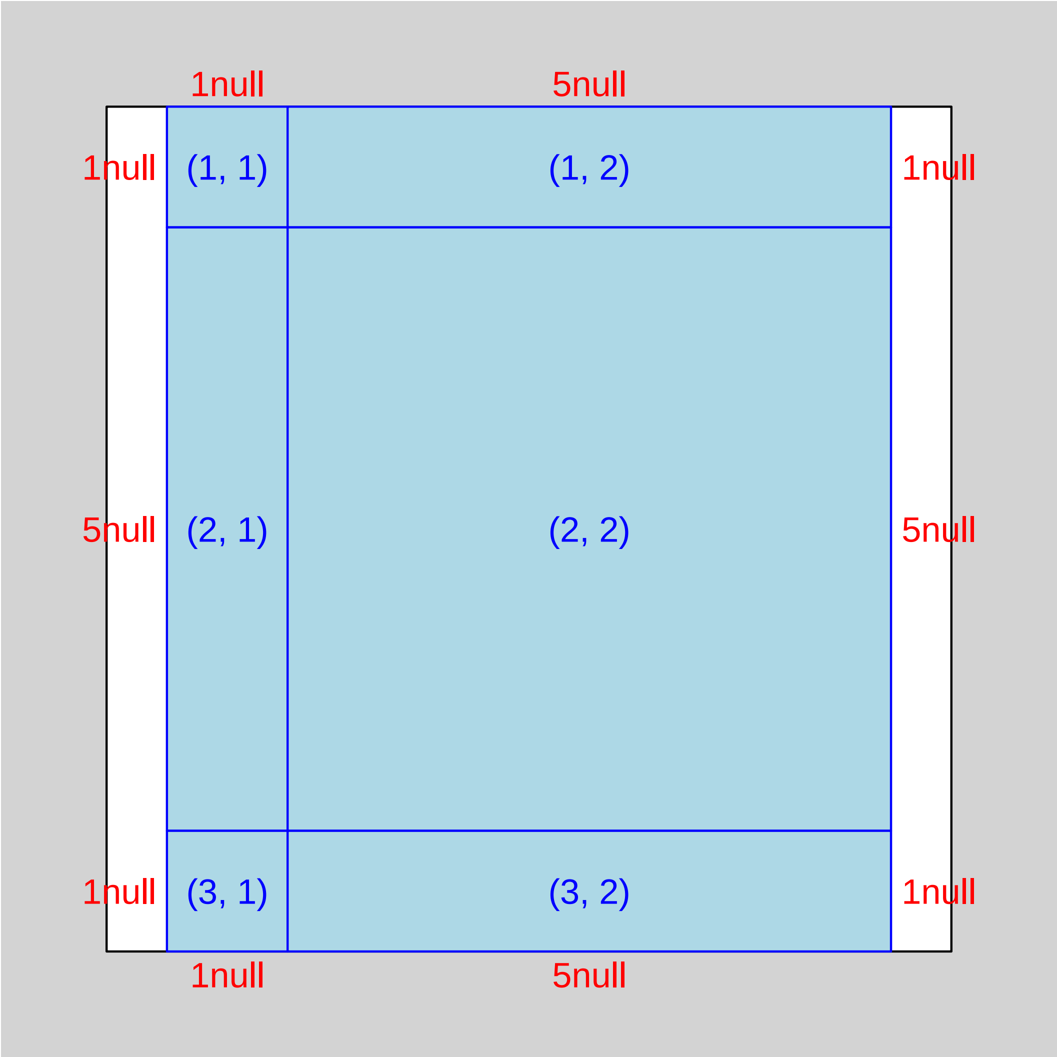
```
# MM-estimator fitting
mmest.fit <- function(y,x,ci=0.95) {
  require(MASS)
  dat <- sortmc(cbind(y,x),2,decreasing=FALSE)
  x <- dat[,2]
  y <- dat[,1]
  result <- rlm(y~x,method="MM")
  print(result)
  fit <- list(x=x)
  fit$y <- result$fitted.values
  fit$lower <- fit$upper <- NULL
  if (length(na.omit(ci))>0)
    for (i in 1:length(ci)) {
      pred <- predict(result,interval="confidence",level=ci[i])
      fit$lower <- cbind(fit$lower,pred[,2])
      fit$upper <- cbind(fit$upper,pred[,3])
    }
  fit
}
```

An example `grid` session

. . . Now we initialize the plotting area

```
library(grid)
usr <- c(1,8,20,100)

pdf("testgrid.pdf",width=5,height=5)

# Set up the layout
# This is optional:  we could instead put viewports
# anywhere we want
overlay <- grid.layout(nrow=3,
                       ncol=2,
                       widths=c(1,5),
                       heights=c(1,5,1),
                       respect=TRUE)

# Push the initial viewport, which includes a tree of
# viewports according to layout
pushViewport(viewport(layout=overlay)
             )
```

