

CSSS 569 · Visualizing Data

GRAPHICAL PROGRAMMING IN R

Christopher Adolph

Department of Political Science

and

Center for Statistics and the Social Sciences

University of Washington, Seattle



Today's outline: Using R graphics functions

Review of R basics

Overview of available high-level plots

Modifying traditional graphics

R graphic devices

Next week: Writing R graphics functions

Philosophy: Start from scratch

Line & color

Annotation

Coordinate systems

General purpose graphics packages to replace base:

The lattice graphics package

The grid graphics package

The ggplot2 graphics package*

Strategy for today

Review basics quickly

Stop and ask for clarification, elaboration, examples

Why R?

Real question: Why programming?

Non-programmers stuck with package defaults

For your substantive problem, defaults may be

- inappropriate (not quite the right model, but “close”)
- unintelligible (reams of non-linear coefficients and stars)

Programming allows you to match the methods to the data & question

Get better, more easily explained results.

Why R?

Many side benefits:

1. Never forget what you did: The code can be re-run.
2. Repeating an analysis n times? Write a loop!
3. Programming makes data processing/reshaping easy.
4. Programming makes replication easy.

Why R?

R is

- free
- open source
- growing fast
- widely used
- the future for most fields

But once you learn one language, the others are much easier

Introduction to R

R is a calculator that can store lots of information in memory

R stores information as “objects”

```
> x <- 2
```

```
> print(x)
```

```
[1] 2
```

```
> y <- "hello"
```

```
> print(y)
```

```
[1] "hello"
```

```
> z <- c(15, -3, 8.2)
```

```
> print(z)
```

```
[1] 15.0 -3.0 8.2
```


Introduction to R

```
> w <- c("gdp", "pop", "income")
> print(w)
[1] "gdp"      "pop"      "income"
>
```

Note the assignment operator, `<-`, not `=`

An object in memory can be called to make new objects

```
> a <- x^2
> print(x)
[1] 2
> print(a)
[1] 4
```

```
> b <- z + 10
> print(z)
[1] 15.0 -3.0  8.2
> print(b)
[1] 25.0  7.0 18.2
```

Introduction to R

```
> c <- c(w,y)
> print(w)
[1] "gdp"      "pop"      "income"
> print(y)
[1] "hello"
> print(c)
[1] "gdp"      "pop"      "income" "hello"
```

Commands (or “functions”) in R are always written `command()`

The usual way to use a command is:

```
output <- command(input)
```

We’ve already seen that `c()` pastes together variables.

A simple example:

```
> z <- c(15, -3, 8.2)
> mz <- mean(z)
> print(mz)
[1] 6.733333
```

Introduction to R

Some commands have multiple inputs. Separate them by commas:

```
plot(var1,var2) plots var1 against var2
```

Some commands have optional inputs. If omitted, they have default values.

```
plot(var1) plots var1 against the sequence {1,2,3,.. . }
```

Inputs can be identified by their position or by name.

```
plot(x=var1,y=var2) plots var2 against var1
```

Entering code

You can enter code by typing at the prompt, by cutting or pasting, or from a file

If you haven't closed the parenthesis, and hit enter, R lets you continue with this prompt +

You can copy and paste multiple commands at once

You can run a text file containing a program using `source()`, with the name of the file as input (ie, in `""`)

I prefer the `source()` approach. Leads to good habits of retaining code.

Data types

R has three important data types to learn now

```
Numeric    y <- 4.3  
Character   y <- "hello"  
Logical     y <- TRUE
```

We can always check a variable's type, and sometimes change it:

```
population <- c("1276", "562", "8903")  
print(population)  
is.numeric(population)  
is.character(population)
```

Oops! The data have been read in as characters, or “strings”. R does not know they are numbers.

```
population <- as.numeric(population)
```

Some special values

| | |
|--------------|------|
| Missing data | NA |
| A “blank” | NULL |
| Infinity | Inf |
| Not a number | NaN |

Data structures

All R objects have a data type *and* a data structure or class

Data structures can contain numeric, character, or logical entries

Important structures:

Vector

Matrix

Dataframe

List

Vectors in R

Vectors in R are simply 1-dimensional lists of numbers or strings

Let's make a vector of random numbers:

```
x <- rnorm(1000)
```

x contains 1000 random normal variates drawn from a Normal distribution with mean 0 and standard deviation 1.

What if we wanted the mean of this vector?

```
mean(x)
```

What if we wanted the standard deviation?

```
sd(x)
```


Vectors in R

What if we wanted just the first element?

```
x[1]
```

or the 10th through 20th elements?

```
x[10:20]
```

what if we wanted the 10th percentile?

```
sort(x)[100]
```

Indexing a vector can be very powerful. Can apply to any vector object.

What if we want a histogram?

```
hist(x)
```

Vectors in R

Useful commands for vectors:

| | |
|--------------------------------|---------------------------------------|
| <code>seq(from, to, by)</code> | generates a sequence |
| <code>rep(x, times)</code> | repeats x |
| <code>sort()</code> | sorts a vector from least to greatest |
| <code>rev()</code> | reverses the order of a vector |
| <code>rev(sort())</code> | sorts a vector from greatest to least |

Matrices in R

Vectors are the standard way to store and manipulate variables in R

But usually our datasets have several variables measured on the same observations

Several variables collected together form a matrix with one row for each observation and one column for each variable

Matrices in R

Many ways to make a matrix in R

```
a <- matrix(data=NA, nrow, ncol, byrow=FALSE)
```

This makes a matrix of $nrow \times ncol$, and fills it with missing values.

To fill it with data, substitute a vector of data for NA in the command. It will fill up the matrix column by column.

We could also paste together vectors, binding them by column or by row:

```
b <- cbind(var1, var2, var3)
```

```
c <- rbind(obs1, obs2)
```

Matrices in R

Optionally, R can remember names of the rows and columns of a matrix

To assign names, use the commands:

```
colnames(a) <- c("Var1", "Var2")  
rownames(a) <- c("Case1", "Case2")
```

Substituting the actual names of your variables and observations (and making sure there is one name for each variable & observation)

Matrices in R

Matrices are indexed by row and column.

We can subset matrices into vectors or smaller matrices

| | |
|------------------------|---|
| <code>a[1,1]</code> | Gets the first element of a |
| <code>a[1:10,1]</code> | Gets the first ten rows of the first column |
| <code>a[,5]</code> | Gets every row of the fifth column |
| <code>a[4:6,]</code> | Gets every column of the 4th through 6th rows |

To make a vector into a matrix, use `as.matrix()`

R defaults to treating one-dimensional arrays as vectors, not matrices

Useful matrix commands:

| | |
|---------------------|--|
| <code>nrow()</code> | Gives the number of rows of the matrix |
| <code>ncol()</code> | Gives the number of columns |
| <code>t()</code> | Transposes the matrix |

Dataframes in R

Dataframes are a special kind of matrix used to store datasets

To turn a matrix into a dataframe (note the extra .):

```
a <- as.data.frame(a)
```

Dataframes always have columns names, and these are set or retrieved using the `names()` command

```
names(a) <- c("Var1", "Var2")
```

You can access a variable from a dataframe directly using `$`:

```
a$Var1
```

Dataframes can also be “attached”, which makes each column into a vector with the appropriate name

```
attach(a)
```

Loading data

There are many ways to load data to R.

I prefer using comma-separated variable files, which can be loaded with `read.csv()`

You can also check the `foreign` library for other data file types

Suppose you load a dataset using

```
data <- read.csv("mydata.csv")
```

You can check out the names of the variables using `names(data)`

And access any variables, such as `gdp`, using `data$gdp`

Benefits and dangers of attach()

If your data have variable names, you can also “attach” the dataset like so:

```
data <- read.csv("mydata.csv")
attach(data)
```

to access all the variables directly through newly created vectors.

Be careful! attach() is tricky.

1. If you attach a variable `data$x` in `data` and then modify `x`, the original `data$x` is unchanged.
2. If you have more than one dataset with the same variable names, `attach()` is a bad idea: only the first will be attached!

Sometimes `attach()` is handy, but be careful!

Missing data

When loading a dataset, you can often tell R what symbol that file uses for missing data using the option `na.strings=`

So if your dataset codes missings as `.`, set `na.strings="."`

If your dataset codes missings as a blank, set `na.strings=""`

If your dataset codes missings in multiple ways, you could set, e.g.,
`na.strings=c(".", "", "NA")`

Missing data

When loading a dataset, you can often tell R what symbol that file uses for missing data using the option `na.strings=`

So if your dataset codes missings as `.`, set `na.strings="."`

If your dataset codes missings as a blank, set `na.strings=""`

If your dataset codes missings in multiple ways, you could set, e.g.,
`na.strings=c(".", "", "NA")`

Missing data

Many R commands will not work properly on vectors, matrices, or dataframes containing missing data (NAs)

To check if a variables contains missings, use `is.na(x)`

To create a new variable with missings listwise deleted, use `na.omit`

If we have a dataset `data` with NAs at `data[15,5]` and `data[17,3]`

```
dataomitted <- na.omit(data)
```

will create a new dataset with the 15th and 17th rows left out

Be careful! If you have a variable with lots of NAs you are not using in your analysis, remove it from the dataset *before* using `na.omit()`

Mathematical Operations

R can do all the basic math you need

Binary operators:

+ - * / ^

Binary comparisons:

< <= > >= == !=

Logical operators (and, or, and not; use parentheses!):

&& || !

Math/stat fns:

log exp mean median min max sd var cov cor

Set functions (see `help(sets)`), Trigonometry (see `help(Trig)`),

R follows the usual order of operations; if in doubt, use parentheses

An R list is a basket containing many other variables

```
> x <- list(a=1, b=c(2,15), giraffe="hello")
```

```
> x$a
```

```
[1] 1
```

```
> x$b
```

```
[1]  2 15
```

```
> x$b[2]
```

```
[1] 15
```

```
> x$giraffe
```

```
[1] "hello"
```

```
> x[3]
```

```
$giraffe
```

```
[1] "hello"
```

```
> x[["giraffe"]]
```

```
[1] "hello"
```

R lists

Things to remember about lists

- Lists can contain any number of variables of any type
- Lists can contain other lists
- Contents of a list can be accessed by name or by position
- Allow us to move lots of variables in and out of functions
- Functions often return lists (only way to have multiple outputs)

lm() basics

```
# To run a regression
```

```
res <- lm(y~x1+x2+x3,
```

```
      data,
```

```
      na.action="")
```

```
# A dataframe containing
```

```
# y, x1, x2, etc.
```

```
# To print a summary
```

```
summary(res)
```

```
# To get the coefficients
```

```
res$coefficients
```

```
# or
```

```
coef(res)
```

```
#To get residuals
```

```
res$residuals
```

```
#or
```

```
resid(res)
```


lm() basics

```
# To get the variance-covariance matrix of the regressors  
vcov(res)
```

```
# To get the standard errors  
sqrt(diag(vcov(res)))
```

```
# To get the fitted values  
predict(res)
```

```
# To get expected values for a new observation or dataset  
predict(res,  
        newdata,                # a dataframe with same x vars  
        # as data, but new values  
        interval = "confidence", # alternative: "prediction"  
        level = 0.95  
        )
```

R lists & Object Oriented Programming

A list object in R can be given a special “class” using the `class()` function

This is just a metatag telling other R functions that this list object conforms to a certain format

For example, suppose we run a linear regression:

```
resLS <- lm(y~x1+x2+x3, data=exampledata)
```

The result `resLS` is a list object of class ‘`lm`’

Other functions like `plot()` and `predict()` will react to `resLS` in a special way because of this class designation

Specifically, they will run functions called `plot.lm()` and `predict.lm()`

Object-oriented programming:

a function does different things depending on class of input object

Help!

To get help on a known command `x`, type `help(x)` or `?x`

To search the help files using a keyword string `s`, type `help.search(s)`

Note that this implies to search on the word `regression`, you should type `help.search("regression")`

but to get help for the command `lm`, you should type `help(lm)`

Installing R on a PC

- Go to the Comprehensive R Archive Network (CRAN)
<http://cran.r-project.org/>
- Under the heading “Download and Install R”, click on “Windows”
- Click on “base”
- Download and run the R setup program.
The name changes as R gets updated;
the current version is “R-3.2.3-win.exe”
- Once you have R running on your computer,
you can add new libraries from inside R by selecting
“Install packages” from the Packages menu

Installing R on a Mac

- Go to the Comprehensive R Archive Network (CRAN)
<http://cran.r-project.org/>
- Under the heading “Download and Install R”, click on “MacOS X”
- Download and run the R setup program.
The name changes as R gets updated;
the current version is “R-3.2.3.pkg”
- Once you have R running on your computer,
you can add new libraries from inside R by selecting
“Install packages” from the Packages menu

Editing scripts

Don't use Microsoft Word to edit R code!

Word adds lots of “stuff” to text; R needs the script in a plain text file.

Some text editors:

- **Notepad:** Free, and comes with Windows (under Start → Programs → Accessories). Gets the job done; not powerful.
- **TextEdit:** Free, and comes with Mac OS X. Gets the job done; not powerful.
- **TINN-R:** Free and fairly powerful. Windows only.
<http://www.sciviews.org/Tinn-R/>
- **Emacs:** Free and very powerful (my preference). Can use for R and Latex. Available for Mac and PC.

For Mac (easy installation): <http://aquamacs.org/>

For Windows (see the README): <http://ftp.gnu.org/gnu/emacs/windows/>

Editing data

R can load many other packages' data files

See the `foreign` library for commands

For simplicity & universality, I prefer Comma-Separated Variable (CSV) files

Microsoft Excel can edit and export CSV files (under Save As)

R can read them using `read.csv()`

OpenOffice is free alternative to Excel & makes CSV files (for all platforms):
<http://www.openoffice.org/>

My detailed guide to installing social science software on the Mac:

<http://thewastebook.com/?post=social-science-computing-for-mac>

Focus on steps 1.1 and 1.3 for now; come back later for Latex in step 1.2

What's a high-level graphics command?

Most of you probably make R graphics by calling a “high-level” command (HLC)

In R, HLCs:

- produce a standard graphic type
- fill in lots of details (axes, titles, annotation)
- have many configurable parameters
- have varied flexibility
- may respond to object class

You don't need to use HLCs to make R graphics.

