Empirical research using R:
in economics, finance, and management
Essentials, real examples, and troubleshooting

Compiled from session05.tex @ 2024-05-15 21:06:14+02:00.

Day 5: Graphics and summaries in R

Andreï V. Kostyrka

2nd of October 2023

UNIVERSITÉ DU LUXEMBOURG

Quick recap

We learned:

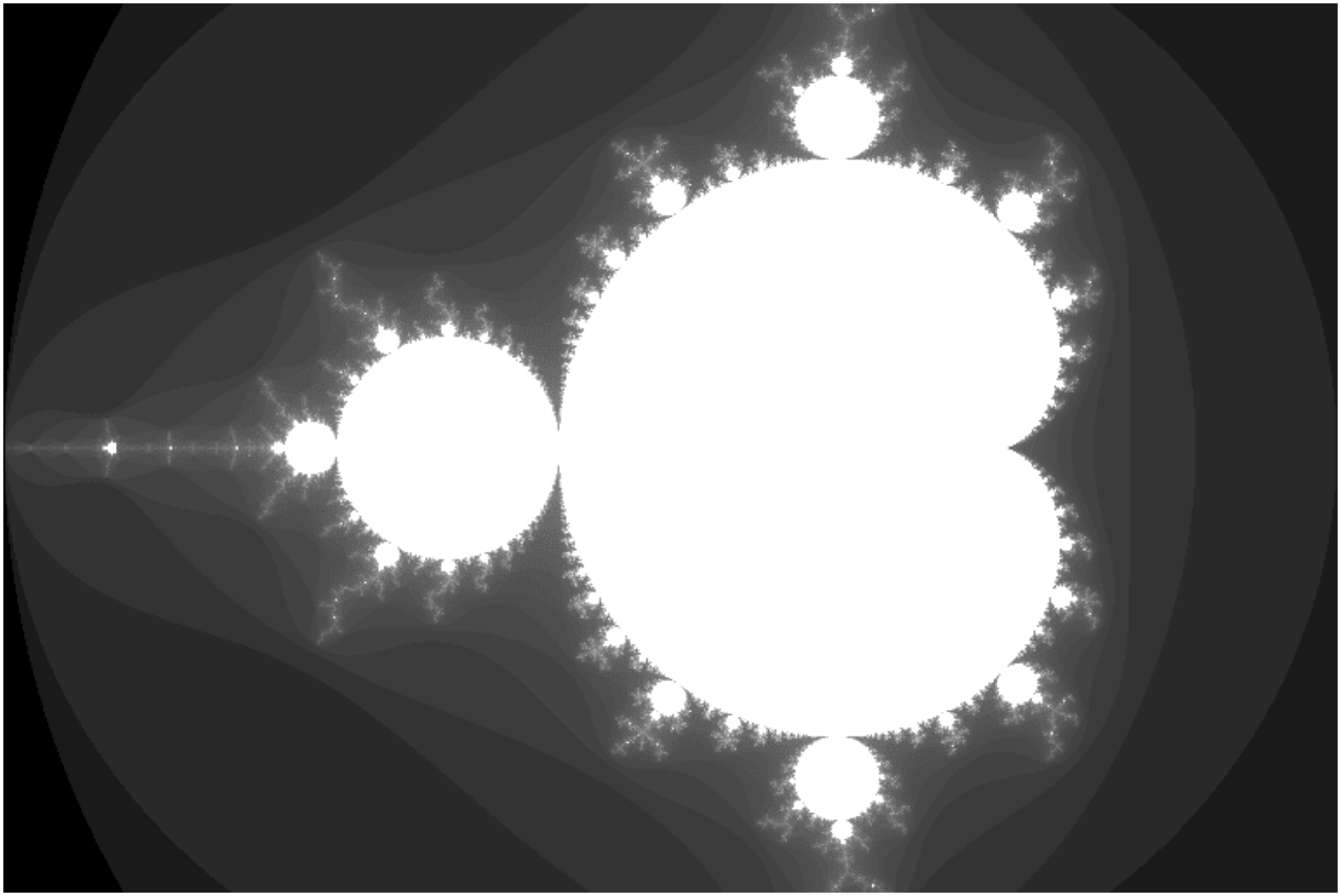
- How to write functions
- How to debug functions
- How to speed up functions

Today, we shall learn how to make beautiful plots and illustrations commonly used in applied research.

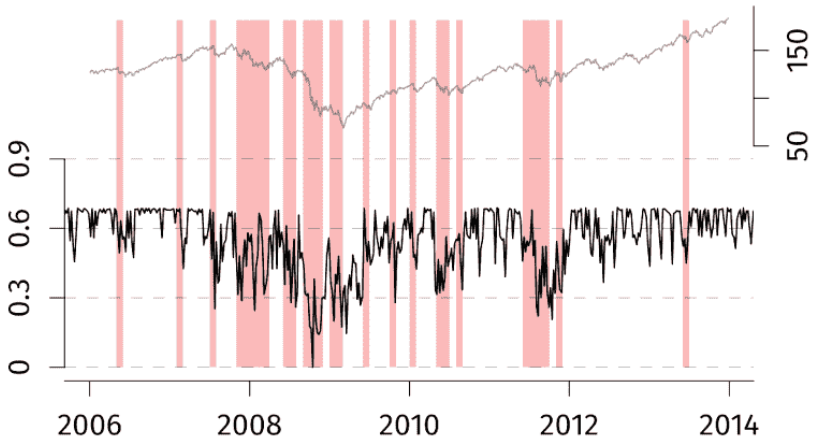
Presentation structure

1. Basics of image processing
2. Plots, parameters, and devices
3. Popular plots and tips for them
4. Summarising and aggregating data
5. 3D graphics, animations, and video encoding

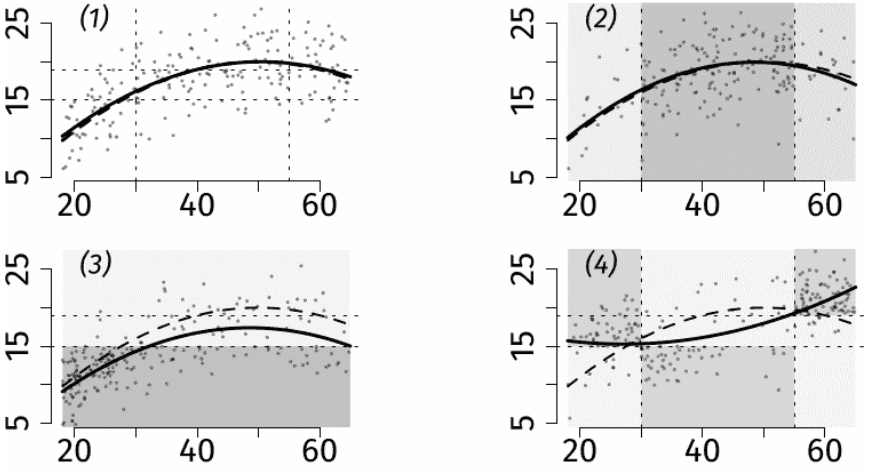
One can draw anything in R



Personal experience: thesis defence



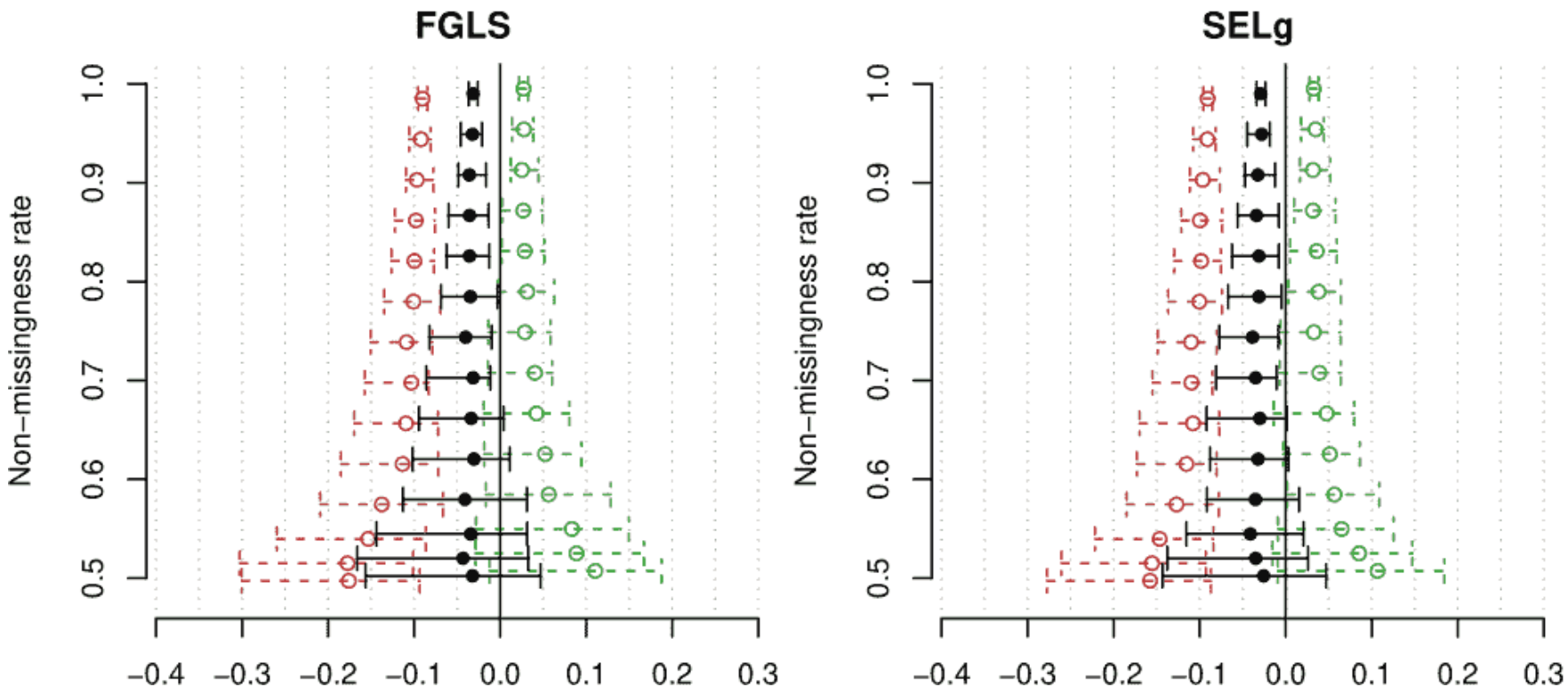
Top: SPDR S&P 500 ETF price. Bottom: conditional correlation between the 'good' and 'bad' market shocks.



Dashed line: true law $\mathbb{E}(Y \mid X)$. Solid lines: OLS model fits. The darker the shade, the higher the retention probability.

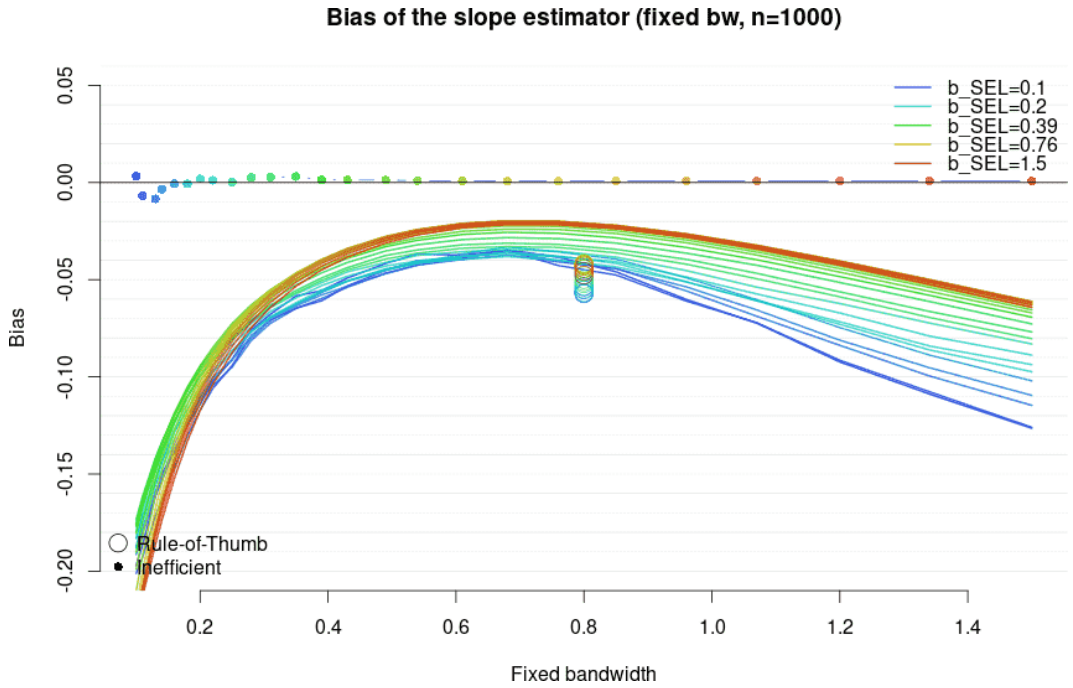
Parsimonious presentation, minimum notation, only one aspect highlighted.

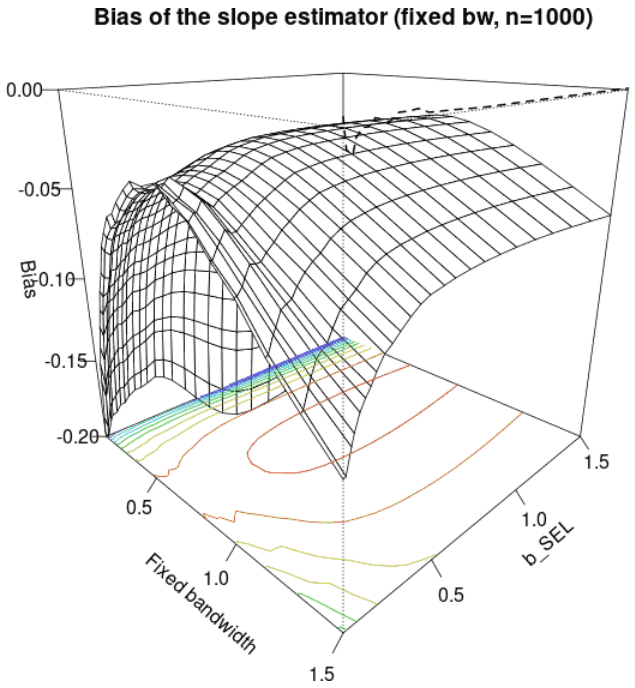
Personal experience: ongoing research



When proposing something new, make extensive comparisons against existing methods / results.

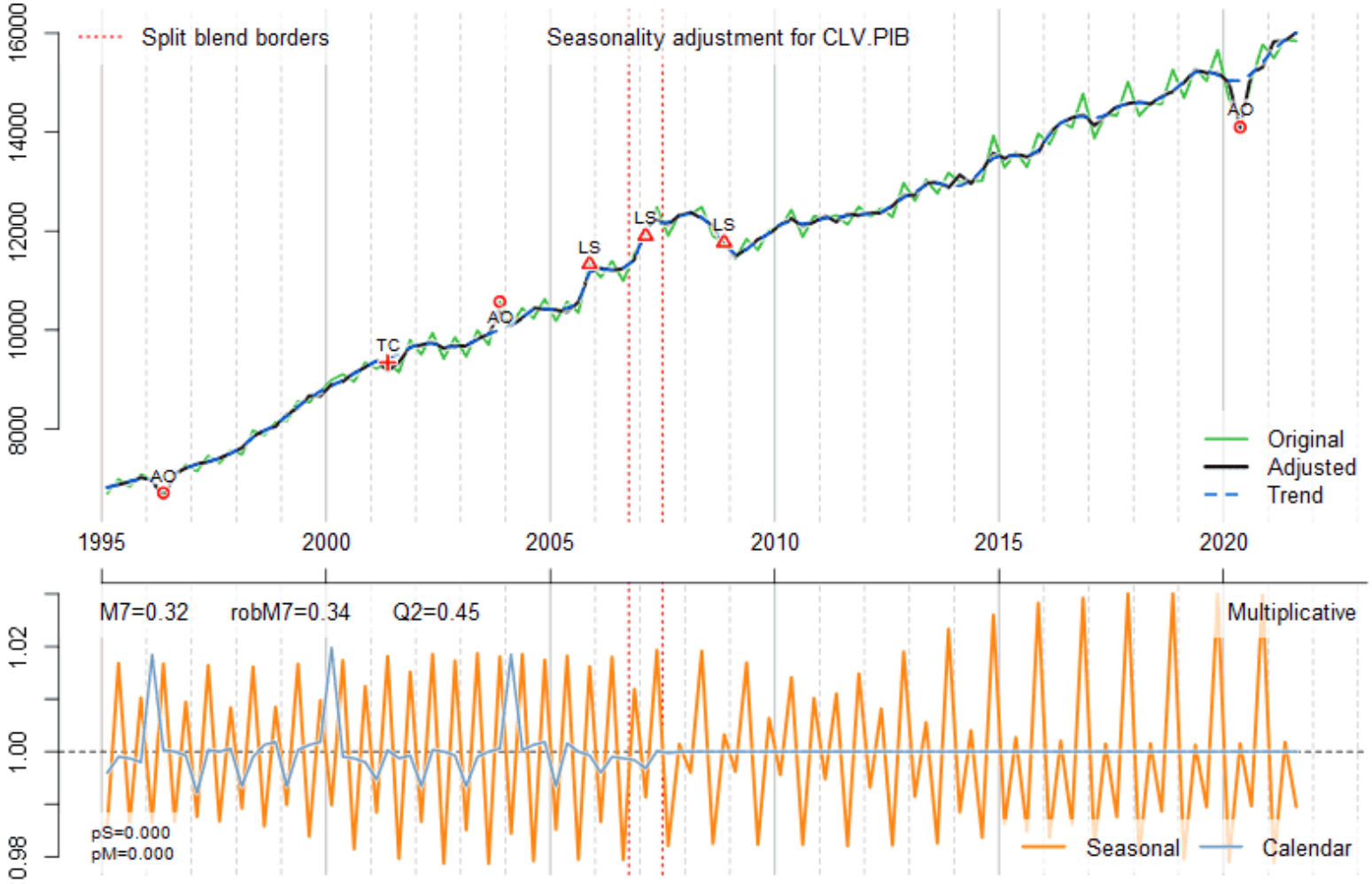
Personal experience: technical discussions





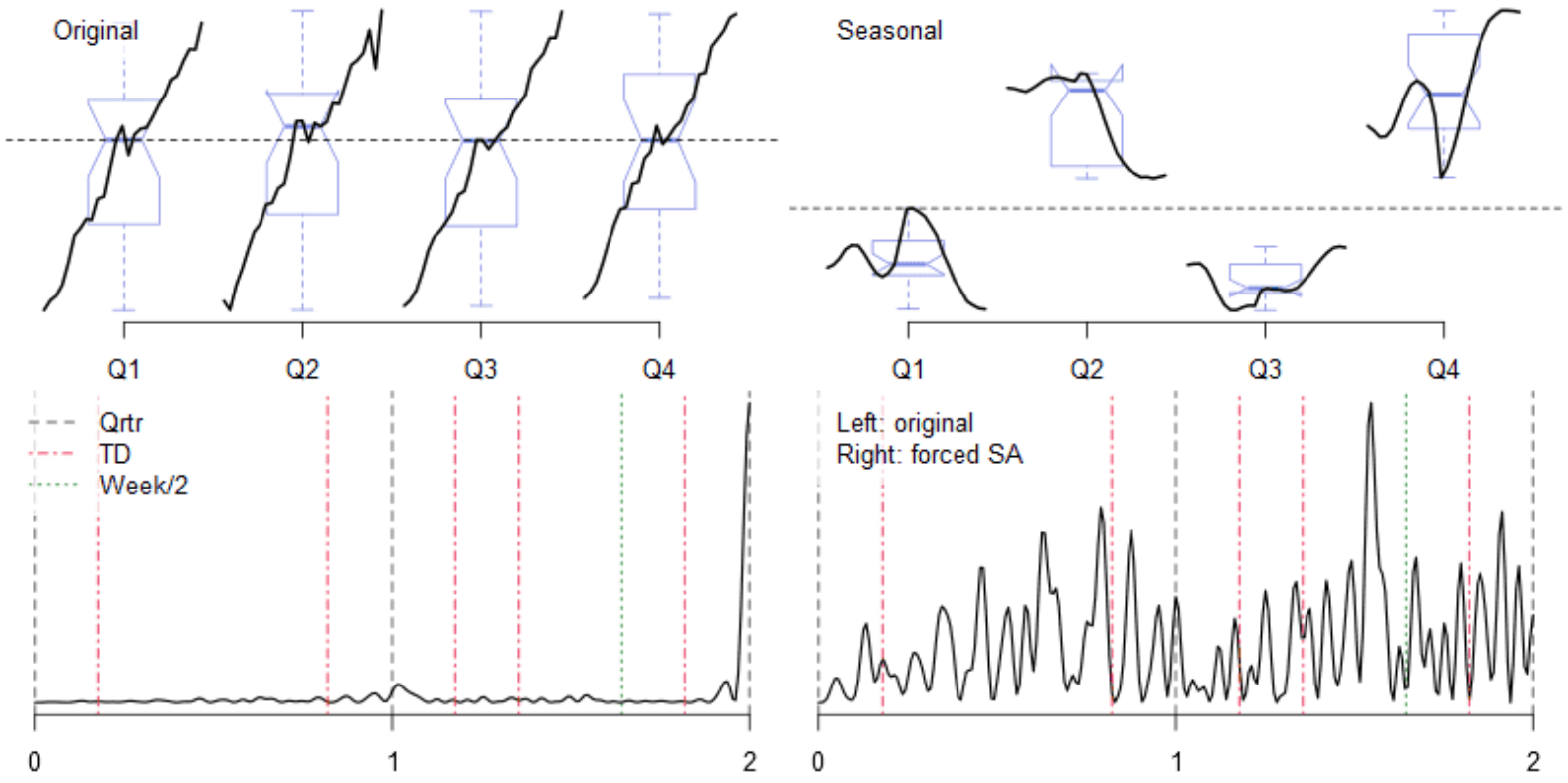
Some presentations are not capable of holding much information without getting cluttered. 3D plots enable comparisons of 3 model aspects.

Personal experience: use in the industry



Statec has of thousands of series with different properties.

Personal experience: use in the industry



Statec needed automating seasonal diagnostics. A 100-line function does the adjustment and returns the visuals to carry out Eurostat-compliant analysis.

Basics of image processing

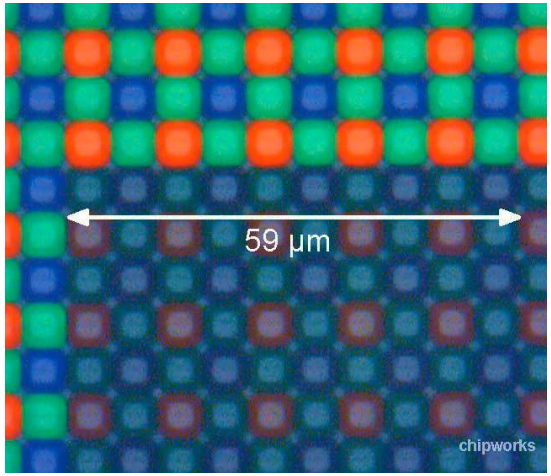
Raster images

Raster digital image is a rectangular grid of uniformly sized pixels taking certain values

- Pixel grid = approximation of a 2D visual phenomenon obtained by sampling different light wavelengths with sensors
  - Capturing sensors are often non-uniform (e. g. RGBG Bayer grid), but output LEDs are often uniform
- Reproduced by shining back R, G, & B
- Measured in **pixels** (usually square) by width and height (not physical units like cm)
- Typical applications (not limited to): photos, images with many colours, images with few sharp elements

Raster pipeline

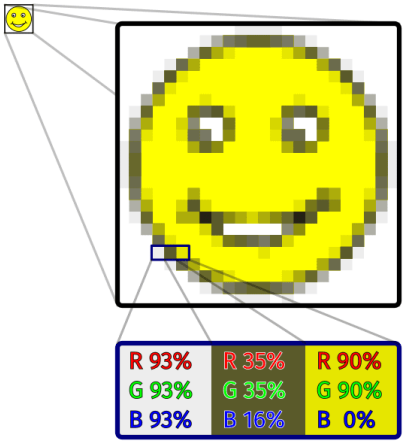
Imperfect input ⇒



Nikon D600 sensor

Credit: iFixit.com.

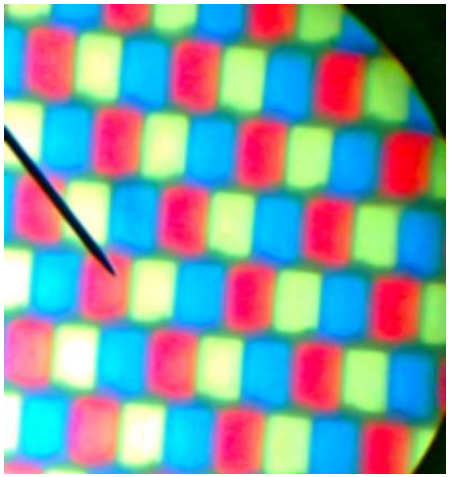
Perfect pixel grid ⇒



Raster image

Credit: Gringer / Wikipedia.

Imperfect output



Casio EX-Z60 screen

Credit: Own work (2007).

Vector graphics

Vector image: graphics defined by analytical descriptions of shapes (points, lines, polygons). Formats: PDF, EPS, SVG...

Raster image

Vector image

Infinitely scalable, suitable for simple shapes and schematic drawings. (We are creating those in R!)

Lossless and lossy compression

Raster RGB graphics with 8 bits / channel (intensity of R, G, & B from 0 to 255) = 3 bytes to encode a pixel.

- 6 MB per 1920×1080 FullHD, 36 MB per photo = too much!

Lossless compression: algorithms to reduce file size by finding redundancies.

- PNG24: run-length encoding (identify long lines or groups of identical adjacent pixels)
- TIFF: ZIP or LZW compression (save repeated sequences in a dictionary and reuse them)

Lossy compression: algorithms allowing certain visual degradation resulting in desirable space savings.

- JPEG: throw out high-frequency info from 8×8 blocks
- GIF/PNG8: use a limited palette (≤ 256 colours)

PDF as a container

R can export vector graphics as PDF, but…

PDF is not a graphics format. PDF is a **container** that can hold raster images, fonts, vector shapes, URLs, sounds, interactive elements etc.

For common use, PDF is the to-go format for vector graphics.

Another one is [E]PS ([encapsulated] Post-Script), which is requested by some academic journals. R can output it, too.

Image format recommendation

- For LaTeX: PDF, PNG, or (for experts) TikZ
- For Word / Writer: PNG
- For picky journals: whatever they request (TIFF, EPS, TikZ)

Don't do: JPG.