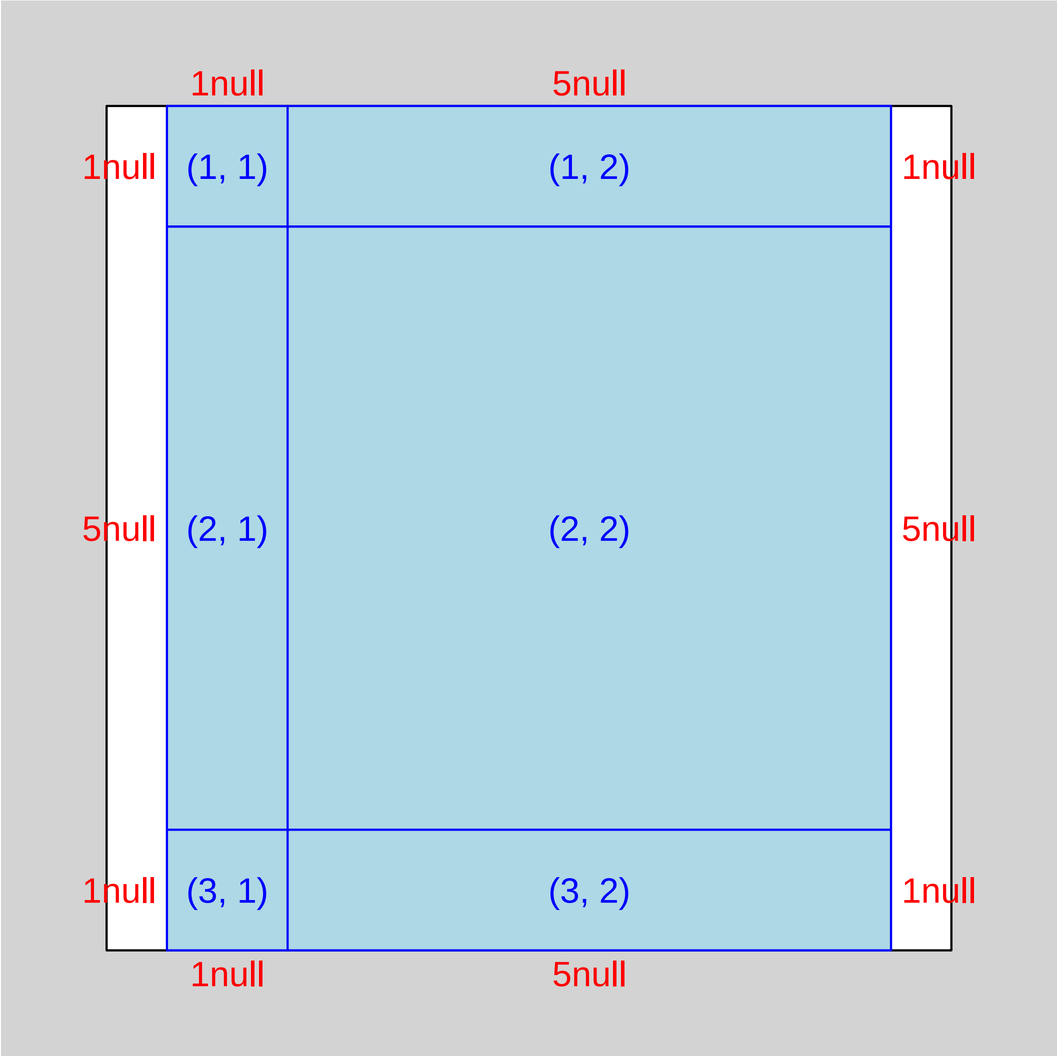
The layout we made



To make this display: `grid.show.layout(overlay)` if our grid.layout is `overlay`

Don't confuse the grid command `grid.layout()` with the base command `layout()`

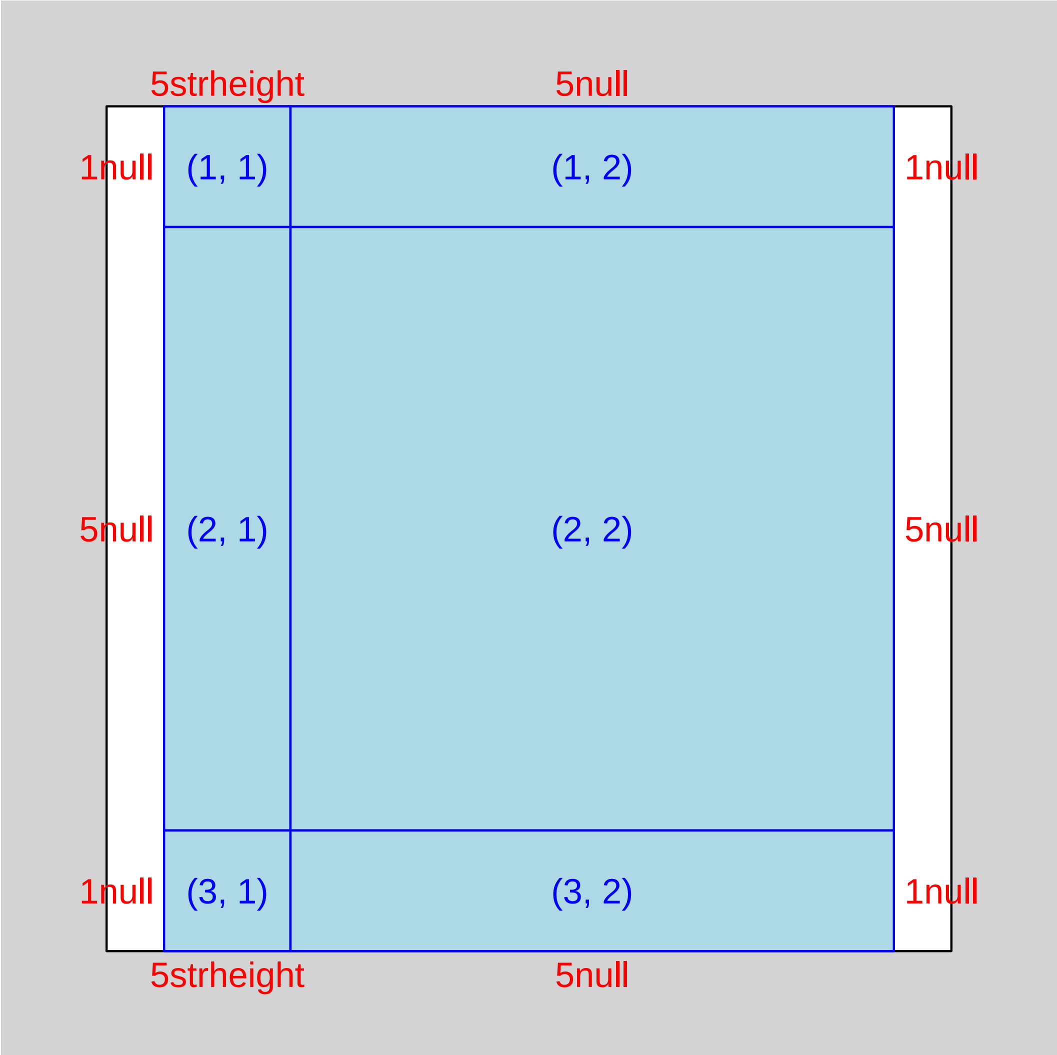
The layout we made



Note the `null` units. The graphic is 6 nulls high and 5 nulls wide

Null is calculated (e.g., in inches) given any fixed layout widths and the device dimensions