Could do from scratch

Some major high-level graphics commands

The two key places to find HLCs:
the base graphics package, and the `lattice` package

Use different graphical primitives

Have distinctive “looks”

Lattice is really good at conditioning and EDA (coplots)

Besides these, there are many HLCs strewn through other packages

Easiest way to find them: `help.search()`

When I first wrote this lecture,
I did `help.search('plot')` on a full install of R packages

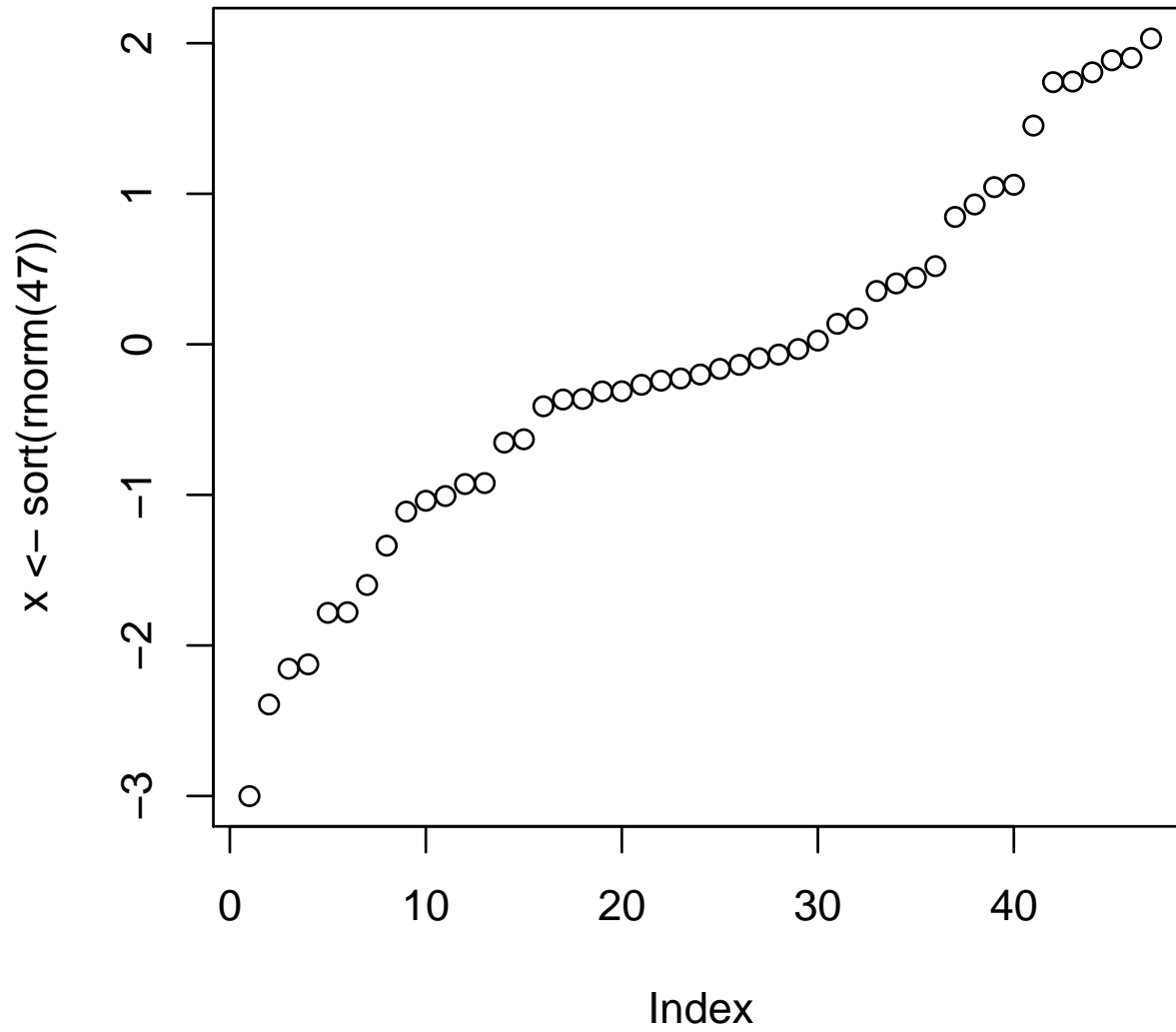
Found lots of neat plotting functions in packages I'd never heard of;
today there are many more

Some major high-level graphics commands

| Graphic | Base command | Lattice command |
|----------------------|------------------------|-------------------------|
| scatterplot | plot() | xyplot() |
| line plot | plot(. . . ,type="l") | xyplot(. . . ,type="l") |
| Bar chart | barplot() | barchart() |
| Histogram | hist() | histogram() |
| Smoothed histograms | plot() after density() | densityplot() |
| boxplot | boxplot() | bwplot() |
| Dot plot | dotchart() | dotplot() |
| Contour plots | contour() | contourplot() |
| image plot | image() | levelplot() |
| 3D surface | persp() | wireframe() |
| 3D scatter | scatterplot3d()* | cloud() |
| conditional plots | coplot() | xyplot() |
| Scatterplot matrix | | splom() |
| Parallel coordinates | | parallel() |
| Star plot | stars() | |
| Stem-and-leaf plots | stem() | |
| ternary plot | ternaryplot() in vcd | |
| Mosaic plots | mosaicplot() in vcd | |

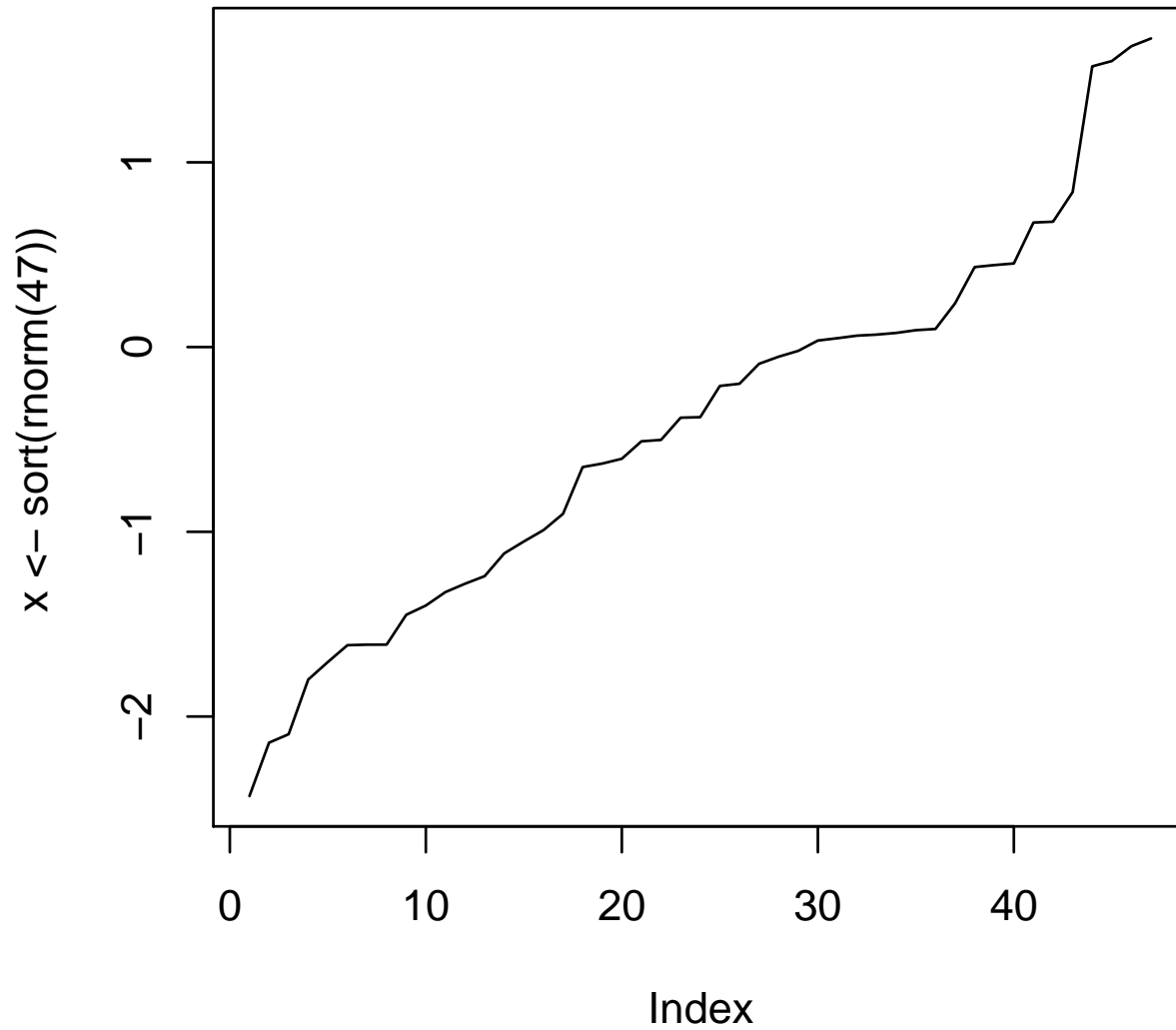
Scatterplot: plot()

`plot(x, type = "p")`

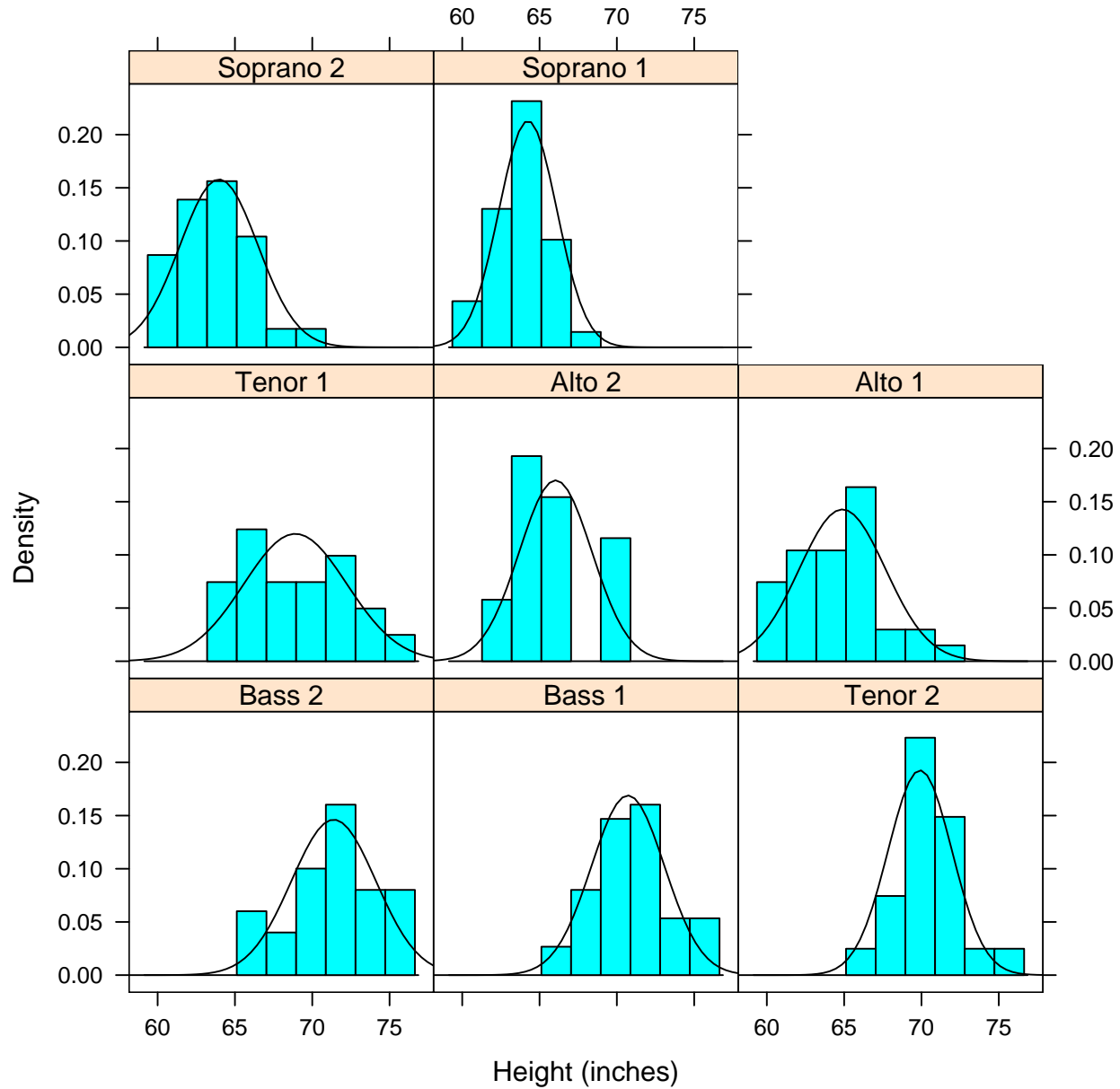


Line plot: `plot(..., type="l")`

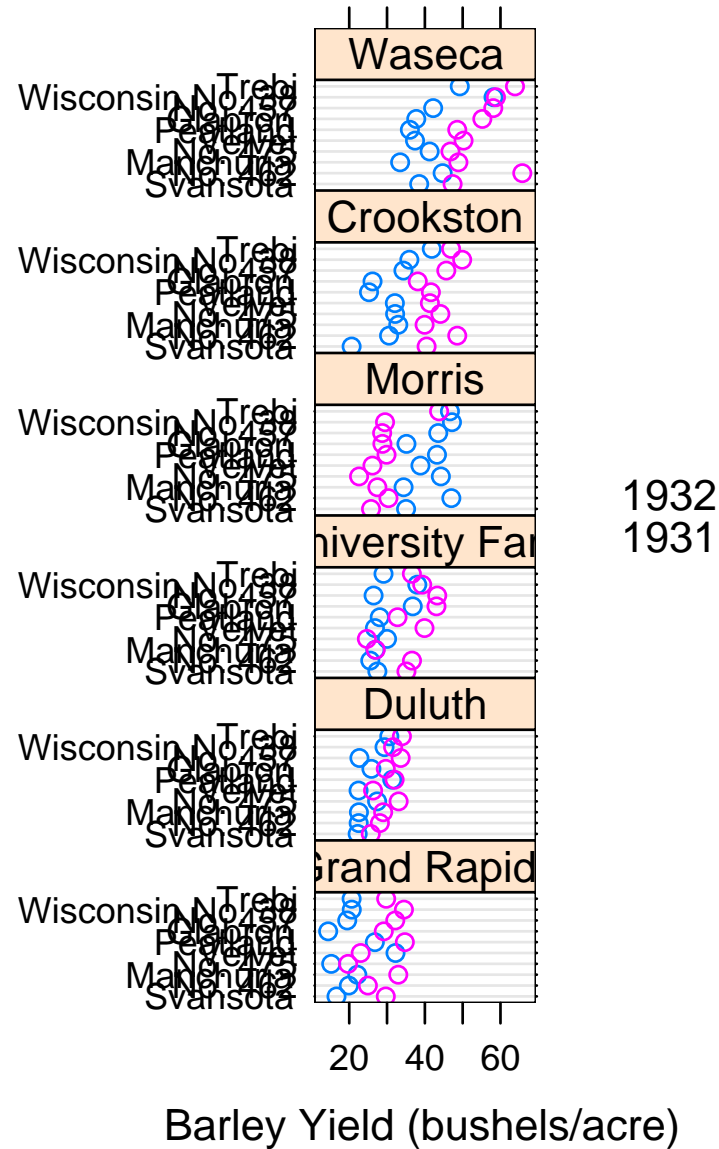
`plot(x, type = "l")`



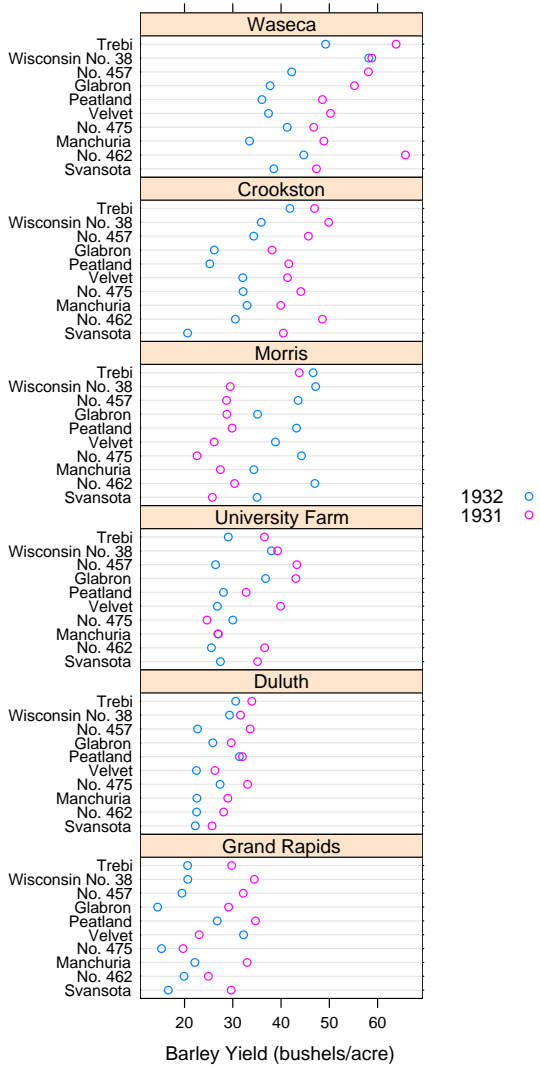
(Smoothed) Histograms: `densityplot()` & others