- JPG encoder discards all high-frequency information in 8×8 blocks, which adds artefacts
  - Typical R graphics output is replete with thin and crisp lines
- Quality loss even for requested Q=100
- Need smaller PNGs? `pngquant` is your friend!
  - Always crisp, quality does not degrade unless one specifically requests a very small (8 colours) palette

PNG saves space for detail-rich plots

Plotting data sets with > 1000 observations is not uncommon. Scatter plots with many semi-transparent elements give the general idea about a distribution.

PDF saves each shape (every circle, every line), but:

- Storing > 1000 coordinates inflates PDF file size
- Rendering > 10 000 vector shapes on a screen is **slow** (may hang PDF viewers)

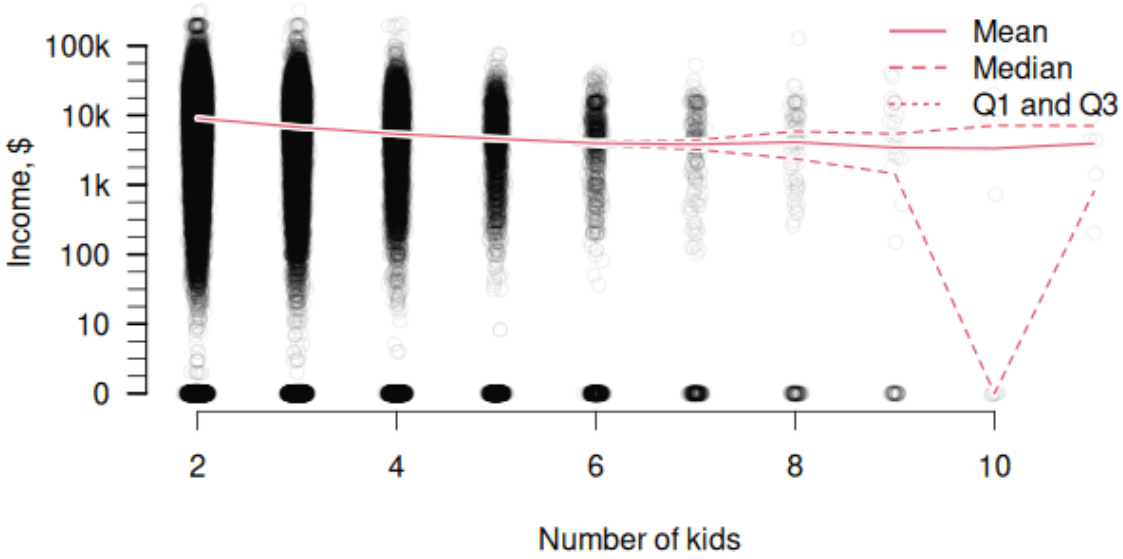
Solution: create a PNG of a fixed size; more shapes → differently coloured pixels, but the file size is well capped.

The same holds for 3D surfaces with many facets.

Real example: Angrist & Evans (1998) data

Sample: n = 380 000 observations of females from 1990 with at least 2 kids.

It takes 20 s to update this preview in RStudio.



PNG24: 31 kB, `pngquant` PNG8: 19 kB. PDF: 21 200 kB (!).

Getting the image tools

imagemagick: the most popular command-line image conversion utility in the world. Automate image format conversion: re-compress TIFF, convert PNG to JPEG and merge to PDF, resize, change DPI, brighten etc.

- Windows: download the binary from imagemagick.org
- Mac: install via Homebrew. To get Homebrew:
```
/bin/bash -c "$(curl -fsSL https://raw.githubuser
↪  content.com/Homebrew/install/HEAD/install.sh)"
brew install imagemagick
```
- Linux: already pre-installed on Ubuntu, otherwise install `imagemagick` with your favourite package manager

imagemagick is quick and intuitive

- Convert all TIFF files to JPG with quality 60%:
  ```
  mogrify -format jpg -quality 60 *.tiff
  ```
- Merge all JPGs into a single PDF (losslessly):
  ```
  convert *.jpg out.pdf
  ```
- Resize an image by 50%, convert to greyscale, increase contrast with levels to get rid of paper noise:
  ```
  convert in.jpg -resize 50% -colorspace Gray
    -level 10%,90% out.jpg
  ```

No online converters or clicking in GUIs; automate locally, avoid menial work! Write once, run always.

Getting the video tools

ffmpeg: the most popular command-line video transcoder.

- Windows: download from ffmpeg.org
  - The binaries are at gyan.dev
- Mac: `brew install ffmpeg`
- Linux: install `ffmpeg` with your favourite package manager
  - Compiling from source adds support for HEVC, FDK AAC, AV1, and other highly efficient modern formats

Very intuitive, too. No need to use Adobe After Effects / Sony Vegas / shady online services that add watermarks to merge, resize, and compress MP4 files.

ffmpeg is quick and intuitive

- Convert a sequence of PNG images into a 24-FPS video:

```
ffmpeg -framerate 24 -pattern_type glob -i '*.png'
  -pix_fmt yuv420p -c:v libx264 -crf 25 out.mp4
```

- Split without re-encoding (how I upload to Moodle):

```
ffmpeg -i session04.mp4 -to 01:18:06 -c:a copy
  -c:v copy session04-p1.mp4
ffmpeg -i session04.mp4 -ss 01:28:26 -c:a copy
  -c:v copy session04-p2.mp4
```

  Blazingly fast: takes 1 s for a 270 000-frame file.

- Shrink a video to 640 px by width, re-compress with the highly efficient codec:

```
ffmpeg -i in.mp4 -filter:v "scale=640:-1"
  -c:v libx265 -crf 32 -c:a copy out.mp4
```

Lossless PNG optimisation

optipng: command-line utility to losslessly re-compress PNG (searching for more optimal encoding schemes when searching for identical neighbours).

Download page at SourceForge.

Optimise PNGs preserving file attributes (modified etc.):

```
optipng -preserve *.png
```

Exhaustively search for the best compression scheme (much slower):

```
optipng -o7 -preserve *.png
```

Lossy PNG optimisation

pngquant: command-line utility to reduce the number of colours in a PNG file from 16 mn to 2–256 with as little visual degradation as possible.

Project page and download.

```
pngquant --quality 50-60 --verbose --speed 1
  --force --ext -opt.png *.png
```

These PDF slides are so lightweight even with this many plots because every PNG is pngquant'ed.

Declare a bash function withh all the parameters:

```
pq () { pngquant --speed 1 --quality 50-60
  --verbose --ext -opt.png --force $1; }
```

System calls

If a utility can be called from the command line on a computer, it can be called from within R – a **system call** is an invocation of any command that can be executed on an OS.

Type this in the terminal to apply `pngquant`:

```
pngquant --quality 60-70 --verbose --speed 1
  --force --ext -opt.png s05-bidgata.png
# Creates s05-bigdata-opt.png
```

Do the same from within R:

```
pngs <- list.files(pattern = "\\.png$")
lapply(pngs, \(v) {
  e <- paste0("pngquant --quality 60-70 --verbose
  --speed 1 --force --ext -opt.png ", v)
  print(e); system(e)})
```

System calls to imagemagick

Compress uncompressed TIFFs losslessly:

```
convert in.tif -density 300 -compress LZW out.tif
```

Use `system()` to do the same from R.

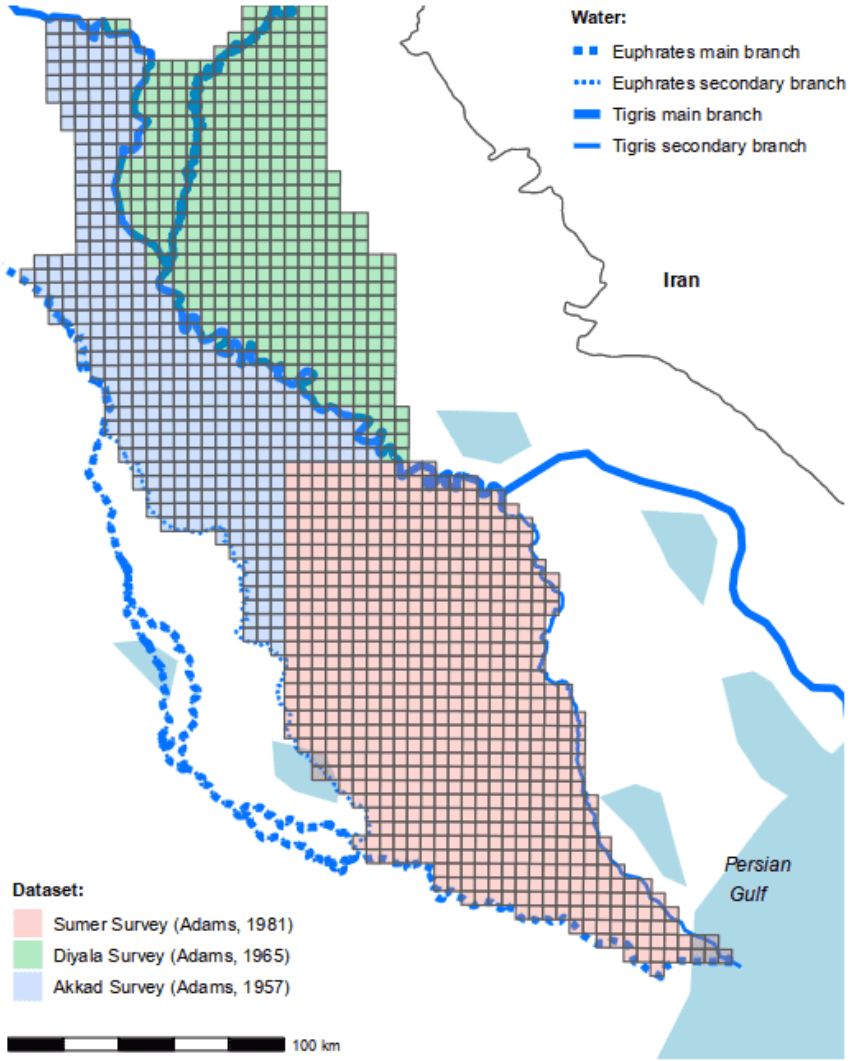
```
tifs <- list.files(pattern = "\\.tiff$")
lapply(tifs, \(v) {
  e <- paste0("convert ", v, " -density 300
    -compress LZW -verbose ",
    gsub("\\.tiff$", "-opt.tiff", v))
  print(e); system(e)})
```

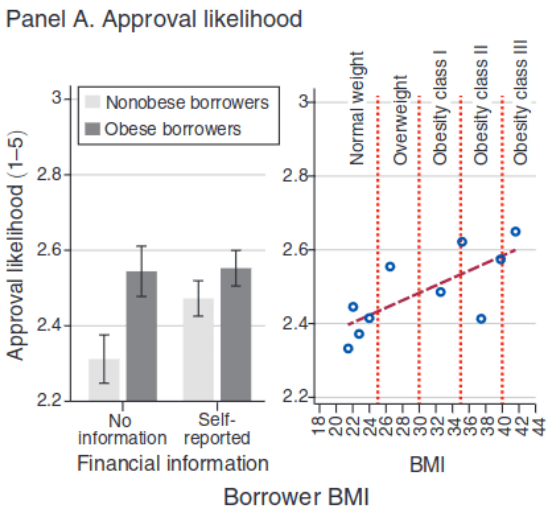
We shall call `ffmpeg` to create MP4 animations, too.

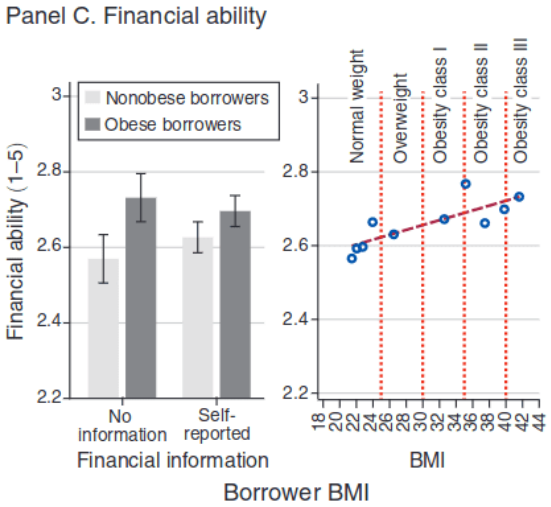
Graphics for journal articles

- The most informative article graphics are non-standard and depend on the research subject
- Space is expensive; each graph should highlight one key point of your research
- For internal use / discussions with your collaborators, plots can be information-dense, technical, highly diagnostic – but be prepared to strip most of it for the publication

American Economic Review (2023) plots

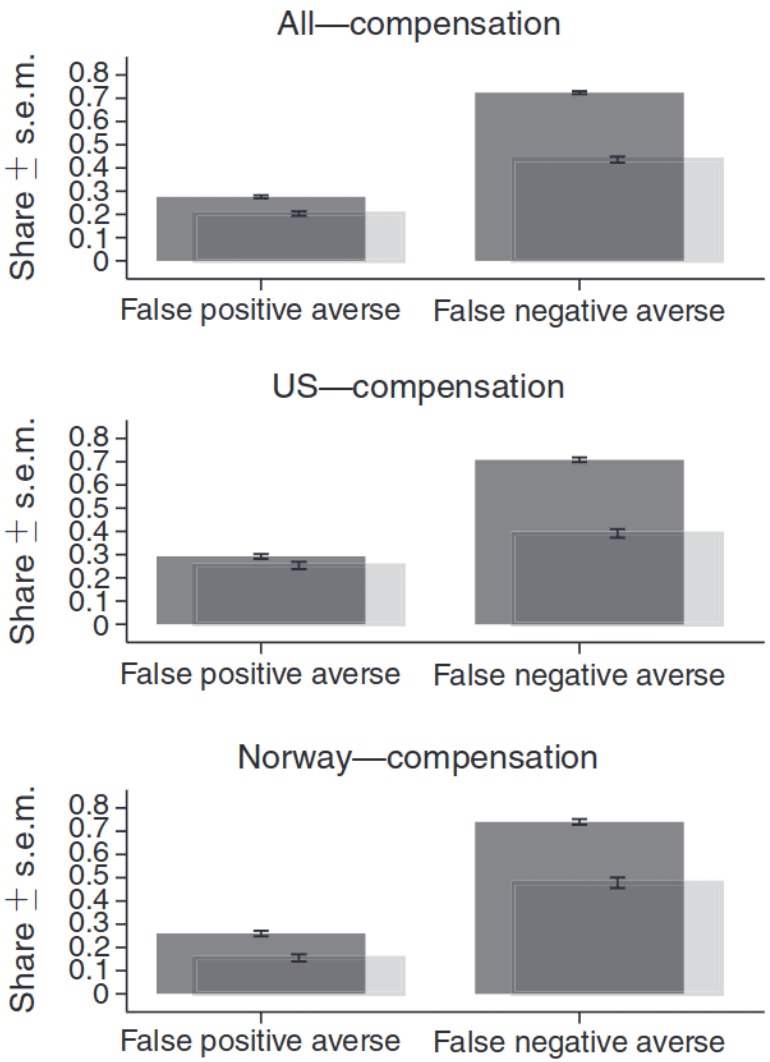


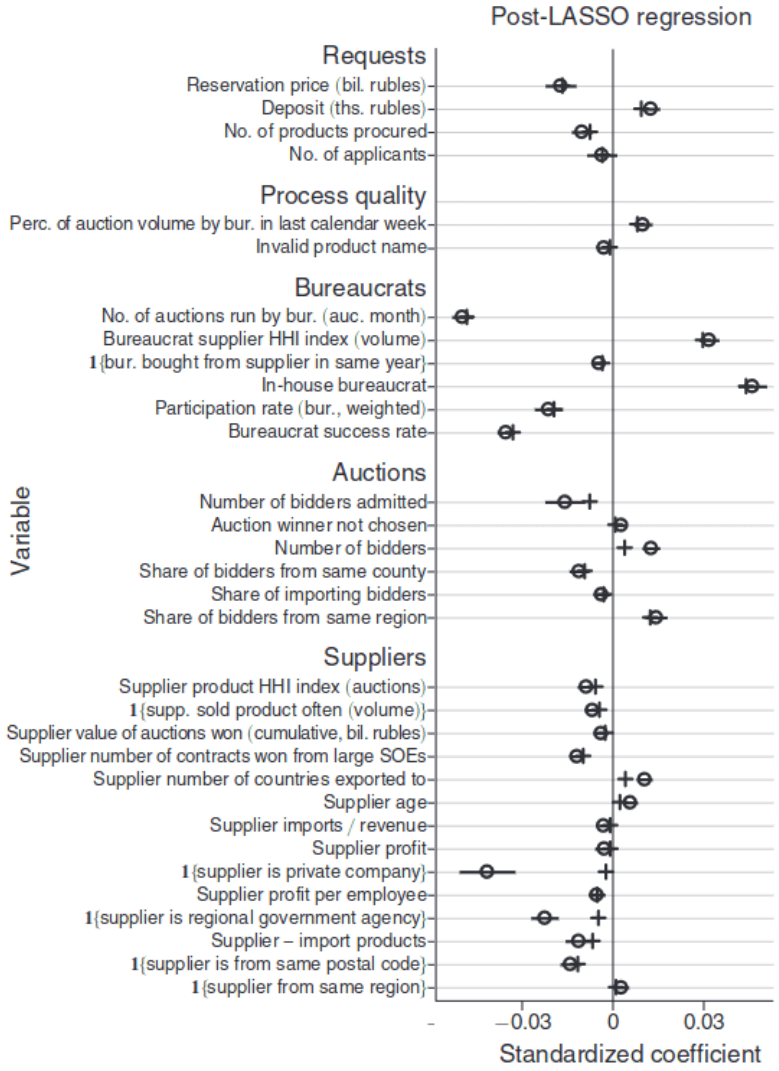




Credit: DOI: 10.1257/aer.20201919, DOI: 10.1257/aer.20211879.

American Economic Review (2023) plots





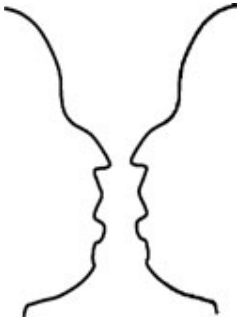
Credit: DOI: 10.1257/aer.20211015, 10.1257/aer.20191598.

Assume readers' minimal visual skills

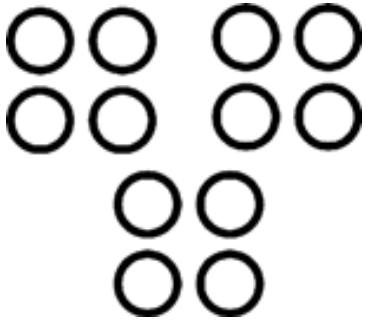
- Deręgowski (1976) on Malawi people: 'Take a picture in black and white and the natives cannot see it'
  - Requires 'This is really a picture of an ox and a dog. Look at the horn of the ox, and there is his tail!'
- People who are not exposed to 3D pictures early on struggle with depth perception (Hudson 1960)

Patterns in perception (1/3)

Foreground / background: Attention is focused on one part of the image, usually smaller or more prominent.

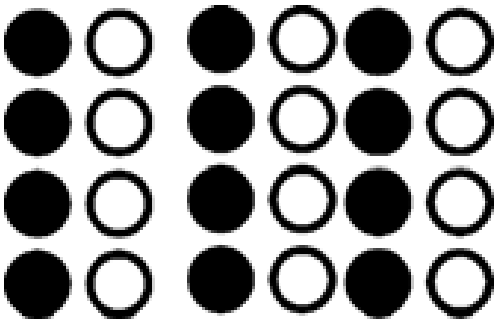


Proximity: Elements that are close together are perceived as a group.

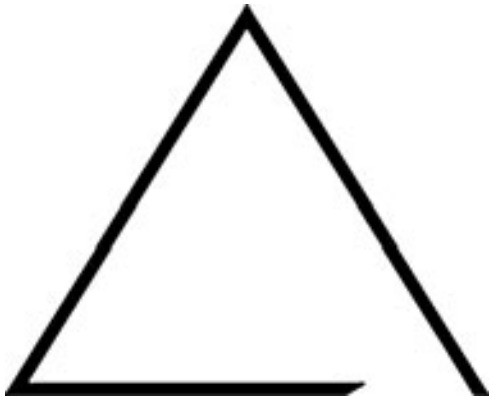


Patterns in perception (2/3)

Similarity: Objects that are alike in form, colour, brightness, or size tend to be grouped in the mind.

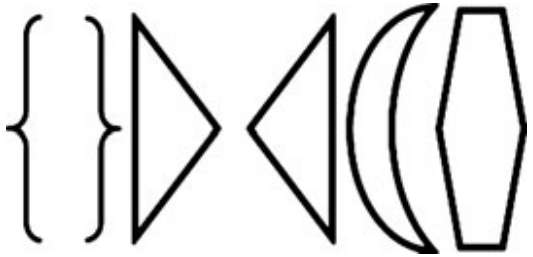


Closure: Tendency to fill in spaces or connect the dots.

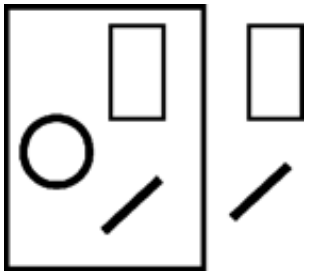


Patterns in perception (3/3)

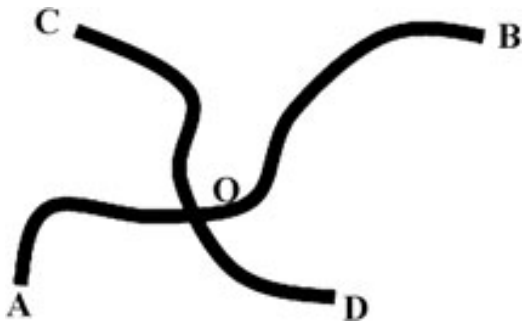
Symmetry: Similar shapes are grouped together, regardless of their proximity.



Common fate: Elements in a common region are perceived as a group, regardless of similarity or proximity.

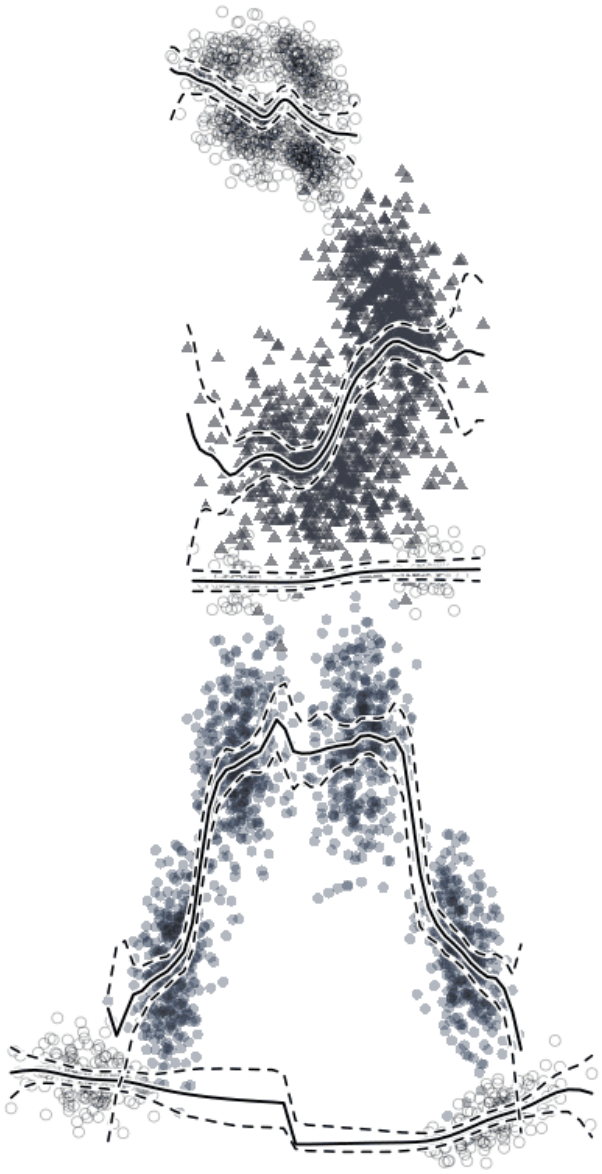


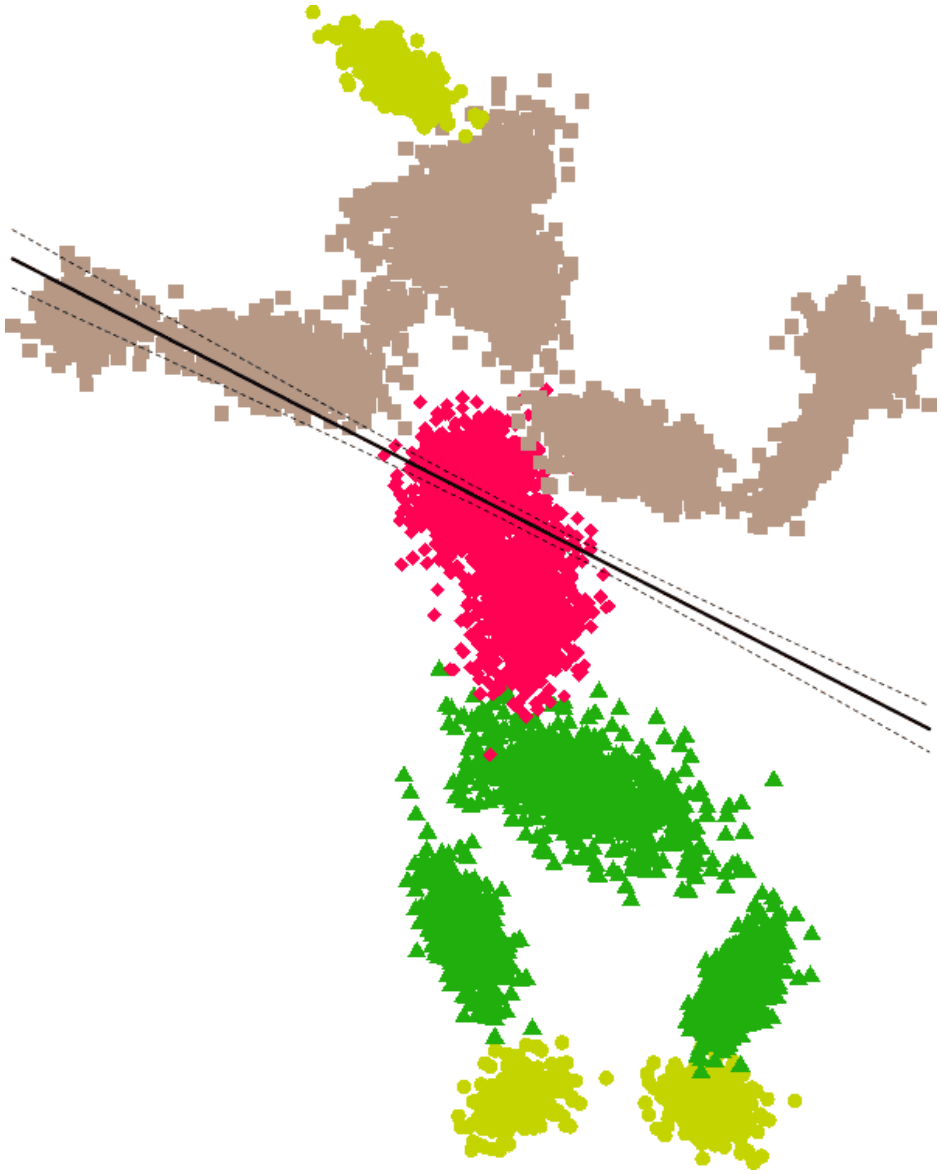
Continuity / smoothness: The mind prefers to perceive continuous and smooth ines (AOB, COD) and avoid cusps (COB).