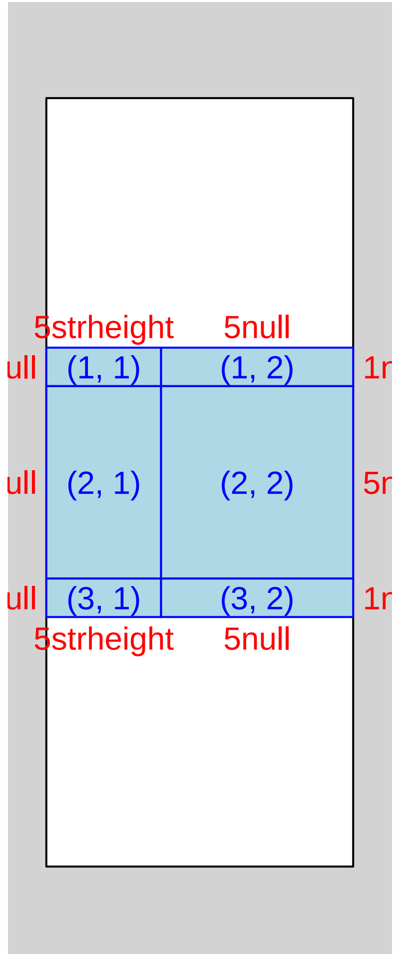
A different layout



Suppose we set the rightmost column to be the $5\times$ the height of the string "Y axis label"

With a 5 inch wide pdf device, this is the resulting layout

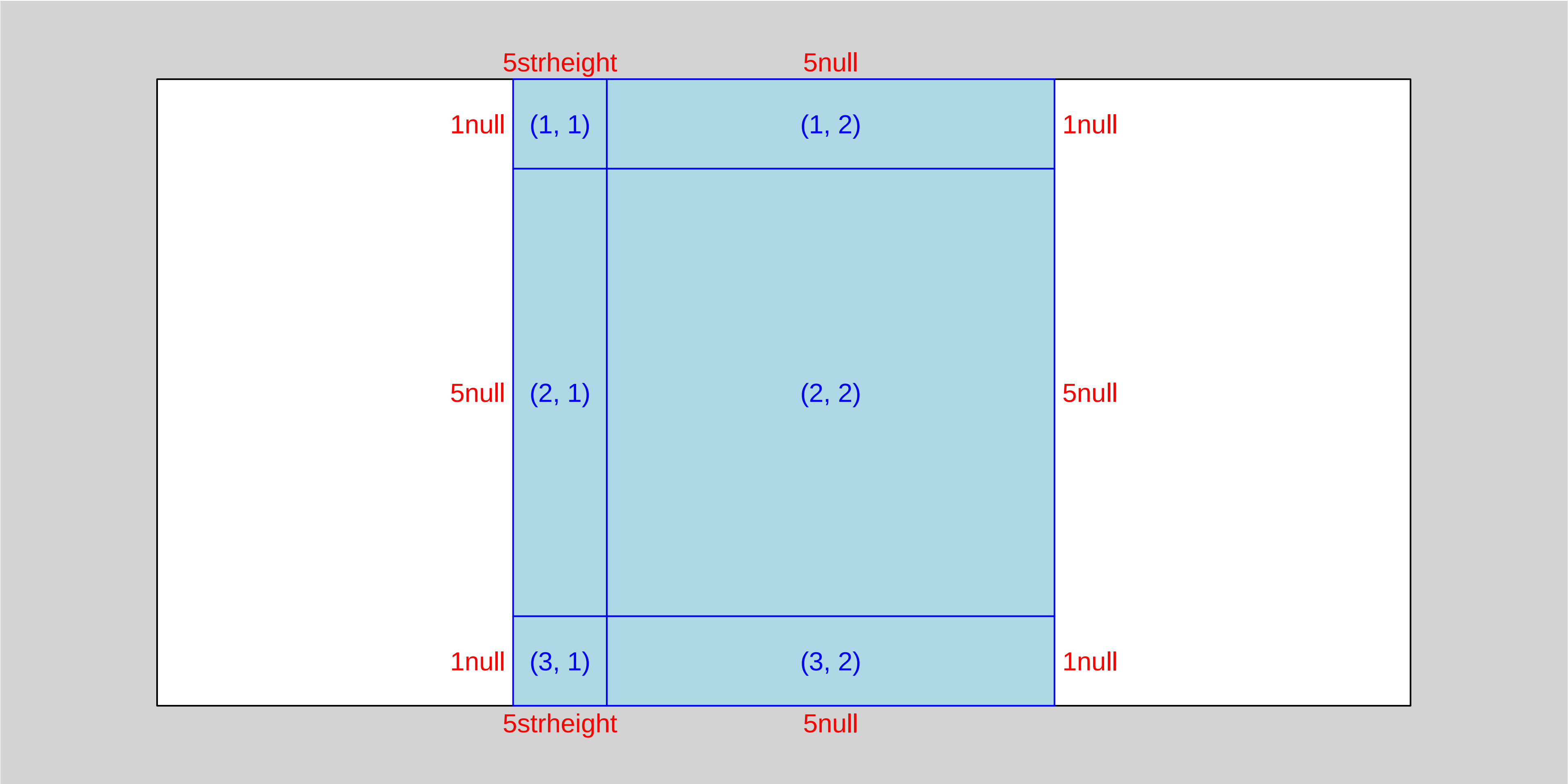
A different layout



But if we narrow the device to 2 inches, look what happens to the nulls

They shrink to fit!