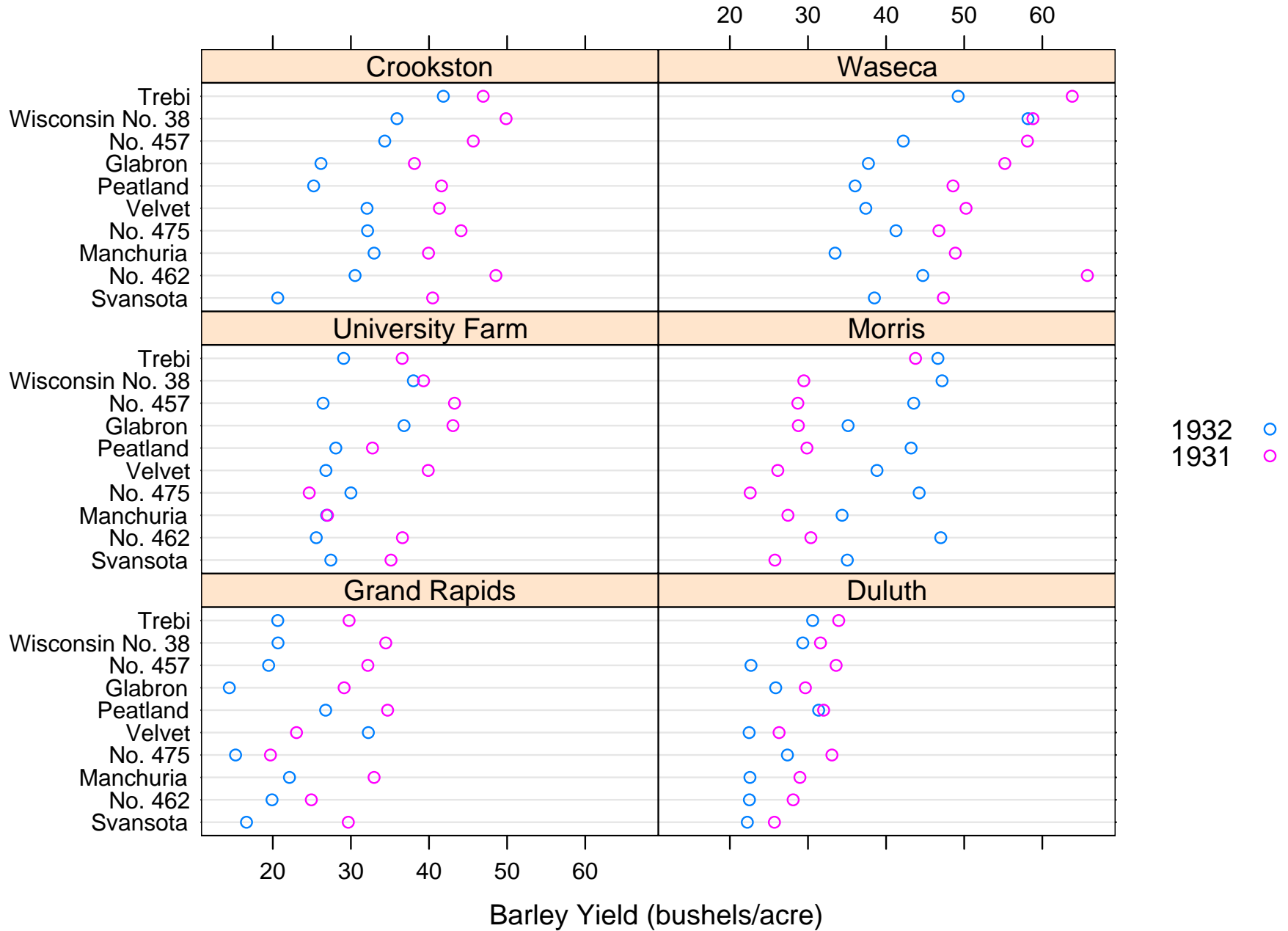
Dot plot: dotplot()



Dot plot is sensitive to device size



Dot plot has a layout option



Contour plot: contour()

Maunga Whau Volcano

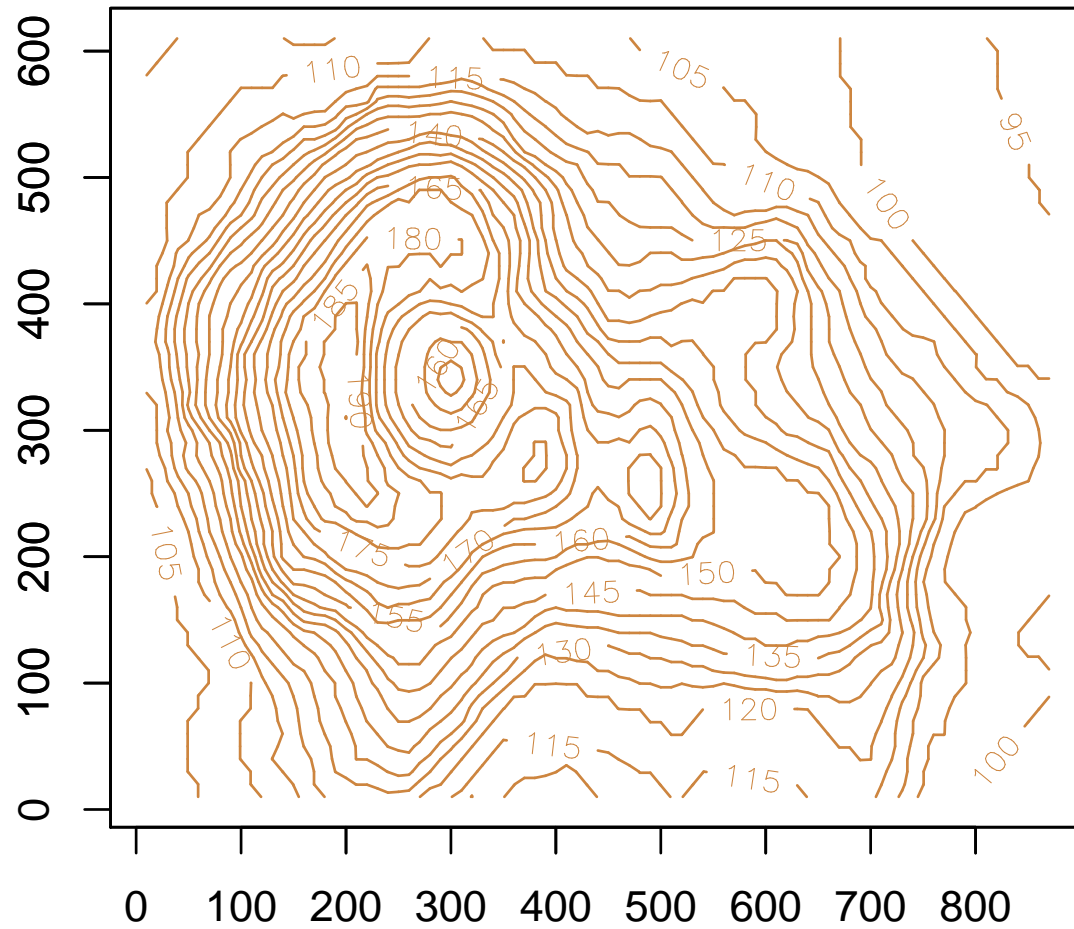


Image plot: image()

Maunga Whau Volcano

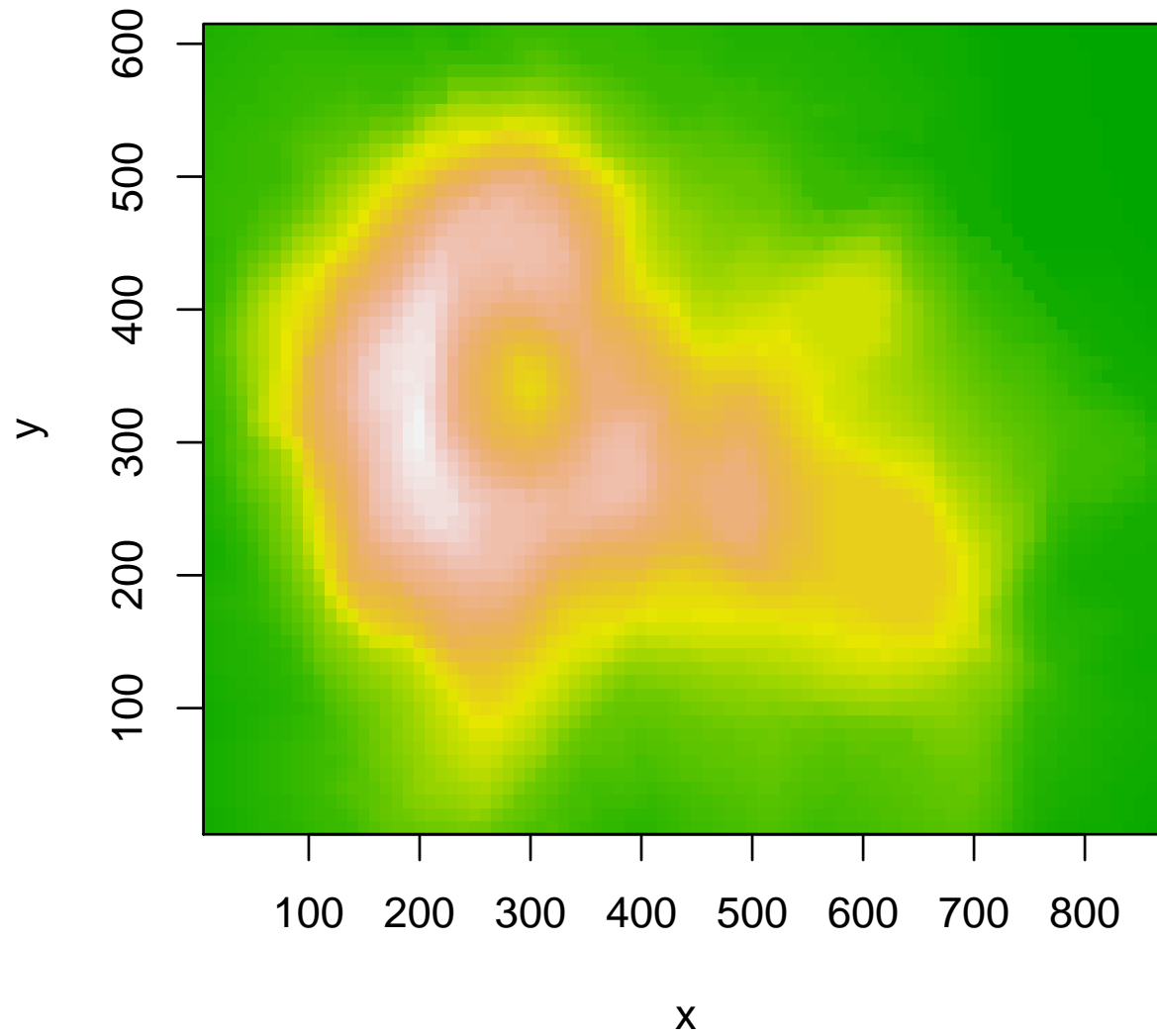
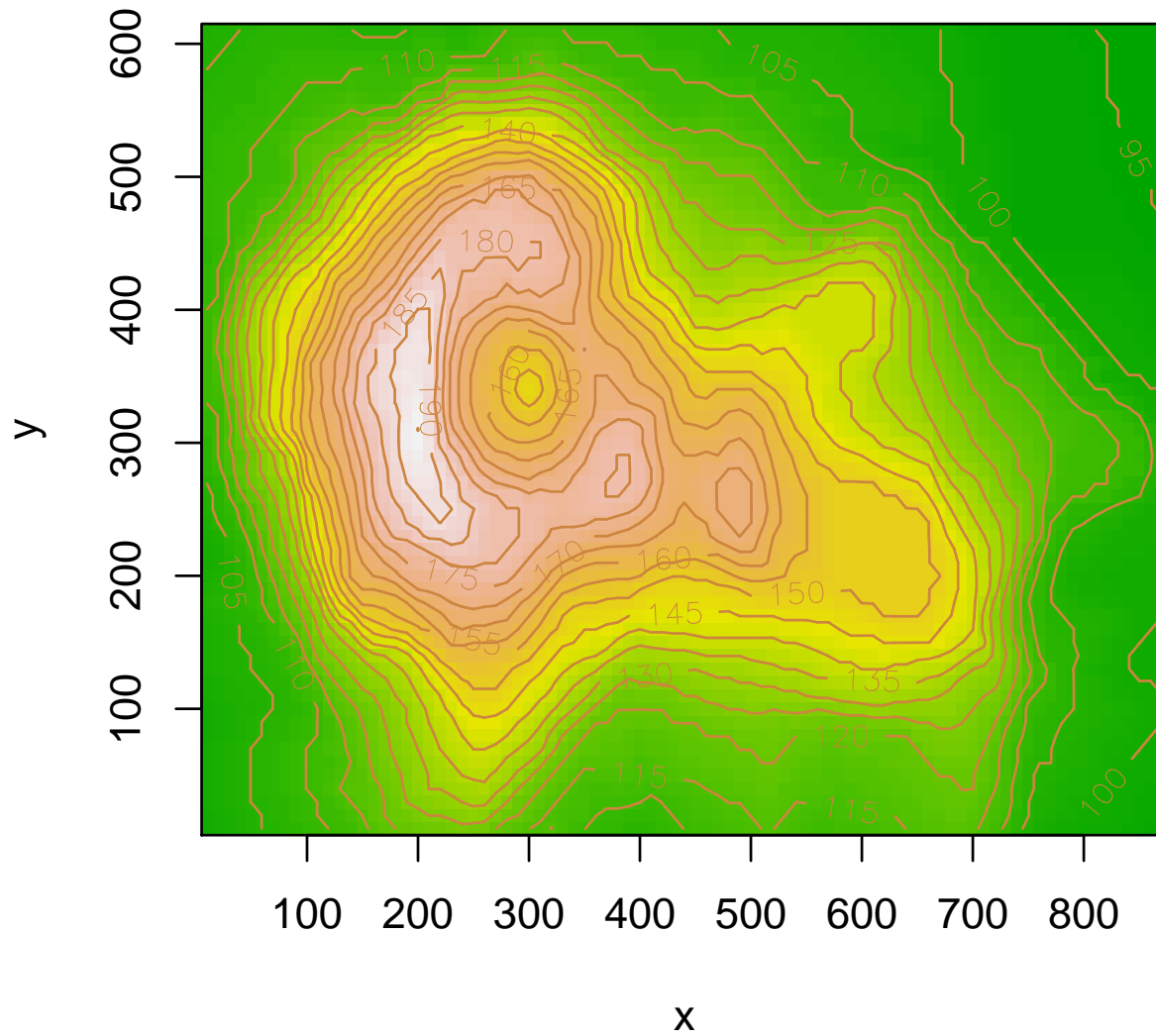
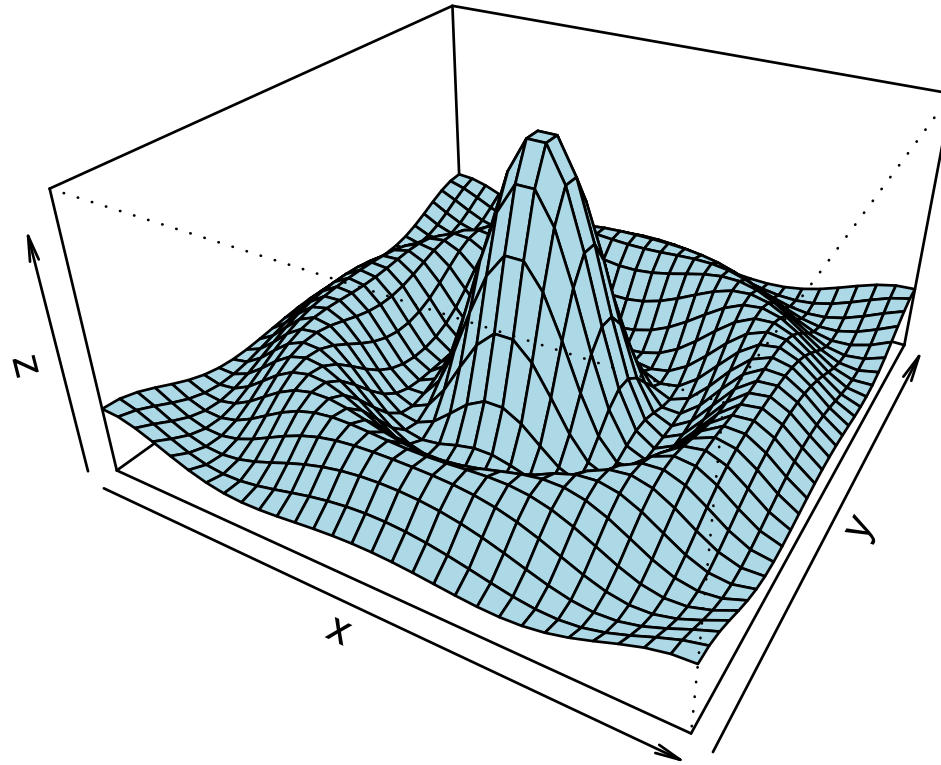