Credit: Sabih et al. (2011). Image perception and interpretation of abnormalities. *Insights into imaging*, 2(1), 47–55..

Plots: complexity ≠ beauty

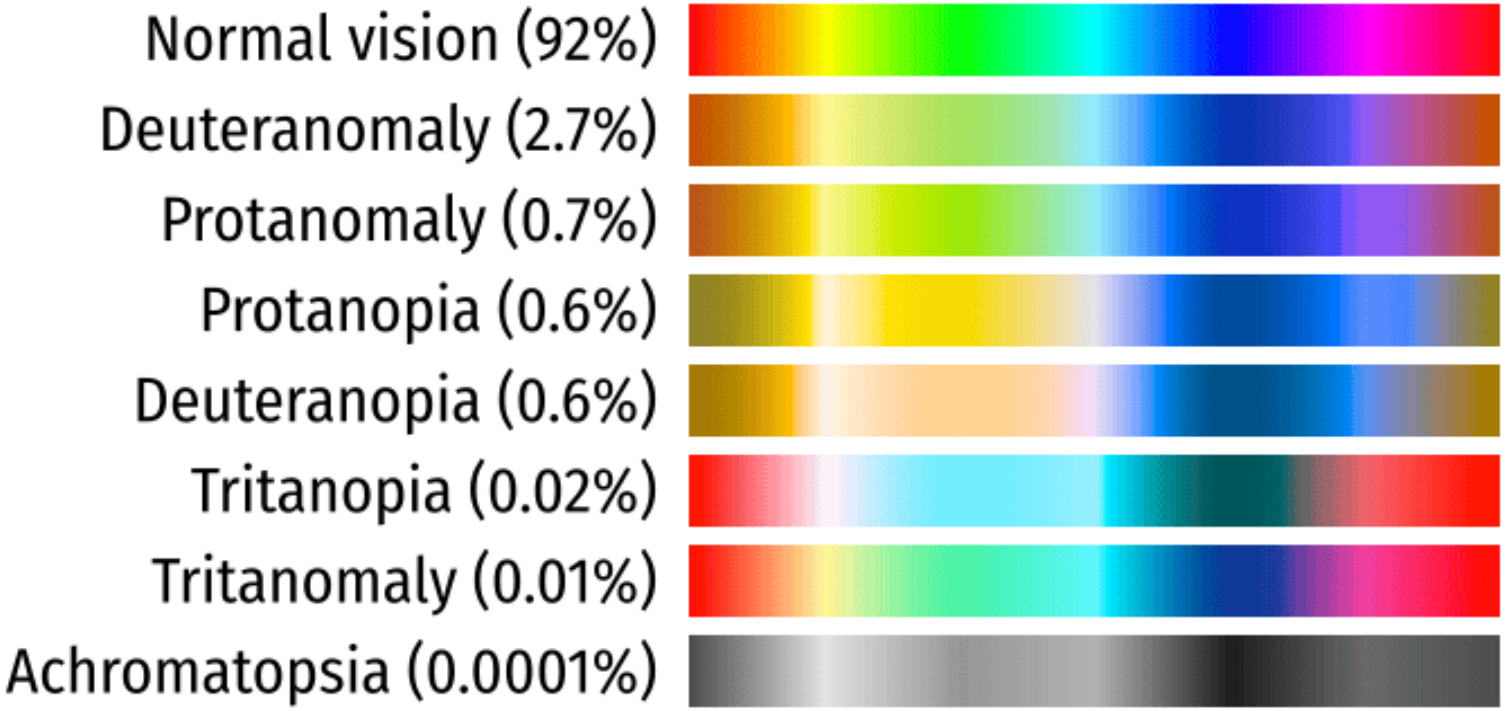




Things to avoid

- Hatching
  - Can produce optical illusions
  - Vestige of the days when mechanical pen plotters could not produce solid fills because they were tearing the paper
  - If you cannot distinguish between the shades of grey in the plot, you are using too many shades of grey
- Pie charts
  - Hard to read, hard to make mental comparisons
  - Use bar charts instead

Colour-vision deficiency



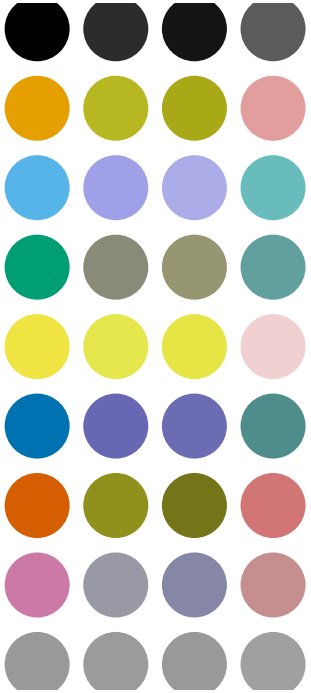
Colour blindness: does the top spectrum look *exactly* like one of the bottom ones?

Colour-blind-friendly colours

- The built-in Okabe–Ito palette is a good start:

```
plot(1:9, 1:9, pch = 16, cex = 5,
  col = palette.colors(9,
    palette = "Okabe-Ito"))
```

- `dichromat` package to convert palettes to colour-deficient ones
- Simulate colour-blind versions of plots at hclwizard.org



Use redundancies

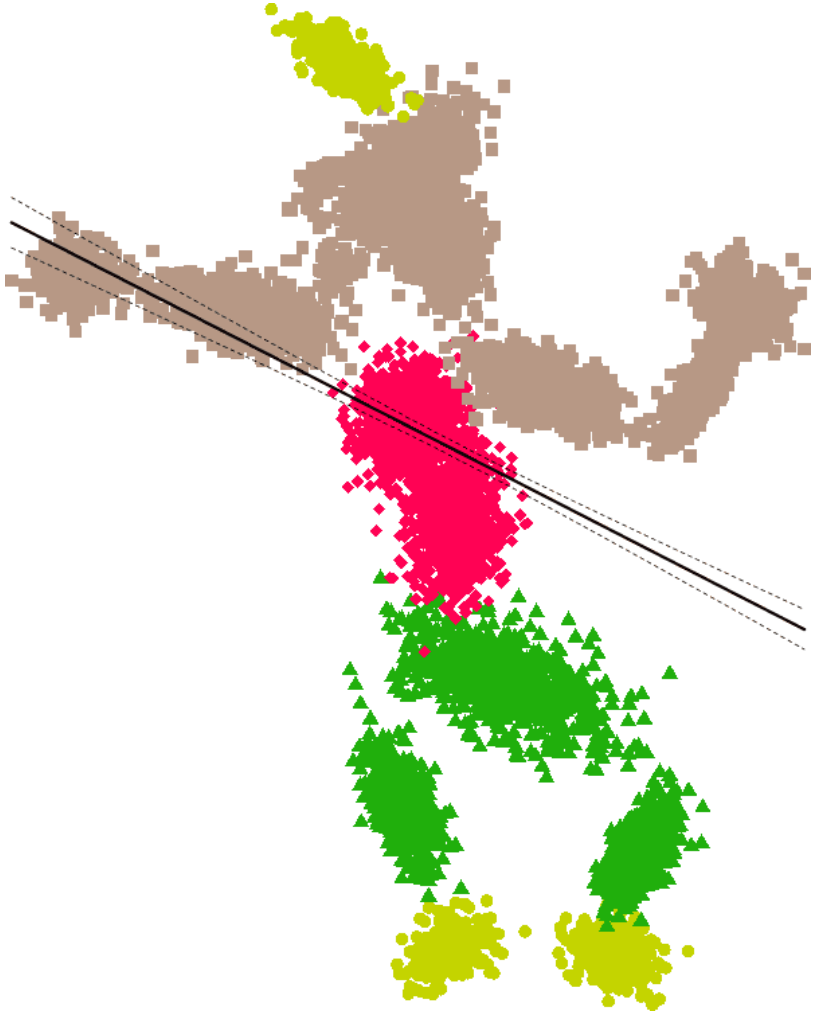
Journal editors are stricter than accessibility guidelines: they often require only B/W graphics.

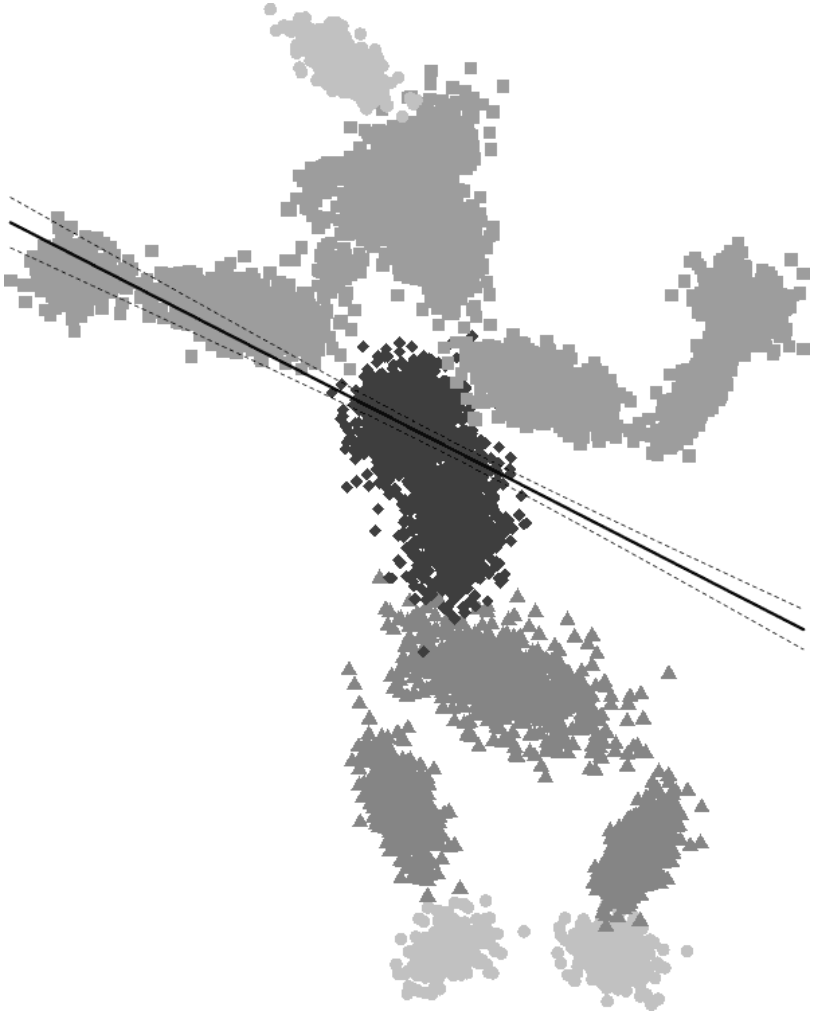
Encode information both in shape and in colour.

- Lines: vary stroke type, thickness, and colour
- Points: vary point character, size, and colour

Double redundancy in action

Distinct: hue + brightness + shape (BW: brightness + shape).





base and ggplot2

2 main plotting syntaxes in R: base graphics and ggplot2.

- **base** graphics are imperative: give instructions to place elements, and R will follow exactly
  - Like the C programming language
- **ggplot2** graphics are declarative: give descriptions of the result that you want to produce
  - Like the Haskell programming language

Visualisations in base R

Base R graphics are extremely powerful and flexible: they allow the user to produce absolutely any kind of plot.

- `plot()` initialises a drawing area on a new device
  - Generic method: some object call class-specific routines
- 95% of everything else can be achieved with 5 commands: `points()`, `lines()`, `arrows()`, `polygon()`, `abline()`
  - Remaining 5%: `contourLines()`, `persp()` + `trans3d()`, `rug()`, `barplot()`, `boxplot()`
- Modify graphic parameters or plotting device settings for fine tuning

Advantages of base over ggplot2

- Easier for newcomers
  - Fewer functions to memorise
  - Break down problems, build visuals up in simple steps
  - Straightforward debugging, no call-stack rabbit holes
- Full control over the plot
  - The canvas is yours, imagination is the limit, no restrictions
  - The next step does not break what has been plotted
  - Changes are instantly visible after a function call
- Universally applicable
  - No need for extra transformations just for plotting, no dependencies on other packages, all classes welcome
  - Works with anything interpretable as a coordinate
- Much faster, less memory-hungry, esp. with big data
- Is extended by the great `lattice` package

Advantages of ggplot2 over base

- Comes with hundreds of pre-defined templates for a wide family of popular plot types
  - For data with regular structure, invocation is often shorten than in base R
- Many easy-to-understand books and learning resources targeted at all audiences
  - Abundant solutions already written for popular applications
  - Base R graphics documentation does not have many good examples
- Change the theme of all elements quickly
- More streamlined margin handling, fewer device errors

Where both ggplot2 and base struggle

- First steps can be tricky
- Default settings can be ugly
  - Always takes many tweaks before the plot looks decent
- Only code and formal declaration of plots, no mouse or free-hand adjustments in a GUI tool

Highest priorities in plotting

People who will tell you to change the plot:

1. Your advisor / supervisor
2. Your journal editor

Pareto 80/20 principle (from experience): making the `ggplot2` output match their suggestions often takes hours of fiddling with deep settings and ugly work-arounds.

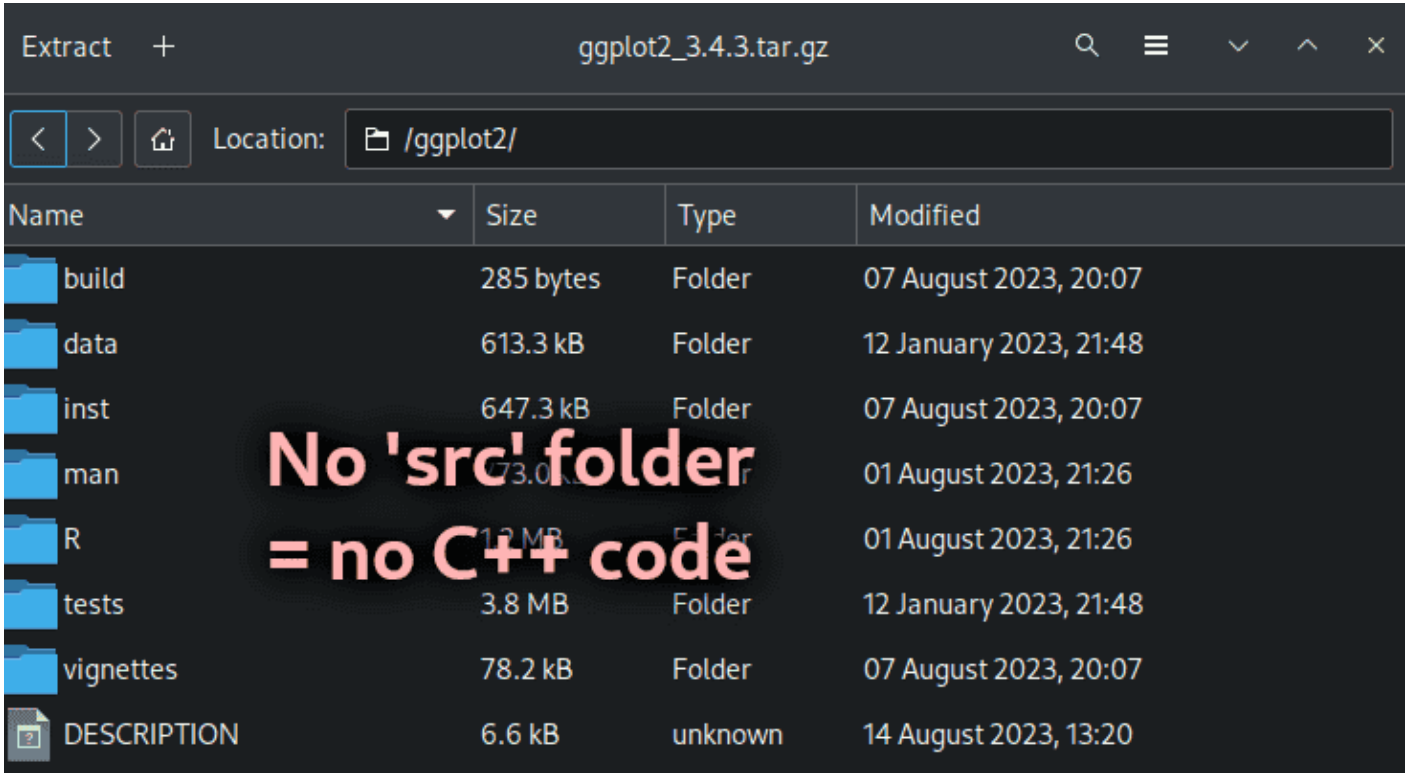
Your Ph.D. time is limited – do not waste it on searching for pre-cooked solutions. *(I have seen people editing PNGs exported from R in MS Paint... over and over.)*

Core principle: if you can compute it, you should be able to add it directly to the plot without re-structuring plot calls.

Recommendation: base first, ggplot2 later

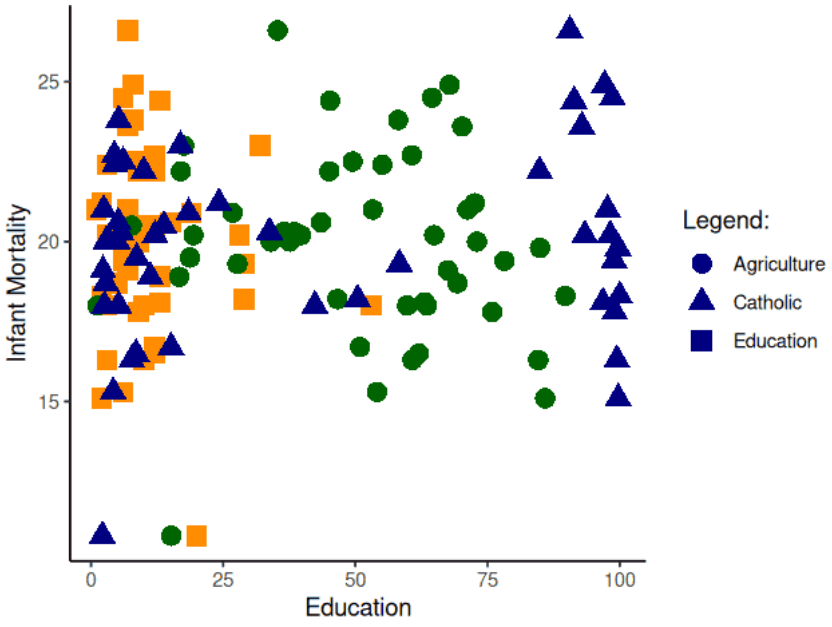
- Anything one can do in `ggplot2`, one can do in base R
  - 5 smaller steps = better than one complex large one
- `ggplot2` imposes too rigid a structure on the inputs
  - Your model may be non-standard and not recognised by `gg*` functions
  - The requested fine tweaks may require looking into `gg*` source code with nested functions calling one another
  - C'mon, I just want to add this small extra notch here... It has to be simple... what, another reshaping?
  - '`ggplot2` produces the legend based on what you used' ⇒ making a custom legend (e. g. with elements different from used in the plot) is *hard* for beginners
- So many packages and custom classes, the logical first step is extracting and reconciling the components

ggplot2 itself is written in base R

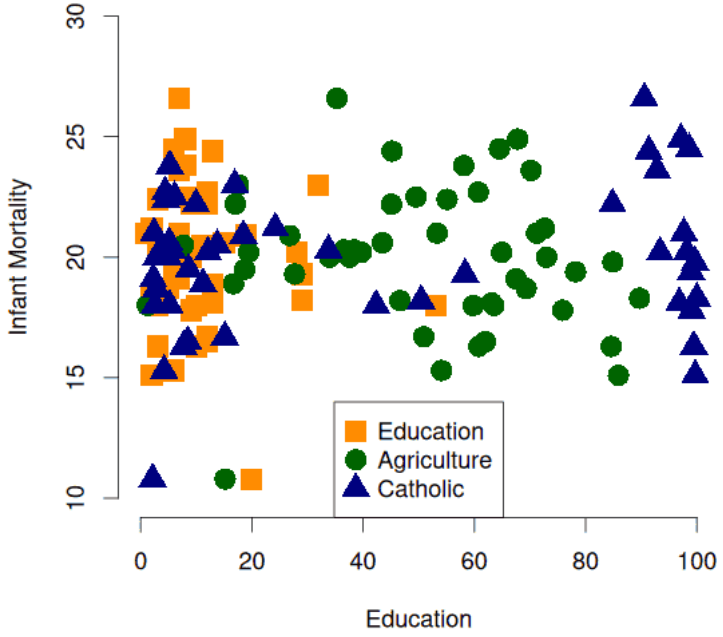


Base R is so powerful, one can write the `ggplot2` package using nothing but base R + core packages.

Recall Session 1: two approaches



ggplot2, 612 bytes
broken legend



base R, 421 bytes
correct legend

Early steps are always hard

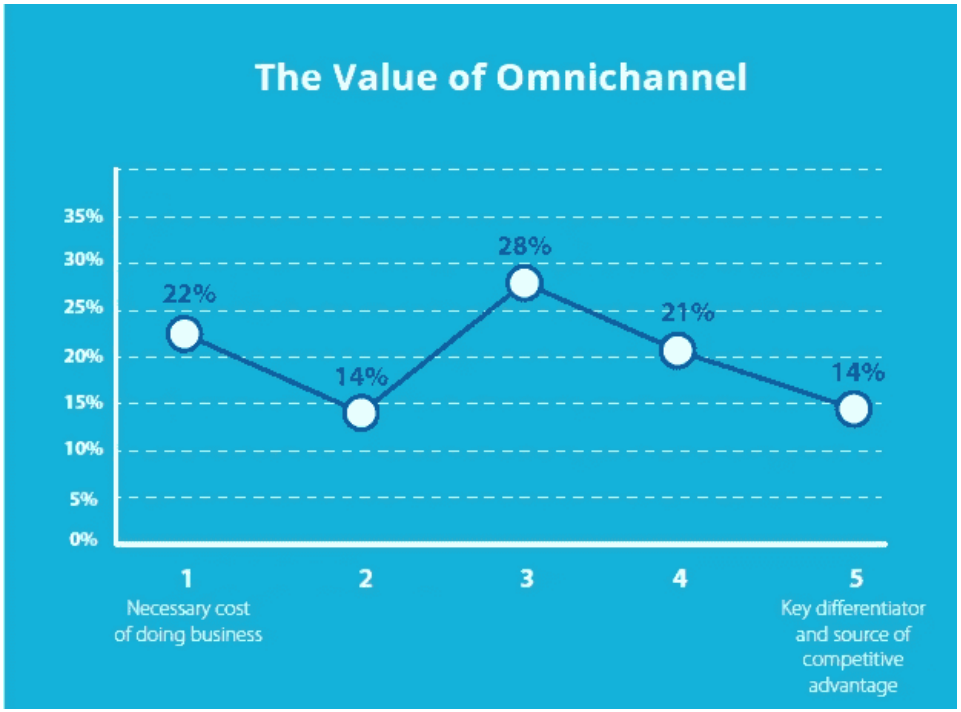
Beginners tend to struggle with `ggplot2`:

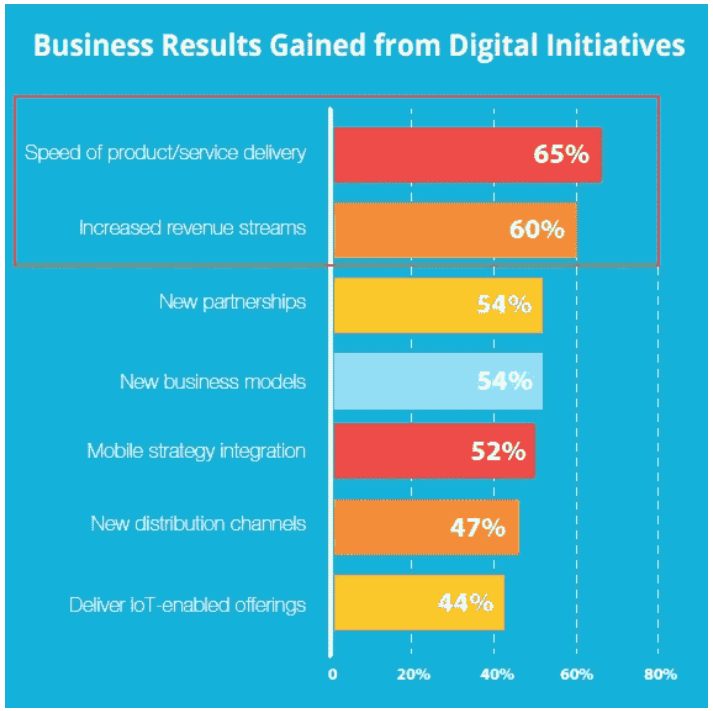
```
ggplot(swiss, aes(x = Education, y = Infant.Mortality)) +
  geom_point(aes(shape = "Education", fill = "Education"),
    size = 4, color = "darkorange") +
  geom_point(aes(x = Agriculture, shape = "Agriculture",
    fill = "Agriculture"), size = 4, color = "darkgreen") +
  geom_point(aes(x = Catholic, shape = "Catholic",
    fill = "Catholic"), size = 4, color = "navy") +
  scale_color_manual(values = c("navy", "darkorange", "darkgreen")) +
  scale_fill_manual(values = c("darkorange", "darkgreen", "navy")) +
  labs(x = "Education", y = "Infant Mortality", shape = "Legend:",
    color = "Legend:", fill = "Legend:") +
  theme_classic()
```

Today, we learn being efficient in base R:

```
y   <- swiss$Infant.Mortality
xs  <- c("Education", "Agriculture", "Catholic")
cls <- c("darkorange", "darkgreen", "navy")
plot(swiss[, xs[1]], y,
     pch = 15, col = cls[1], cex = 2, bty = "n",
     xlab = xs[1], ylab = "Infant Mortality",
     xlim = c(0, 100), ylim = c(10, 30))
points(swiss[, xs[2]], y, pch=16, col=cls[2], cex=2)
points(swiss[, xs[3]], y, pch=17, col=cls[3], cex=2)
legend("bottom", xs, col=cls, pch=15:17, pt.cex=2)
```