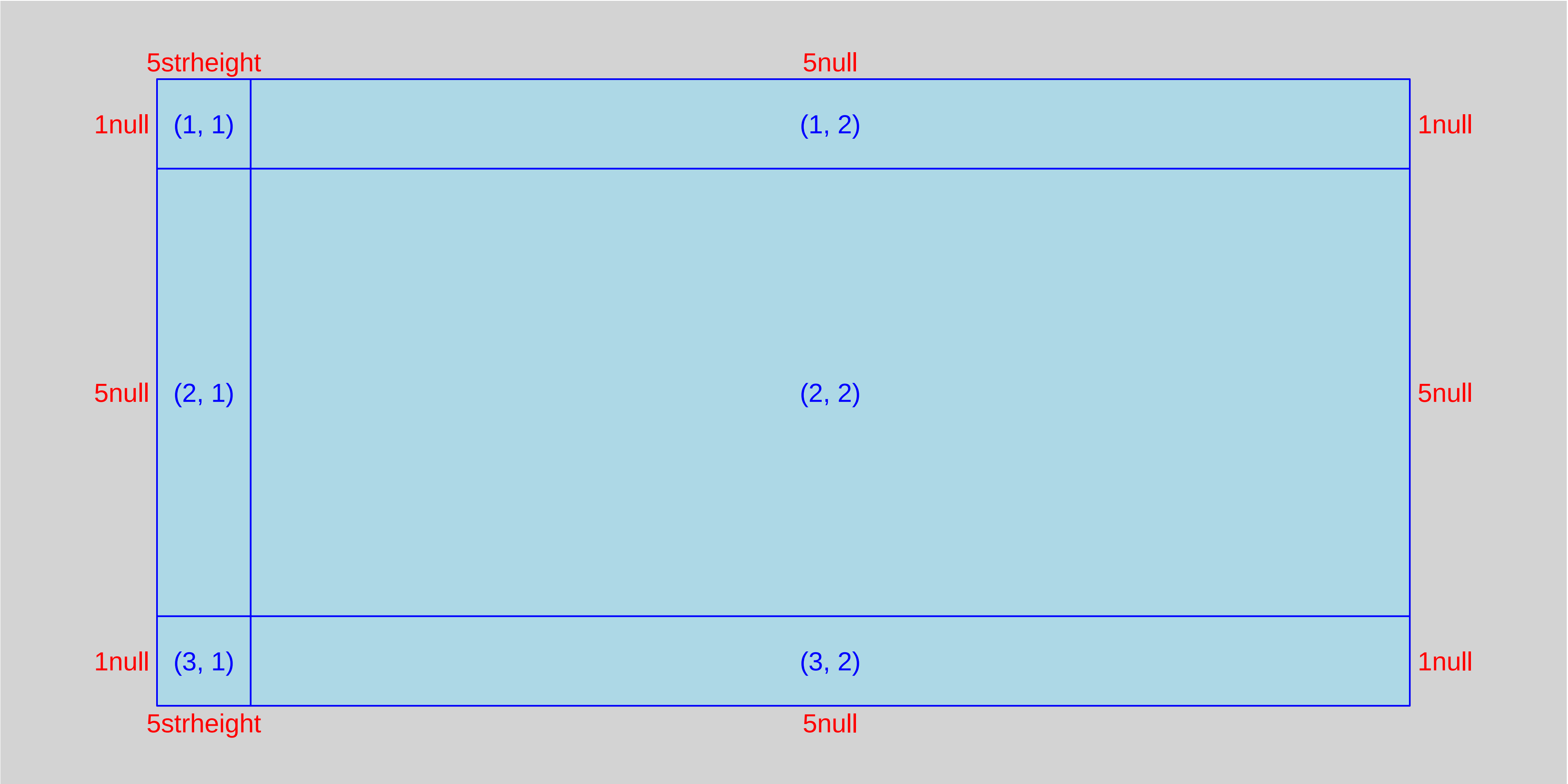
A different layout



Give the device more space—say, 10 inches—they expand

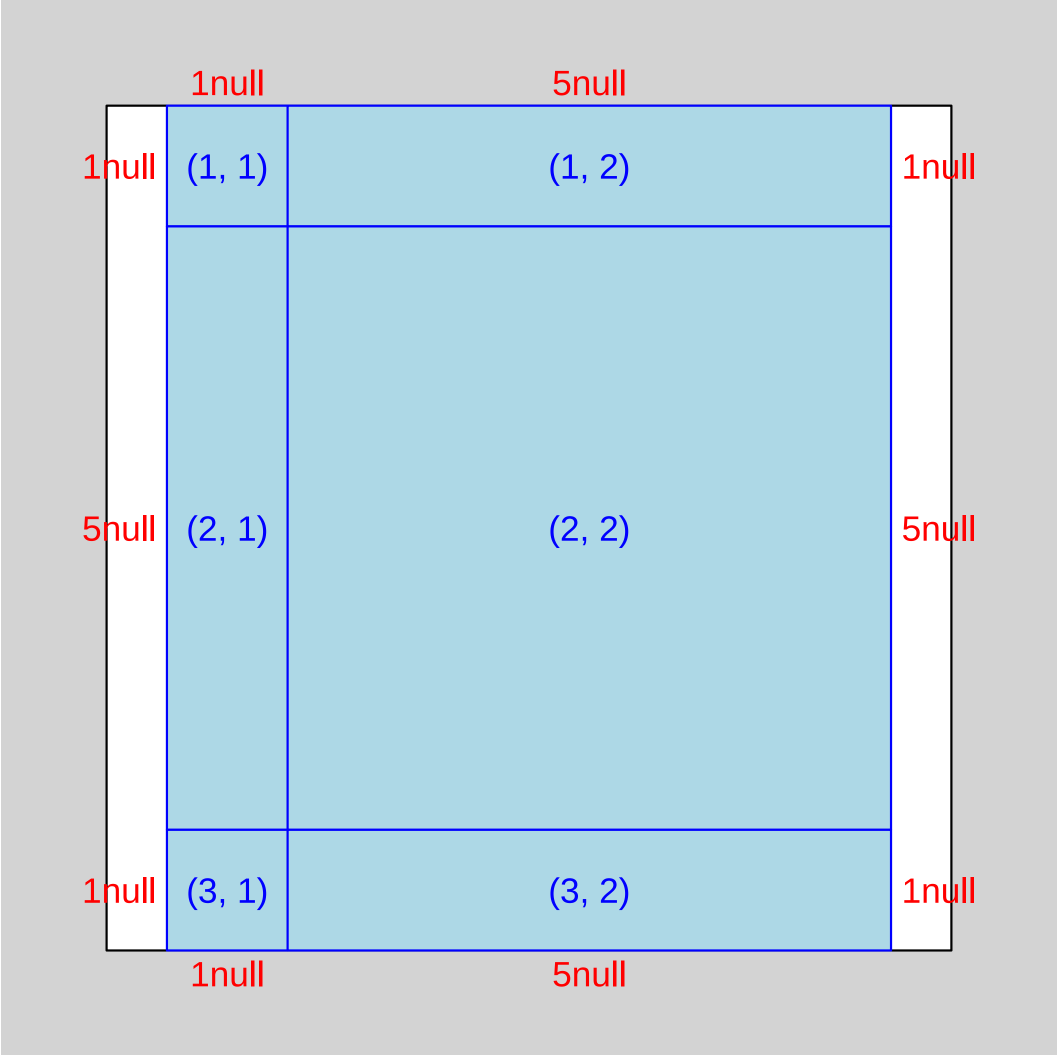
The nulls have been bound to the same size vertically & horiztonally because we set `respect=TRUE` in `layout()`

