CSSS 569: Visualizing Data

Graphical Programming, Part I. Using R graphics functions

Christopher Adolph*
University of Washington, Seattle

January 26, 2011

*Assistant Professor, Department of Political Science and Center for Statistics and the Social Sciences.

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 `ggplot` 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 $\{1,2,3,\ldots\}$

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` let's 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

Vector is 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:

`seq(from, to, by)`	generates a sequence
`rep(x,times)`	repeats `x`
`sort()`	sorts a vector from least to greatest
`rev()`	reverses the order of a vector
`rev(sort())`	sorts a vector from greatest to least

Matrices in R

Vector 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

`a[1,1]`	Gets the first element of a
`a[1:10,1]`	Gets the first ten rows of the first column
`a[,5]`	Gets every row of the fifth column
`a[4:6,]`	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:

`nrow()`	Gives the number of rows of the matrix
`ncol()`	Gives the number of columns
`t()`	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")
```

Dataframes can 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

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

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

to access the variables directly

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 comparisions:

```
< <= > >= == !=
```

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

```
&& || !
```

Math/stat fns:

```
log exp mean median mode min max sd var cov cor
```

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

R follows the usual order of operations; if it 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,                     # A dataframe containing
                                    # y, x1, x2, etc.
          na.action="")

# 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~x, 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()`

OOP: functions respond to class of objects

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()`

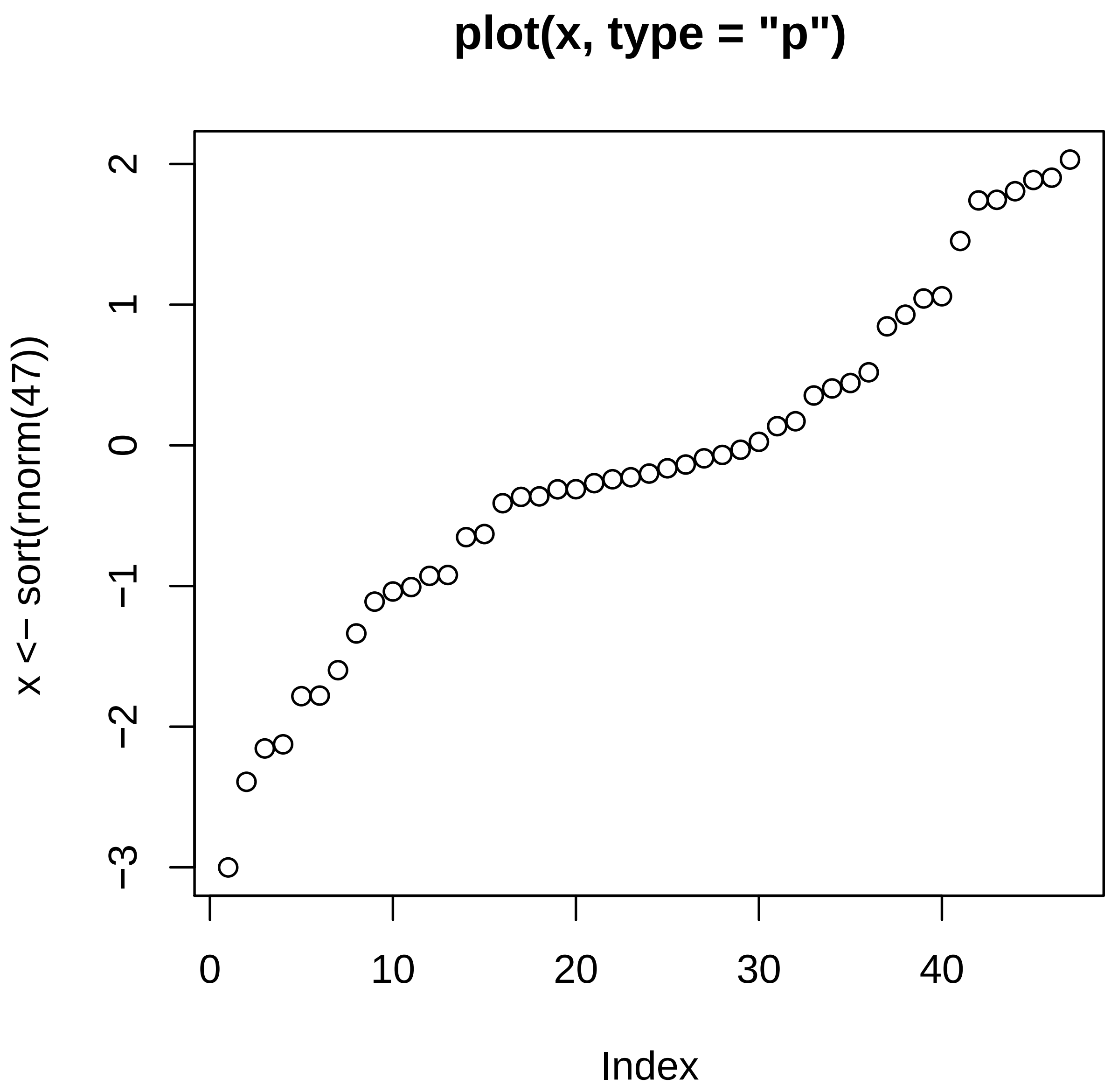
I did `help.search(‘‘plot’’)` on a full install of `R` packages

Found lots of neat plotting functions in packages I'd never heard of.

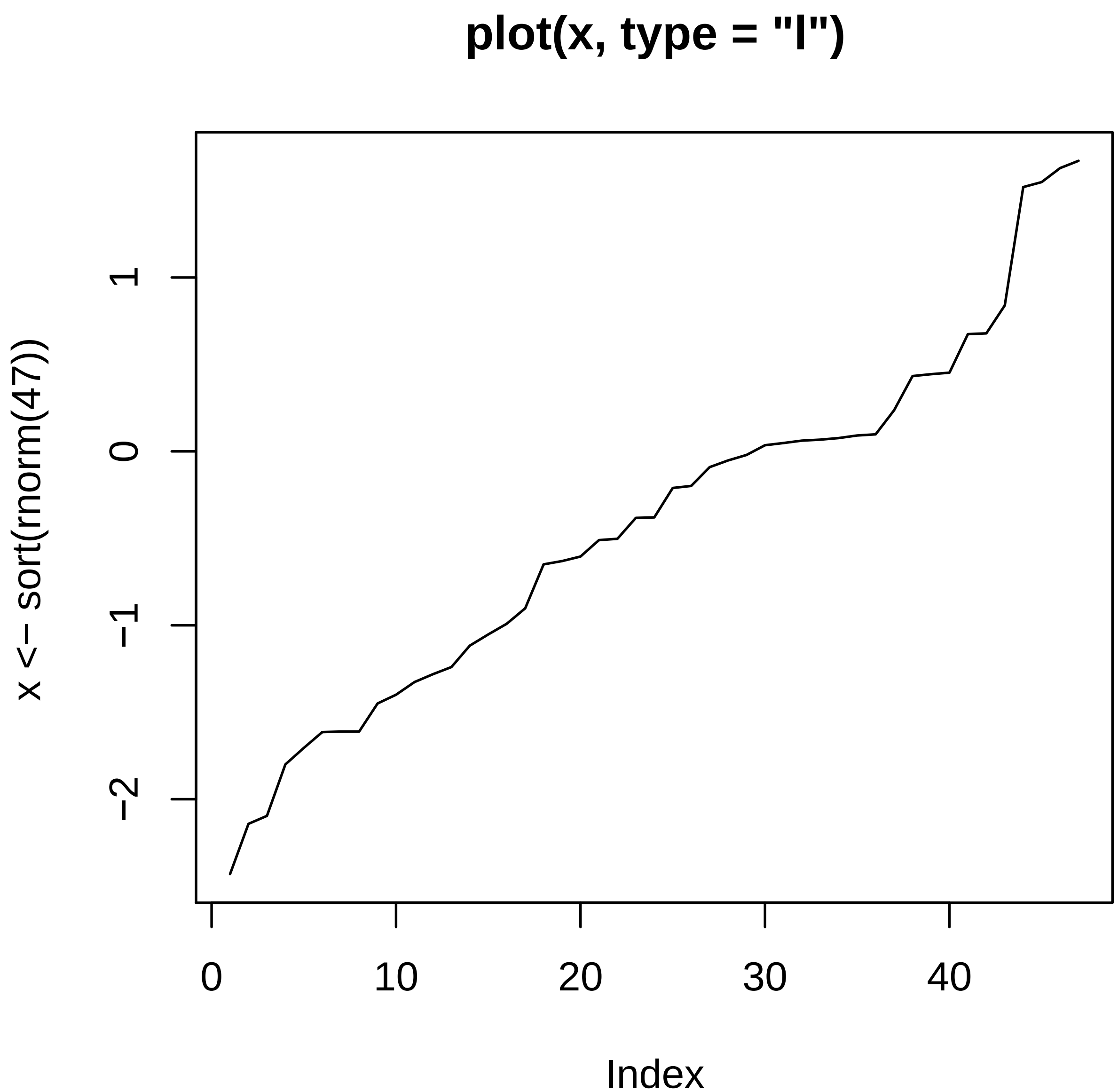
Some major high-level graphics commands

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	

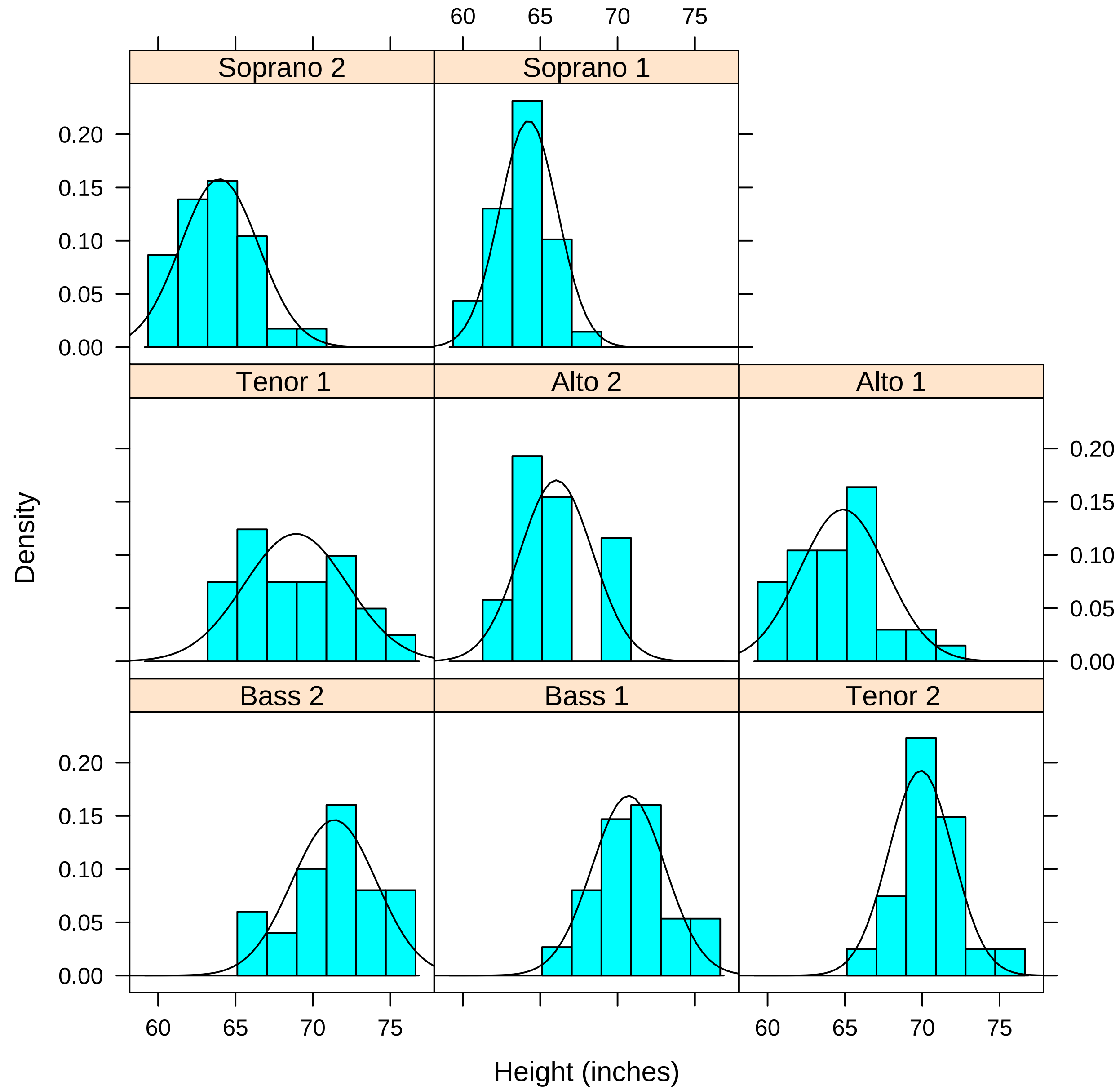
Scatterplot: `plot()`