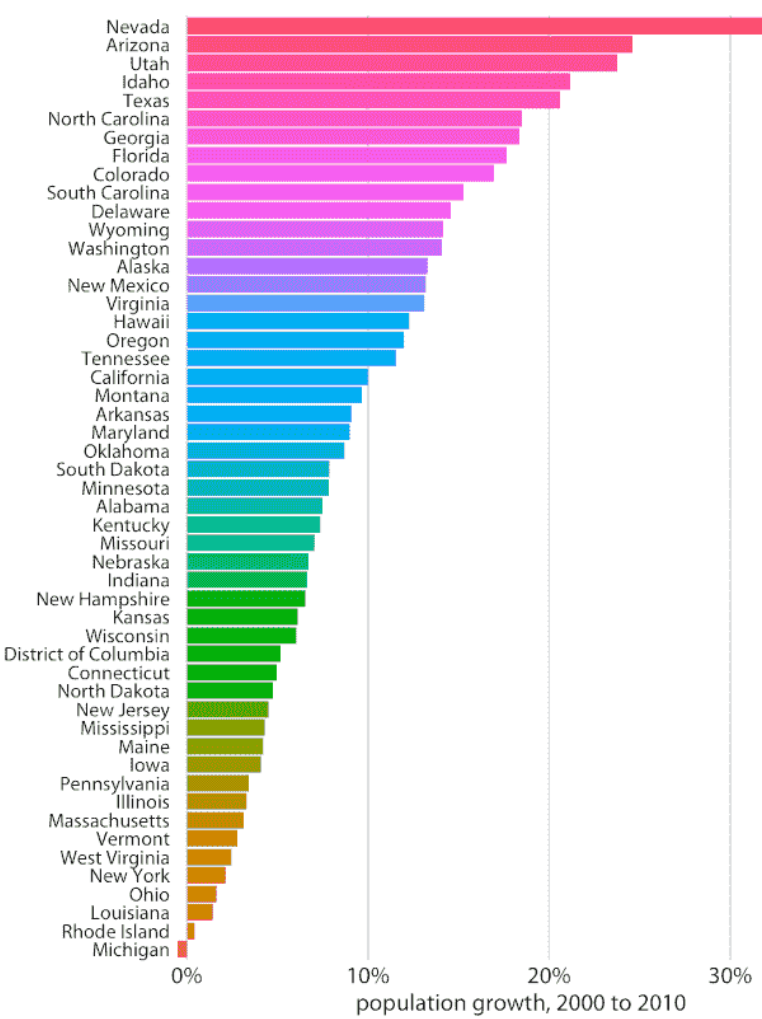
Ugly plots (1/5) – colour abuse

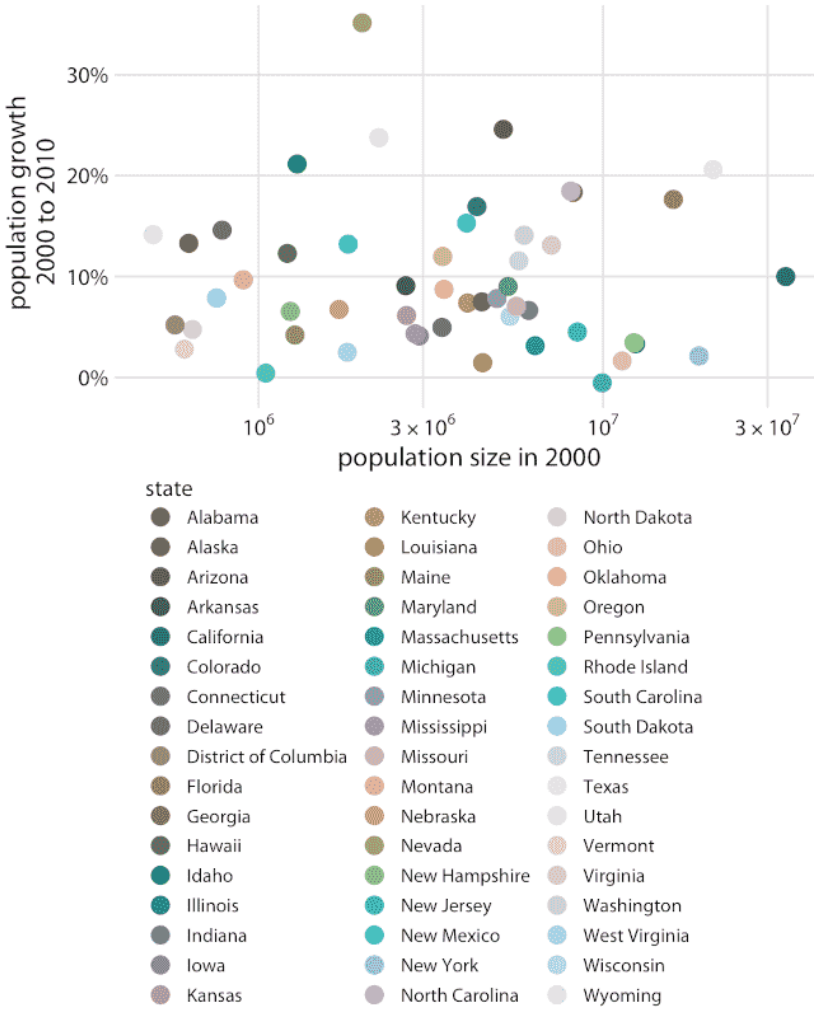




Credit: Towards Data Science.

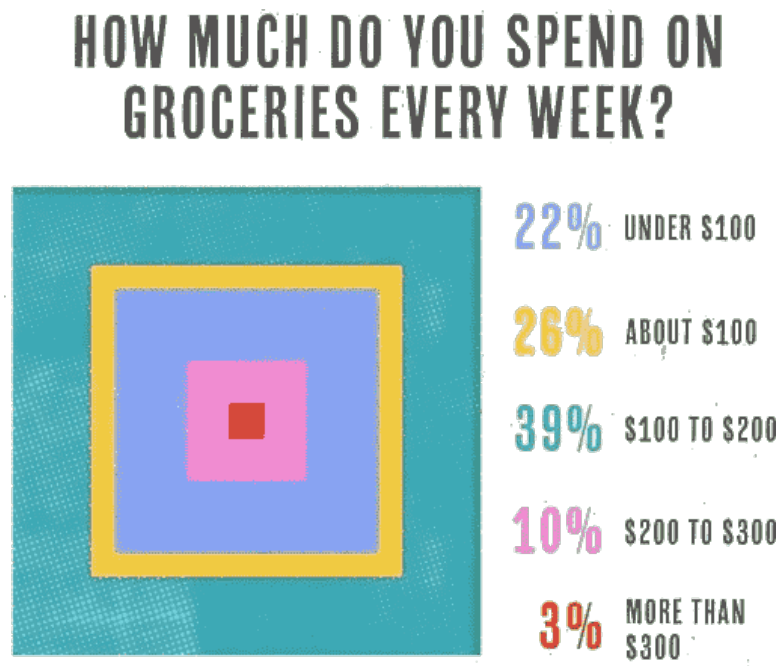
Ugly plots (1/5) – colour abuse



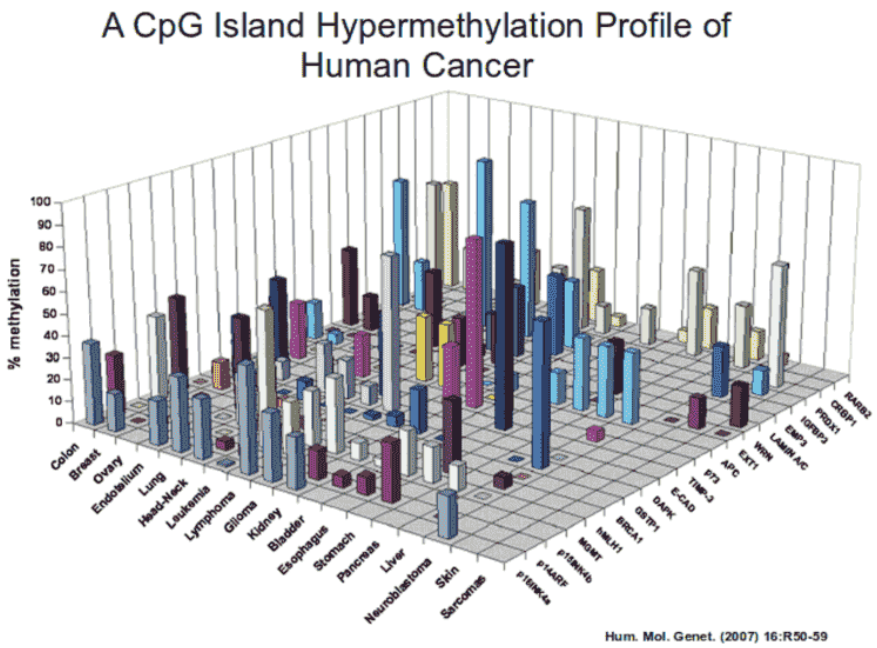


Credit: Claus Wilke.

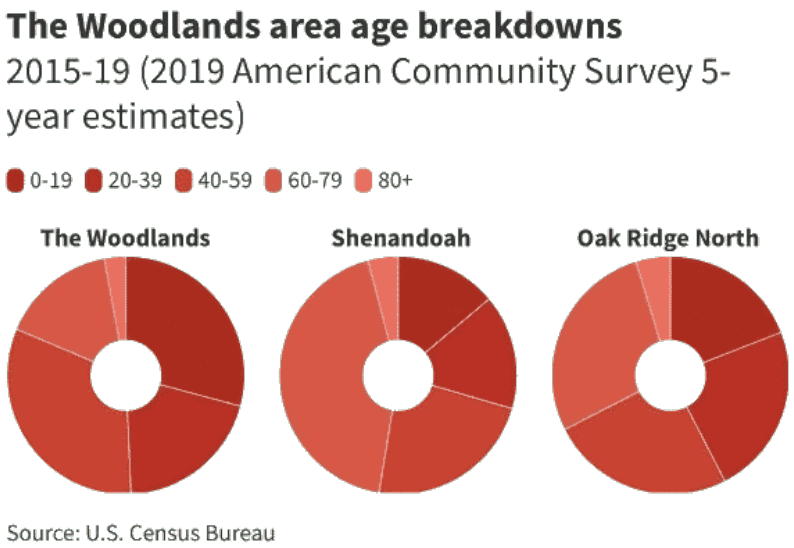
Ugly plots (2/5) – illegible representation

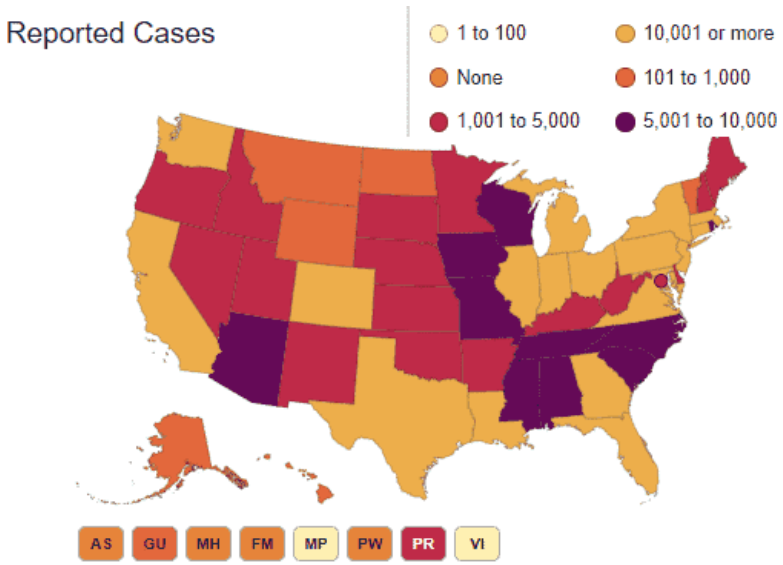


Credit: Old Streets Solutions.



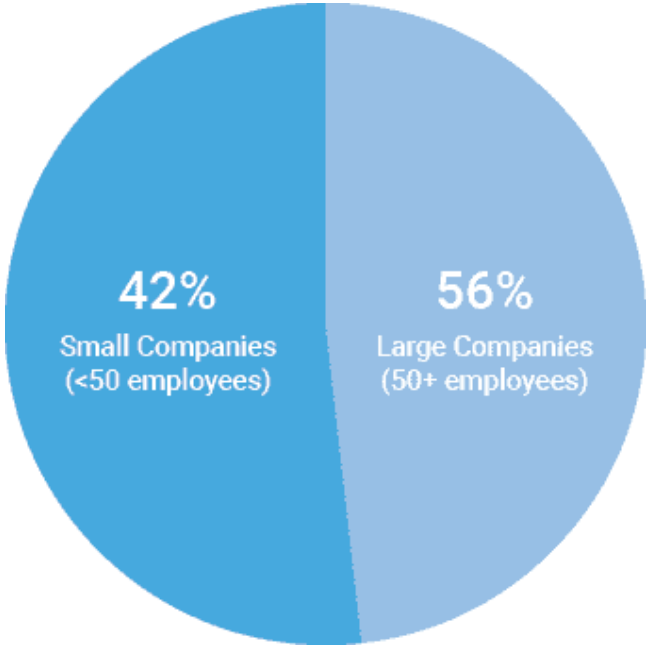
Ugly plots (3/5) – colour misuse





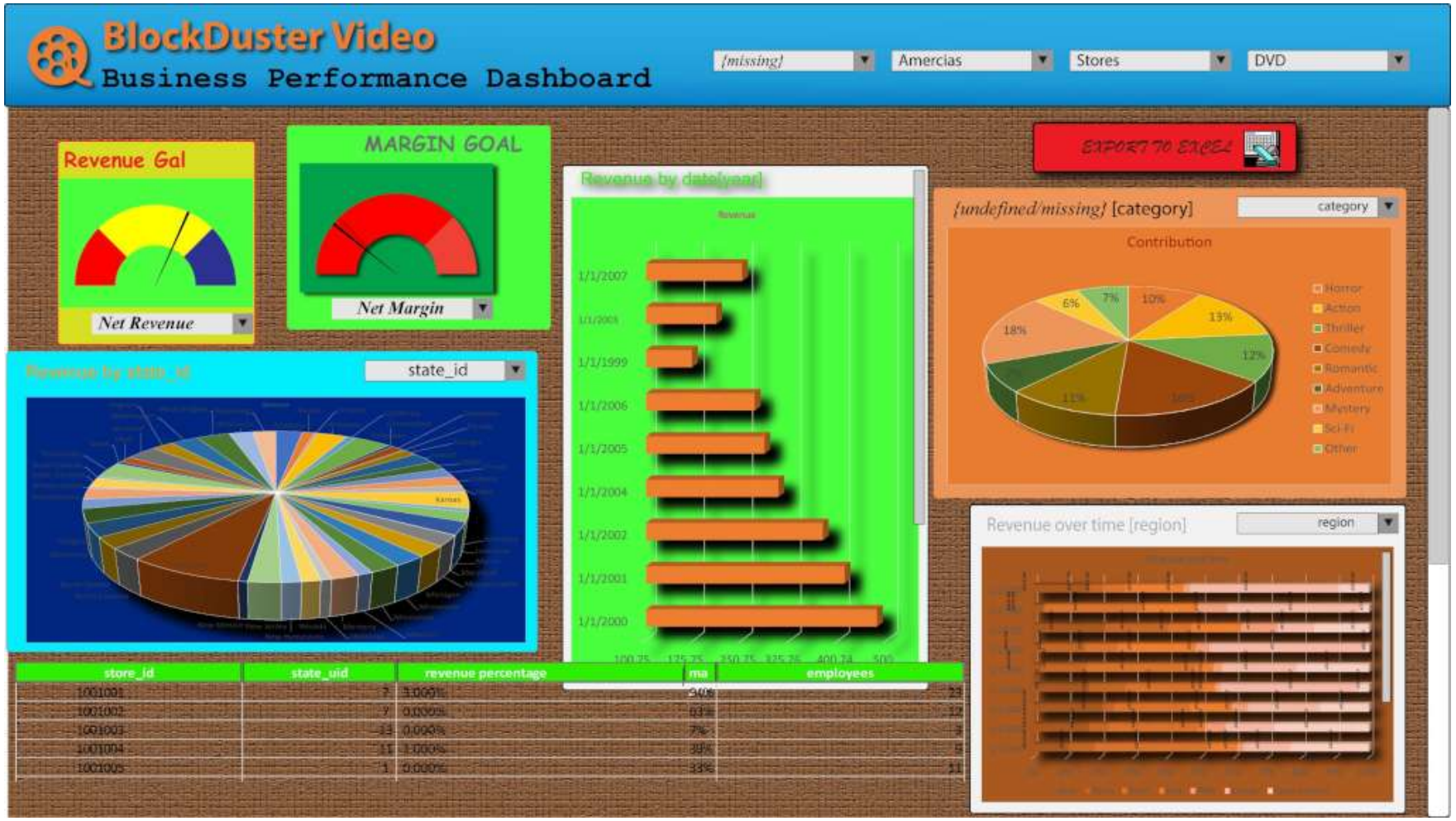
Credit: Old Streets Solutions.

Ugly plots (4/5) – wrong inputs / labels



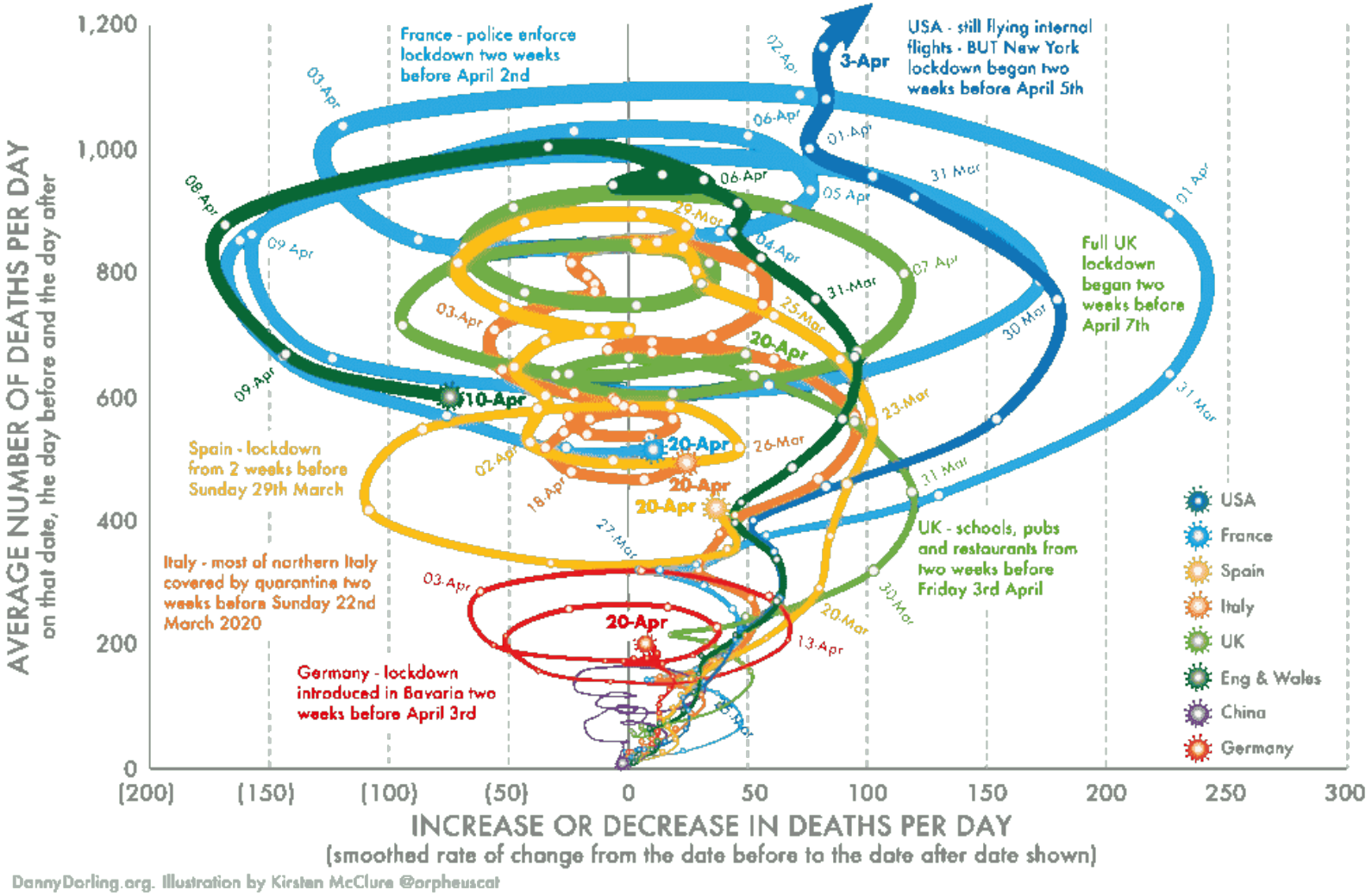
Credit: Old Streets Solutions.

Ugly plots (5/5) – WAT



Credit: Delivering Data Analytics.

Ugly plots (5/5) – WAT



Credit: Old Streets Solutions.

Any questions on raster and digital images?

Plots, parameters, and devices

Five pillars of base graphics

1. Points
2. Lines
3. Polygons
4. Text
5. Layout parameters

Anything can be drawn with these 5 elements.

Caveat: sacrifices for comprehensibility

In these plots, the effects of changing specific parameters will be shown.

However, certain extra parameters *not shown in the code* are used to change the layout to make these plots look better in the slides.

- The code blocks showcase the effect of changing a certain parameter on the final look
- To reproduce the plots *exactly* (with margins, prettifications, omissions etc.), see `session05.R`

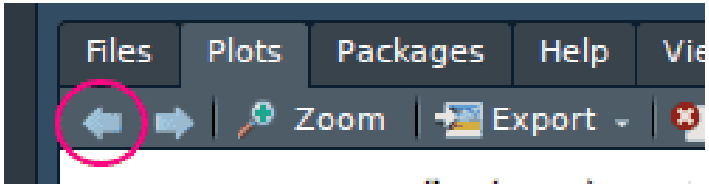
Transform data and plot

A family of functions automatically pre-process data to create something plottable:

- `boxplot()` – box-and-whisker plot, `barplot()` – barplot
  - Compute and return points used to produce the bars
- `density()` – density, `hist()` – histogram
- `acf()` – auto-correlation, `hclust()` – dendrograms, `ecdf()` – empirical cumulative distribution, `prcomp()` – principal-component decimposition, `stl()` – season-trend LOESS decomposition

Workhorse: plot()

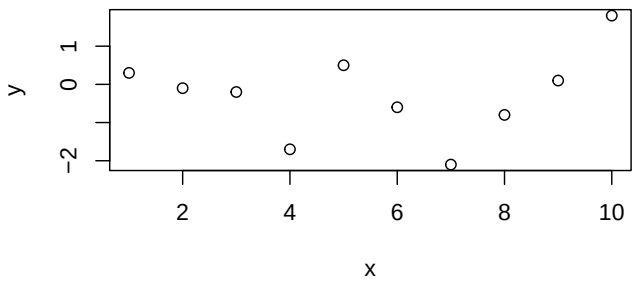
- Almost every plot starts with calling `plot(...)`
- For numeric vectors x and y, `plot(x, y)` produces a scatter plot (with points) by default (no recycling)
  - Optional arguments change the style of `plot()`
- If `class(a) == "XYZ"` and the method `plot.XYZ()` is defined, `plot(a)` will produce a special kind of plot pre-defined by the developer
  - To see how it is achieved, run `pkgname:::plot.XYZ`
  - Copy and change the definition for full customisability
- In an interactive environment (e. g. RStudio), `plot()` starts a new plot and overwrites the old one
  - This button goes to the previous interactive plot:



Basic scatter plot

Given two vectors of the same length, produce a simple scatter plot:

```
x <- 1:10
y <- c(0.3, -0.1, -0.2,
       -1.7,  0.5, -0.6,
 -2.1, -0.8,  0.1, 1.8)
plot(x, y)
```

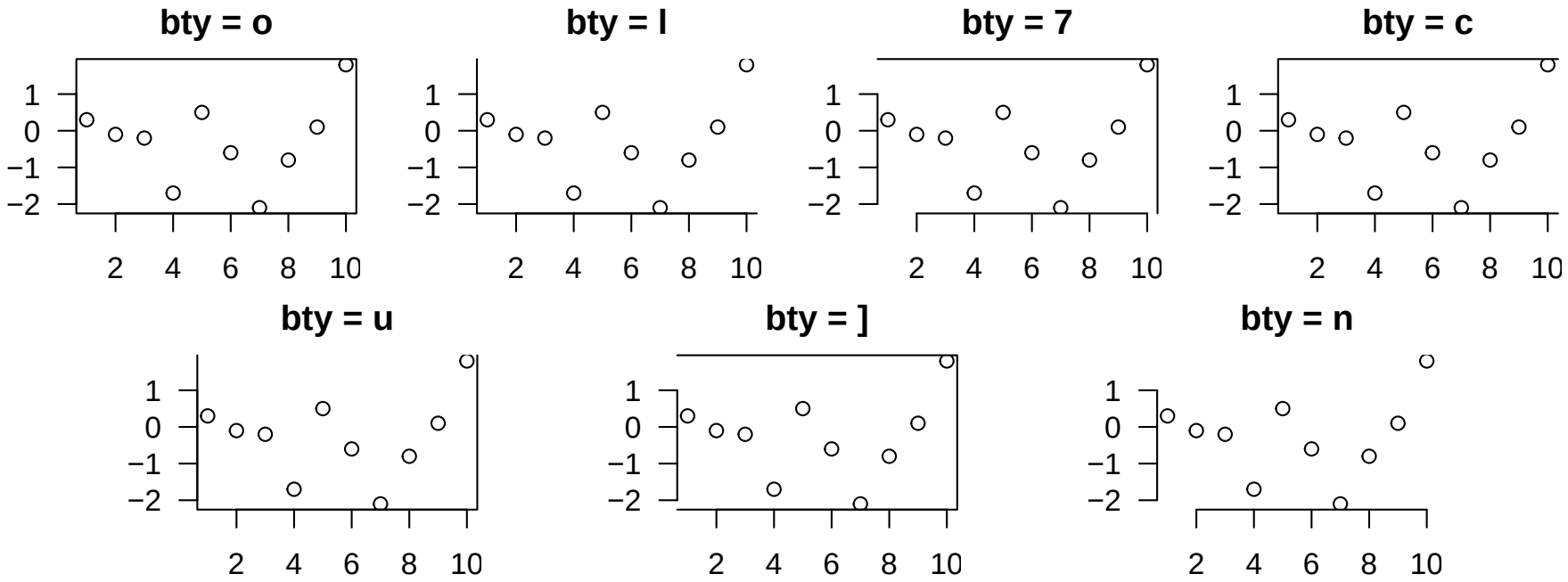


Looks ugly? No problem, we can make it much prettier with a couple of custom parameters.