A different layout



Setting `respect=FALSE` allows the nulls to fill the whole device

The layout we made



Okay, back to the original layout for now

An example `grid` session

```
# Push the main title viewport
pushViewport(viewport(layout.pos.col=2,
                      layout.pos.row=1,
                      xscale=c(0,1),
                      yscale=c(0,1),
                      gp=gpar(fontsize=12),
                      name="maintitle",
                      clip="on"
                      )
             )

# Note the use of a grid primitive
grid.text("Main title",
          x=unit(0.5,"npc"),     # Why NPC?
          y=unit(0.5,"npc"),
          gp=gpar(fontface="bold")
          )

# Go back up to the top level Viewport
upViewport(1)
```

The plot so far

An example grid session

```
# Go to the y-axis title viewport
pushViewport(viewport(layout.pos.col=1,
                      layout.pos.row=2,
                      xscale=c(0,1),
                      yscale=c(0,1),
                      gp=gpar(fontsize=12),
                      name="ytitle",
                      clip="on"
                      )
             )

grid.text("Y-axis label",
          x=unit(0.15,"npc"),
          y=unit(0.5,"npc"),
          rot=90
          )

upViewport(1)
```

The plot so far

An example grid session

```
# Go to the x-axis title viewport
pushViewport(viewport(layout.pos.col=2,
                      layout.pos.row=3,
                      xscale=c(0,1),
                      yscale=c(0,1),
                      gp=gpar(fontsize=12),
                      name="xtitle",
                      clip="on"
                      )
             )

grid.text("X-axis label",
          x=unit(0.5,"npc"),
          y=unit(0.25,"npc")
          )


upViewport(1)
```

The plot so far

Main title

Y−axis label

X−axis label

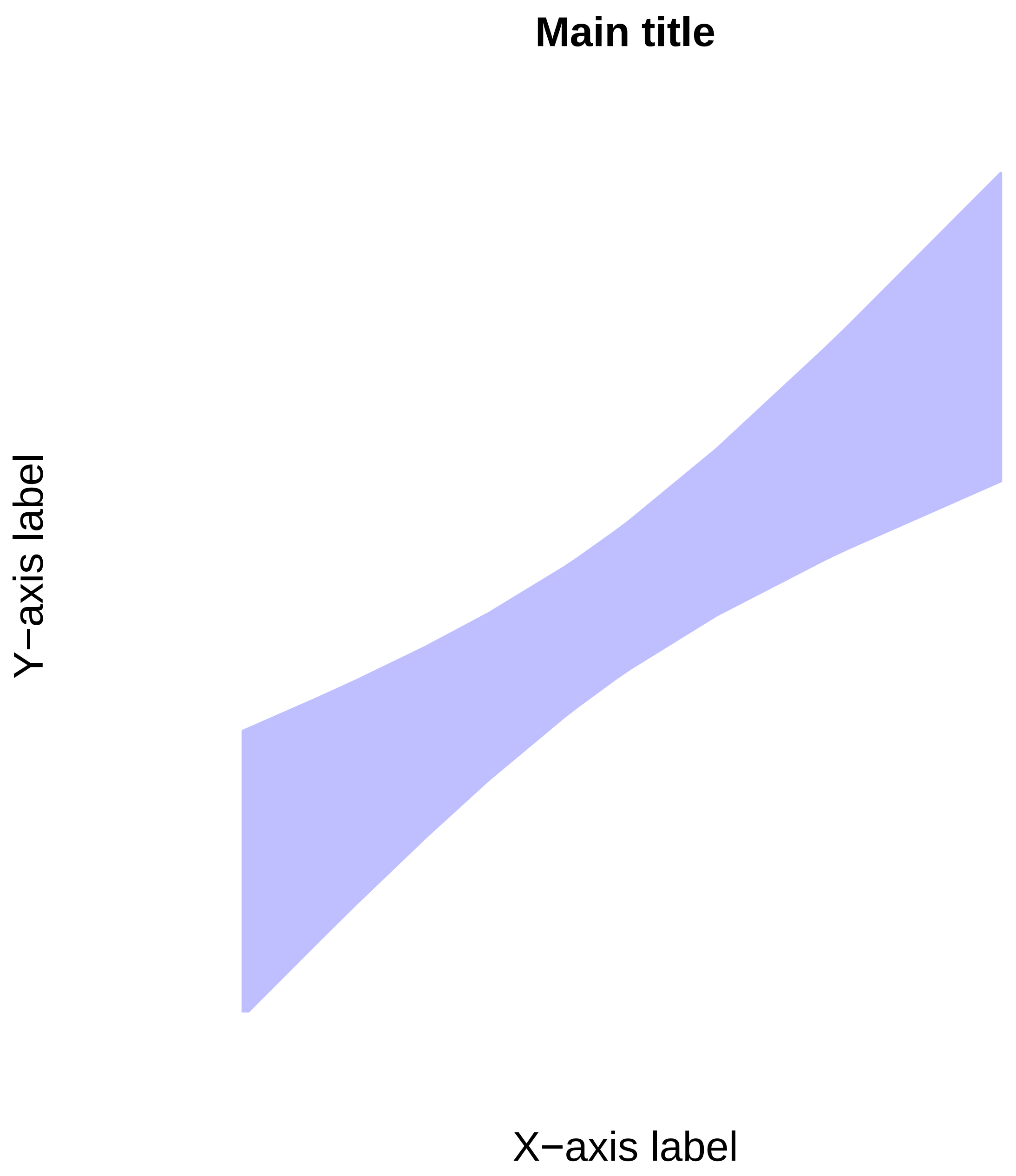
An example `grid` session

```
# Push the main plot Viewport.  Note the scales
pushViewport(viewport(layout.pos.col=2,
                      layout.pos.row=2,
                      xscale=c(usr[1],usr[2]),
                      yscale=c(usr[3],usr[4]),
                      gp=gpar(fontsize=12),
                      name="mainplot",
                      clip="on"
                      )
             )

# get the fit from the data
fit <- mmest.fit(y,x,ci=0.95)

# Make the x-coord of a confidence envelope polygon
xpoly <- c(fit$x,
           rev(fit$x),
           fit$x[1])
```

An example grid session

```
# Make the y-coord of a confidence envelope polygon
ypoly <- c(fit$lower,
           rev(fit$upper),
           fit$lower[1])

# Choose the color of the polygon
pcol <- lighten("blue")

# Plot the polygon first
grid.polygon(x=xpoly,
             y=ypoly,
             gp=gpar(col=pcol,
               border=FALSE,
               fill=pcol
               ),
             default.units="native")
```