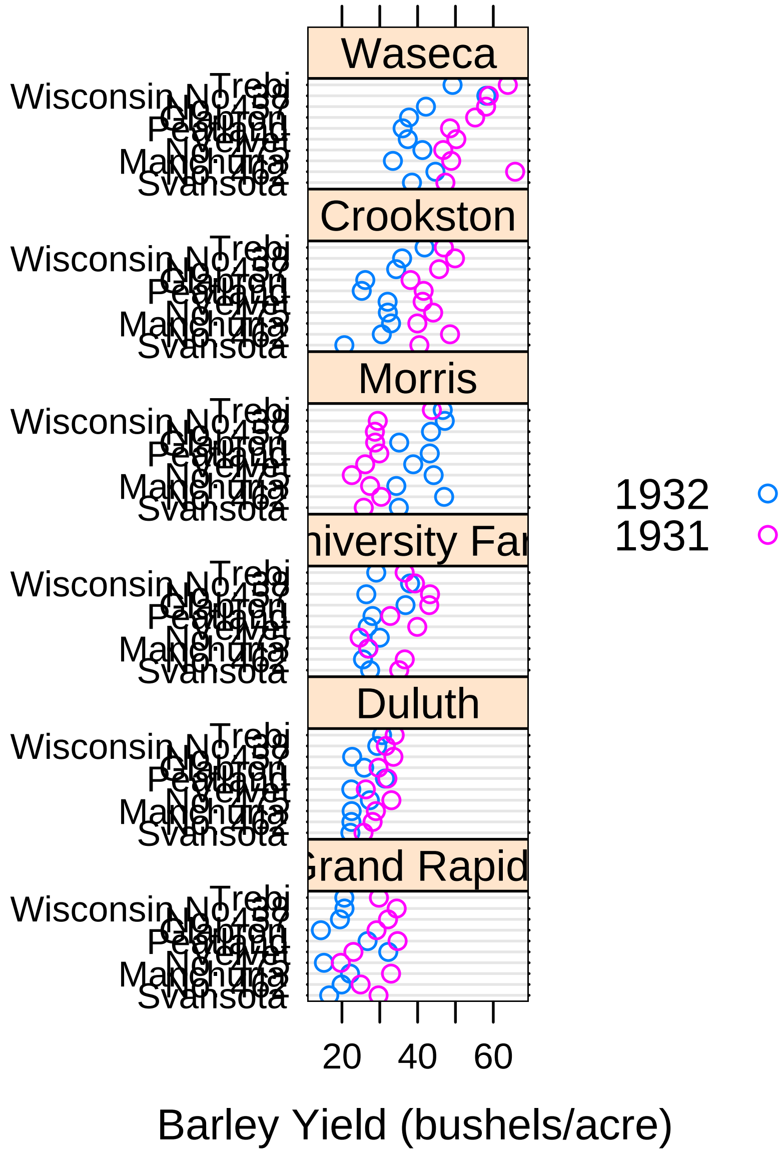
Line plot: `plot(...,type="l")`



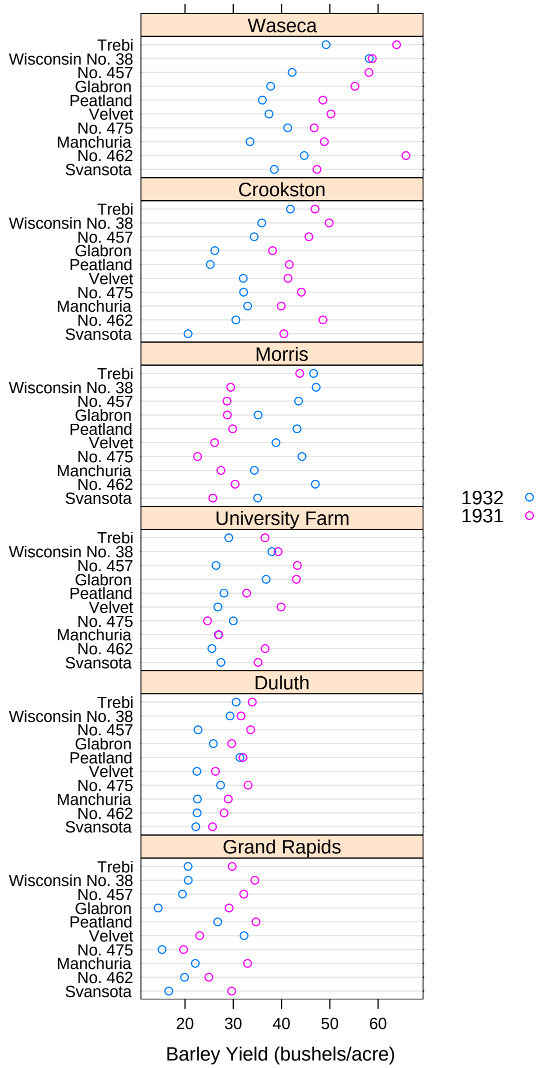
(Smoothed) Histograms: `densityplot()` & others



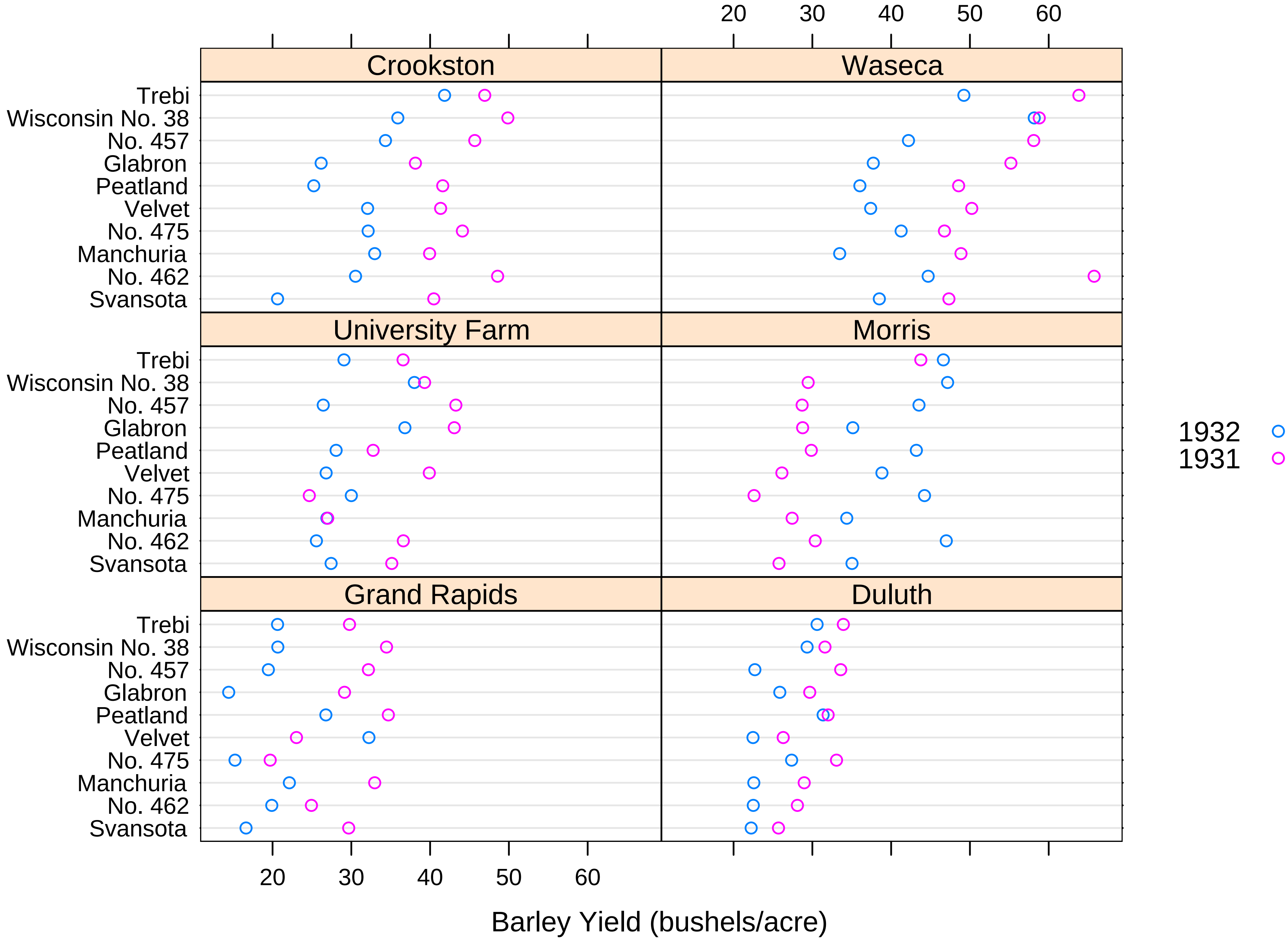
Dot plot: `dotplot()`



Dot plot is sensitive to device size



Dot plot has a `layout` option



Contour plot: `contour()`

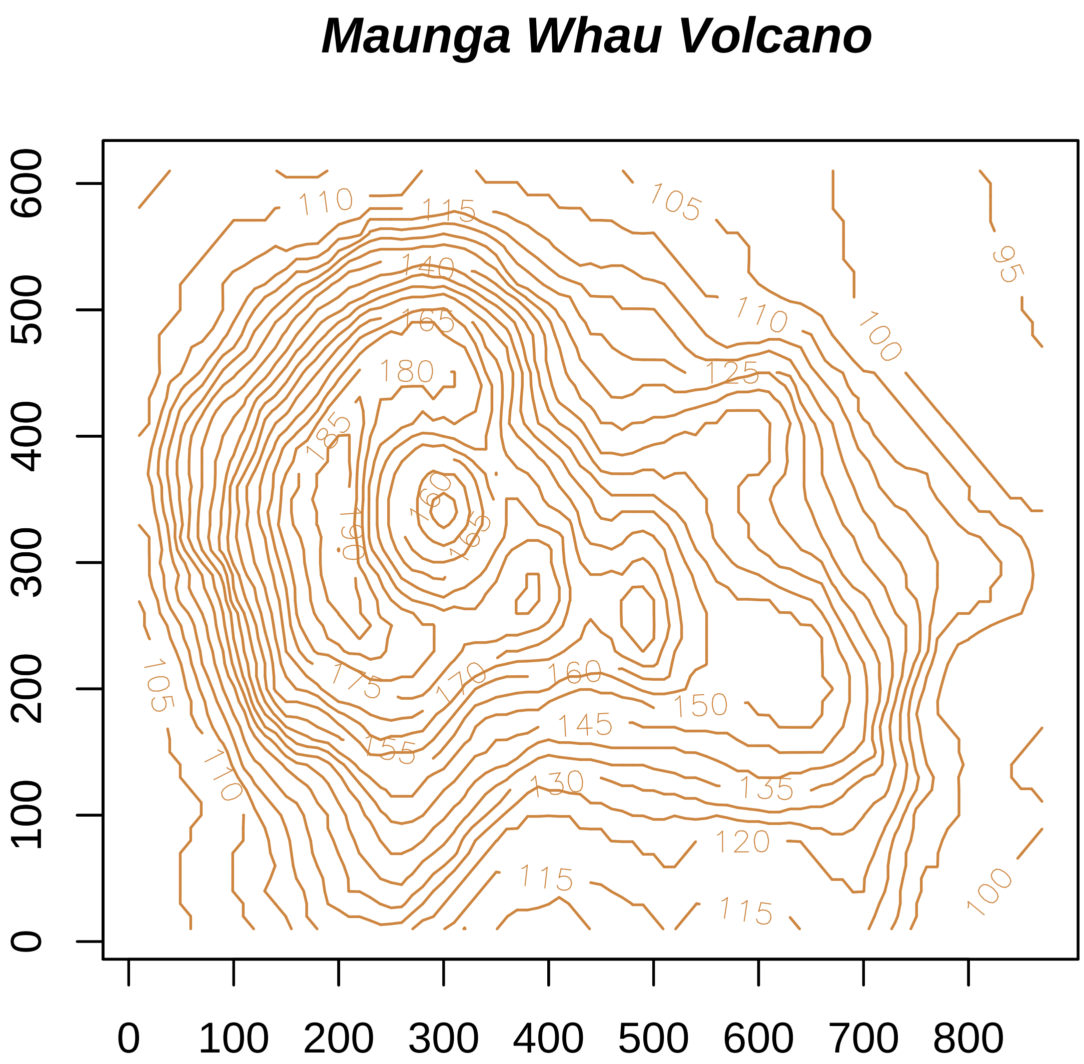


Image plot: `image()`

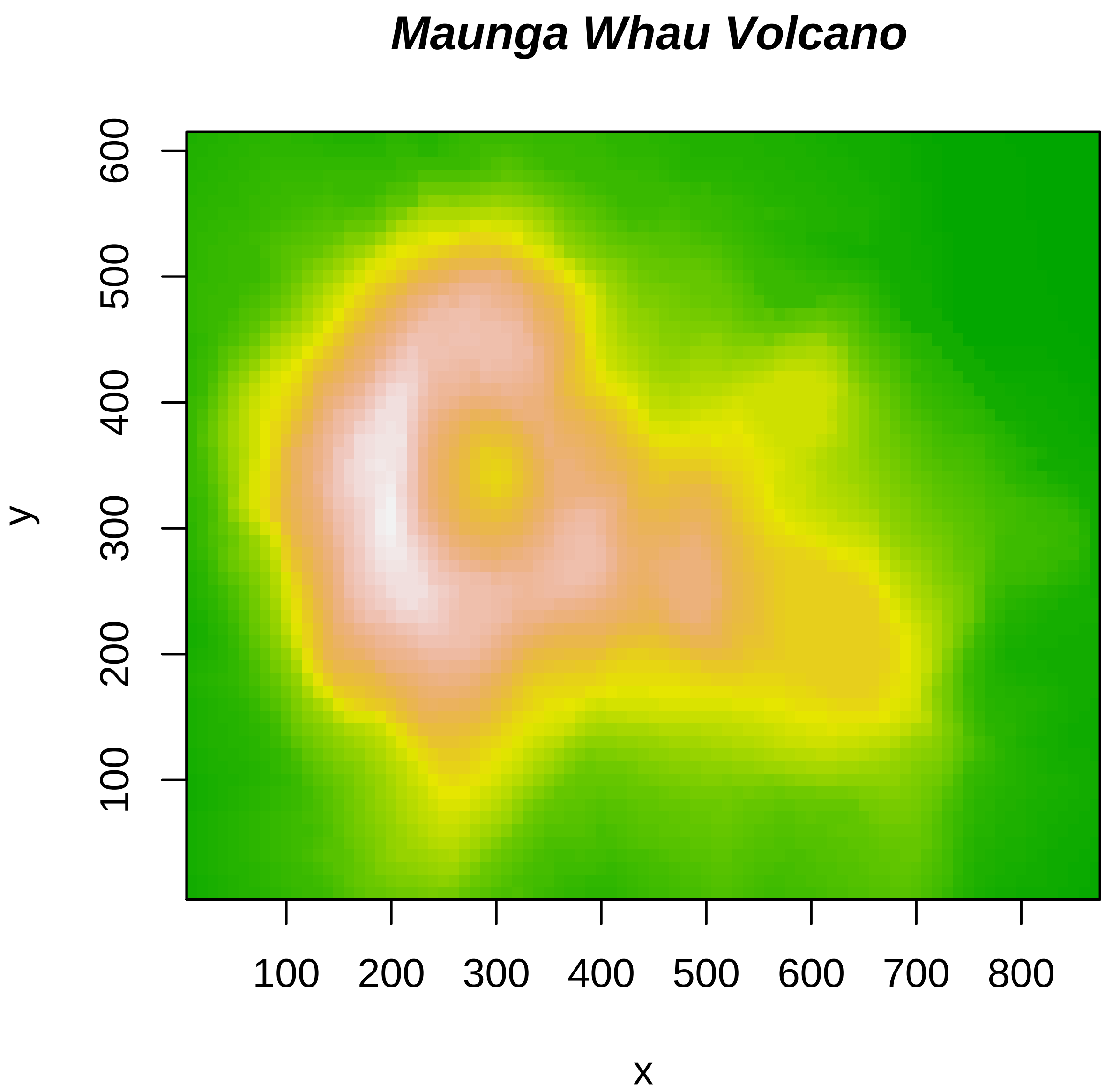
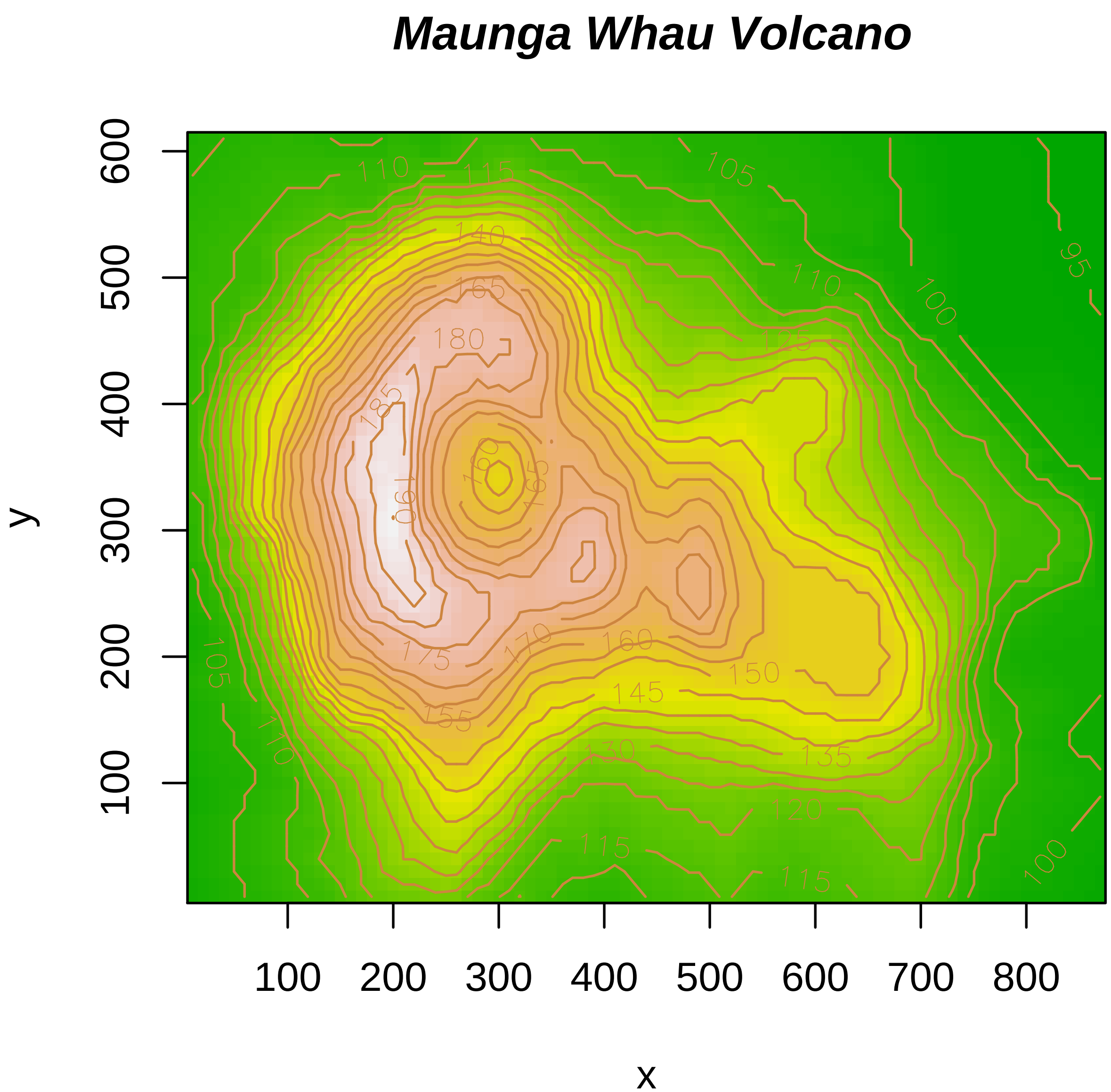
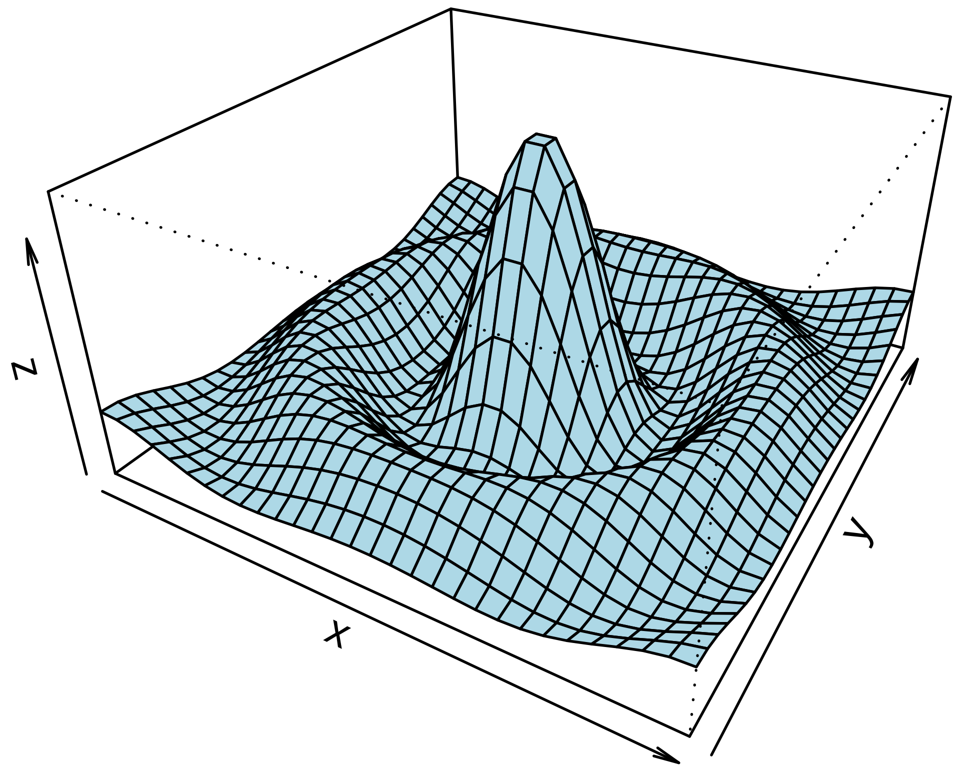


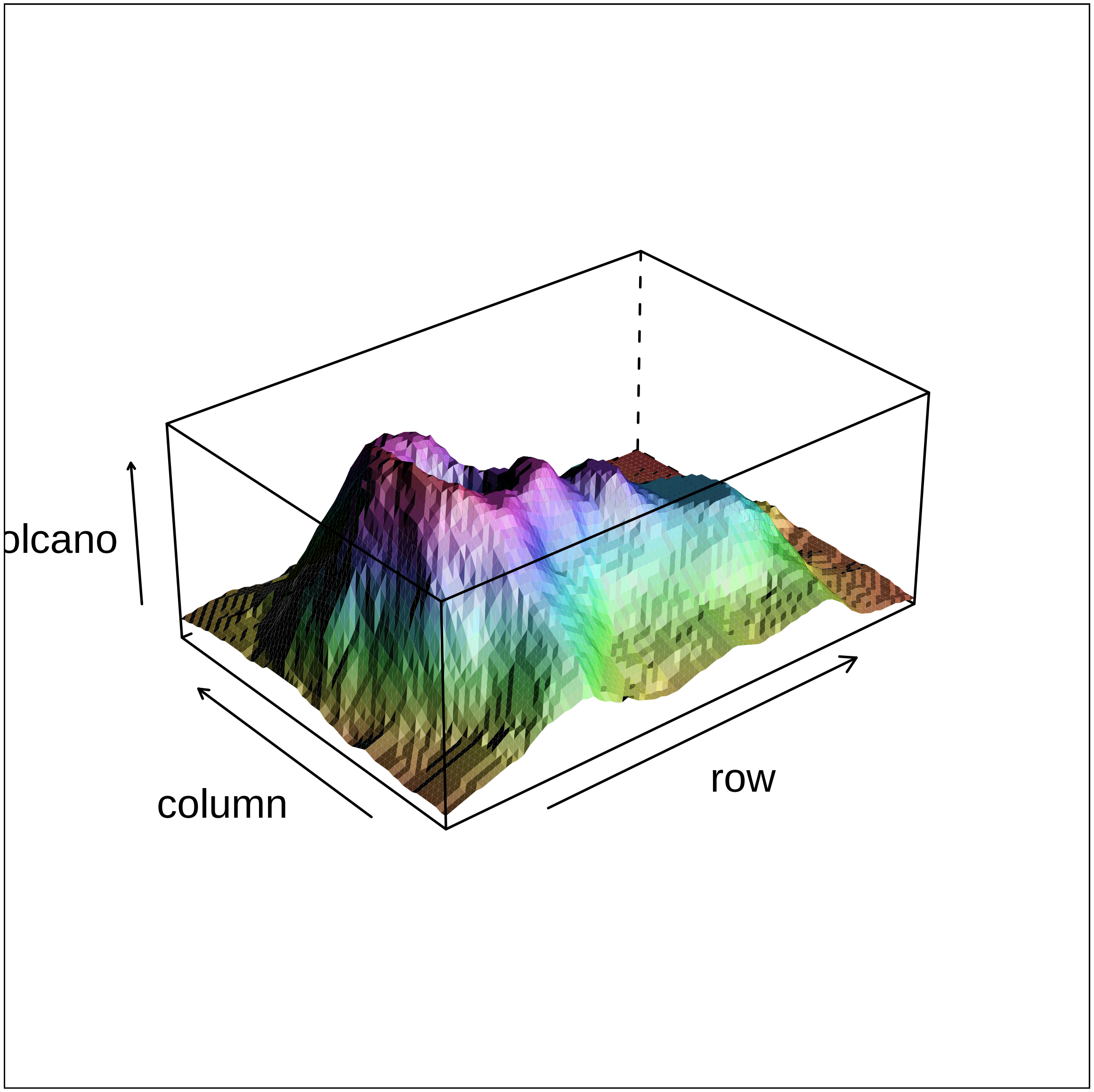
Image plot with contours: `contour(...,add=TRUE)`



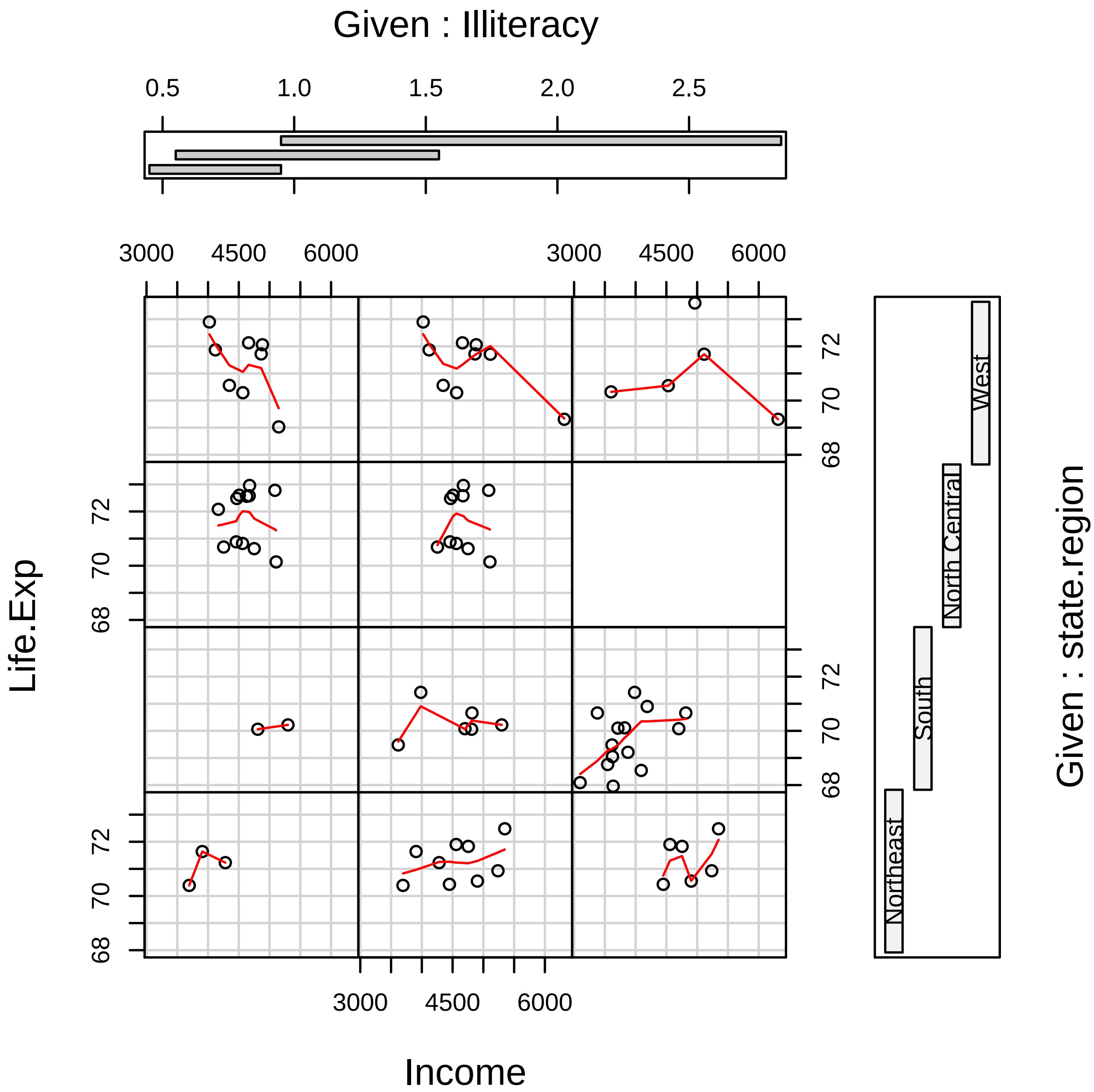