Changing the box type

The argument `bty = "o"` defines the **b**ox **ty**pe around the plot. Use `bty = "n"` to remove it. Other types exist:

```
btys <- c("o", "l", "7", "c", "u", "]", "n")
for (b in btys) plot(x, y, bty = b)
```



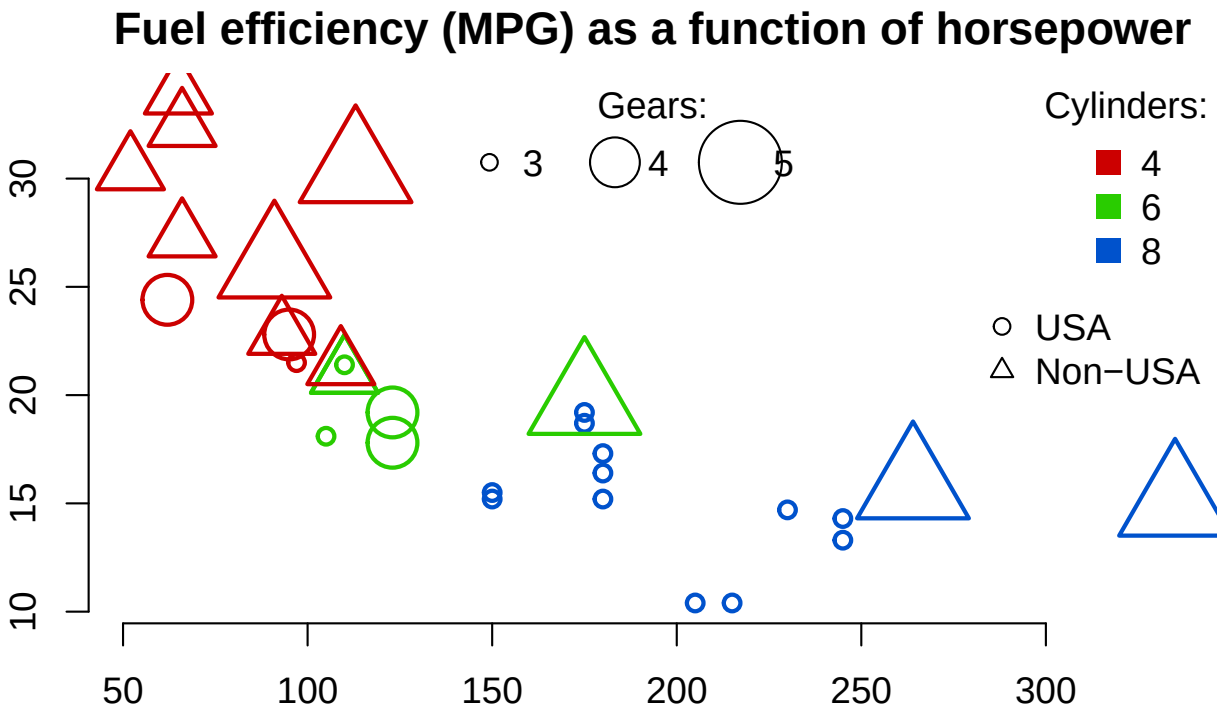
Idea behind plot vectorisation

Main concept: the optional arguments of `plot()` related to points / lines correspond to the input vectors. The arguments in the call are vectorised as follows:

$$\text{plot}\left(\text{x} = \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{pmatrix}, \text{y} = \begin{pmatrix} y_1 \\ y_2 \\ \vdots \\ y_n \end{pmatrix}, \text{pch} = \begin{pmatrix} pch_1 \\ pch_2 \\ \vdots \\ pch_n \end{pmatrix}, \text{col} = \begin{pmatrix} col_1 \\ col_2 \\ \vdots \\ col_n \end{pmatrix}, \cdots\right)$$

`plot()` can produce various visuals for all points at once without any loops / repeated calls: just pass the desired visual parameters as vectors.

Vectors of point parameters

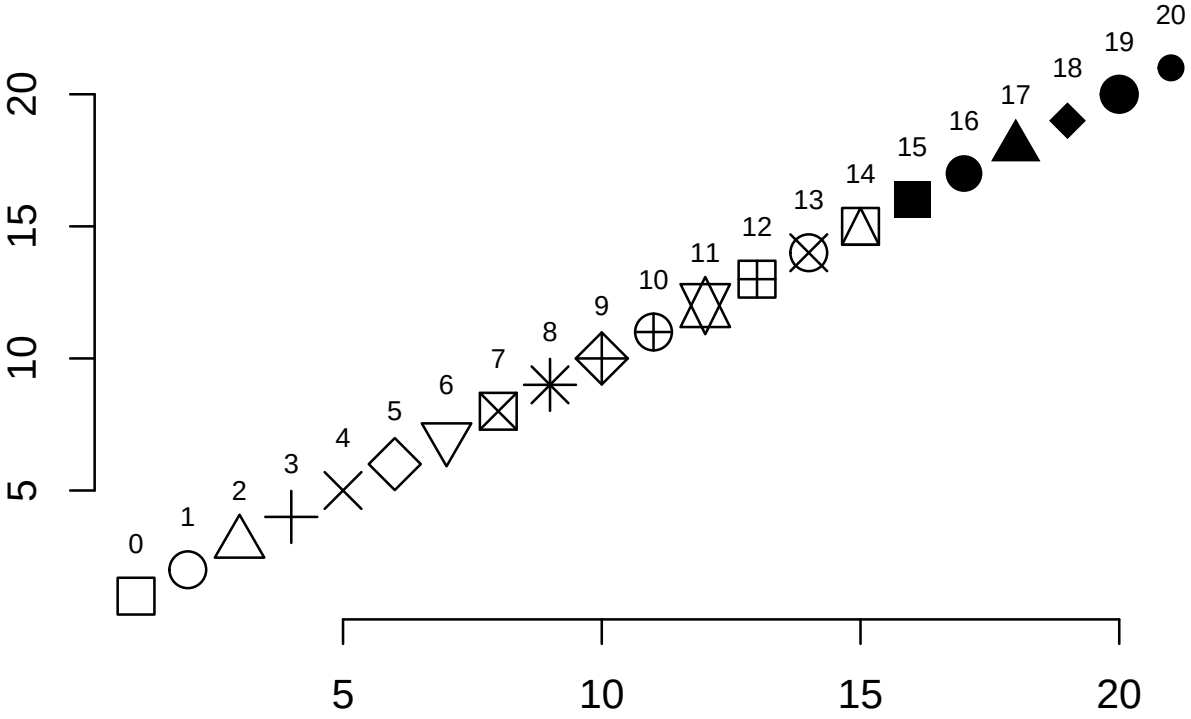



```r
mycols <- rainbow(3, end = 0.6, v = 0.8)
plot(mtcars$hp, mtcars$mpg,
     pch = 1 + mtcars$am,  # am is 0 or 1
     col = mycols[as.factor(mtcars$cyl)],
     cex = (mtcars$gear-3)*2+1)
```

Plotting character

In R, there are 21 distinct characters (0–20) in the plotting font. They are selected via `pch = <integer>`.

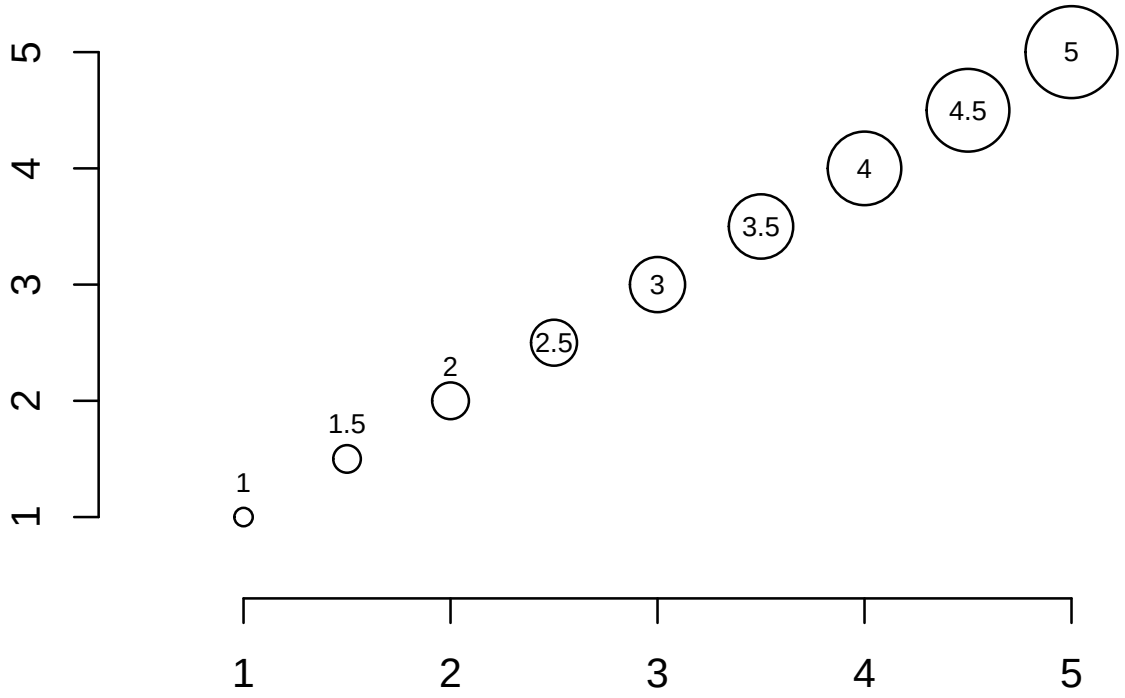
```
plot(0:20, 0:20, pch = 0:20)
```



Plotting character size

To set a different plotting character size, use `cex = <positive number>`; default = 1.

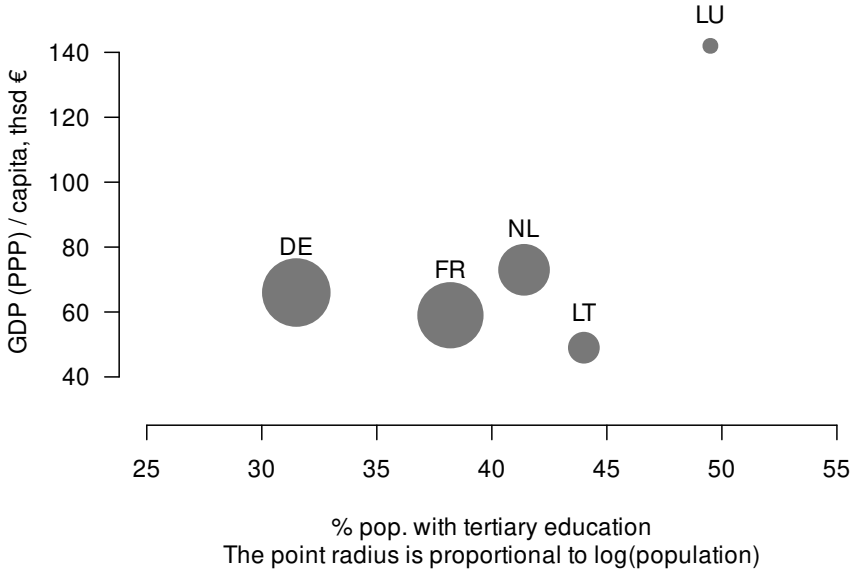
```
plot(2:10/2, 2:10/2, cex = 2:10/2)
```



Point size for importance

```r
d <- data.frame(
  country = c("DE", "FR", "LU", "LT", "NL"),
  edu3  = c(31.5, 38.2, 49.5, 44, 41.4),
  gdppc = c(66,   59,  142,   49, 73),
  pop   = c(83191, 67287, 639, 2786, 17501)
)

par(mar = c(5, 4, 0, 0)+.1)
psize <- log(d$pop) - 5
# psize = 6.3 6.1 1.5 2.9 4.8
plot(d$edu, d$gdppc,
  col = "#00000088",
  pch = 16,
  cex = psize)  # <== KEY LINE
text(x = d$edu,
  y = d$gdppc + psize + 8,
  labels = d$country)
```

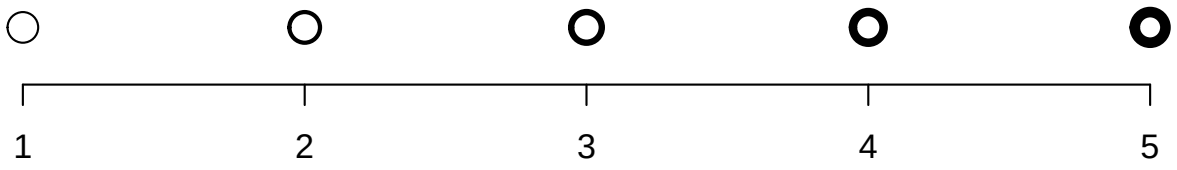


Making scatter plot circles thicker

Since the hollow circles, squares, and other plotting characters consist of lines, those lines can be made thicker if `lwd` is changed.

It may seem that `points()` does not do anything with line parameters, but specifically `lwd` changes the line thickness in the plotting character:

```
plot(rep(0, 5), pch = 1:5, cex = 2, lwd = 1:5)
```



Plotting colours

A colour in R is defined as a character vector starting with # followed by the hexadecimal value (also known as HTML colour). Use 6 characters for solid colours and 8 characters for semi-transparent ones.

- Red, green, blue: `"#FF0000"`, `"#00FF00"`, `"#0000FF"`
- Dark red: `"#880000"`
- Light blue: `"#880000"`
- Semi-transparent dark green: `"#00008888"`
  - Default: full opaqueness; `880000 = 880000FF`

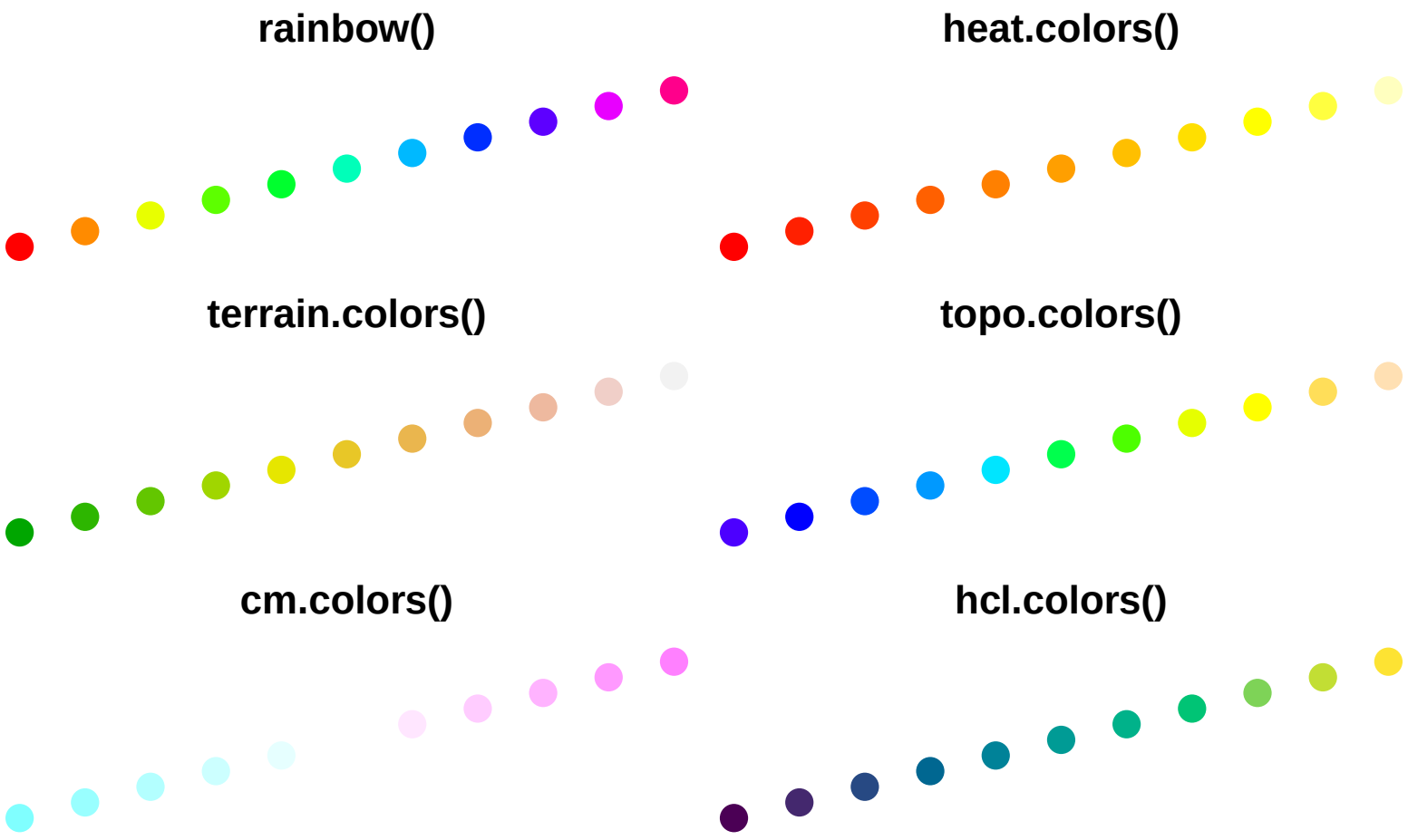
Names `"red"`, `"blue"` etc. are also supported.

Colour palettes

- `rainbow(10, v=0.8, start=0.1, end=0.6)` creates a uniform rainbow progression that starts at 0.1 and ends at 0.6 (0 = red, 0.2 = yellow, 0.4 = green, 0.6 = blue, 0.8 = purple), with lightness value 0.8 (1 = brightest)
- `heat.colors(n, alpha)` create a red-yellow-white gradient (good for heat maps)
- `terrain.colors(n)`, `topo.colors(n)`, `cm.colors(n)` return green-yellow-beige, blue-green-yellow, and azure-white-pink gradients respectively
- `hcl.colors(n, palette = "XYZ")` returns colours from a built-in palette

See `?hcl.colors`; `hcl.pals()` produces a list of palettes.

Colour examples



Adding transparency to colours

Since colours are characters, adding transparency to them is as easy as pasting 2 more characters for the alpha channel.

```
mycols <- c("#845ec2", "#0081cf", "#008f7a")
mycols1 <- paste0(mycols, "AA")
mycols2 <- paste0(mycols, "44")
plot(1:3, rep(1, 3), pch = 16, col = mycols)
points(1:3, rep(1.3, 3), pch = 16, col = mycols1)
points(1:3, rep(1.6, 3), pch = 16, col = mycols2)
```

Generate good palettes

Search ‘HTML palettes’ or ‘HTML colour picker’ online to get good colours!

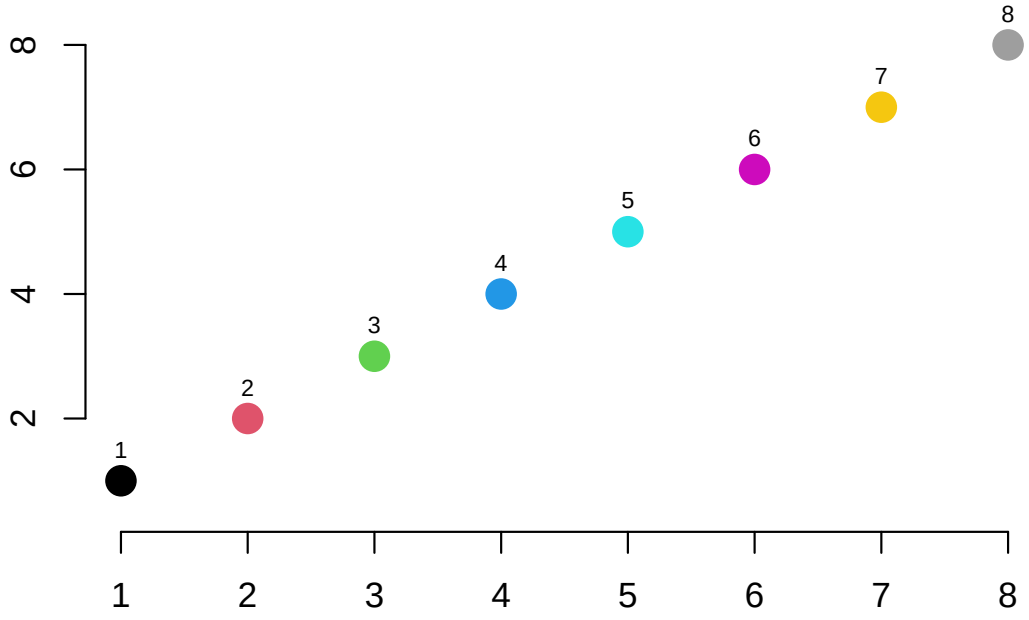
- Adobe colour wheel
- mycolor.space
- coolors.co

Test the palettes for accessibility (if possible).

Built-in numeric colours

Built-in colours 1–8 are simply integers – they look almost not too bad:

```
plot(1:8, 1:8, col = 1:8, pch = 16)
```



Adding things to plots

A plot might contain multiple lines, points etc.

To add elements:

- `points(x, y, ...)` adds more points
- `lines(x, y, ...)` adds lines going through the indicated points
- `text(x, y, labels ...)` adds text labels at the given coordinates

These command accept some common and some distinct elements.

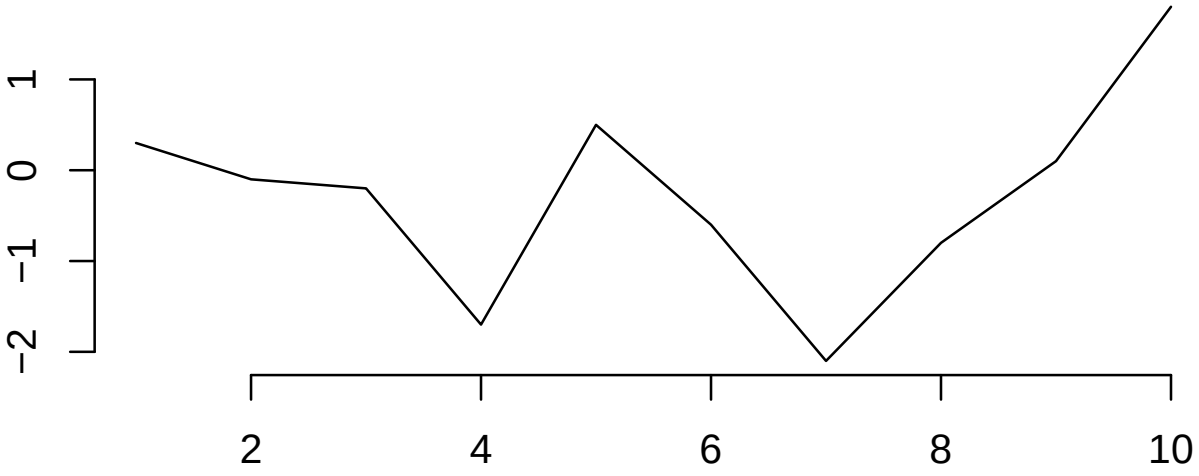
Changing plot type

- `plot(x, y, type = "l", ...)` creates a line plot
- `plot(x, y, type = "b", ...)` creates a line-and-point plot

Creating a line plot

`plot(x, y, type = "l", ...)` connects the points with a line:

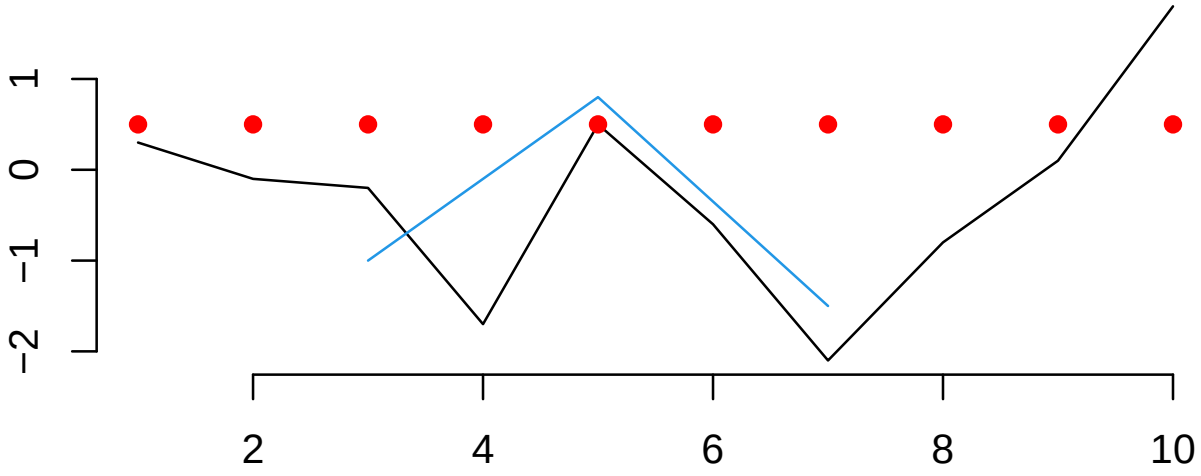
```
plot(x, y, type = "l")
```



Combining lines and dots

Invoke `points()` and `lines()` to gradually fill the plot with more elements.

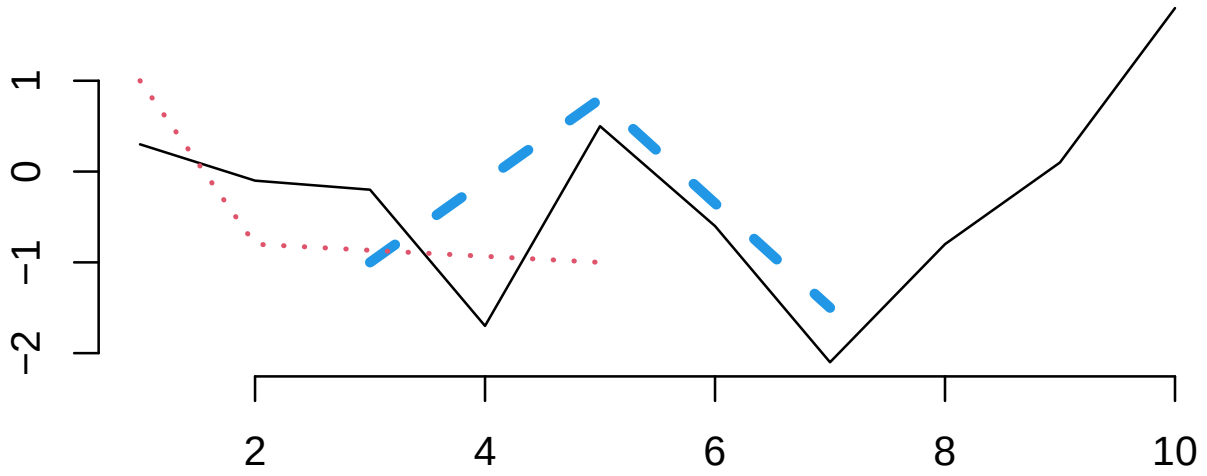
```
plot(x, y, type = "l")
points(1:10, rep(0.5, 10), pch = 16, col = "red")
lines(c(3, 5, 7), c(-1, 0.8, -1.5), col = 4)
```



Line parameters

- `lwd = <num>` modifies the line thickness (default: 1)
- `lty = <num>` modifies the line stroke (1: solid, 2: dash, 3: dotted, 4: dot-dash)

```
plot(x, y, type = "l", bty = "n")
lines(c(3,5,7), c(-1, 0.8, -1.5), col=4, lwd=4, lty=2)
lines(c(1,2,5), c(1, -0.8, -1),   col=2, lwd=2, lty=3)
```



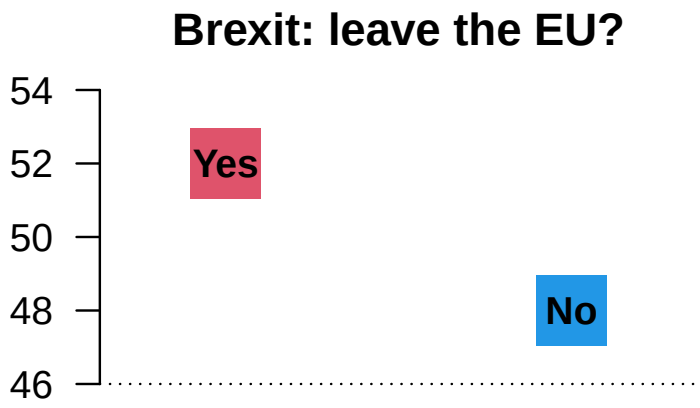
Plot limits

The arguments `xlim` and `ylim` of `plot()` are length-2 vectors defining the horizontal and vertical plotting range.