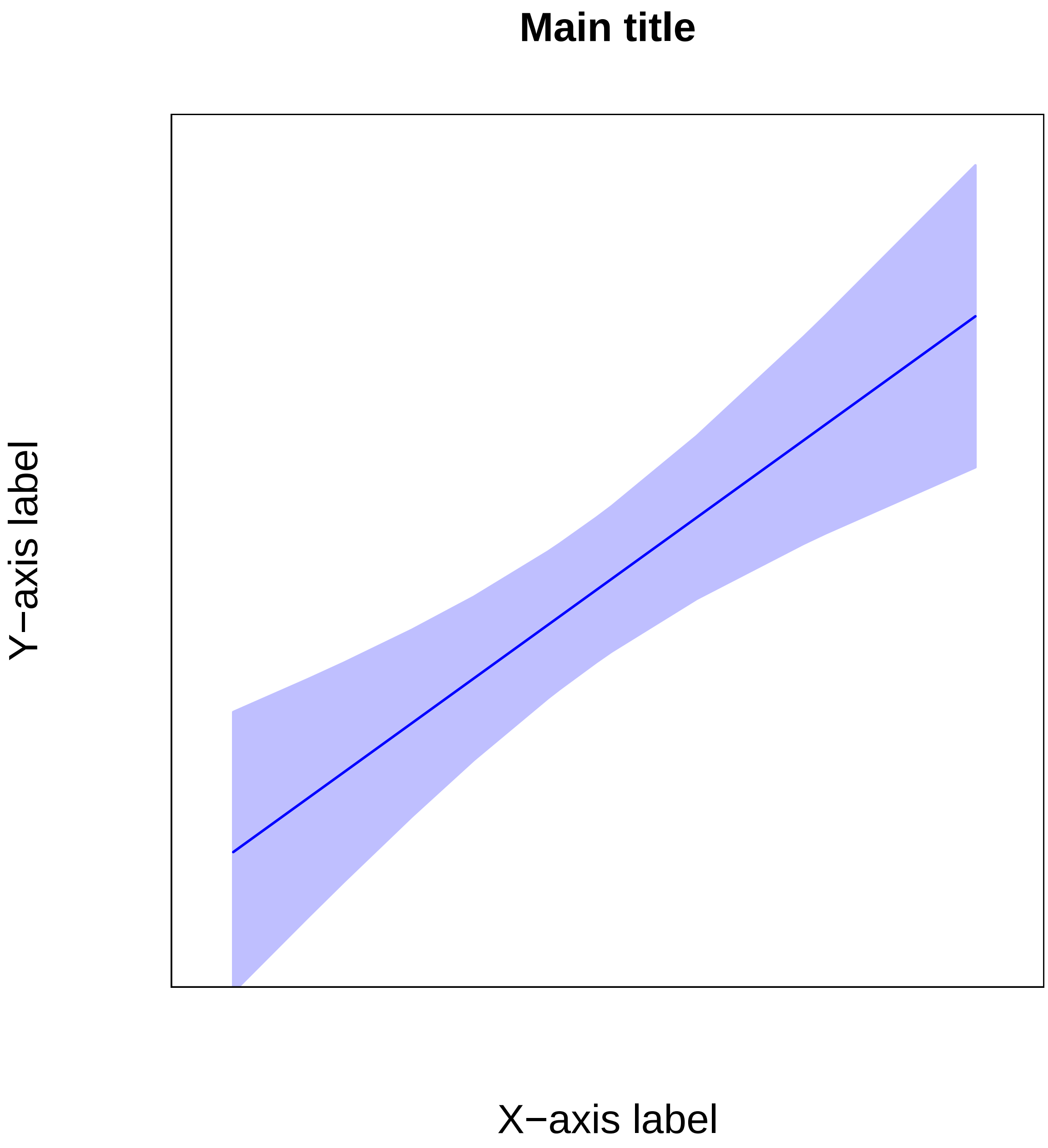
The plot so far



An example `grid` session

```
# Then plot the line
grid.lines(x=fit$x,
           y=fit$y,
           gp=gpar(lty="solid",
             col="blue"),
           default.units="native")

# Finally add a box around the plot
grid.rect(gp=gpar(linejoin="round"))

upViewport(1)
```

The plot so far



An example grid session

```
# Wait!  We haven't made axes!
# Axes plot facing out of the Viewport
pushViewport(viewport(layout.pos.col=2,
                      layout.pos.row=2,
                      xscale=c(usr[1],usr[2]),
                      yscale=c(usr[3],usr[4]),
                      gp=gpar(fontsize=12),
                      name="mainplot",
                      clip="off"       # Needed for ticks to show
                      )
             )
# let's make the axis, but not draw it yet
yaxis <- yaxisGrob(at = c(20,40,60,80,100),   # Where to put ticks
          label = TRUE,      # Argh!  Only takes logical values
          main = TRUE        # Left axis (TRUE) or right axis (FALSE)
          #gp = gpar(),      # Any gpars to change
          )
```