Image plot with contours: `contour(..., add=TRUE)`

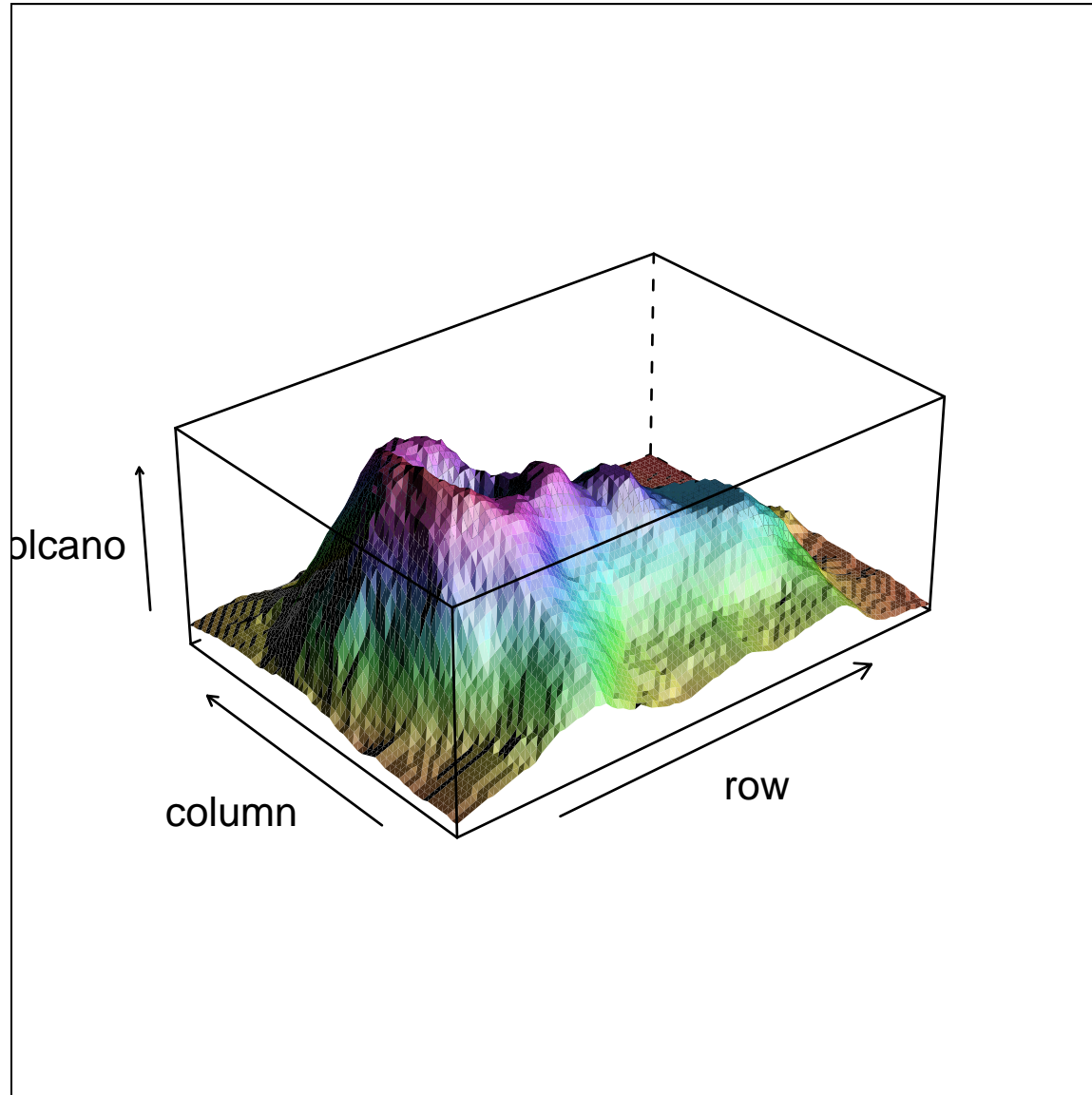
Maunga Whau Volcano



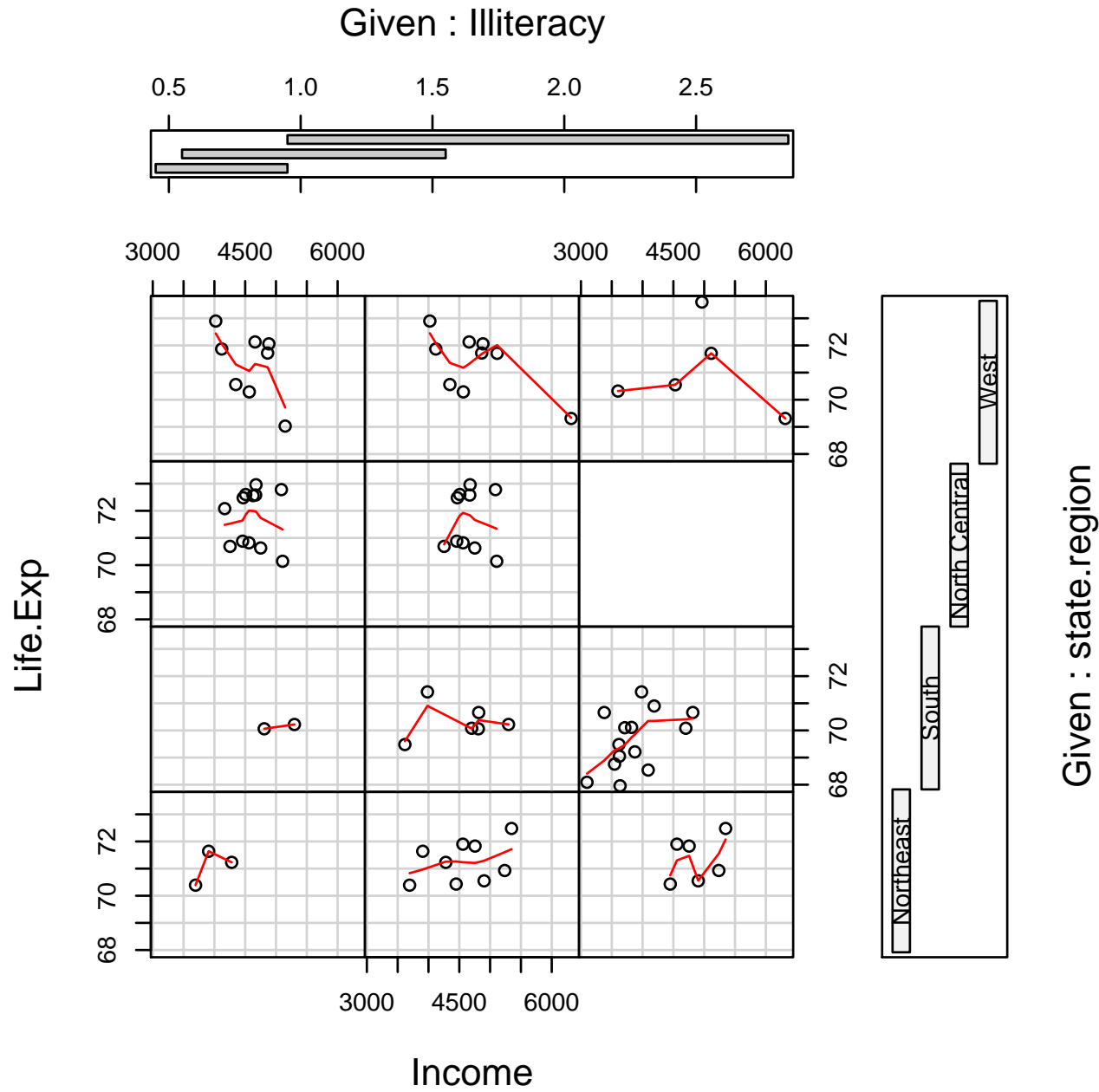
3D surface: persp()



3D surface: wireframe()

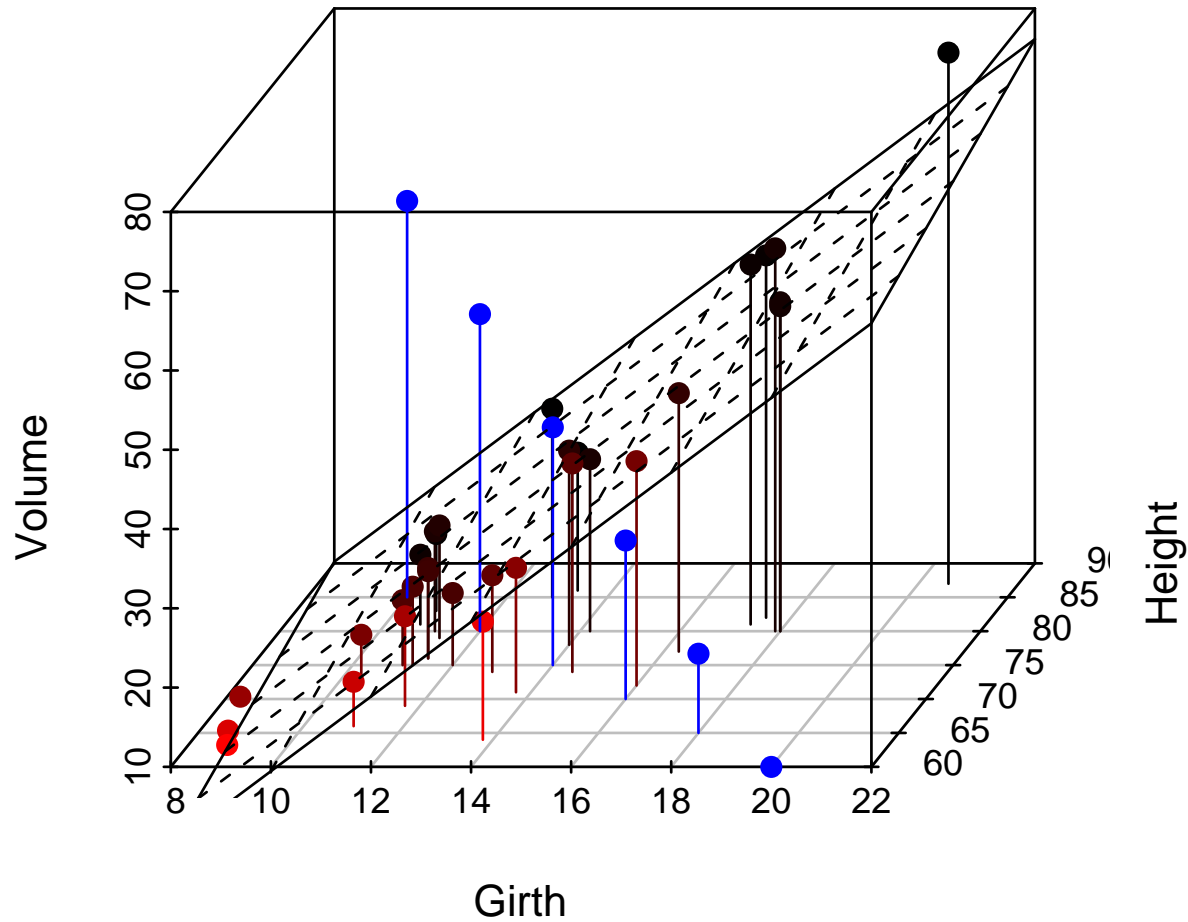


Conditional plots: coplot()