Default: `xlim = range(x), ylim = range(y)`.

Tip: if possible, include zero: it gives the perspective.

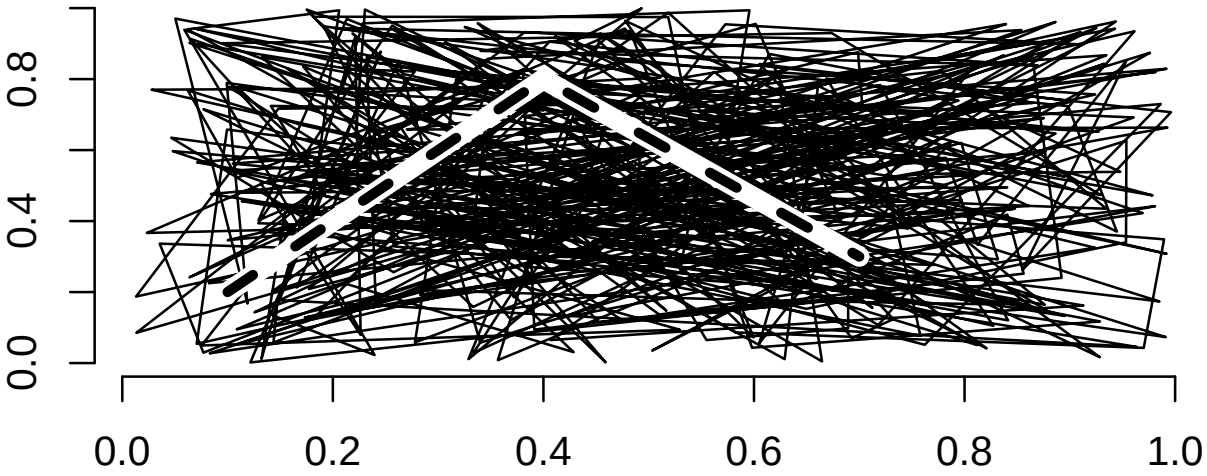
```
plot(1:2, c(52, 48), pch = 15, col = c(2, 4),
  xlim = c(0.7, 2.3), ylim = c(46, 54), cex = 4)
text(1:2, c(52, 48), c("Yes", "No"), font = 2)
# Right: the same, BUT ylim = c(0, 60)
```



Tip: create contours

If there are many points, create a thick white outline to make the line stand out. Exaggerated example:

```
set.seed(1); x1 <- runif(400); y1 <- runif(400)
plot(x1, y1, type = "l", bty = "n", col = "#000000")
lines(c(.1,.4,.7), c(.2,.8,.3), lwd = 8, col = "white")
lines(c(.1,.4,.7), c(.2,.8,.3), lwd = 4, lty = 2)
```



Plot any list with x and y

If a list contains named components x and y, R will attempt to automatically plot `plot(a)` as `plot(a$x, a$y)`.

```
a <- list(x = 1:10, y = 1:10, s = "Thing")
plot(a)
```

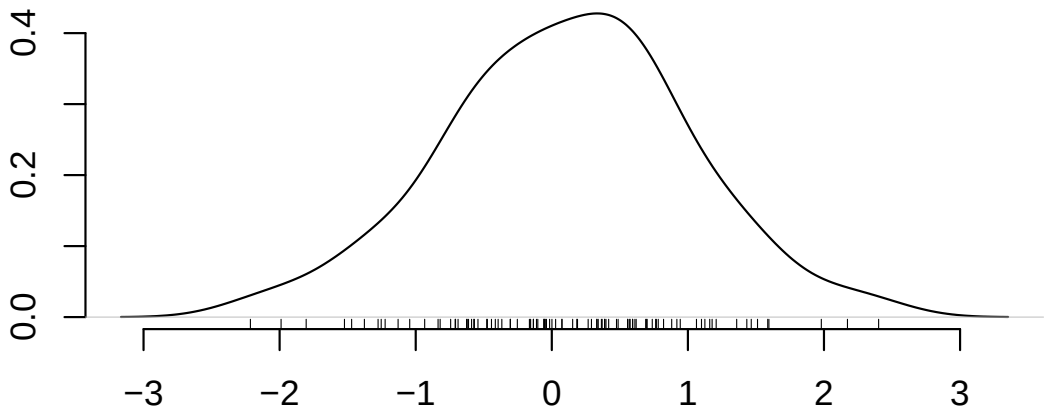
This is why many popular functions return lists containing x and y components:

```
a <- density(1:10)
head(cbind(a$x, a$y))
#>       [,1]         [,2]
#> -4.157859 0.0003034332
#> -4.120059 0.0003245074
#> ...
```

Density plots and rugs

Very popular are density plots: visualise a smooth estimated distribution of your data. Add rugs to show the original values. Remove the title with `main = ""`.

```
set.seed(1); xs <- rnorm(100)
d1 <- density(xs)
plot(d1, bty = "n", main = "")
rug(xs)
```



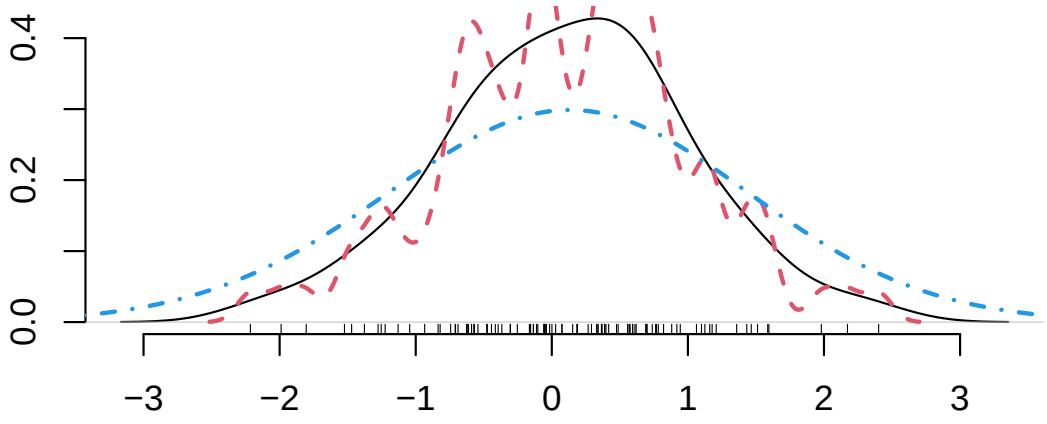
Adding density lines

```
d2 <- density(xs, bw=0.1); d3 <- density(xs, bw=1)
```

If one needs two density plots, calling `plot(d1)` and `plot(d2)` will cause the new plot to overwrite the old one.

Add lines by examining `str(d1)` and noting the `names(d1)` contains elements named `"x"` and `"y"`.

```
plot(d1, bty = "n", main = ""); rug(xs)
lines(d2$x, d2$y, col = 2, lty = 2, lwd = 2)
lines(d3$x, d3$y, col = 4, lty = 4, lwd = 2)
```



Starting with a clean slate

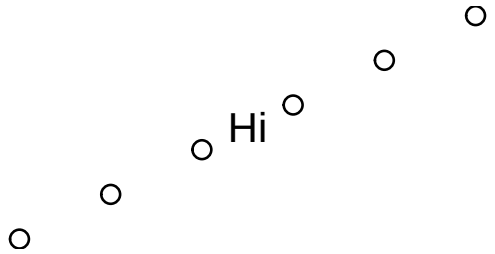
`plot.new()` opens an empty canvas: (1) no points, (2) both ranges [0, 1], (3) no box, (3) no axes, (5) no axis labels:

```
plot(NULL, NULL,
  xlim = 0:1, ylim = 0:1,
  bty = "n",
  xaxt = "n", yaxt = "n",
  xlab = "", ylab = "")
```

⇔

```
plot.new()
# Gives
# the same
# blank
# canvas
```

```
points(0:5/5, 0:5/5)
text(0.5, 0.5, "Hi")
```



However, the solution in the left column must be used if a custom plotting range – `xlim` and `ylim` – is necessary.

Preventing clipping with ranges

In the example above, the red density (`d2`, with the smallest bandwidth) was partially clipped because `ylim` for this plot was computed for `d1`.

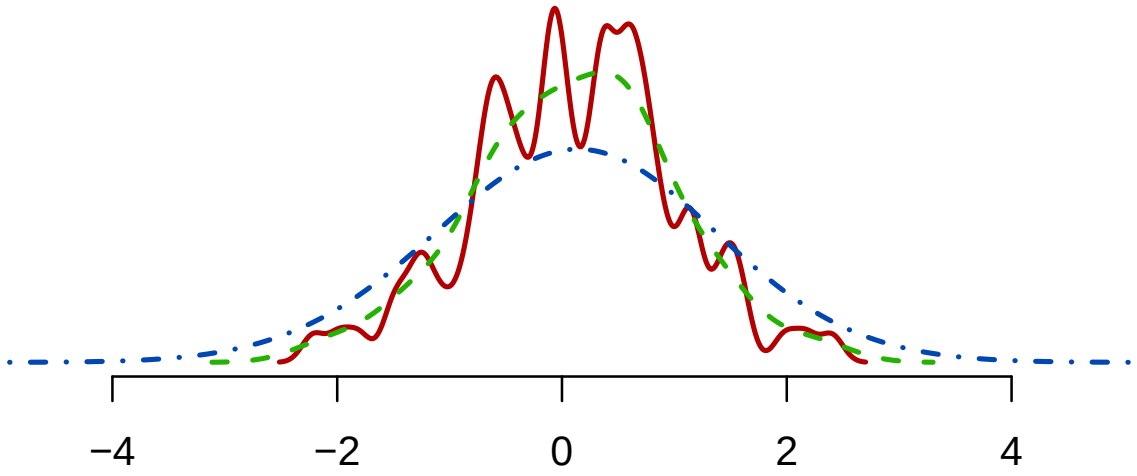
It is a good idea to compute the limits in advance if multiple objects are to be plotted. Note that `range(x1, x2) = range(range(x1), range(x2))`.

An empty plot with `x = NULL` and `y = NULL` requires the mandatory ranges.

Tip: for multi-element plots, combine them into the list and (l,s)apply `range()`.

No-hardcoded-ranges example

```
bws <- c(0.1, 0.3, 0.9)
d.list <- lapply(bws, \(b) density(xs, bw = b))
xl <- range(sapply(d.list, \(d) range(d$x)))
yl <- range(0, sapply(d.list, \(d) range(d$y)))
plot(NULL, NULL, xlim = xl, ylim = yl, bty = "n")
mycols <- rainbow(3, end = 0.6, v = 0.7)
ltys <- c(1, 2, 4)
for (i in 1:3)
  lines(d.list[[i]], col = mycols[i], lty = ltys[i])
```



Saving graphics

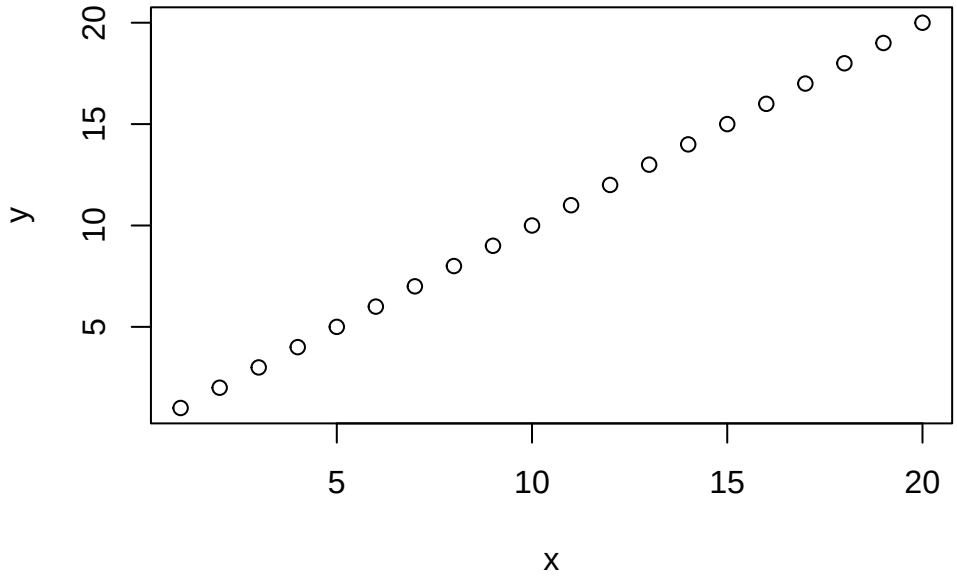
R features devices that can be written to.

(On Linux, everything is a file – a device is a file, too.)

To save graphics to an external PDF or PNG file,

1. Open a device with the desired parameters and the file path
2. Run all the plotting command
3. **Close the device**

How to save a plot to PDF in R




```r
x <- y <- 1:20
pdf("test.pdf", width = 5, height = 3)
plot(x, y)
dev.off()  # Finalise writing the file
```

Default width and height units for PDF: inches.

Saving in other formats

- PNG: use the cairo device for smooth plots (default Windows plots look jagged). Width and height: pixels.
```r
png("test.png", width = 800, height = 480)
# Plotting commands
dev.off()
```
- PDF with full Unicode character support. Width and height: inches.
```r
cairo_pdf("test.pdf", width = 8, height = 5)
```
- TIFF: turn on lossless compression to avoid having huge files. Width and height: pixels.
```r
tiff("test.tif", compress = "lzw",
     width = 800, height = 480)
```

At the end: always `dev.off()`.

Troubleshooting: nothings plots / updates

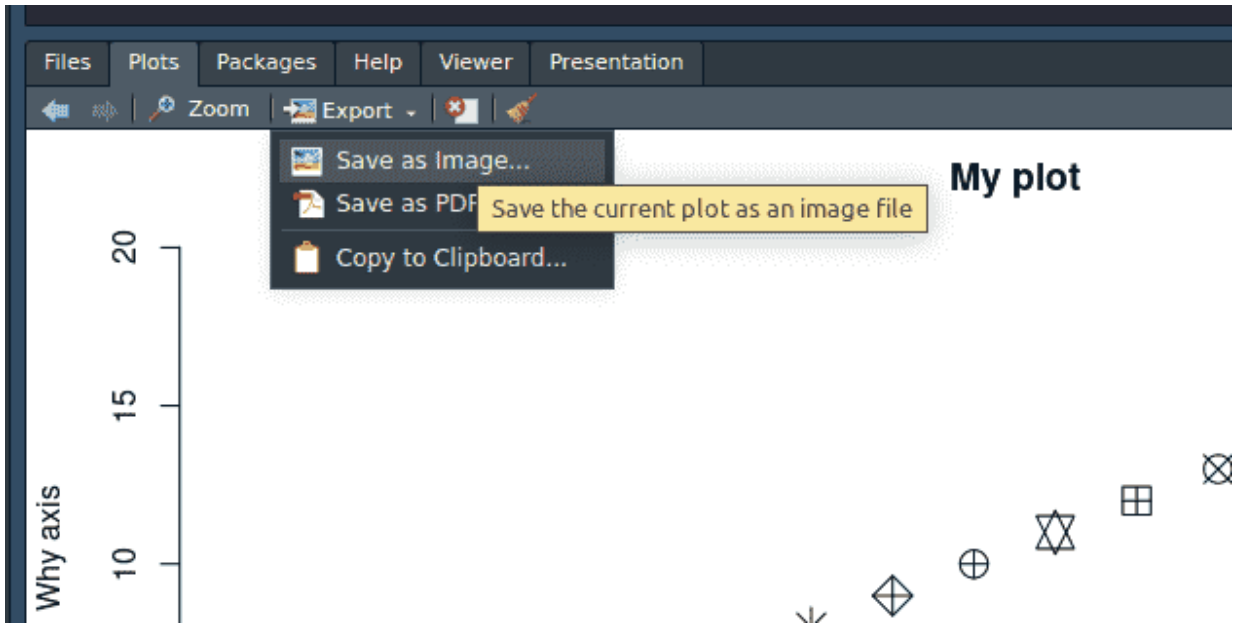
If `png()`, `pdf()`, or something similar is used to write to an external device, then, the device must be closed so that the plot window starts refreshing again.

If plotting commands do not show any new plots or additions to the plots, **do not panic**. Some device is open and is being written to instead of RStudio GD.

Solution: if plotting stopped working, run `dev.off()` multiple times to close all graphics devices (until 'cannot shut down device 1' appears).

RStudio tip: quick-and-dirty plot export

The plot panel allows one to open a plot in full screen, or copy it, or export.



The legend or text spacing may be broken after manual rescaling, though.

Image resolution

Raster images come in **pixels**. When transferred to screens or paper, the density of pixels on the physical medium (how many fit into 1 inch) is measured in DPI (dots per inch).

A source image printed at 300 DPI will be 2× smaller than the same one printed at 150 DPI.

Changing the DPI of a digital will only change the size of its physical output, but nothing in its screen representation.

Convert centimetres at desired DPI into pixels:
A4 paper (21 × 29.7 cm) @ 300 DPI =
$(21/2.54 \cdot 300) \times (29.7/2.54 \cdot 300) = 2480 \times 3508$ pixels

Without physical reference units, DPI is meaningless.

Creating files with specified resolution

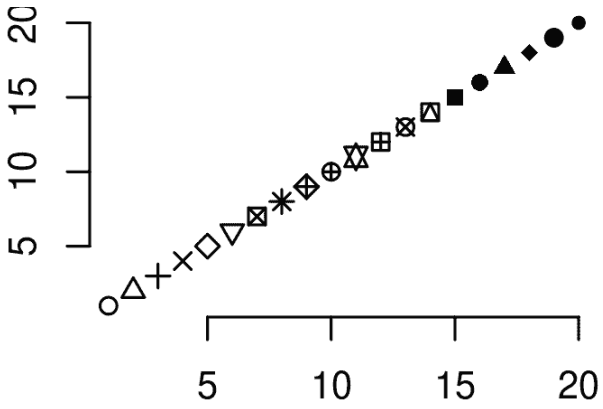
Sometimes, journals require PNG/TIFF images with resolution *X* DPI (usually 300 or 600).

Suppose that the text area is 12 cm wide and there 2 plots to be placed side-by-side (5.7 cm each) and the desired W:H ratio is 3:2.

2.54 cms in an inch. We use inches because DPI is universal, whereas dots-per-cm is not.

Example: using DPI to create PNG

```r
w <- 5.7 / 2.54 # Inches
png("s05-test-res.png",
  width = w, height = w/3*2,
  units = "in", res = 300,
  pointsize = 10, type = "cairo")
par(mar = c(2, 2, 0, 0)+.1)
# We talk about this par later
plot(x = 1:20, y = 1:20,
  pch = 1:20, cex = 1,
  bty = "n", main = "",
  xlab = "", ylab = "")
dev.off()
```



Troubleshooting: wrong TIFF resolution

Sometimes, on Mac, if `tiff()` is requested to write an image at high DPI, the output dimensions may be correct, but the DPI in the file properties can be wrong (72 DPI).

Reason: quirks of X11, `cairo`, and `quartz` device detection / selection.

Foolproof solution: call ImageMagick to fix any broken resolution! Process multiple files at once:

```
fn <- c("Fig1.tiff", "Fig2.tiff")
for (v in fn) {
  e <- paste0("convert ", v, " -density 300
    -compress LZW -verbose ",
    gsub("\\.tiff$", "-opt.tiff", v))
  print(e); system(e)}
```

Setting plot margins

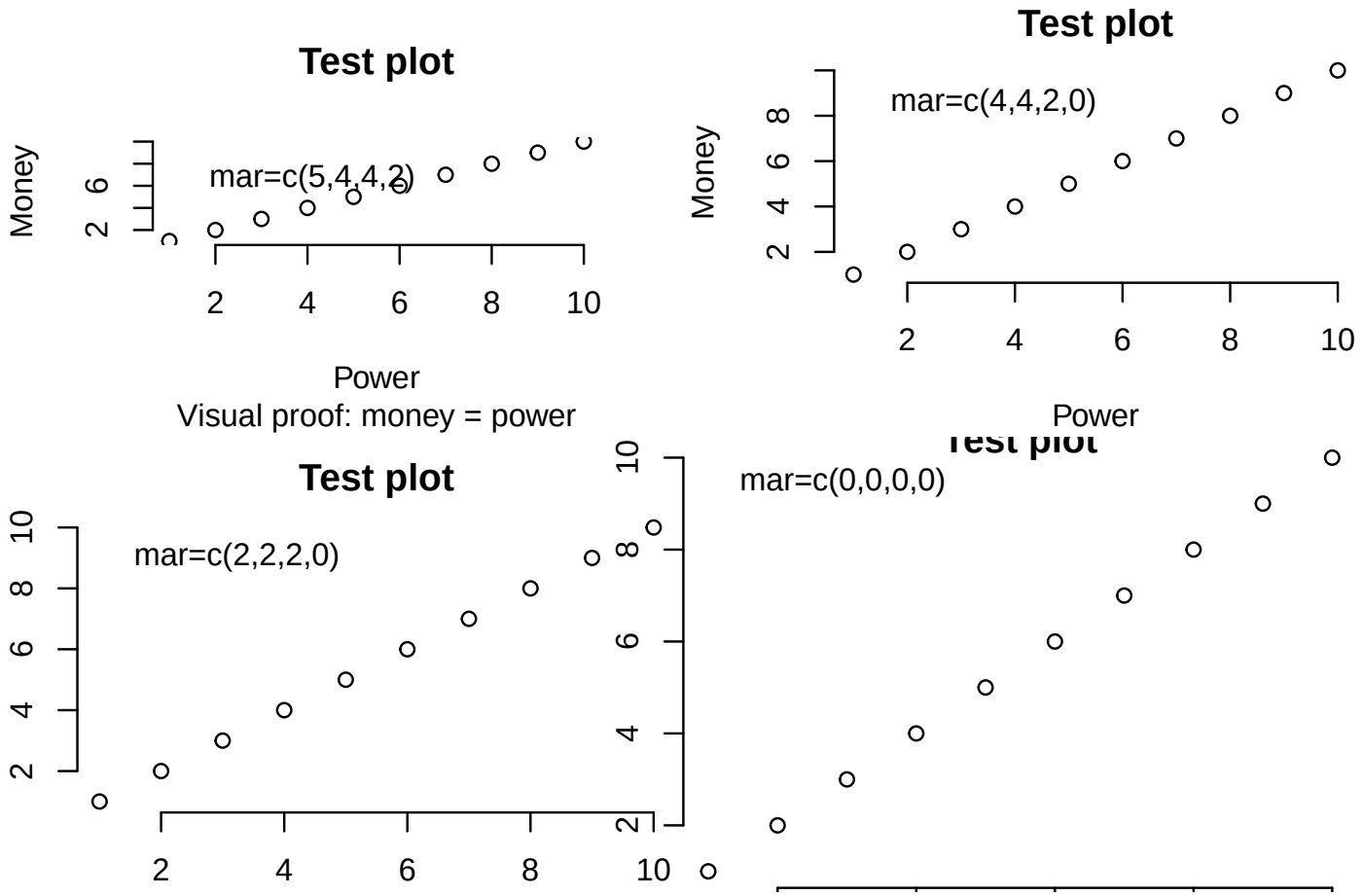
`par()` is a flexible function that can set many graphical parameters. Call `par()` before any `plot()` or similar calls.

The most useful one is plot margins (a length-4 numeric vector). They go in this order: bottom, left, top, right.

Default value: `c(5, 4, 4, 2) + 0.1`.

- To plot axis + marks + axis name, use 4.1; for axis + marks, use 2.1; to plot nothing, use 0.1; if `plot(..., sub = "Subtitle")` is called, use 5.1
- Top: to plot the main title, use 2.1 (otherwise 0.1)

Plot margins example

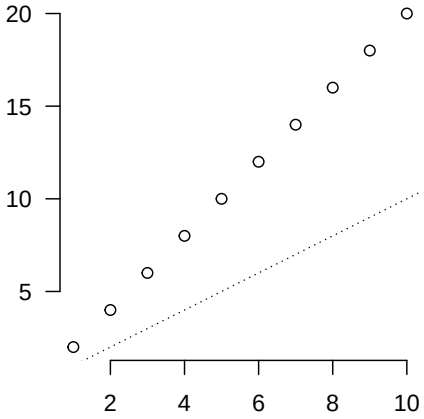


NB: even though zero margins were requested at times, the extra elements are still visible! Remove them with `main = ""`, `xaxt = "n"`, `xlab = ""` etc.

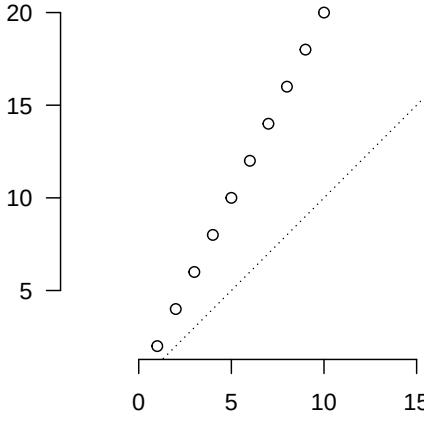
Plot aspect ratio

A plot is bounded by `xlim` and `ylim`, and the result is produced on a canvas of fixed size (sans the margins). As a result, 1 cm or pixel in the horizontal direction may contain more or fewer units than 1 cm in the vertical direction.