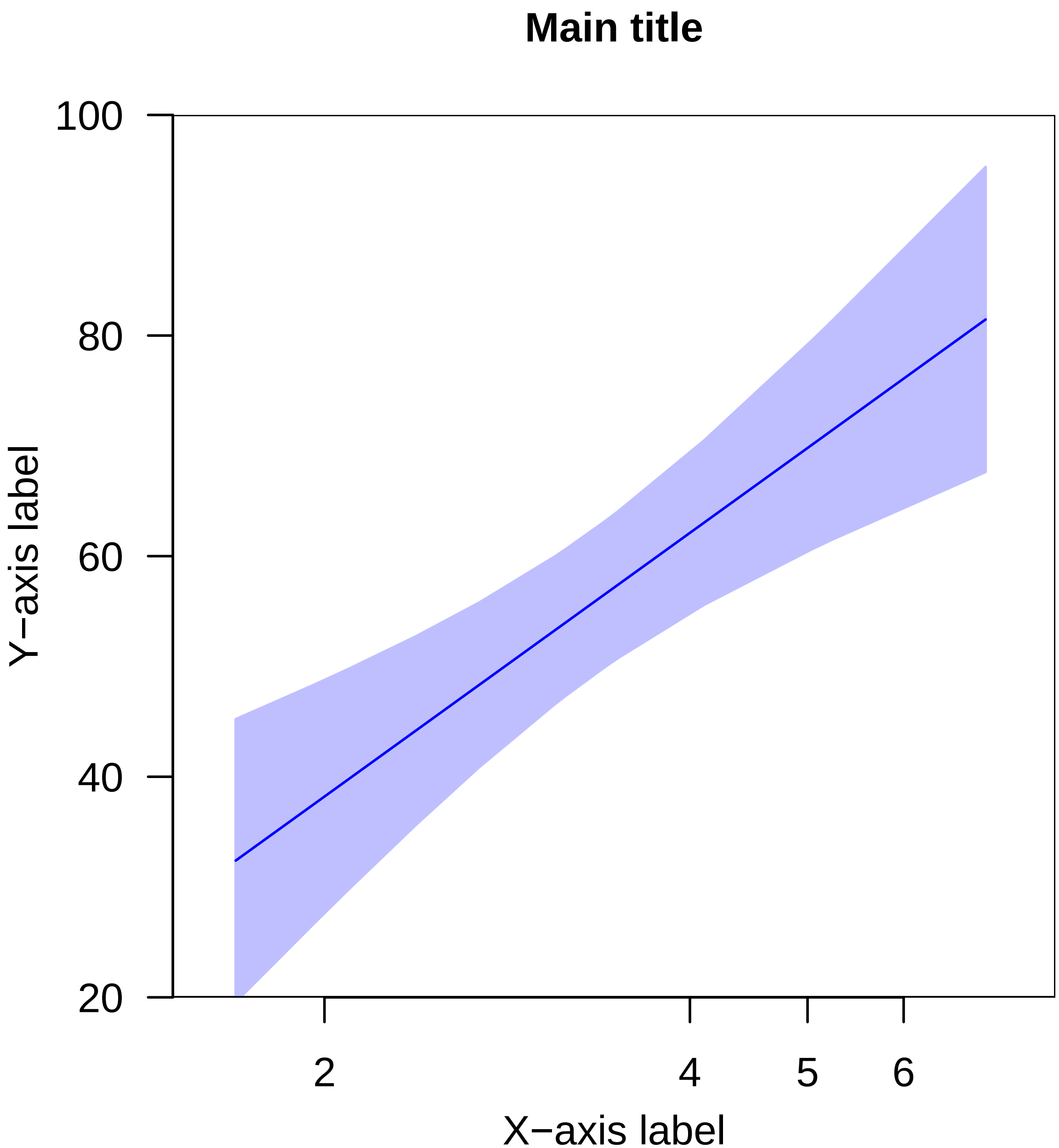
An example `grid` session

```
# Now we draw
grid.draw(yaxis)

# x-axis is tricky.  Log scaling
xaxis <- xaxisGrob(at = log(c(2,4,5,6)),   # Where to put ticks
          label = TRUE,       # Argh!  Only takes logical values
          main = TRUE         # Bottom axis (TRUE) or top axis (FALSE)
          #gp = gpar(),       # Any gpars to change
          )

# Edit the (undrawn) axis grob to have right ticks
xaxis <- editGrob(xaxis,
                  gPath("labels"),
                  label=c(2,4,5,6)
                  )

# Now draw it
grid.draw(xaxis)

upViewport(1)
```

An example `grid` session

```
# Done! Whew
dev.off()
```

The final product



Who uses grid?

Two packages written in grid:

Lattice

Tile

It's a shame grid wasn't original to R

Clearly superior to base graphics, but a bit steeper learning curve

Multiple plots

Most social science graphics should be small multiples

We have multidimensional data; usually we make many comparisons

Our graphics package should make small multiples easy

`R` does not.

It's possible to make multiplot layout in the base package

Use the `mfrow` `mfcol`, or `mfg` options in `par`

Use the layout command

But these methods require **lots** of work from the user to look good

One answer: `lattice`

The `lattice` package implements a set of techniques pioneered by Bell Labs/Bill Cleveland.

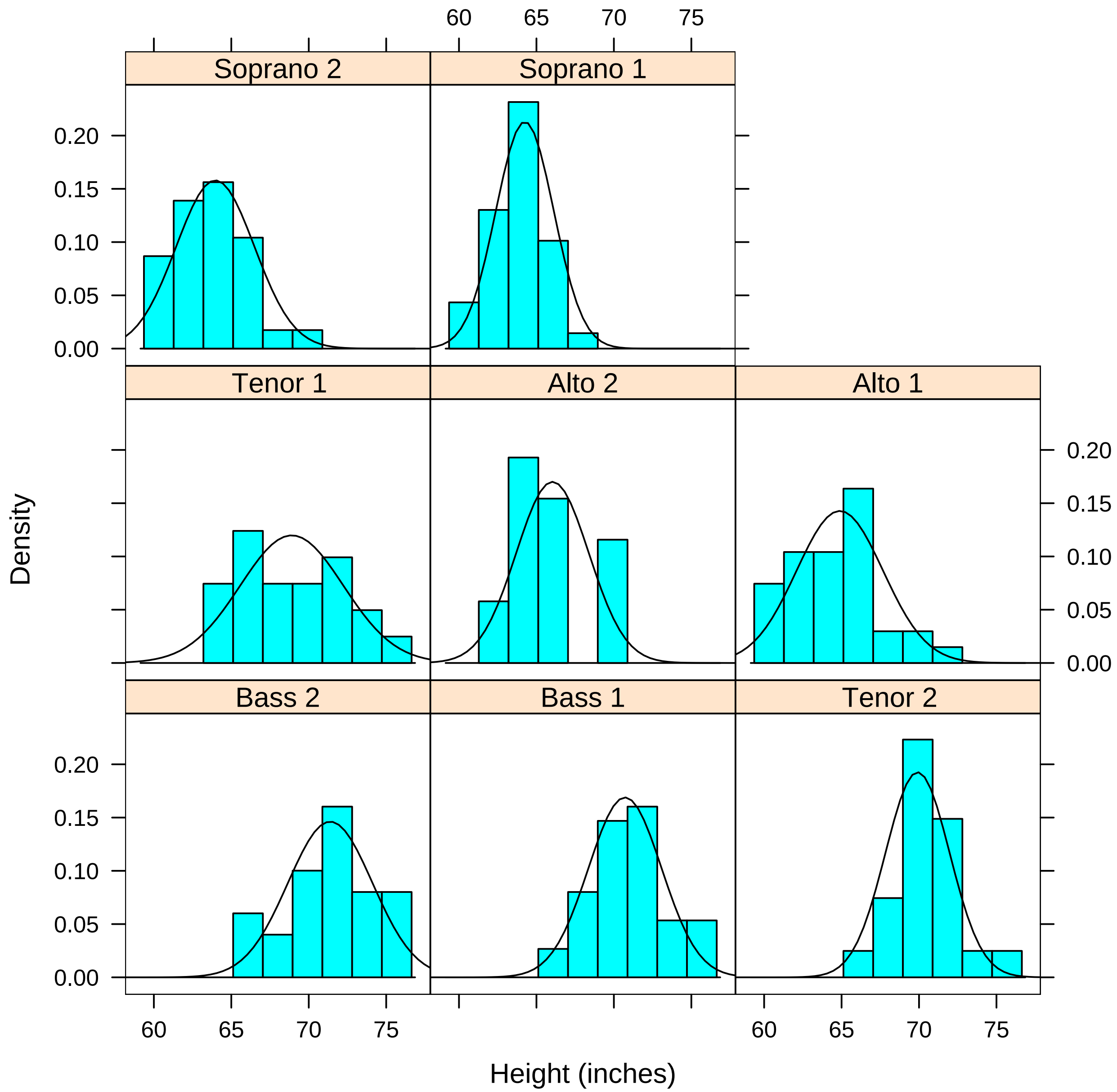
Basic idea: small multiples that show relations between x and y conditioning on z, and perhaps w, etc.