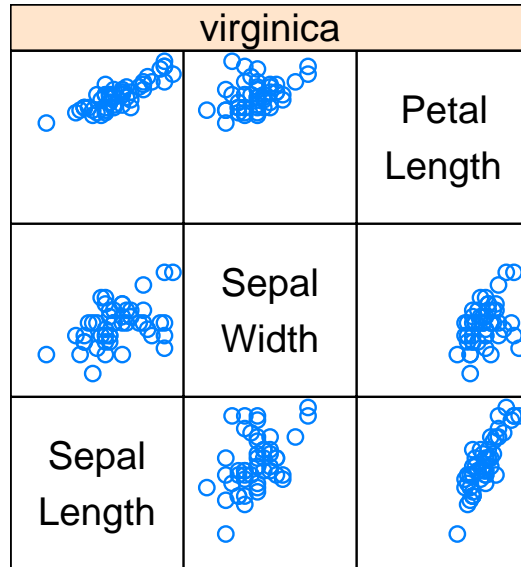


3D scatter: scatterplot3d() in own library

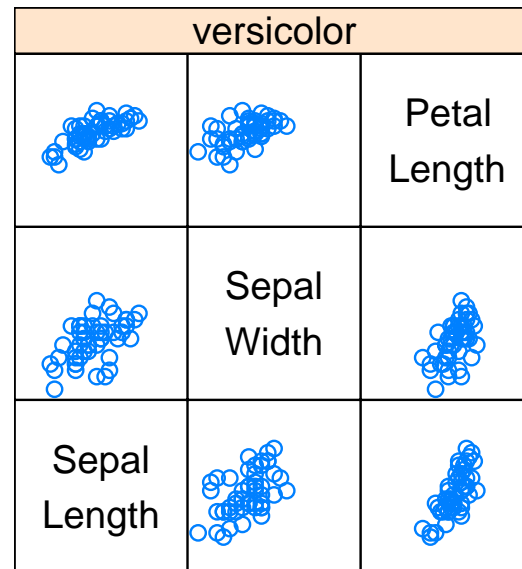
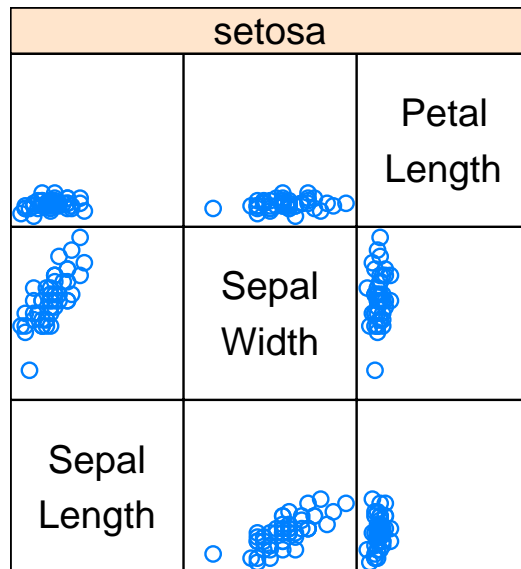
scatterplot3d – 5



Scatterplot matrix: `spLOm()`

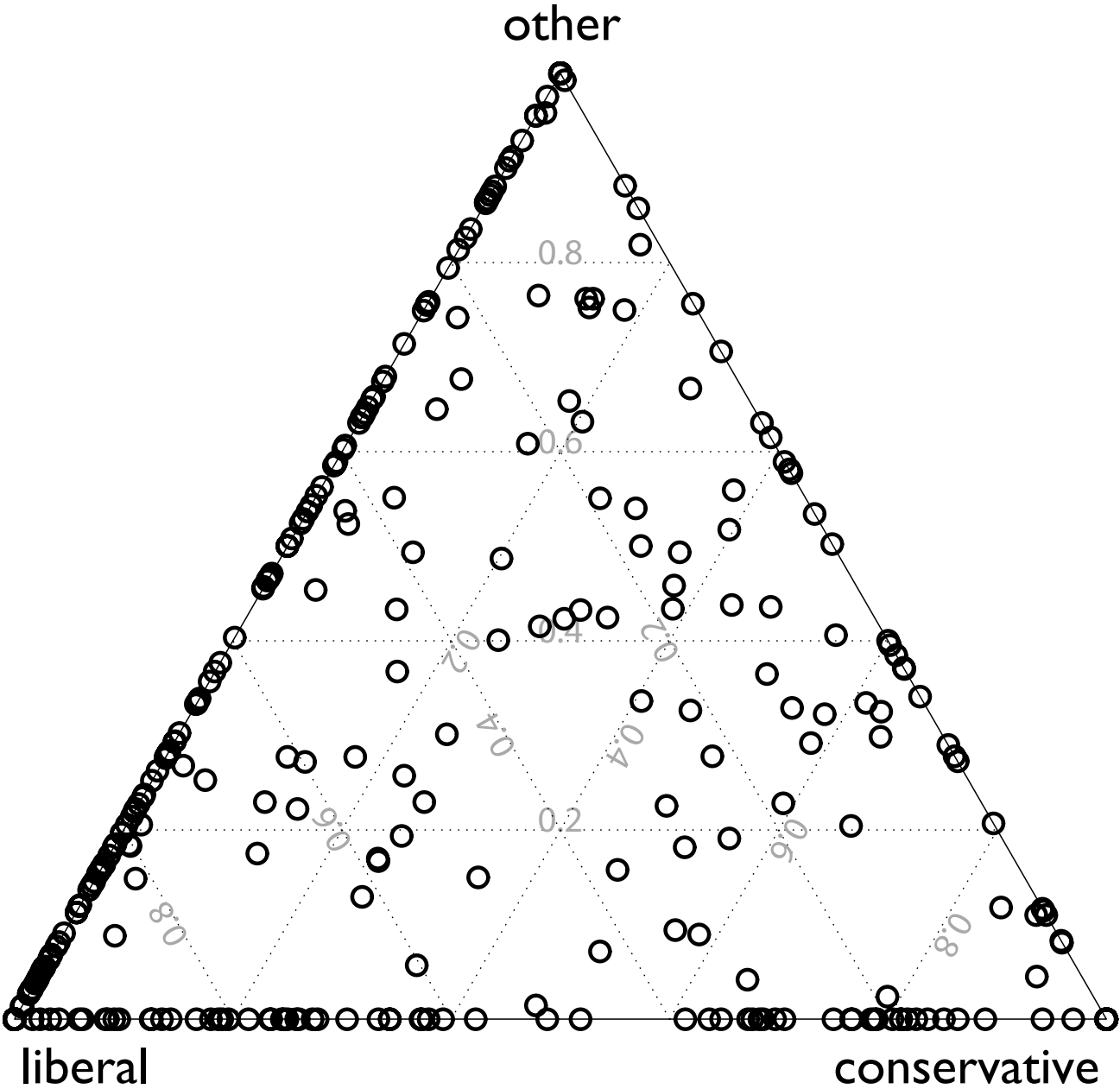


Three
Varieties
of
Iris



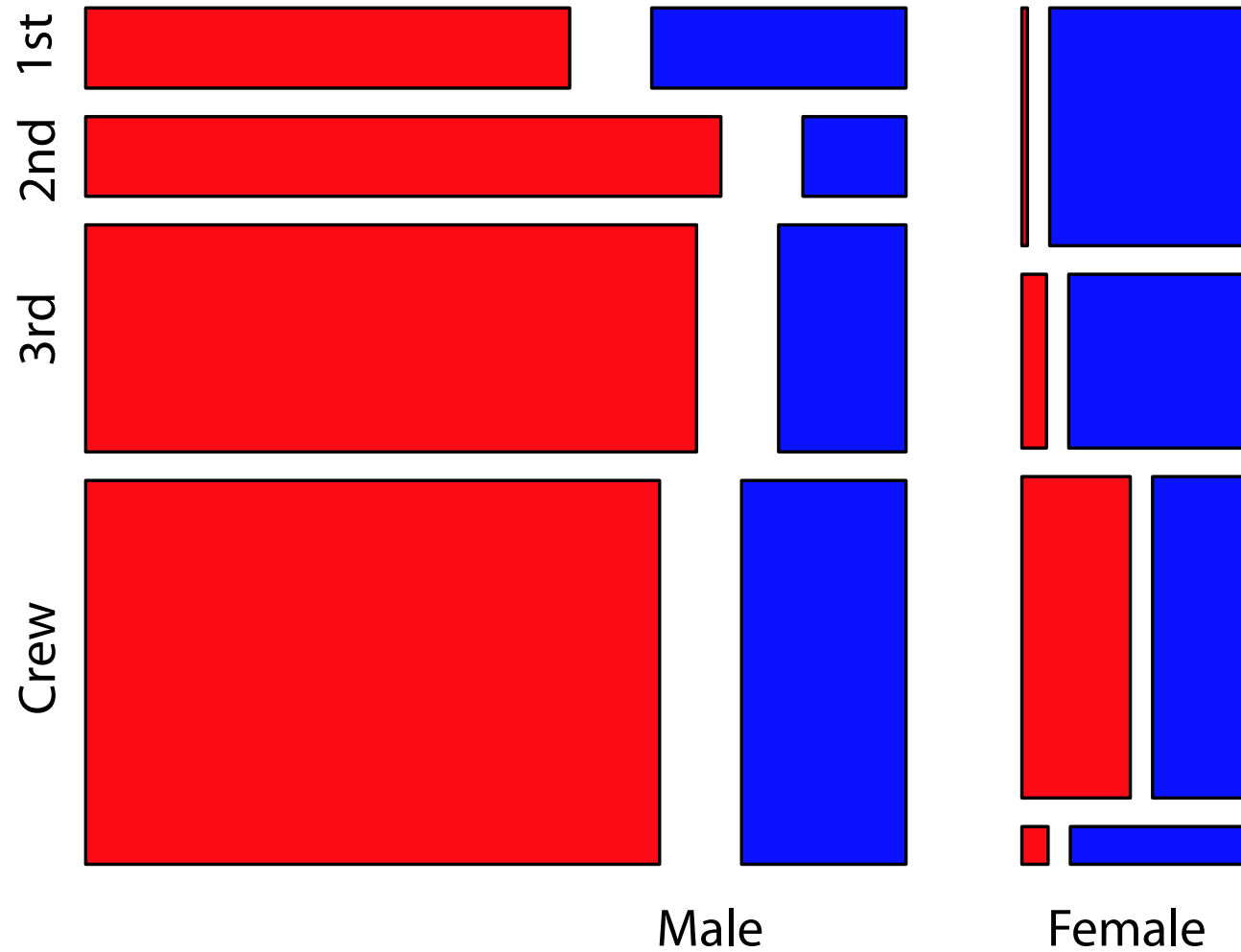
Scatter Plot Matrix

Ternary plot: ternaryplot() in vcd



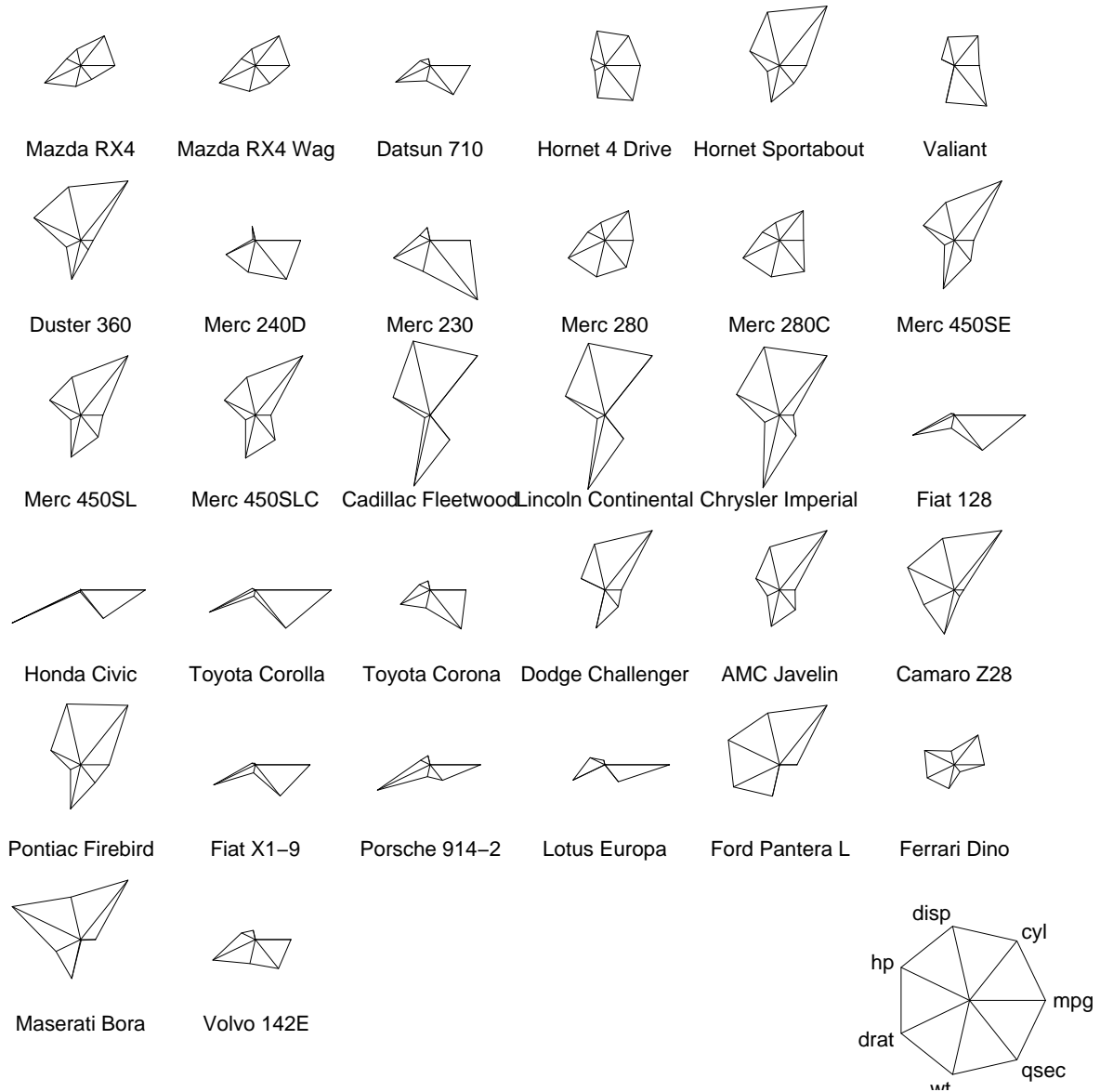
Mosaic plot: mosaic() in vcd

Titanic Survival Proportions: Deaths vs Survivors



Star plot: stars()

Motor Trend Cars : full stars()



Some major high-level graphics commands

```
stem> stem(log10(islands))
```

The decimal point is at the |

1 | 1111112222233444

1 | 5555556666667899999

2 | 3344

2 | 59

3 |

3 | 5678

4 | 012

Some major high-level graphics commands

| Graphic | Base command | Lattice command |
|----------------------|------------------------|-------------------------|
| scatterplot | plot() | xyplot() |
| line plot | plot(. . . ,type="l") | xyplot(. . . ,type="l") |
| Bar chart | barplot() | barchart() |
| Histogram | hist() | histogram() |
| Smoothed histograms | plot() after density() | densityplot() |
| boxplot | boxplot() | bwplot() |
| Dot plot | dotchart() | dotplot() |
| Contour plots | contour() | contourplot() |
| image plot | image() | levelplot() |
| 3D surface | persp() | wireframe() |
| 3D scatter | scatterplot3d()* | cloud() |
| conditional plots | coplot() | xyplot() |
| Scatterplot matrix | | splom() |
| Parallel coordinates | | parallel() |
| Star plot | stars() | |
| Stem-and-leaf plots | stem() | |
| ternary plot | ternaryplot() in vcd | |
| Fourfold plot | fourfoldplot() in vcd | |
| Mosaic plots | mosaicplot() in vcd | |

Basic customization

For any given high-level plotting command, there are many options listed in help

```
barplot(height, width = 1, space = NULL,  
        names.arg = NULL, legend.text = NULL, beside = FALSE,  
        horiz = FALSE, density = NULL, angle = 45,  
        col = NULL, border = par("fg"),  
        main = NULL, sub = NULL, xlab = NULL, ylab = NULL,  
        xlim = NULL, ylim = NULL, xpd = TRUE,  
        axes = TRUE, axisnames = TRUE,  
        cex.axis = par("cex.axis"), cex.names = par("cex.axis"),  
        inside = TRUE, plot = TRUE, axis.lty = 0, offset = 0, ...)
```

Just the tip of the iceberg: notice the ...

This means you can pass other, unspecified commands through `barplot`

Basic customization

The most important (semi-) documented parameters to send through `...` are settings to `par()`

Most base (traditional) graphics options are set through `par()`

`par()` has no effect on grid graphics (e.g., `lattice`, `tile`)

If you never have, consult `help(par)` now!