3D surface: `persp()`



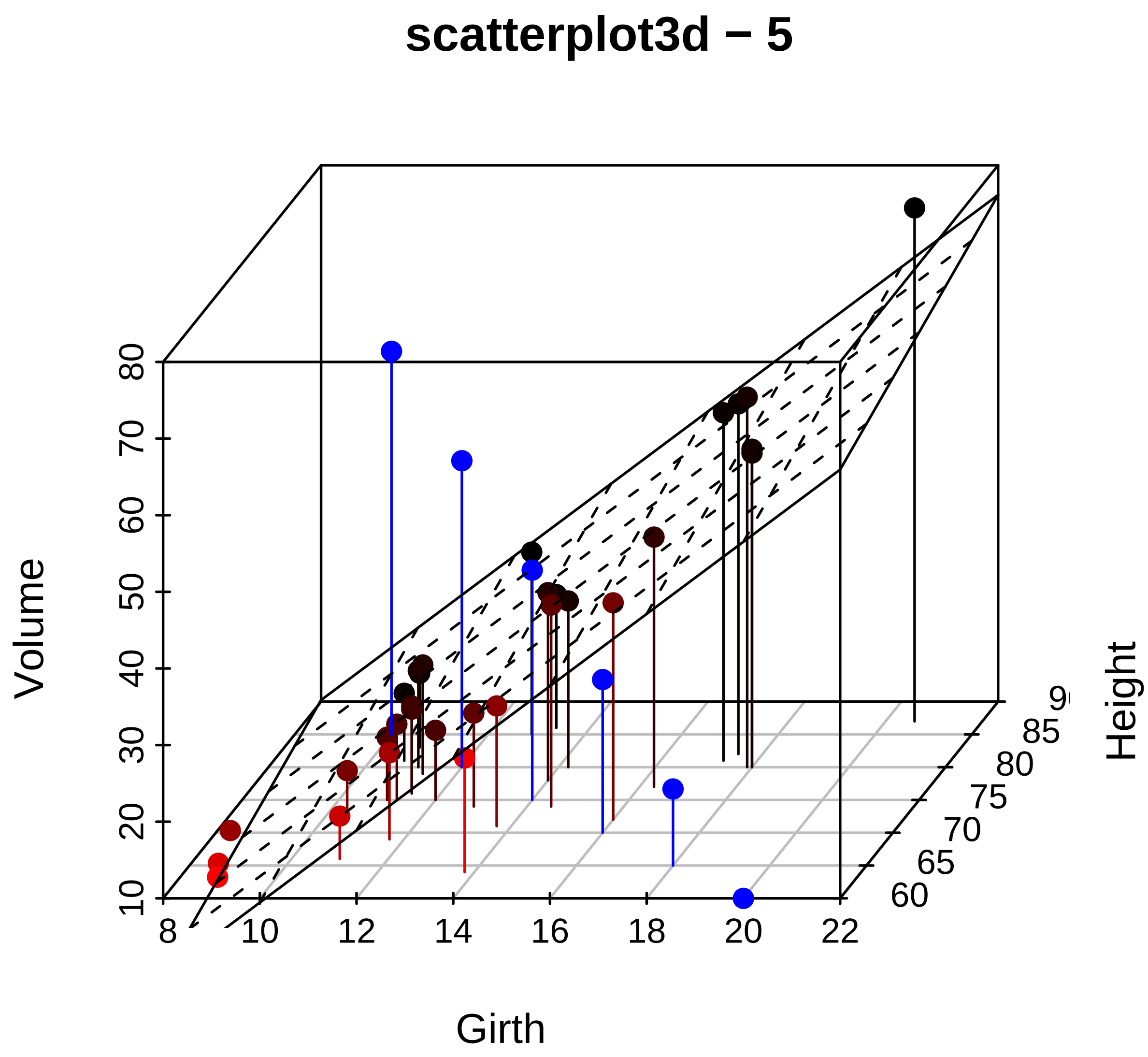
3D surface: `wireframe()`



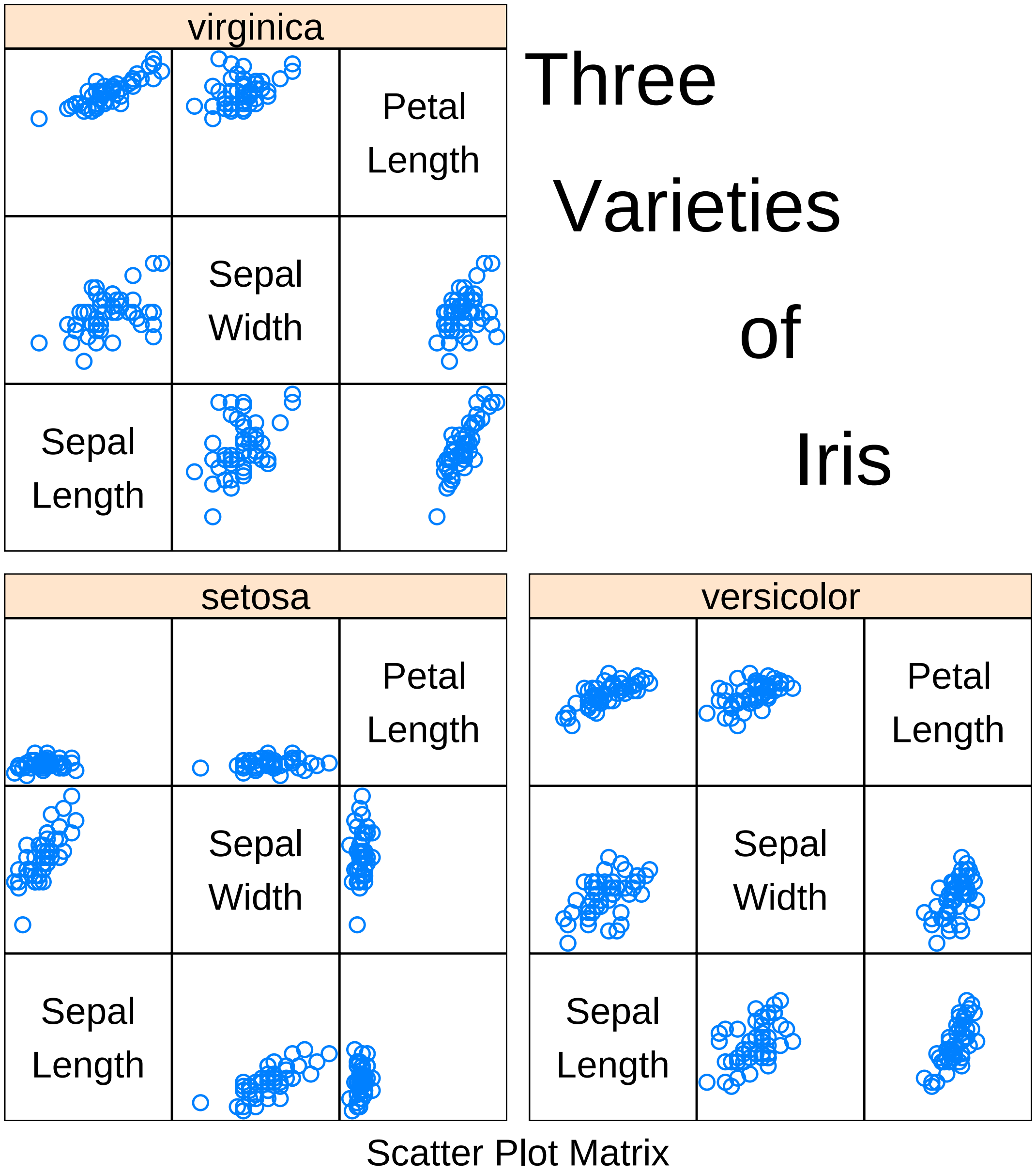
Conditional plots: `coplot()`



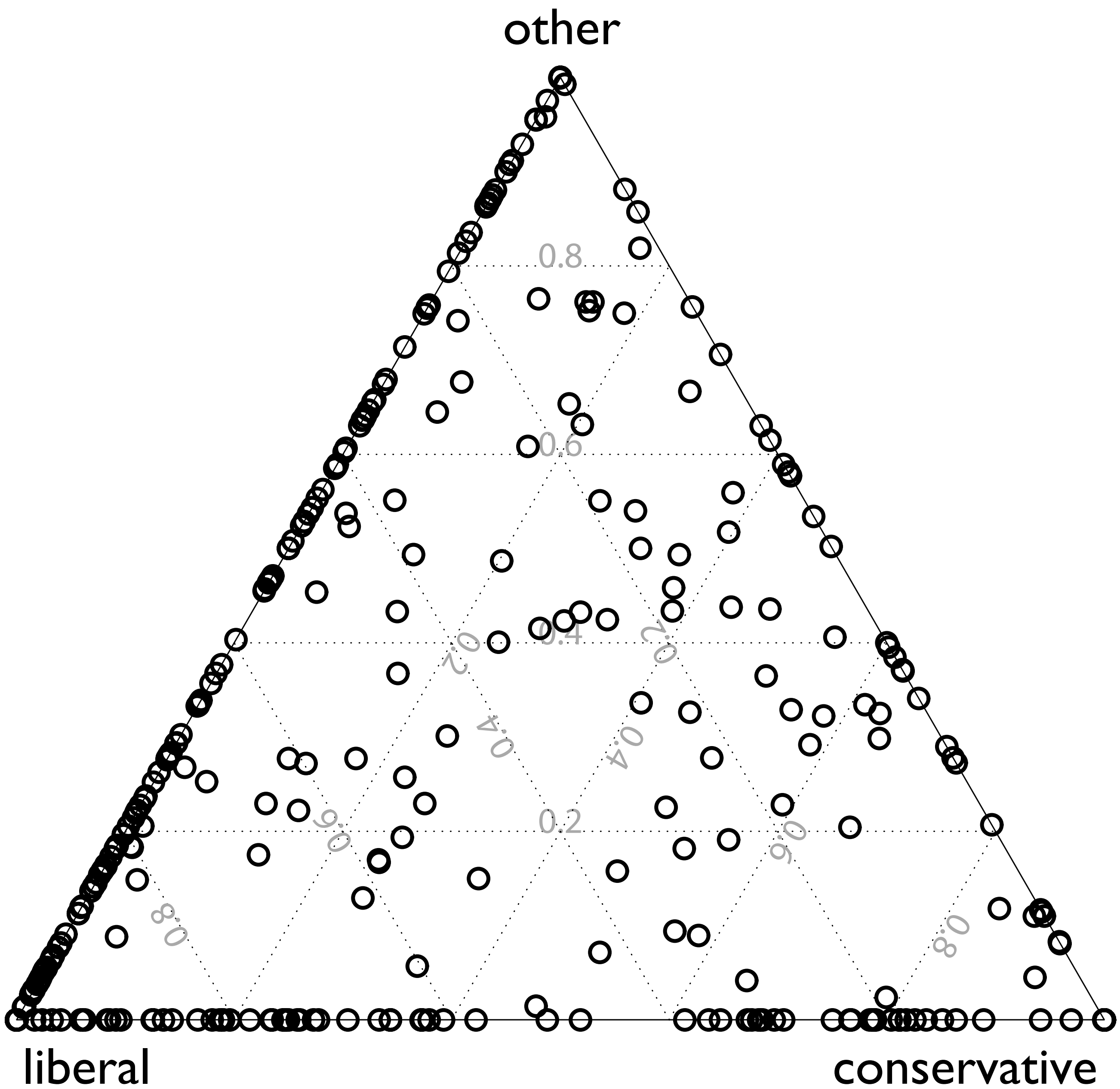
3D scatter: `scatterplot3d()` in own library



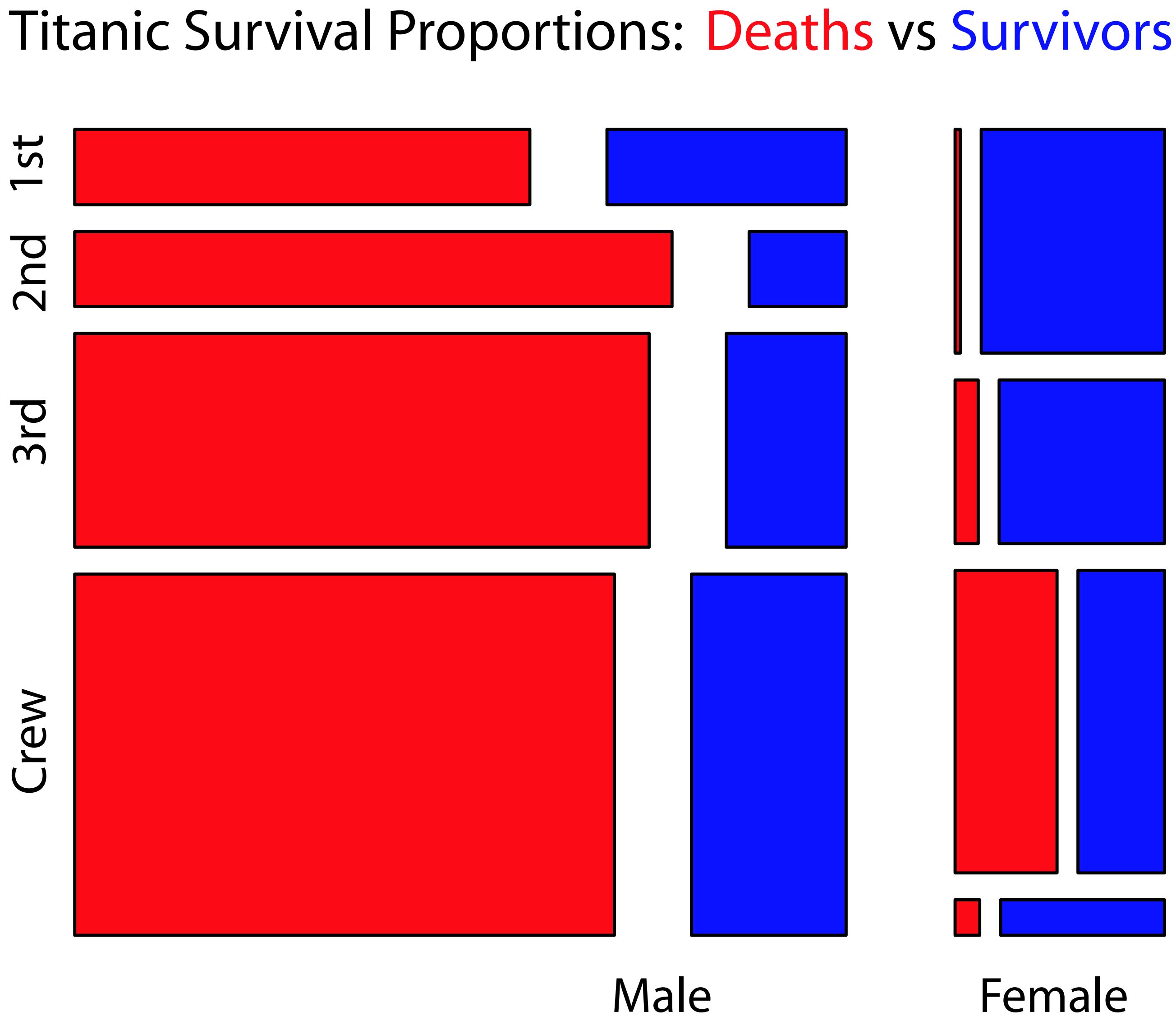
Scatterplot matrix: `splom()`



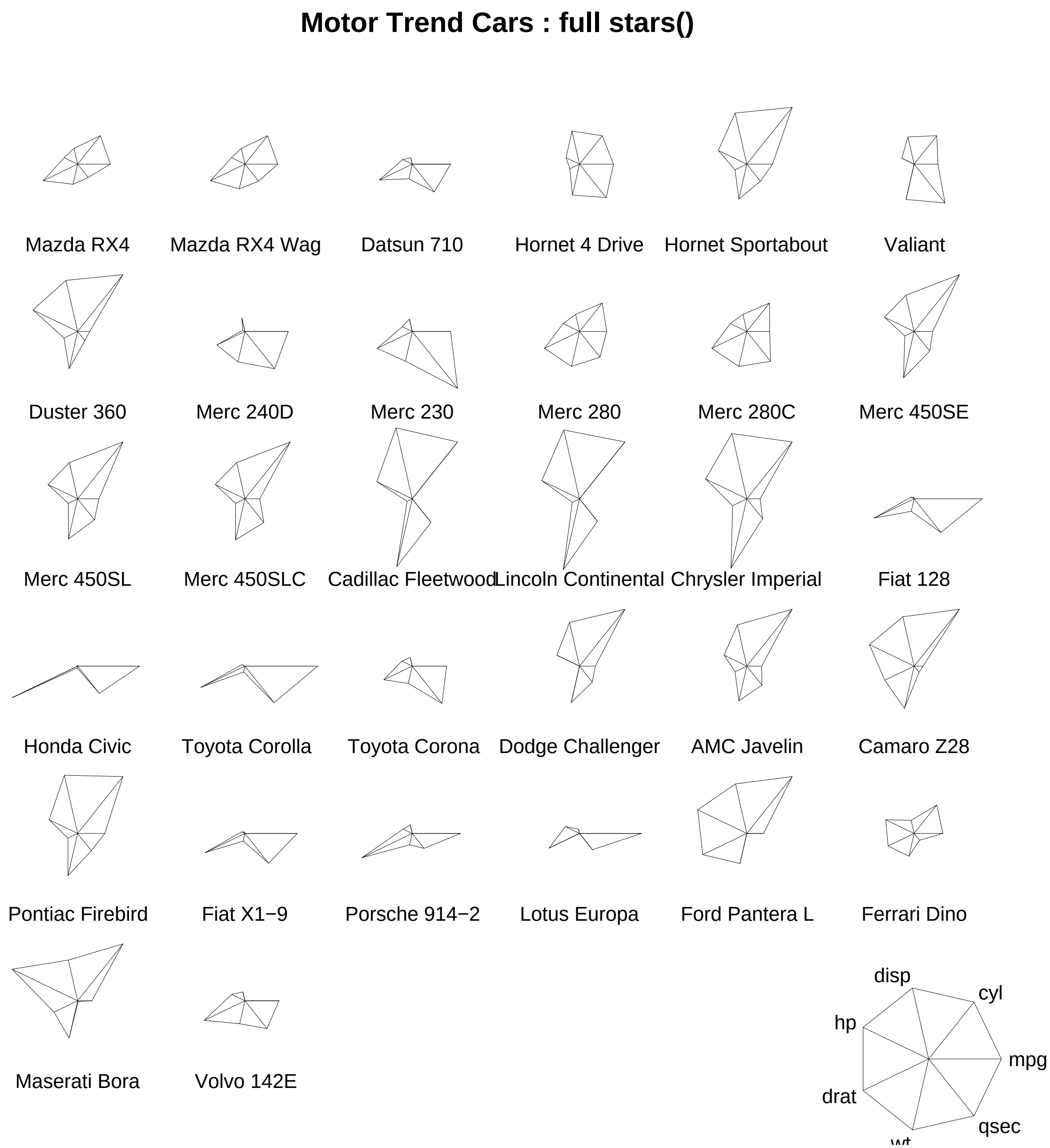
Ternary plot: `ternaryplot()` in `vcd`



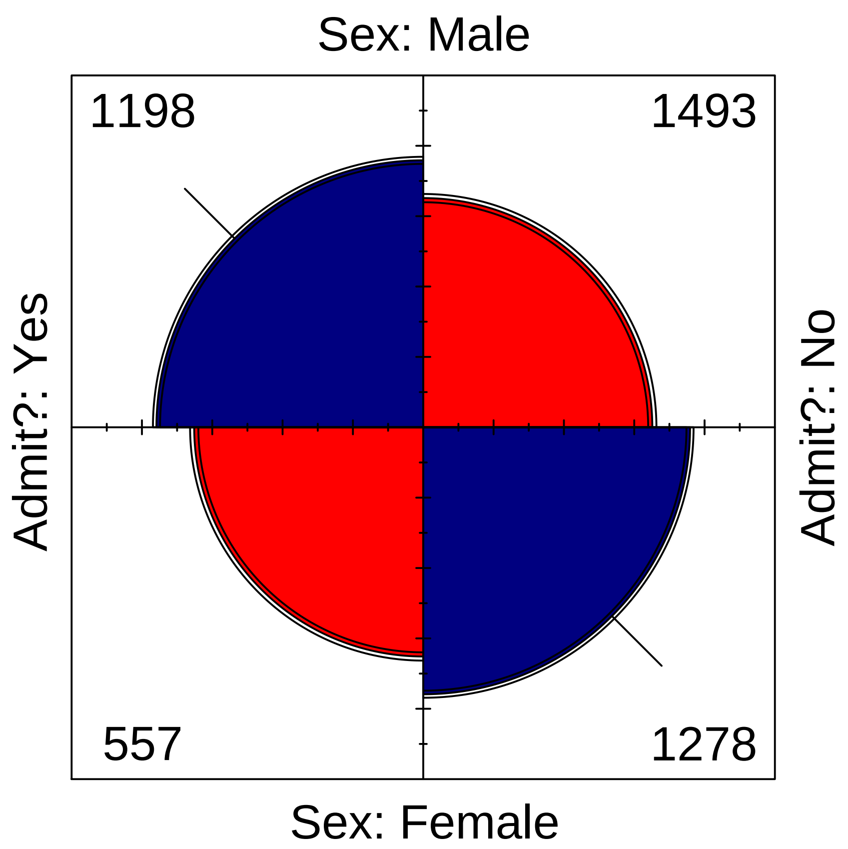
Mosaic plot: `mosaic()` in `vcd`



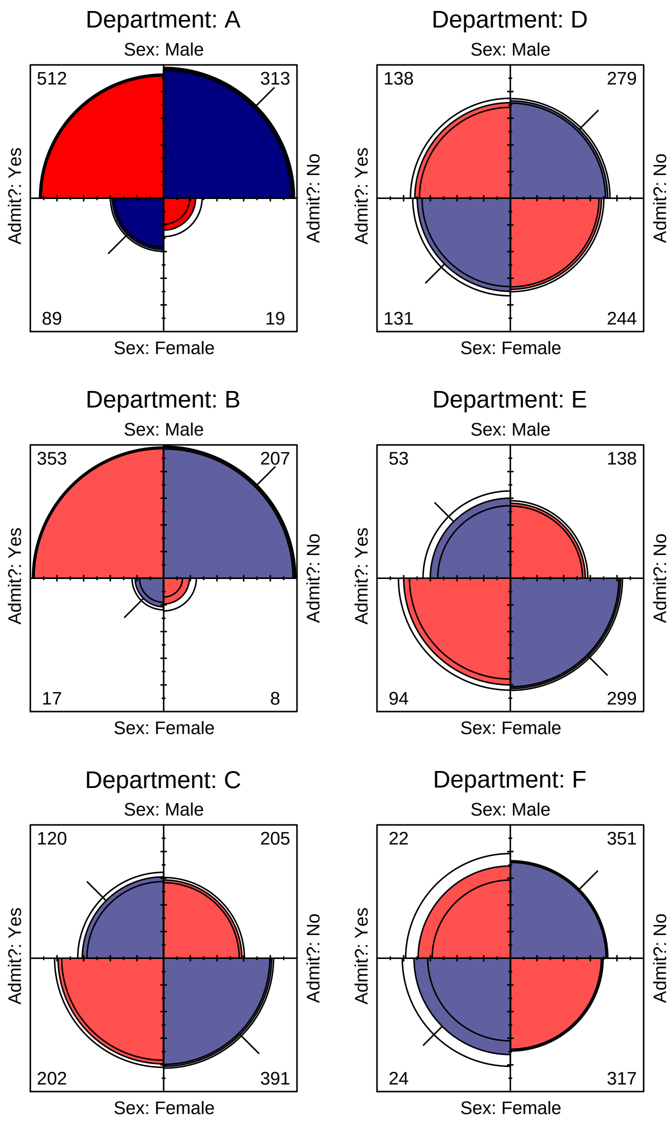
Star plot: `stars()`



Fourfold plot: `fourfoldplot()` in `vcd`



Fourfold plot: `fourfoldplot()` in `vcd`