Lattice plots consist of multiple panels of plotted data

The panels are linked to strips which identify a conditioning variable

We saw several examples above. This histogram:

Lattice in action



Key lattice options

```
histogram( ~ height | voice.part, data = singer,
          xlab = "Height (inches)", type = "density",
          panel = function(x, ...) {
             panel.histogram(x, ...)
             panel.mathdensity(dmath = dnorm, col = "black",
                     args = list(mean=mean(x),sd=sd(x))
                              )
             }
          )

dev.off()
```

Notice two trademark elements of lattice:

- the use of a formula to input the data
- the presence of a customizable panel function

Lattice

Key parameters for lattice plots often hide in `panel.XXX()` where `XXX()` is the function of interest

Example: the key parameter for 3D plots (how to spin them) is `screen`, which is documented in `panel.cloud()` only

`par()` doesn't work for lattice.

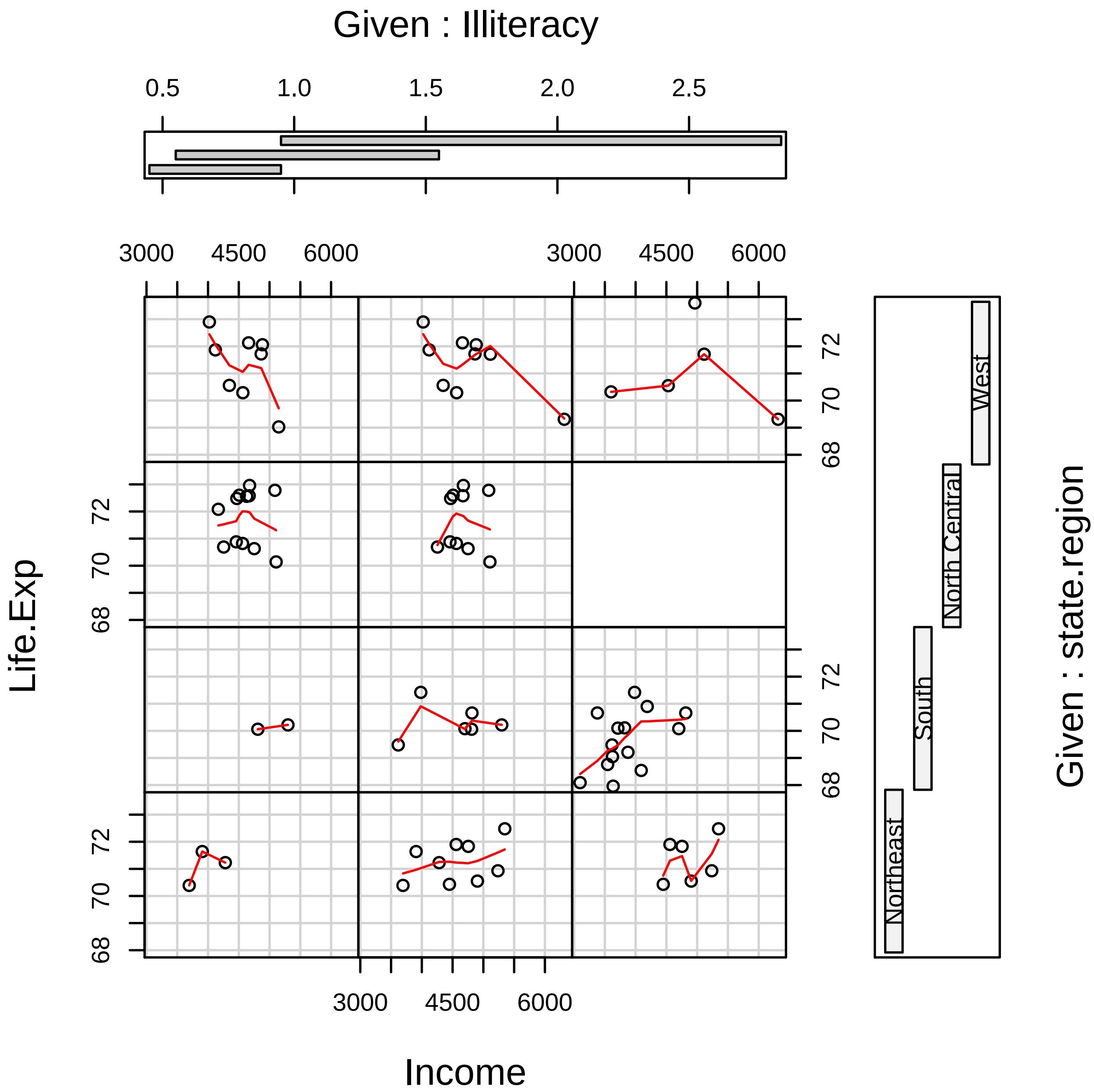
Use `trellis.par.get()` and `trellis.par.set()` to modify lattice parameters

What are the lattice parameters? Mostly undocumented!

`print(trellis.par.get())` gives a list of them, for what it's worth

We'll talk more about lattice next week

Another example, this time from base



Lattice-like graphics in base

```
attach(data.frame(state.x77))
coplot(Life.Exp ~ Income | Illiteracy * state.region,
       number = 3,  # of conditioning intervals
       panel = function(x, y, ...)
                 panel.smooth(x, y, span = 0.8, ...)
       )
```

Notice the use of two conditioning variables

Notice the smoother added by panel