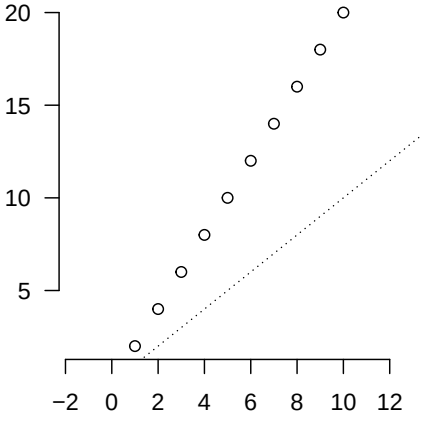
To force an aspect ratio, use `plot(..., asp = 1)` (or other number) – compare the axis ranges and the 45° line:




```
plot(1:10, (1:10)*2)
abline(0, 1, lty=3)
```




```
plot(1:10, (1:10)*2,
   asp = 1)
abline(0, 1, lty=3)
```




```
plot(1:10, (1:10)*2,
   asp = 0.8)
abline(0, 1, lty=3)
```

Customising axes

To change axis parameters, remove the respective axis by calling `plot(xaxt = "n")` (or `yaxt = "n"`) and draw a new one with customisation,

`axis(1)` is a command that puts an axis at the bottom, `axis(2)` on the left side, `axis(3)` and `axis(4)` on the top / right side respectively.

`axis(1, at=c(1,4,7), labels=c("L", "M", "H"))` places the horizontal axis at the bottom with three marks and three custom labels.

Label orientation: `las=1` creates horizontal labels, `las=2` rotates them by 90°. Do not break the reader's neck!

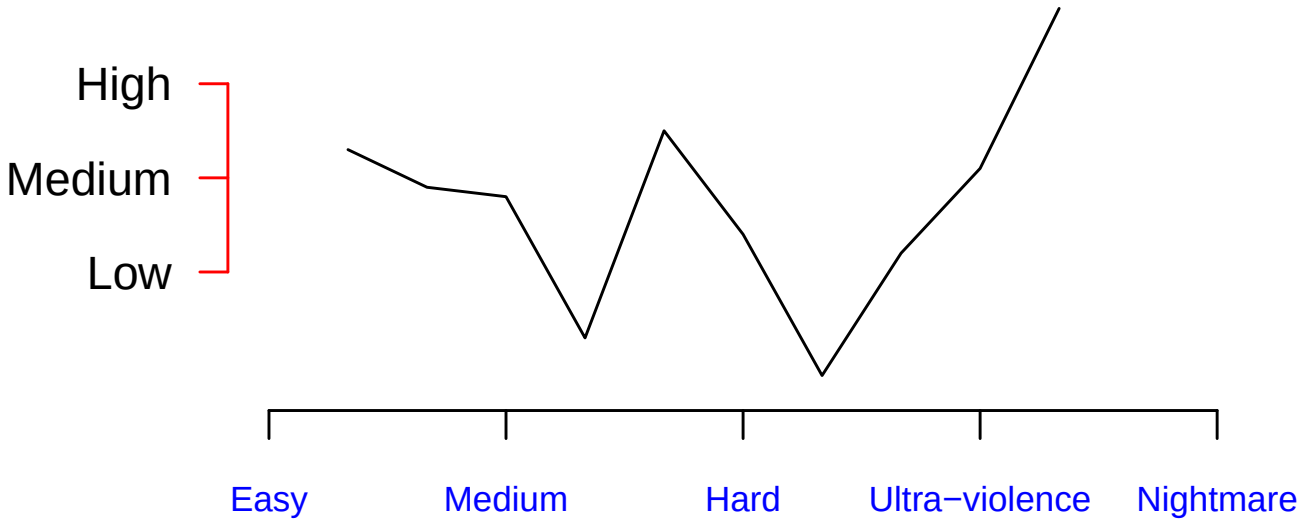
Separation between axis and labels

The notation is a bit confusing:

- `axis(..., cex.axis = 0.8)` or `plot(..., cex.axis = 0.8)` scales only the labels
- `axis(..., col.axis = "red")` or `plot(..., col.axis = "red")` changes only the label colour
- `axis(..., col = "red")` changes only the axis line colour

Axis customisation example

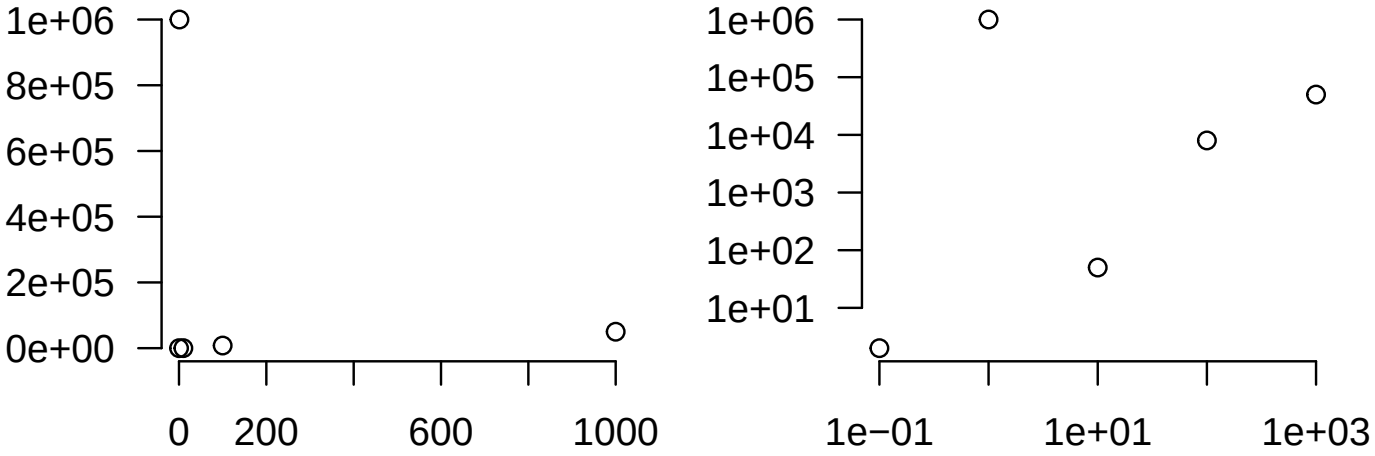
```r
x <- 1:10  # Full unabridged code, nothing simplified
y <- c(0.3, -0.1, -0.2, -1.7, 0.5, -0.6, -2.1, -0.8,  0.1, 1.8)
pdf("s05-plot-axis.pdf", 5, 2)
par(mar = c(2, 5, 0, 0)+.1)  # Large left margin for horiz. text
plot(x, y, type = "l", bty = "n", xaxt = "n", yaxt = "n",
   xlim = c(0,13), ylim = range(y) - 0.2*c(1,-1), xlab="", ylab="")
axis(1, at = seq(0, 12, 3),labels = c("Easy", "Medium",
  "Hard", "Ultra-violence", "Nightmare"), las = 1,
  cex.axis = 0.75, col.axis = "blue")
axis(2, -1:1, c("Low", "Medium", "High"), las = 1, col ="red")
dev.off()
```



Logarithmic axes

If the data (1) are positive and (2) span several orders of magnitude, using logarithms helps showing exponential changes on a linear scale. `plot(..., log = "xy")` for both log axes, or `log = "x"` / `log = "y"` for only one.

```
x <- 10^(-1:3)
y <- c(2, 1e6, 50, 8000, 5e4)
plot(x, y)
plot(x, y, log = "xy")
```



Adding orthogonal lines

- Grids can be ugly, redundant, and noisy
- Grids can be useful in certain cases to spot tiny deviations

`abline(...)` draws many types of lines:

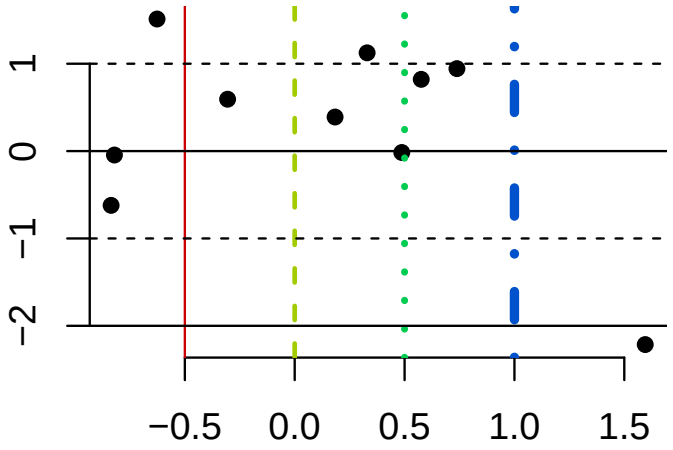
- `v = c(x1, x2, ...)` puts vertical lines at $x_1, x_2, \ldots$
- `h = c(y1, y2, ...)` puts horizontal lines at $y_1, y_2, \ldots$
- `a = 0.5, b = 1` (both are necessary) draws $y = a + bx$

abline() styles are vectorised

For a vector of vertical or horizontal positions, `abline()` can use a vector of colours, widths, or types to produce multiple styles at once.

NB. Arguments `a` and `b` for sloped lines are not vectorised.

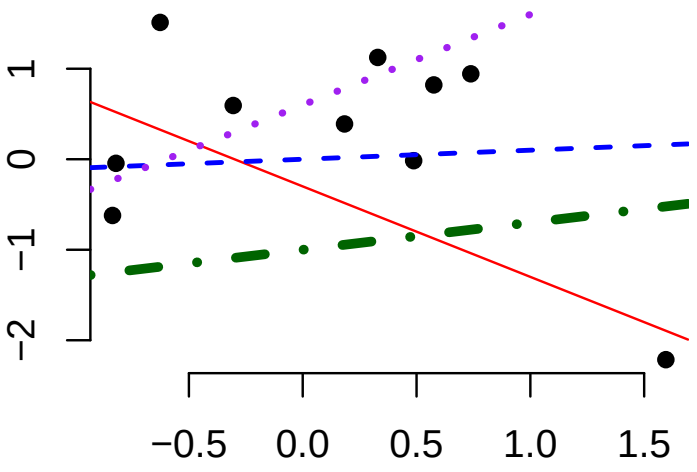
```
set.seed(1)
x <- rnorm(10); y <- rnorm(10)
plot(x, y, pch = 16)
abline(v = seq(-0.5, 1, 0.5),
  col = rainbow(4,
    end = 0.6, v = 0.8),
  lwd = 1:4, lty = 1:4)
abline(h = -2:1, lty = 1:2)
# lty is recycled 2 to 4
```



Prepare DFs for non-vectorised lines

Multiple sloped lines in various styles from 1 DF in a loop:

```
ab <- data.frame(a = c(-0.3, 0, 0.6, -1),
  b = c(-1, 0.1, 1, 0.3), lty = 1:4, lwd = 1:4,
  col = c("red", "blue", "purple", "darkgreen"))
plot(x, y, bty = "n", xlab = "", ylab = "", pch = 16)
for (i in 1:nrow(ab))
  abline(a = ab[i,"a"], b = ab[i,"b"], # With $ or ""
  lty = ab$lty[i], lwd = ab$lwd[i], col = ab$col[i])
```

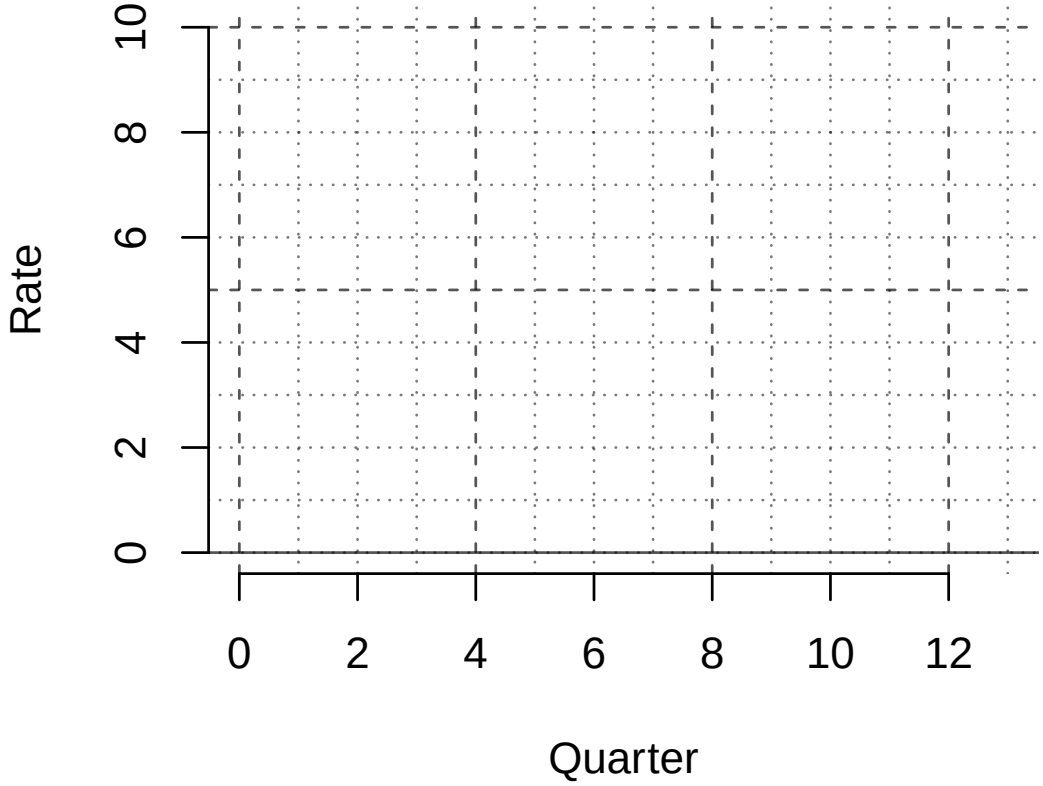


Primary and secondary grid

Recall Session 4: we used ‘round’ numbers to put solid grid lines, and added several faint lines in between. If the number of elements is too high to allow convenient vectorisation, repetition leads to more readable code:

```
v1 <- seq(0, 12, 4); v2 <- setdiff(0:13, v1)
h1 <- c(5, 10); h2 <- setdiff(0:10, h1)
# v1, h1 = coarse grids, v2, h2 = fine grids
cs <- c("#000000AA", "#00000088")
plot(NULL, NULL, xlim = c(0, 13), ylim = c(0, 10),
  bty = "n", xlab = "Quarter", ylab = "Rate")
abline(v = v1, col = cs[1], lty = 2)
abline(v = v2, col = cs[2], lty = 3)
abline(h = c(0, h1), lty = c(1, 2, 2),
  col = c("black", cs[1]))
abline(h = h2, col = cs[2], lty = 3)
```

Double grid



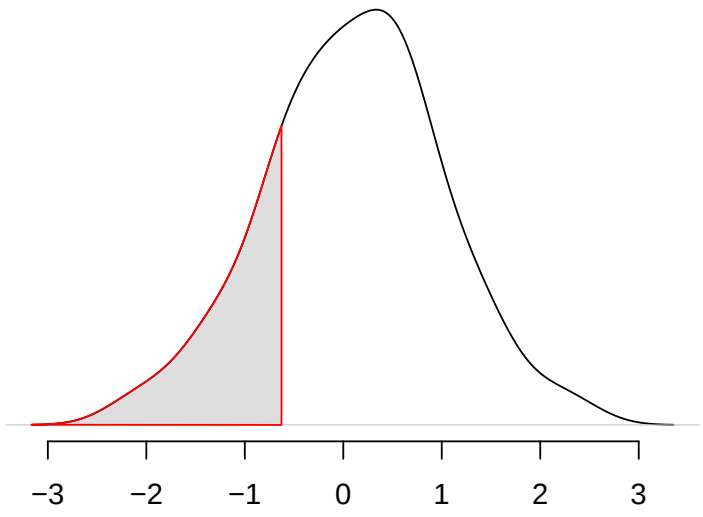
The grid must not obstruct the plotted object. Resist the temptation to put a grid on everything.

Tip: zero lines ($x = 0$, $y = 0$) + axis labels are usually enough.

Polygons

`polygon(x, y, ...)` adds a shaded area defined by the shape formed by the points (x, y).

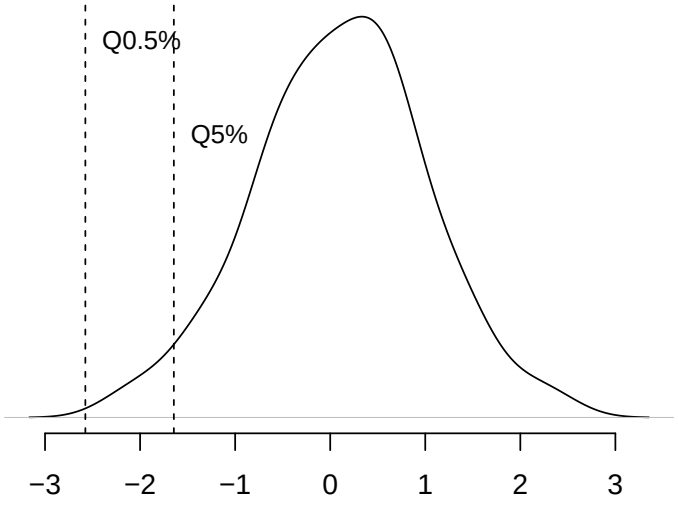
```
set.seed(1); xs <- rnorm(100)
d1 <- density(xs)
plot(d1, bty = "n", yaxt = "n")
polygon(d1$x[c(1:200, 200)], c(d1$y[1:200], 0),
  border = "red", col = "#00000022")
```



Text

- Same syntax as with `points()`
- Adjust the placement with `pos = i`: 1 = bottom, 2 = left, 3 = top, 4 = right
- Rotate text with `srt = <angle>` in degrees

```
plot(d1)
xs <- qnorm(c(0.005, 0.05))
abline(v = xs, lty = 2)
text(xs, c(.4, .3), cex = 0.9,
  c("Q0.5%", "Q5%"), pos = 4)
```



Simple formulæ with expression()

Put complex formulæ in the article text, use the simplest notation in graphs.

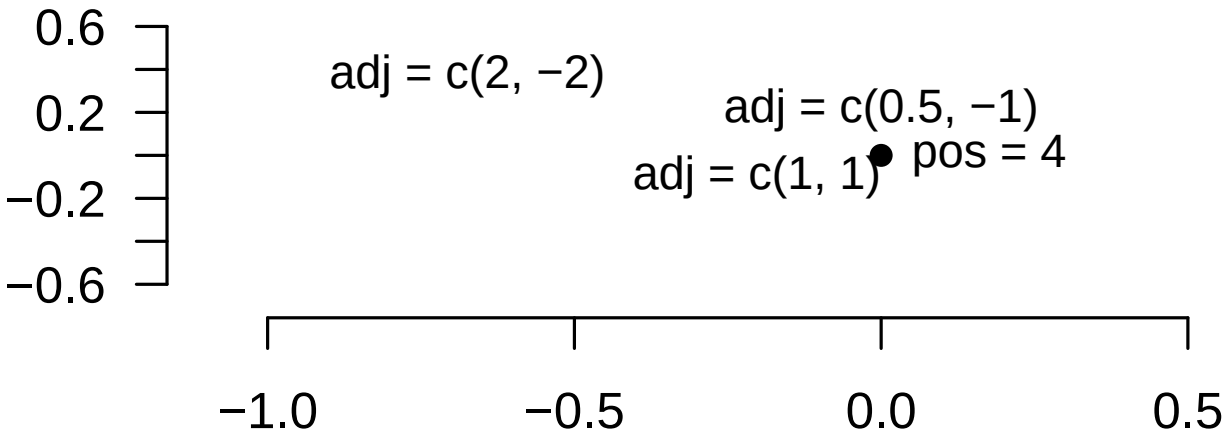
`expression()` supports simple formulæ. Run `demo(plotmath)` for the comprehensive demonstration.

- x_i → `x[i]`, $x^{\alpha\beta}$ → `x^{alpha * beta}`
- Var X → `Var ~ italic(X)`, $a = b$ → `a == b`
- $\tilde{\theta}_1$ → `tilde(theta)[1]`, $\widehat{\mathrm{Cov}}$ → `widehat(Cov)`

Adjusting text position

Apart from `pos = i`, there is a more flexible way to adjust the text position: `text(..., adj = c(a, b))`.

- Default: `adj = c(0.5, 0.5)` centres the text
- `a<0.5` moves the text to the right, `a>0.5` – left
- `b<0.5` shifts the text to the top, `b>0.5` – bottom



Legend

A legend clarifies the notation by creating a rectangle with information inside the plotting region.

- `x`: position (character): `"topright"`, `"bottom"`
- `legend`: character vector
- `pch`, `lty`, `col`, `lwd`, `cex`: styles to show
  - Provide `pch = 16` for filled dots
  - Provide `lwd = 2` for lines
- `box.col = "#FFFFFF88"` + `bg = "#FFFFFF88"` to make it semi-opaque, `bty = "n"` to remove the box
- `cex` changes label sizes, `pt.cex` changes point sizes

Check the help: the amount of customisations is overwhelming. A legible legend is essential for publications.

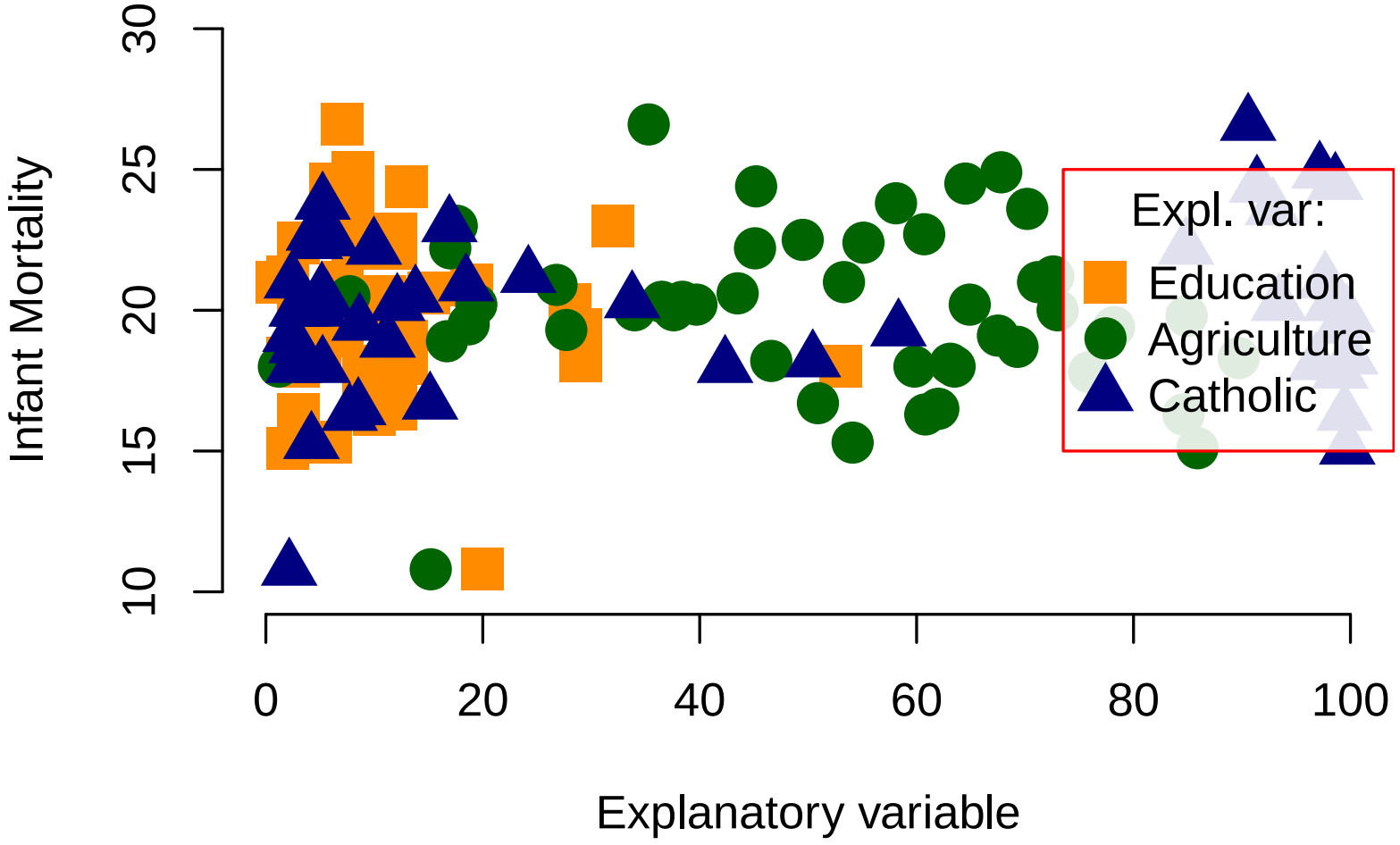
Legend code

Recall this example from earlier:

```
par(mar = c(4, 4, 0, 0)+.1)
y   <- swiss$Infant.Mortality
xs  <- c("Education", "Agriculture", "Catholic")
cls <- c("darkorange", "darkgreen", "navy")
plot(swiss[, xs[1]], y,
     pch = 15, col = cls[1], cex = 2, bty = "n",
     xlab = "Explanatory variable",
ylab = "Infant Mortality",
     xlim = c(0, 100), ylim = c(10, 30))
points(swiss[, xs[2]], y, pch=16, col=cls[2], cex=2)
points(swiss[, xs[3]], y, pch=17, col=cls[3], cex=2)

legend("right", xs, col=cls, pch=15:17, pt.cex=2,
       box.col = "#FF0000", bg = "#FFFFFFE1",
       title = "Expl. var:")
```

Plot with a legend (ugly example)



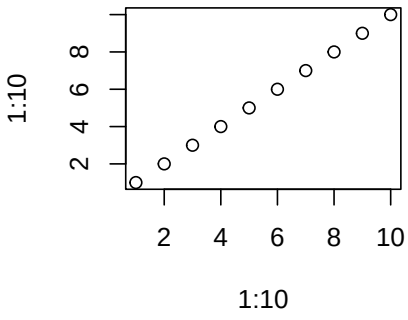
General de-cluttering

- Drop the box (`bty = "n"`)
  - Drop the legend box; make the legend background white, maybe slightly transparent
- Drop the axis names for plots in papers (`xlab = ""`, `ylab = ""` or make them minimalistic)
  - Write in human language: 'This plot shows the dependence of labour income (vertical axis) on age (horizontal axis)'
  - Orient the value labels (`las = 1`) to save many necks
- Drop the plot titles in papers (`main = ""`), add the caption in the document
  - `par(mar = c(2, 2, 0, 0)+.1)` seems appropriate for axes and tick labels
- Drop the vertical axis for densities and histograms
  - By definition, they integrate / add up to 1

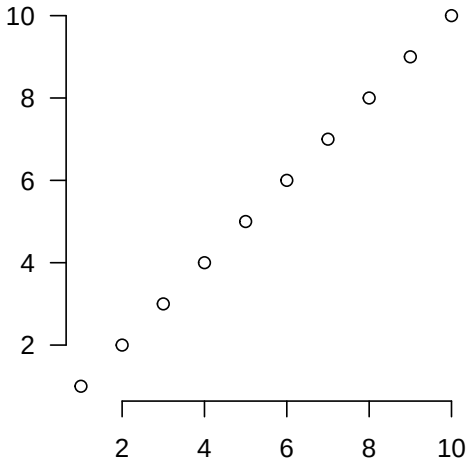
Automating clean default settings

- Functions are useful in automating customisation
- Ellipses allow passing arguments further

```
myPlot <- function(..., mar = c(2,2,0,0)+.1, bty = "n",
                   xlab = "", ylab = "", main = "", las = 1) {
  par(mar = mar)
  plot(..., xlab = xlab, ylab = ylab, main = main,
       bty = bty, las = las)}
```




```
plot(1:10, 1:10)
```




```
myPlot(1:10, 1:10)
```

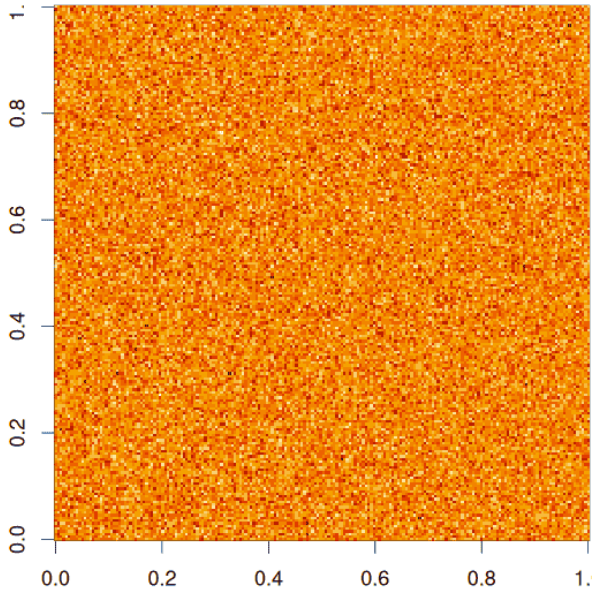
Any questions on points, lines, text, polygons, or margins?