Some key examples, grouped functionally

par() settings

Customizing text size:

| | |
|----------|----------------------------|
| cex | Text size (a multiplier) |
| cex.axis | Text size of tick numbers |
| cex.lab | Text size of axes labels |
| cex.main | Text size of plot title |
| cex.sub | Text size of plot subtitle |

note the latter will multiply off the basic cex

`par()` settings

More text specific formatting

`font` Font face (bold, italic)

`font.axis` etc

`srt` Rotation of text in plot (degrees)

`las` Rotation of text in margin (degrees)

Note the distinction between text in the plot and outside.

Text in the plot is plotted with `text()`

Text outside the plot is plotted with `mtext()`, which was designed to put on titles, etc.

Aside on margins

`mtext()` expects to be told which side of the plot & how many margin lines away the text is

This is kind of hopeless

A work-around to get stuff in the margins:

1. Turn off “clipping”, the function that keeps data outside the plotting region from showing up in the margin.

We do this by setting `par(xpd=TRUE)` for the current plot

2. Then plot your text using the usual `text()` command, but with coordinates outside the plot region
3. Now, if you want to rotate, use `par(srt)` as normal
4. You could turn clipping on and off to get only certain marginal data plotted.

`grid` offers a much better way

More `par()` settings

Formatting for most any object

| | |
|-----------------------|---------------------------------|
| <code>bg</code> | background color |
| <code>col</code> | Color of lines, symbols in plot |
| <code>col.axis</code> | Color of tick numbers, etc |

Want to color the axes? You'll need to draw them yourself (next time)

Aside: Colors in R

Three ways to specify a color to an R function (for all R graphics tools):

1. color names, like `'red'` or `'lightblue'`
(see `colors()` for a list of hundreds of color names)
2. numerical color codes from `rgb()`, `hsv()`, or `hcl()`
(`hcl()` gives CIEluv equal perceptual changes for unit changes in chroma, value, or brightness)

Also useful: `col2rgb()`, `rgb2hsv()`, etc. for conversions among these functions

3. numerical color codes offered by packages for selecting cognitively valid palattes, optimized to your required number of colors and level of measurement (categorical, ordered, interval):

| Package | Key function(s) |
|--------------|--|
| RColorBrewer | <code>brewer.pal()</code> |
| colorspace | <code>sequential_hcl()</code> and <code>diverge_hcl()</code> |

RColorBrewer is fast and easy; colorspace is very powerful

More `par()` settings

Formatting for lines and symbols

`lty` Line type (solid, dashed, etc)

`lwd` Line width (default too large; try really small, e.g., 0)

`pch` Data symbol type; see `example(points)`

`lty` can take complex inputs, see the help for `par()`

You will very often need to set the above

More `par()` settings

Formatting for axes

| | |
|----------------------|--|
| <code>lab</code> | Number of ticks |
| <code>xaxp</code> | Number of ticks for xaxis |
| <code>tck,tcl</code> | Length of ticks relative to plot/text |
| <code>mgp</code> | Axis spacing: axis title, tick labels, axis line |

These may seem trivial, but affect the aesthetics of the plot & effective use of space

R defaults to excessive `mgp`, which looks ugly & wastes space

Most HLCs forget to rotate the y-axis labels. This is a bit harder to fix

More `par()` settings

More formatting for axes

The following commands are special:

they are primitives in `par()` that can't be set inside the `...` of high-level commands

You must set them with `par()` first

`usr` Ranges of axes: `c(xmin, xmax, ymin, ymax)`

`xlog` Log scale for x axis?

`ylog` Log scale for y axis?

Getting math on plots

Getting mathematics on the plots is sometimes possible

See `example(text)` for ideas

The key command is `expression()`

For example,

| | |
|---------------------------------|-----------|
| <code>expression(bar(x))</code> | \bar{x} |
| <code>expression(x[i])</code> | x_i |
| <code>expression(x^{-2})</code> | x^{-2} |

etc

Vaguely Latex-like, but less powerful

Give up and use Illustrator and/or Latex?

R **graphics devices**

Everything you draw in R must be drawn on a canvas

Must create the canvas before you draw anything

Computer canvases are **devices** you draw to

Devices save graphical input in different ways

Most important distinction: raster vs. vector devices

Vector vs. raster



Pointalism = raster graphics. Plot each pixel on an n by m grid.

Vector vs. raster

Pixel = Point = Raster

Good for pictures. Bad for drawings/graphics/cartoons.

(Puzzle: isn't everything raster? In display, yes. Not in storage)

Advantages of vector:

- Easily manipulable/modifiable groupings of objects
- Easy to scale objects larger or smaller/ Arbitrary precision
- Much smaller file sizes
- Can always convert to raster (but not the other way round, at least not well)

Disadvantages:

- A photograph would be really hard to show (and huge file size)
- Not web accessible. Convert to PNG or PDF.

Some common graphics file formats

| | Lossy | Lossless |
|--------|-------------|----------------------------|
| Raster | .gif, .jpeg | .wmf, .png, .bmp |
| Vector | — | .ps, .eps, .pdf, .ai, .wmf |

Lossy means during file compression, some data is (intentionally) lost

Avoid lossy formats whenever possible

Avoid copy-and-paste on PC: rasterizes vector graphics in lossy way!

Some common graphics file formats

In R, have access to several formats:

| | |
|-----------------------------|------------------------------------|
| <code>win.metafile()</code> | wmf, Windows media file |
| <code>pdf()</code> | pdf, Adobe portable data file |
| <code>postscript()</code> | postscript file (printer language) |

| | |
|------------------------|-------------------------------|
| <code>x11()</code> | opens a screen; all computers |
| <code>windows()</code> | opens a screen; PC only |
| <code>quartz()</code> | opens a screen; Mac only |

Latex, Mac or Unix users can't use wmf

`windows(record=TRUE)` let's you cycle thru old graphs with arrow keys

Best to make final graphics directly through `pdf()` or `postscript()`

Avoids rasterization

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 thought 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.

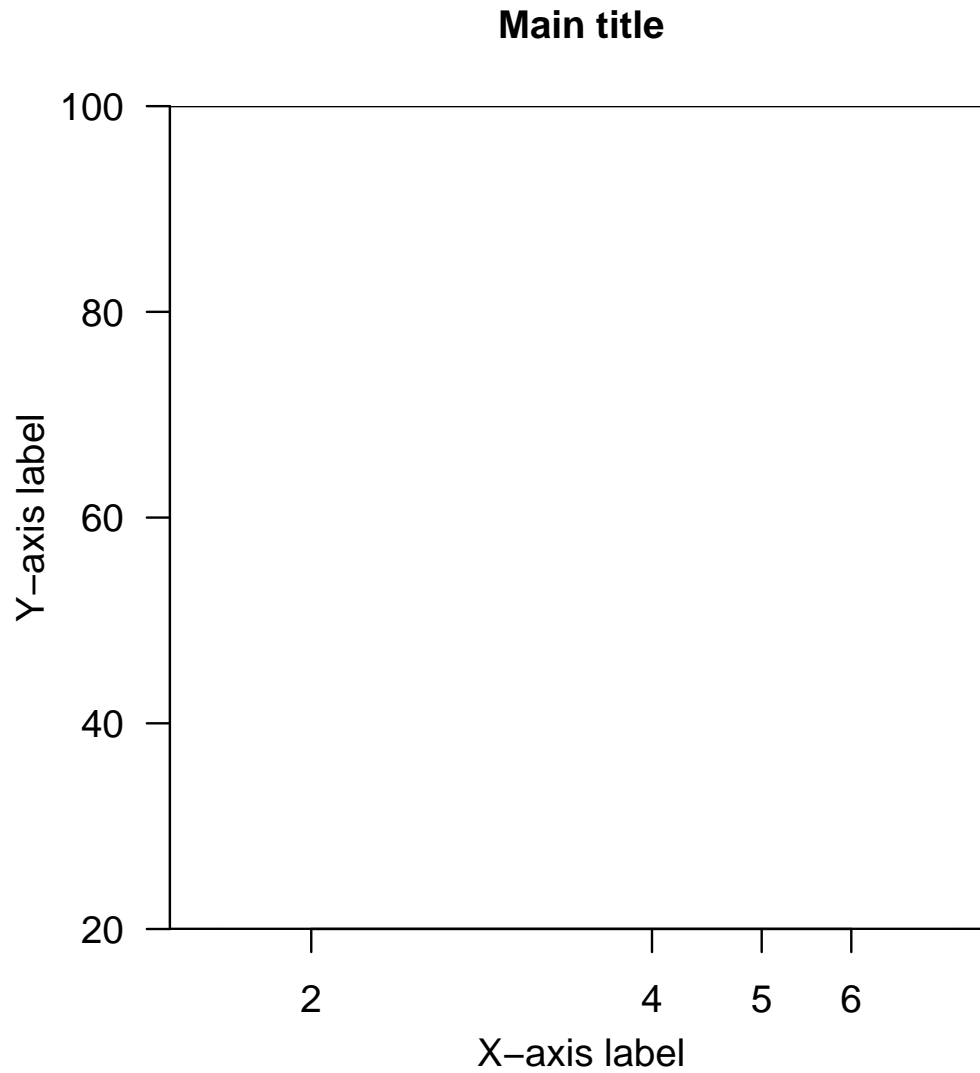
Note this is not an "axis system".

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

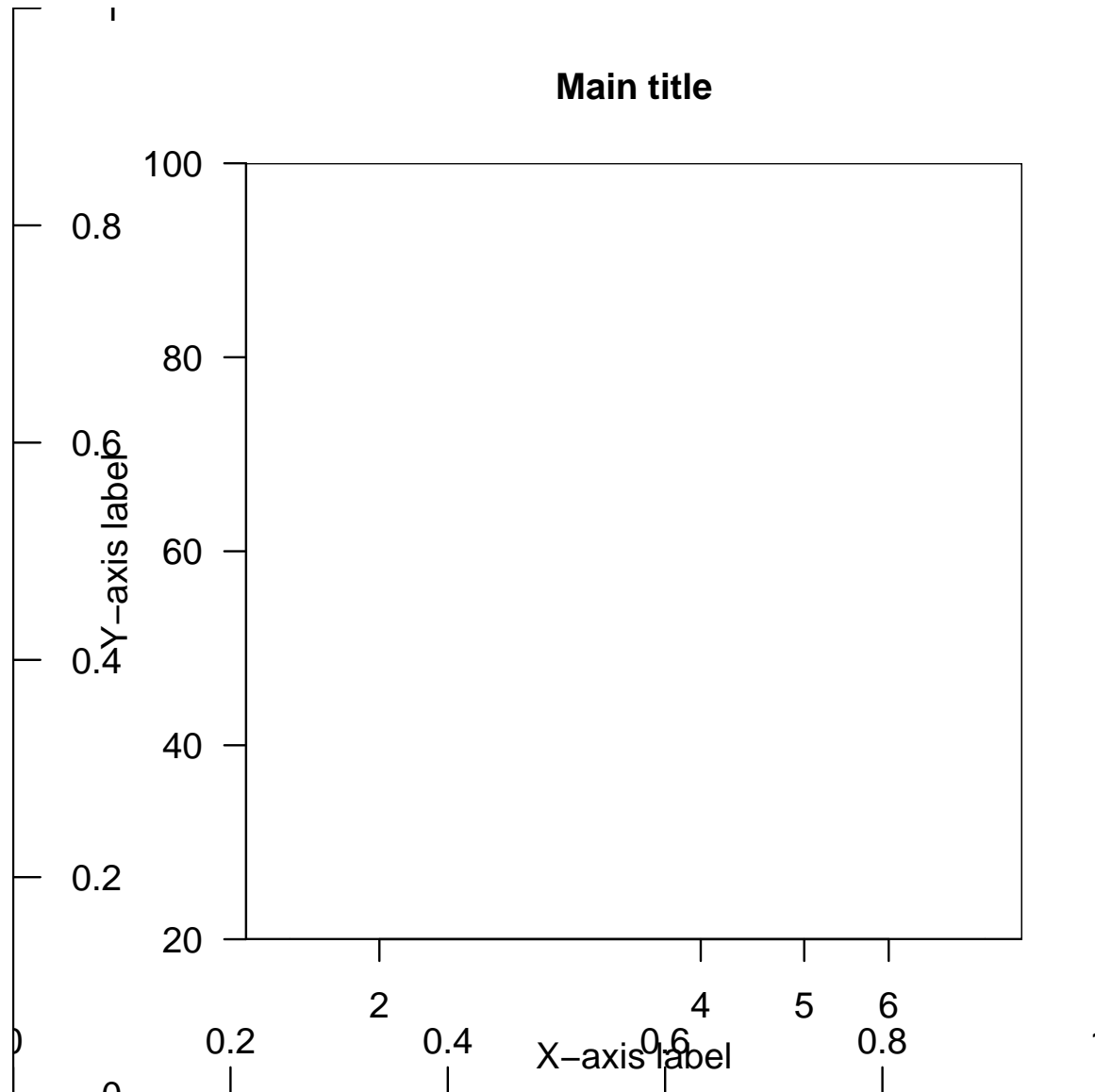
This coordinate system complete not just for 2D images, but for representing "3D" images (ie 2D with false perspective), or for showing movies, or for interactive displays.