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 throough `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

bg	background color
col	Color of lines, symbols in plot
col.axis	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)

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)

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

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`	`brewer.pal()`
`colorspace`	`sequential_hcl()` and `diverge_hcl()`

   `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

lab	Number of ticks
xaxp	Number of ticks for xaxis
tck,tcl	Length of ticks relative to plot/text
mgp	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 formating 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,

`expression(bar(x))` $\bar{x}$
`expression(x[i])` x_i
`expression(x^{-2})` 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

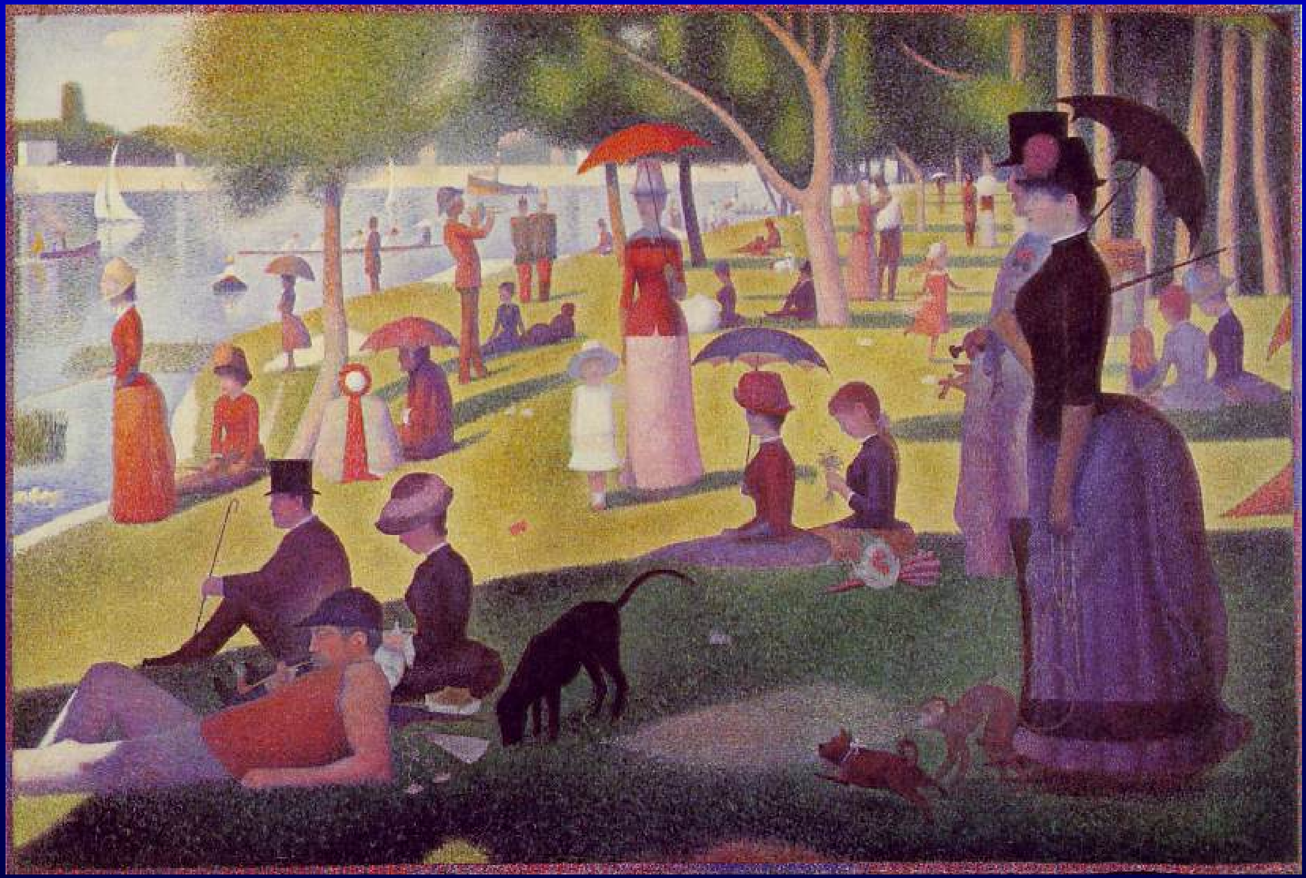
Must create the canvas before you draw anything

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:

win.metafile()	wmf, Windows media file
pdf()	pdf, Adobe portable data file
postscript()	postscript file (printer language)
x11()	opens a screen; all computers
windows()	opens a screen; PC only
quartz()	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