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

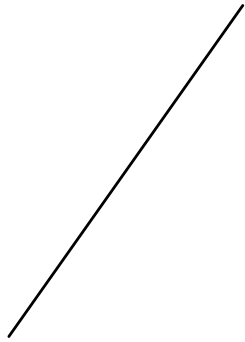
The base system: Example

```
# 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)
```

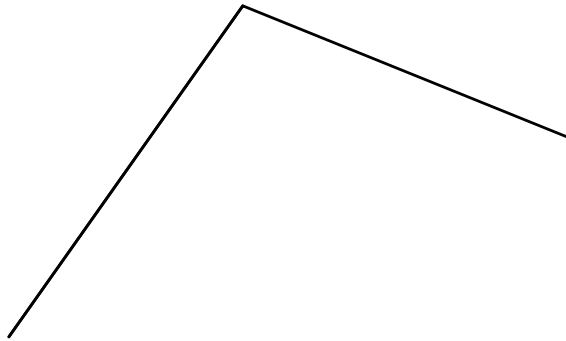
The plot so far



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

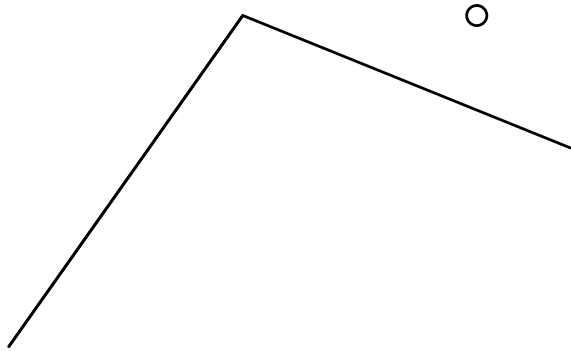


We connected the points $(0,0)$, $(0.25,0.5)$, and $(0.6,0.3)$
Using `lines()` we can draw any shape

The base system: Example

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

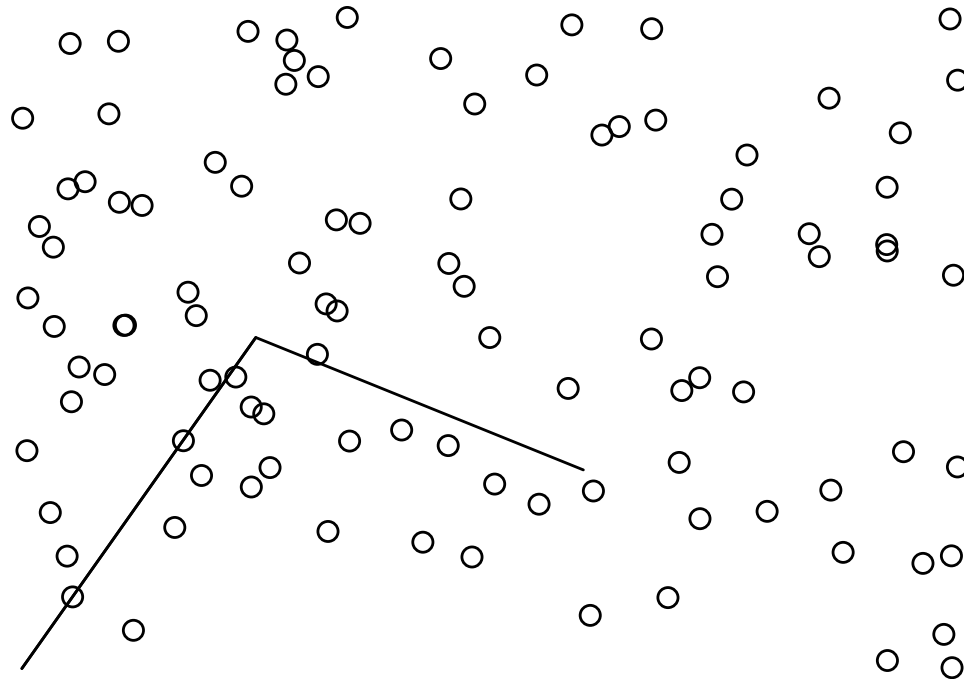
The plot so far



The base system: Example

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

The plot so far



The base system: Example

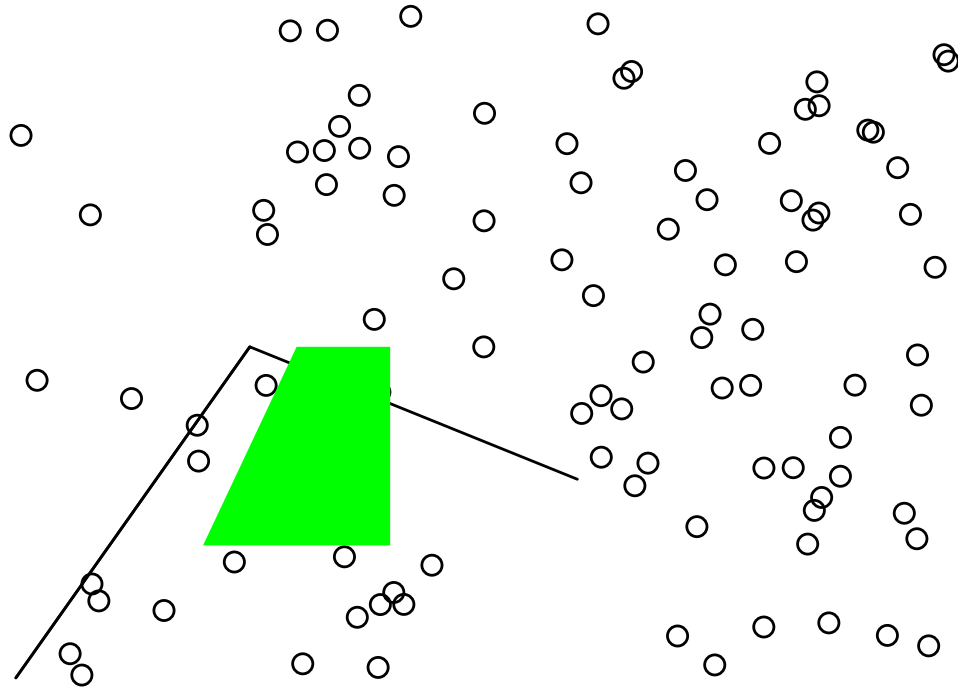
```
# Or a polygon?
```

```
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
)

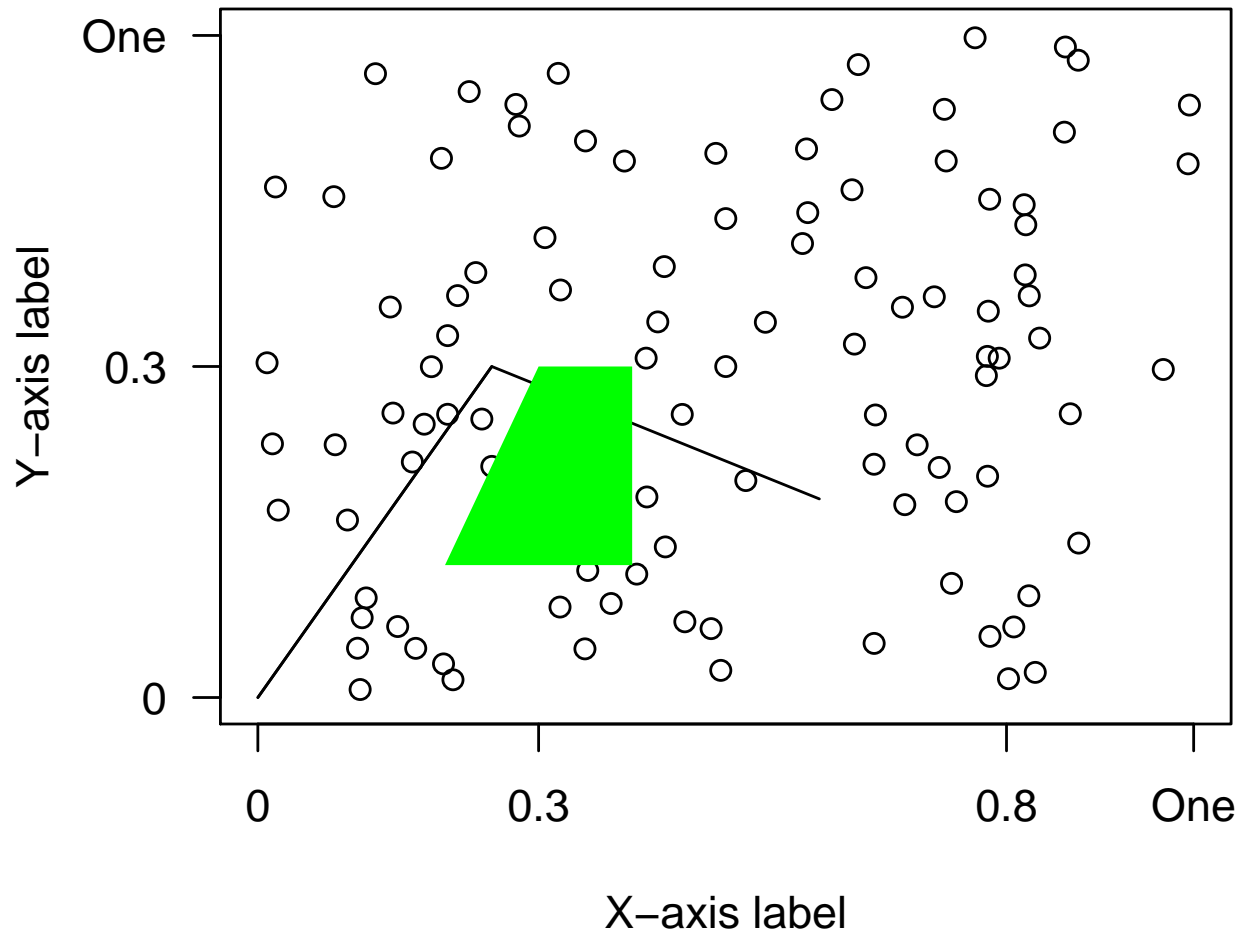
axis(side=2,                # 2 = y.  Obviously
      at=c(0,0.5,1),
      labels=c(0,0.3,"One"),
      las=1                 # rotate labels
)

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

box()
```

The final plot

A scatterplot made from scratch



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:

| | |
|------------------------|--|
| <code>native</code> | Based on the current x, y scales (e.g., your data) |
| <code>npc</code> | Treats the current viewport as (0,0) to (1,1) |
| <code>inches</code> | This and other physical unit available, given device |
| <code>strwidth</code> | Multiples of the width of a given string |
| <code>strheight</code> | Multiples of the height of a given string |
| <code>null</code> | 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 |

Let's use `grid` to plot a regression line with a shaded confidence region

`grid` is really meant for designing totally new graphics,
and especially for building functions for end-users

Although you wouldn't normally write a plot from scratch in `grid`,
it's the best way to learn how `grid` works

Let's use grid to plot a regression line with a shaded confidence region

grid is really meant for designing totally new graphics,
and especially for building functions for end-users

Although you wouldn't normally write a plot from scratch in grid,
it's the best way to learn how grid works

We begin by loading libraries and helper functions...

```
# Clear memory of all objects
rm(list=ls())

# Load libraries
library(RColorBrewer) # For nice colors
library(grid)         # For graphics
```



```
# MM-estimator fitting
mmest.fit <- function(y, x, ci, quiet=TRUE) {
  require(MASS)
  y <- y[order(x)]
  x <- x[order(x)]
  result <- rlm(y~x, method="MM")
  if (!quiet) 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
}
```

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

```
# Open a new pdf device
pdf("testgrid.pdf", width=5, height=5)

# Set up plot limits (used later)
limits <- c(log(1.5), log(8), 20, 100)

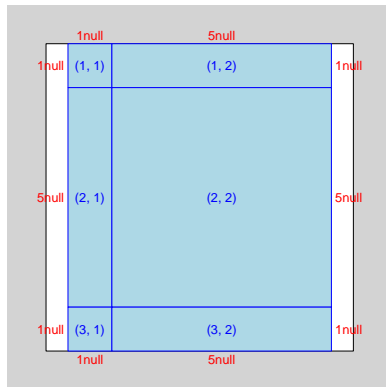
# Set up tick locations (used later)
yat <- c(20, 40, 60, 80, 100)
xat <- c(2, 3, 4, 5, 6, 7)

# Establishing limits and tick locations
# up front makes them easier to change
# -- otherwise, you need to make lots of edits
```

*PDF output is blank so far:
Nothing plotted yet*

```
# Create layout
overlay <- grid.layout(nrow=3,
                       ncol=2,
                       widths=c(1, 5),
                       heights=c(1, 5, 1),
                       respect=TRUE)

# Display layout
grid.show.layout(overlay)
```

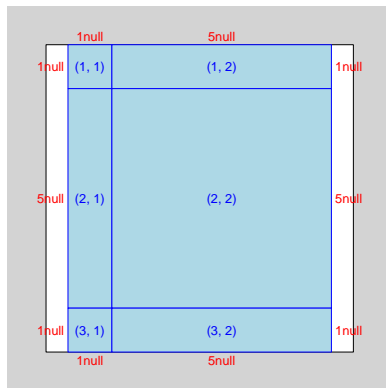


Diagnostic plot of overlay layout

```
# Create layout
overlay <- grid.layout(nrow=3,
                      ncol=2,
                      widths=c(1, 5),
                      heights=c(1, 5, 1),
                      respect=TRUE)

## Display layout
#grid.show.layout(overlay)

# "Push" layout to create initial viewport
pushViewport(viewport(layout=overlay))
```

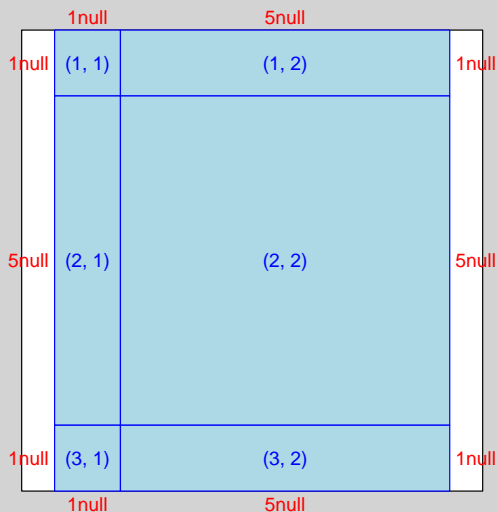


Diagnostic plot of overlay layout

If you make a `grid.layout()` you can take a look at it with `grid.show.layout()`

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

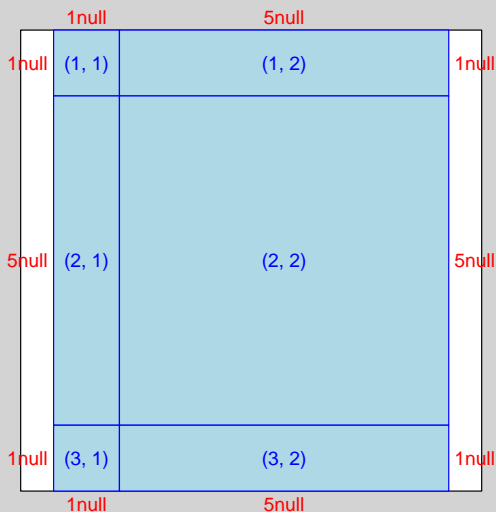
Let's use this diagnostic plot to understand null units better...



This graphic is 7 null's high and 6 null's wide

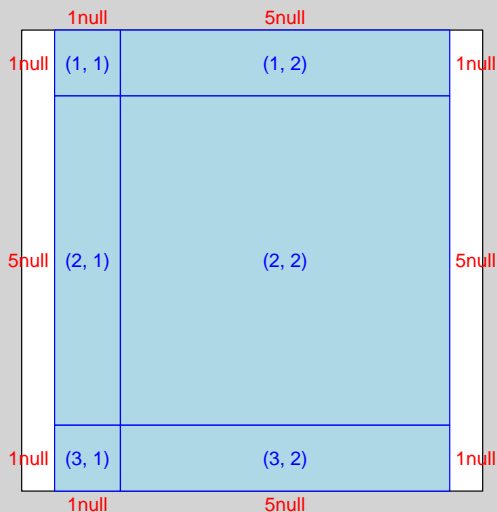
After allocating any fixed units (like inches), grid will assign remaining space to null

Thus the size of 1 null varies based on the fixed content of the graph



Recall this graphic exists in
a pdf device that is 5 inches
× 5 inches

So how big is a null in
inches?

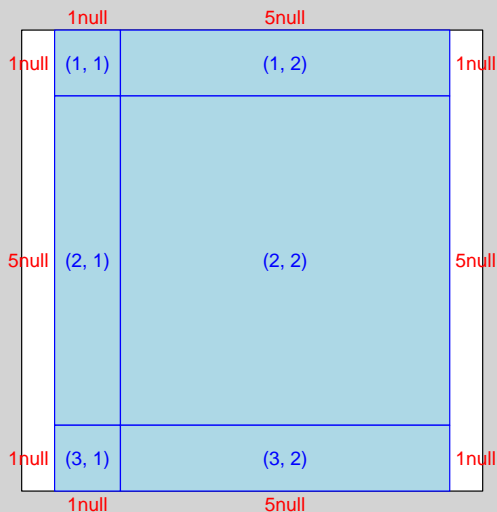


Recall this graphic exists in
a pdf device that is 5 inches
× 5 inches

So how big is a null in
inches?

The graph is 7 null's tall,
so 7 null = 5 inches

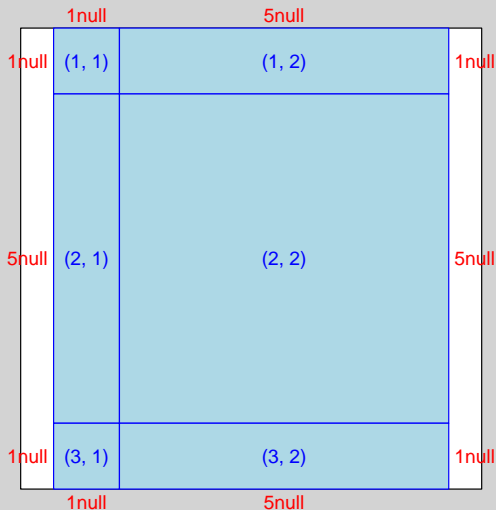
Thus 1 null = 5/7 inches
(≈ 0.714 inches)



Why didn't we set the width of 6 nulls equal to 5 inches?

In our layout, we set `respect=TRUE`, which forces null widths and null heights to be equal

Where nulls might differ, this forces the tightest dimension to "bind" them both to the smaller of the two possible lengths

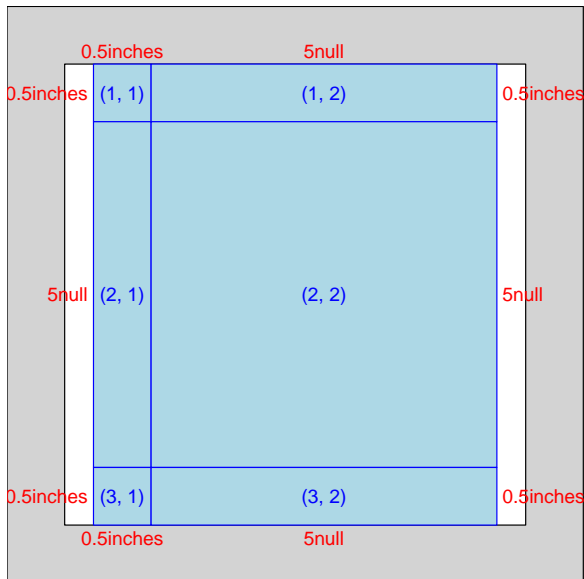


To understand null's better, let's change the layout so the first column is 0.5 inches wide...

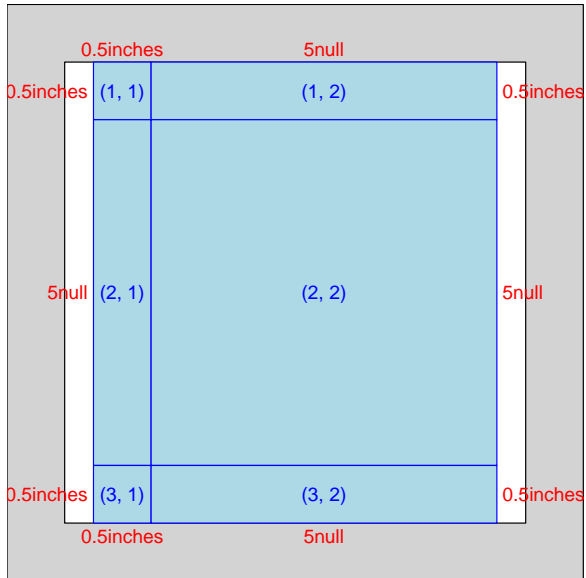
...and the first and last rows are 0.5 inches tall

Why? Because these are spaces for text labels

Fixing to 0.5 inch is one (inelegant) way to ensure they're readable & not too big



What is the size of a null in inches now?



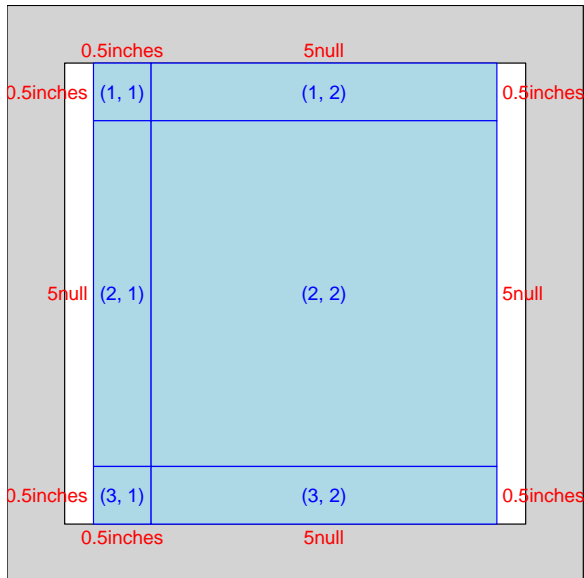
What is the size of a null in inches now?

Height is still the binding dimension

Layout is 1 inch and 5 null's tall, and must fit a 5 inch tall pdf

This leaves 4 inches to devote to null's, so 4 inches = 5 null

A null is now $\frac{4}{5}$ inches (0.8 in), increasing the space of our graphic devoted to data

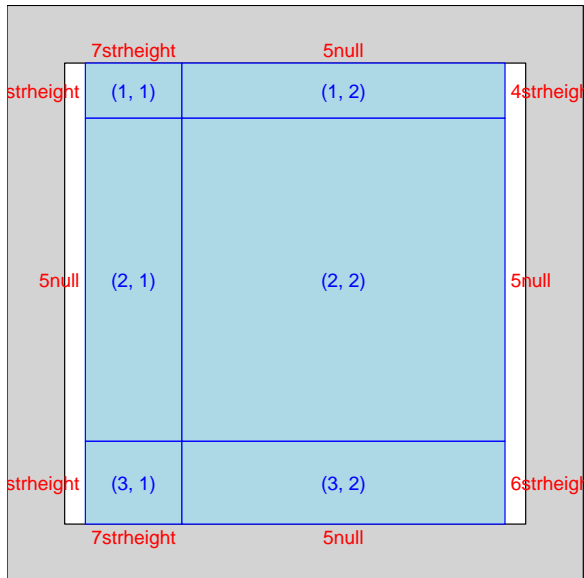


A still more powerful grid unit allows us to tile layouts to the heights and widths of specific strings

We can snugly fit text areas around the text as plotted in the device

...Leaving the maximum possible space for the graphic

Critical for tiling small multiples effectively - why base R does this badly



```

# Strings for titles
mainTitle <- "Avoid plot titles, except for small multiples"
yaxisTitle <- "Poverty Reduction"
xaxisTitle <- "Effective Number of Parties"

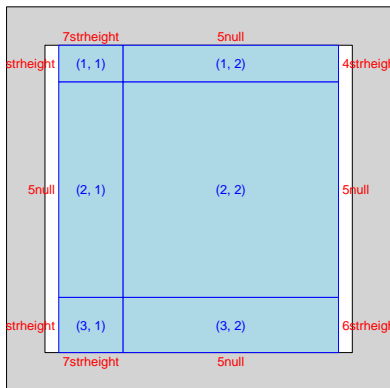
# Set up layout widths
wd <- unit.c(unit(7,"strheight", yaxisTitle),
             unit(5,"null"))

# Set up layout heights
ht <- unit.c(unit(4,"strheight", mainTitle),
             unit(5,"null"),
             unit(6,"strheight", xaxisTitle))

# Create layout
overlay <- grid.layout(nrow=3, ncol=2,
                      widths=wd, heights=ht,
                      respect=TRUE)

# "Push" layout to create initial viewport
pushViewport(viewport(layout=overlay))

```



Diagnostic plot of
revised layout

```
# 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"
                      ))
```

Avoid plot titles, except for small multiples

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

Our plot so far...

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



```
# Push 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(yaxisTitle,
          x=unit(0.15, "npc"),
          y=unit(0.5, "npc"),
          rot=90
          )
```

```
upViewport(1)
```

Avoid plot titles, except for small multiples

Poverty reduction

Our plot so far...

```
# Push 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(xaxisTitle,
         x=unit(0.5, "npc"),
         y=unit(0.25, "npc")
         )

upViewport(1)
```

Avoid plot titles, except for small multiples

Poverty reduction

Effective number of parties

Our plot so far...

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

```
# get the fit from the data
fit <- mmest.fit(y=data$povertyReduction,
                x=log(data$effectiveParties),
                ci=0.95)
```

Avoid plot titles, except for small multiples

Poverty reduction

Effective number of parties

Our plot so far...

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

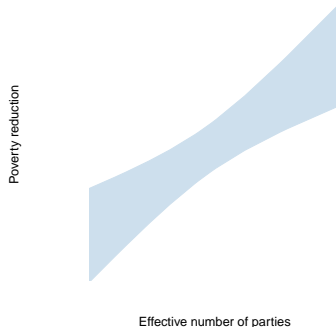
# Make the y-coord of a CI polygon
ypoly <- c(fit$lower, rev(fit$upper), fit$lower[1])

# Pick a nice blue
niceblue <- brewer.pal(3,"Set1")[2]

# Choose a pastel color for the polygon
pastelblue <- lighten("blue")

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

Avoid plot titles, except for small multiples



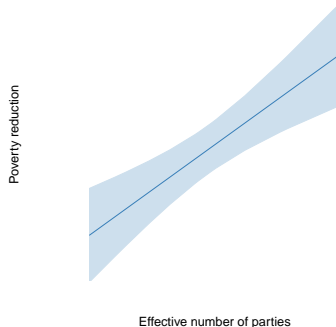
Our plot so far...

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

# If you want a box around the graph,
# now's the time to plot it
#grid.rect(gp=gpar(linejoin="round"))

# Return to layout level
upViewport(1)
```

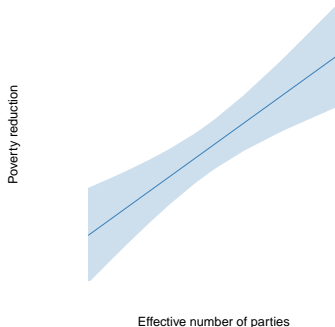
Avoid plot titles, except for small multiples



Our plot so far...

```
# 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(limits[1], limits[2]),  
                    yscale=c(limits[3], limits[4]),  
                    gp=gpar(fontsize=12),  
                    name="mainplot",  
                    clip="off"  
                ))  
  
# Make axis as a "grob" (graphical object),  
# but don't plot yet  
yaxis <- yaxisGrob(at = yat,    # Where to put ticks  
                  label = TRUE, # Only takes logical values!  
                  main = TRUE  # Top (TRUE) or bottom (FALSE)  
                  )
```

Avoid plot titles, except for small multiples

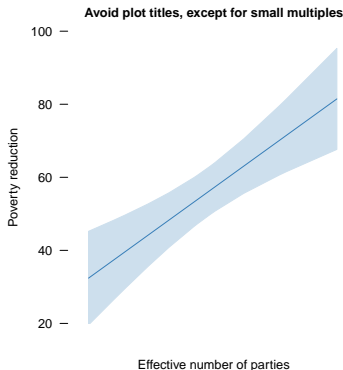


Our plot so far...

```
# Edit the y-axis to remove the major line  
yaxis <- editGrob(yaxis,  
                 gPath("major"),  
                 major=FALSE  
                 )
```

```
# Edit the y-axis to remove the major line  
yaxis <- removeGrob(yaxis, gPath("major"))
```

```
# Draw the y-axis grob  
grid.draw(yaxis)
```



Our plot so far...

```

# Recall that our tick locations are:
# xat <- c(2, 3, 4, 5, 6, 7)

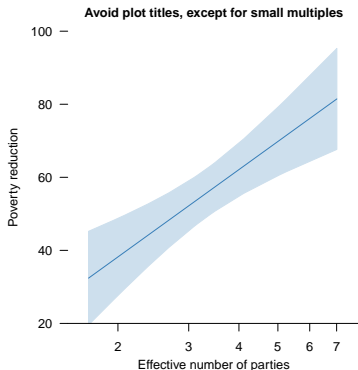
# Make an x-axis as a "grob"
xaxis <- xaxisGrob(at = log(xat),
                  label = TRUE,
                  main = TRUE)

# Edit the x-axis to show unlogged units
xaxis <- editGrob(xaxis,
                 gPath("labels"),
                 label = xat)

# Edit the x-axis to fit the plot
xaxis <- editGrob(xaxis,
                 gPath("major"),
                 x=unit(c(0, 1), "npc")
                 )

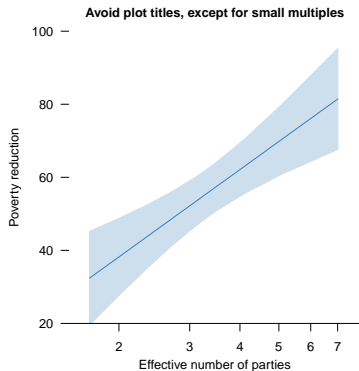
# Draw the x-axis grob
grid.draw(xaxis)

```



Our plot so far...


```
# Return to layout viewport  
upViewport(1)  
  
# Save the graph!  
dev.off()
```



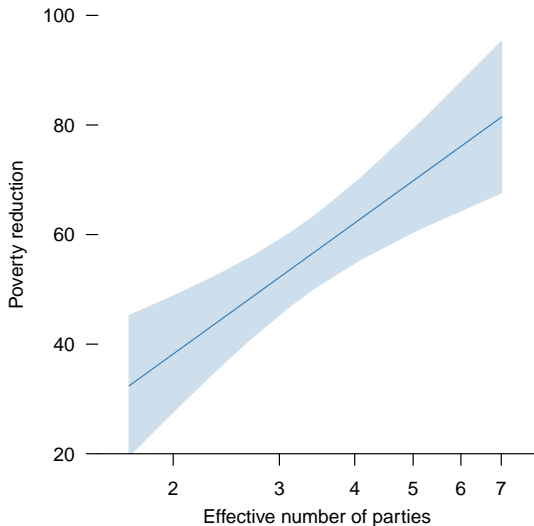
Our plot so far...

grid provides detailed control over coordinates, plotting elements, layout, spacing, alignment, etc.

Essential for legible small multiples

Vastly more powerful than base graphics

And vastly more time consuming to make



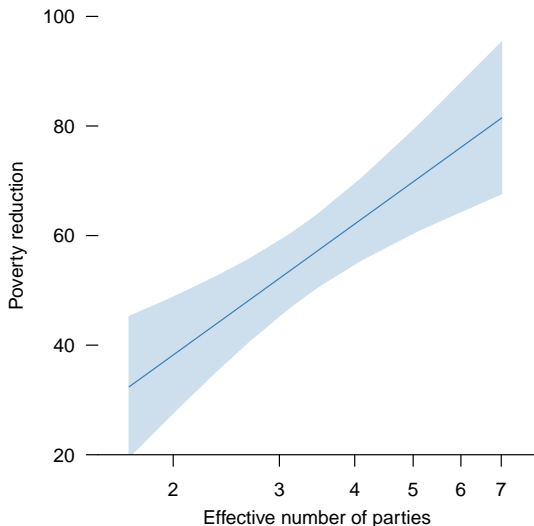
grid is the workhouse behind
powerful but easier to use
graphics packages:

lattice

ggplot2

tile

If you are writing a new
graphics *function*, write it
using either `grid` primitives or
a package based on `grid`



Suppose we wanted to add more features to the plot, such as replacing the y ticks with thin gridlines?

Turns out to be a lot of work!
(Code on course site)

My package `tile` provides users easy ways to make these sorts of changes, but only because `grid` is in the background doing the lifting