Popular plots and tips for them

optipng and pngquant in action

Compressing a complex image:

```
png("v1.png", 512, 512,
  pointsize = 18, type = "cairo")
par(mar = c(2, 2, 0, 0)+.1)
set.seed(1)
image(matrix(rnorm(40000), 200))
dev.off()
```




```
system("optipng -preserve -o5 -out v2.png -clobber v1.png")
system("pngquant --quality 50-60 --speed 1 --force--output v3.png
 ↪  v1.png")
```

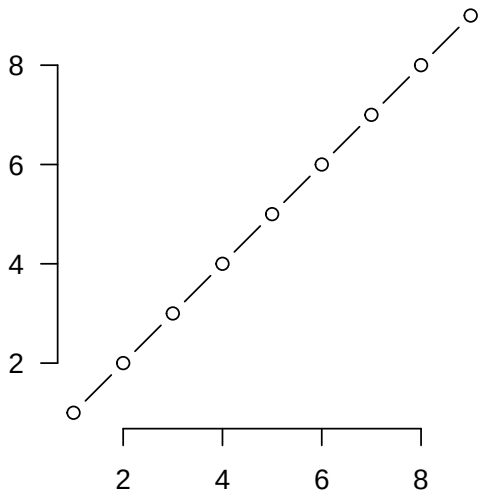
`file.size(paste0("v", 1:3, ".png"))` shows: original – 57 kB, optipng – 41 kB, pngquant – 27 kB.

Plot types vs. points / lines

Do not worry about setting `type = "l"` or `type = "b"`.

These 3 snippets return exactly the same output:

```
plot(1:9, 1:9, type = "b")
plot(NULL, NULL,
  xlim = c(1, 9), ylim = c(1, 9))
points(1:9, 1:9, type = "b")
plot(NULL, NULL,
  xlim = c(1, 9), ylim = c(1, 9))
lines(1:9, 1:9, type = "b")
```



`type = "b"` produces **b**oth points and lines with gaps; `type = "o"` **o**verlays them without gaps. Both `points()` and `lines()` accept the `type` argument.

Margin text

The labels `xlab` and `ylab`, the `main` and `sub` text in `plot()` are plotted in the outer margins.

- Change the space between to the axis label
- Change the margin text style

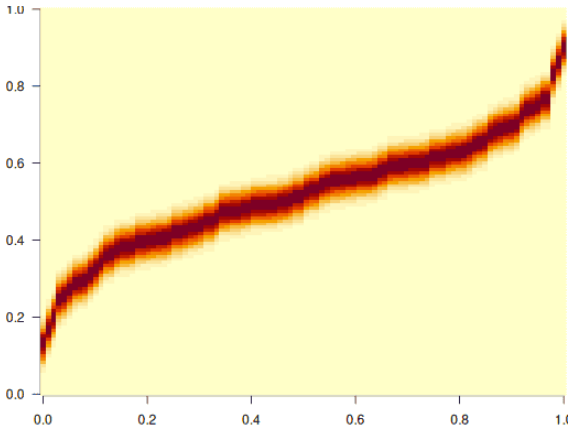
`mtext(side = ..., line = ...)` does exactly that: puts the text in the corresponding margin at the requested line.

Visualising matrices

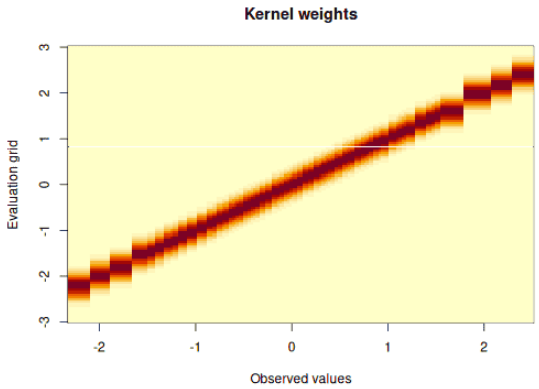
Use `image()` to show matrices as heat maps.

- By default, with 1 argument, colours a matrix
- `image(x, y, z)` adjusts the widths by using x and y as grid point (x and y must be sorted)

```
x <- sort(rnorm(100)); xgrid <- seq(-3, 3, 0.05)
d <- outer(x, xgrid, function(x, y) dnorm(5*(x-y)))
```



`image(d)`

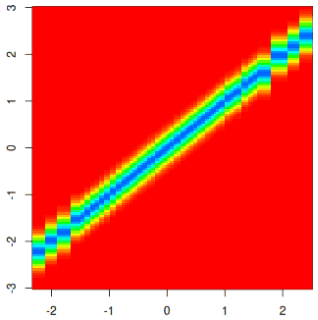


`image(x, xgrid, d)`

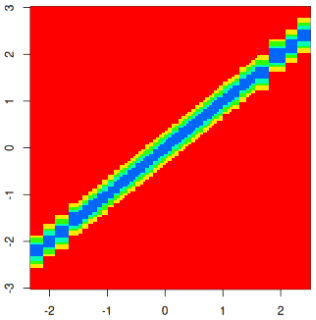
Changing image visuals

Custom palettes can be supplied as colours (smallest value = 1st, largest = last colour, intermediate = uniform ramp). More colours = smoother transitions.

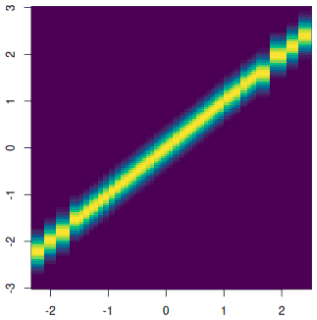
For positive matrices spanning multiple orders of magnitude, `image(log(d))` might look clearer. `image(d^2)` or `image(sqrt(d))` might be more informative about the function shape – try and experiment.



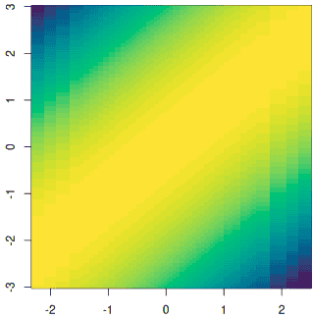
```
image(d, col =
 rainbow(50,
  end = 0.6))
```



```
image(d, col =
 rainbow(5,
  end = 0.6))
```



```
image(d, col =
 hcl.colors(50,
  "viridis"))
```



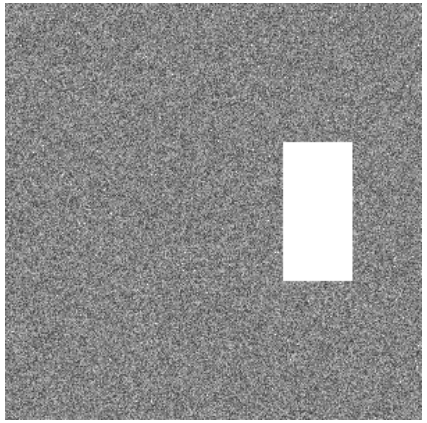
```
image(log(d),
 col =
 hcl.colors(50,
  "viridis"))
```

Pixel-perfect large matrix images

Depending on the number of columns / rows in a matrix, the image will have more or fewer pixels per row / column.

`png::writePNG()` write matrices as images with exactly 1 pixel per element – especially useful for sparse / network matrices. (Clamp the input to [0, 1] first.)

```
set.seed(1)
m <- matrix(rnorm(300^2), 300)
m <- (m - min(m)) / (max(m) - min(m))
m[101:200, 201:250] <- 1
library(png)
writePNG(m, "matrix.png")
```



The input to `writePNG` can be a 3D array (for an RGB image) or 4D array (RGBα).

Too much love for Gaussians

Densities are not enough to visualise data distributions, even with rugs. Often, researchers compare their distributions with the Gaussian.

NB. Normality is not required for most economic analyses:

- Residuals **need not** be normal or homoskedastic
- Normality is an *emerging* property of the output, not input
  - Under general conditions (some finite 4th moments), in repeated experiments on similar large data sets coming from the same DGP, the arg min or arg max of smooth objective functions that depend on the entire sample tends to be approximately normally distributed

Visualising distributions

Visualisations of inputs / residual / variable distribution is useful:

- Detect discontinuities
- Detect anomalies and aberrations

Compare the quantiles of one distribution with the quantiles of a different distribution visually to examing the closeness.

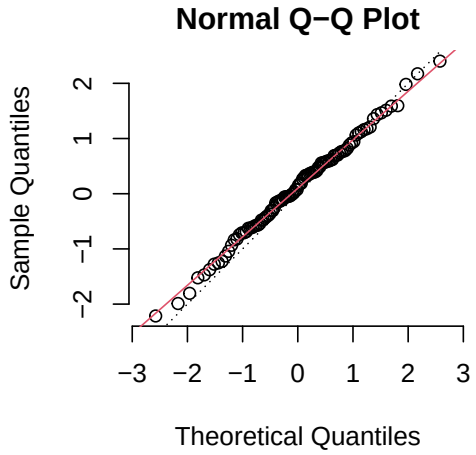
- Usually, something empirical (the observed X_i) are compared with something theoretical
- If $X \sim \mathcal{N}(\mu, \sigma^2)$, and $n = 100$, then, $X_{(1)} \approx Q_{\mathcal{N}(\mu,\sigma^2)}(0.005)$, $X_{(2)} \approx Q_{\mathcal{N}(\mu,\sigma^2)}(0.015)$; in general, $X_{(i)} \approx Q_{\mathcal{N}(\mu,\sigma^2)}(\frac{i-0.5}{n})$

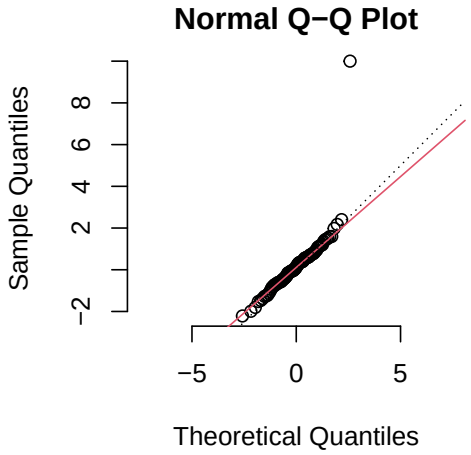
Normal Q-Q plots

`qqnorm(x)` plots the empirical distribution $\{X\}_{i=1}^{n}$ against the quantiles of a Gaussian with $\bar{X}$ and $\hat{\sigma}_X^2$.

- `qqline(x)` adds a straight line through Q_1 and Q_3
- Add a 45° line with `abline(0, 1)`

```
set.seed(1); x <- rnorm(100); xx <- c(x, 10)
```






```
qqnorm(x, asp = 1)
qqline(x, col = 2)
abline(0, 1, lty = 3)
```

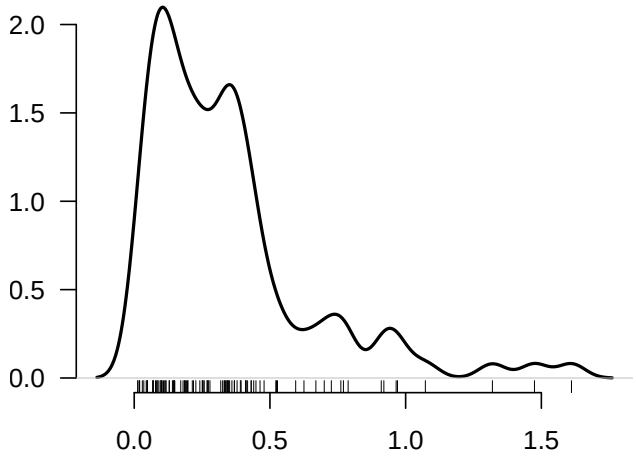
```
qqnorm(xx, asp = 1)
qqline(xx, col = 2)
abline(0, 1, lty = 3)
```

Arbitrary Q-Q plot

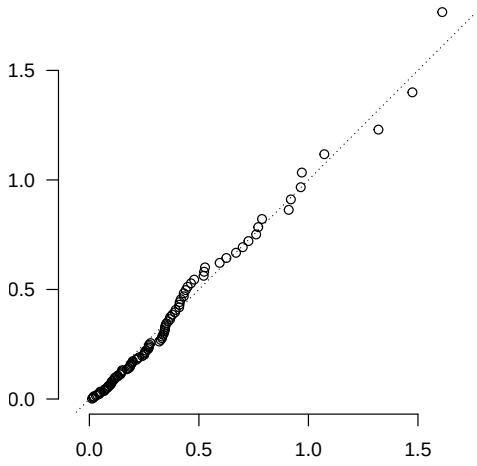
`qqplot()` plots the sorted quantiles of two distributions. Apply the quantile function of the hypothesised distribution to `ppoints(n)` that generates the sequence $\left\{\frac{i-0.5}{n}\right\}_{i=1}^{n}$.

Suppose that $X \sim \exp$ with rate $\lambda = 3$.

```
set.seed(1); x <- rexp(100, rate = 3)
```




```
plot(density(x))
rug(x)
```




```
y <- qexp(ppoints(length(x)), 3)
qqplot(x, y, asp = 1)
abline(0, 1, lty = 3)
```

Densities

The function `density()` computes the kernel density estimator (KDE) of the input variable.

- To change the bandwidth, use `density(x, bw = 0.5)`
- To auto-select the bandwidth with the reliable Sheather–Jones rule, use `density(x, bw = "SJ")`
  - Alternatively, compute `b <- bw.SJ(x)` and `density(x, bw = b)`
- If x contains extreme values, the KDE may look bad
  - Try dropping values near the ends:

```
xl <- quantile(x, c(0.01, 0.99))
xx <- x[x > xl[1] & x < xl[2]]
plot(density(xx))
```

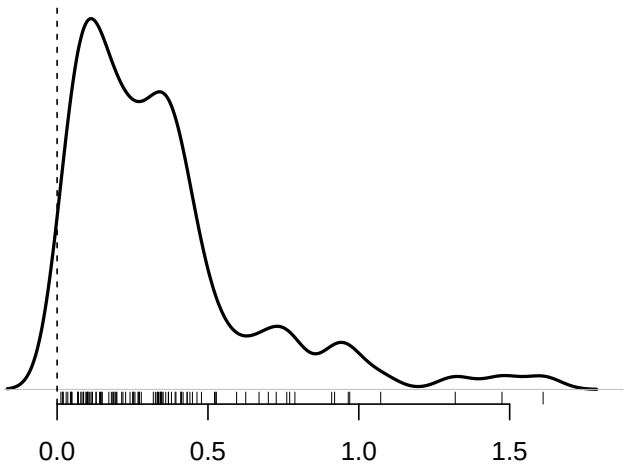
  - `density(x, from = a, to = b)` restricts the range

Densities for bounded variables

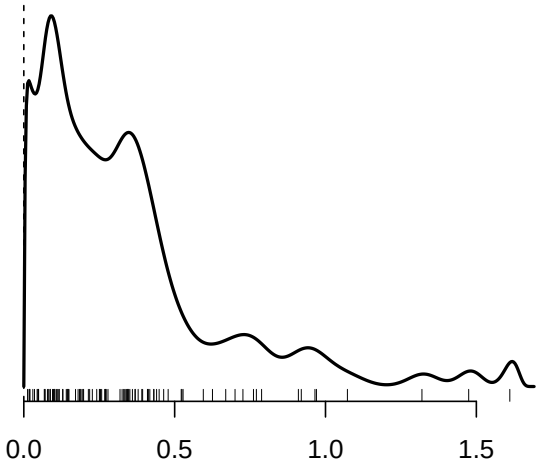
Many economic variables are bounded: wages, population, capital ($X > 0$), which is why negative 'tails' of the density estimator are wrong.

Use the bounded KDE from the `ks` package:

```
bd <- ks::kde.boundary(x, h = bw.SJ(x),
  xmin = 0, xmax = max(x)*1.05)
```




```
plot(density(x, bw = "SJ")); rug(x)
```




```
plot(bd); rug(x)
```

Importance of density tails

Visual comparison of theoretical densities in finance can be hard: the rare events (market crashes) have low probabilities, and linear plots do not show tail behaviour.

Two random variables with mean 0 and variance 1:

1. f_1 = standard normal
2. f_2 = skewed Student with 5 DoF and left tail (the left of the mode) 4 times longer than the right tail

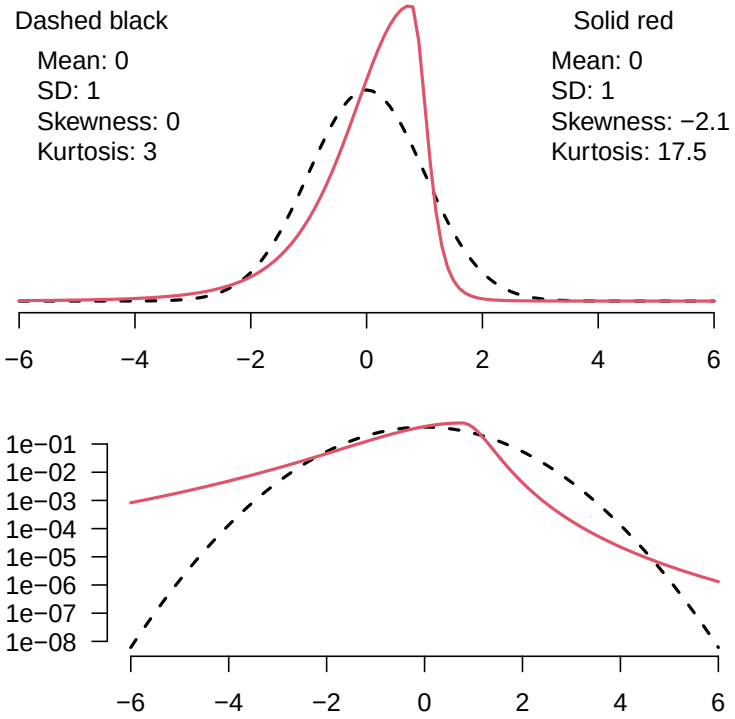
The former decays very quicky (super-exponentially), the latter decays hyperbolically. Can it be seen?

Visualising logarithmic densities

```r
library(rugarch)
f1 <- dnorm
f2 <- \(x) rugarch:::dsstd(x,
  shape = 5, skew = 0.5)

x <- seq(-6, 6, 0.1)
plot(x,  f1(x))
lines(x, f2(x), col = 2)

plot(x,  f1(x), log = "y")
lines(x, f2(x), col = 2)
```

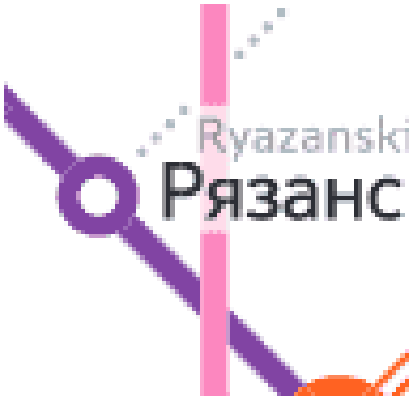


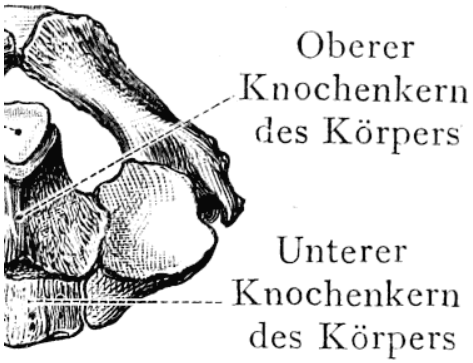
In logarithmic axes, f_1 is concave, f_2 is quasi-concave; the asymmetry and tail heaviness of f_2 is more pronounced.

Text with halo

Before placing text on varicoloured background, create a halo by plotting identical text in a circle.

Halos around text and lines are the standard legibility-improving tool in map-making and atlases.





Credit: Carl Toldt (1900). Anatomischer Atlas: für Studierende und Ärzte.

Goal: create a function that allows a different number of steps, halo colour, and horizontal / vertical spread.

Improved text-halo function

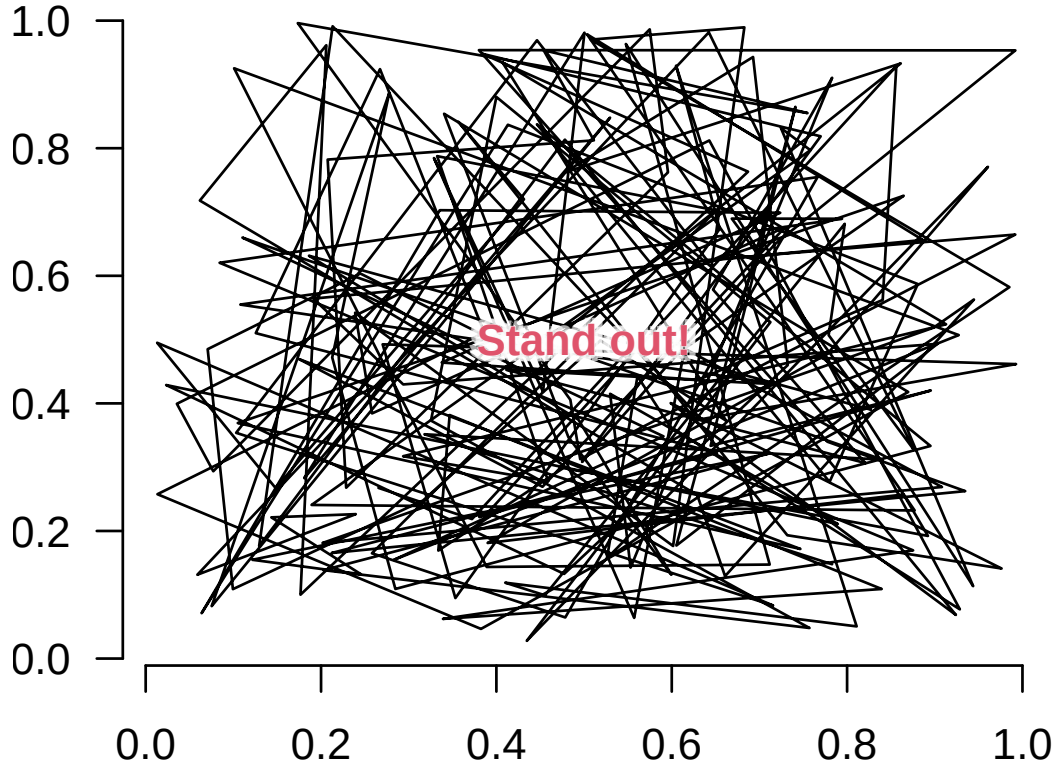
```
textWithHalo <- function(
  x, y, labels, col = "black",     # Passed to text()
  n = 16, col.halo = "#FFFFFFAA", # Halo elements
  hscale = 0.01, vscale = NULL,   # Halo radii (rel.)
  ... ) { # ... are passed to all text()'s
  xlim <- par("usr")[1:2]; ylim <- par("usr")[3:4]
  if (is.null(vscale)) vscale <- hscale
  angls <- seq(0, 2*pi, length.out = n+1)[-(n+1)]
  shifts <- cbind(cos(angls)*hscale*xlim,
                  sin(angls)*vscale*ylim)
  for (i in 1:n) text(x+shifts[i,1], y+shifts[i,2],
    labels = labels, col = col.halo, ...)
  text(x, y, labels = labels, col = col, ...)
}
```

`par("usr")` returns a vector `c(xl, xr, yb, yt)`;

`xlim = c(xl, xr)`, `ylim = c(yb, yt)`.

Text haloing in action

```r
set.seed(1)
plot(runif(200), runif(200), type = "l")
textWithHalo(0.5, 0.5, "Stand out!",
  col = 2, font = 2)
```



Loops for complex plots

If there are multiple elements, data sets, groups to be plotted, do not jump at the problem at once. With 99.9% probability, a function that will output the plot that you need with default settings does not exist.

- Prepare the colours, line types, character types etc. separately
- If there are many similar elements, prepare lists with plottable numbers
- Add those elements onto the picture

Parallel coordinates plot

In experiments, it is sometimes important to visualise 'chains' of characteristics for units, especially is multiple outcomes were measured / multiple tests carried out.

The `survival` package has `survival::lung` data set on survival in patients with advanced lung cancer:

- Survival time in days, status (dead / censored)
- Age and sex (1 = male, 2 = female)
- ECOG performance score (severity of symptoms), Karnofsky performance scores
- Calories consumed and weight loss

Preparations for parallel coordinates

```r
d0 <- survival::lung[, -1]
s <- d0$sex # Extract variable for colouring
d0 <- d0[, -which(colnames(d0) == "sex")]
d <- sapply(d0, scale) # (x - mean) / sd
d[, "status"] <- d0[, "status"] # Restor. 0/1
cs <- c("#1177EEBB", "#EE28C0BB")
cs <- c("#1177EEBB", "#EE28C0BB") # Colours
```

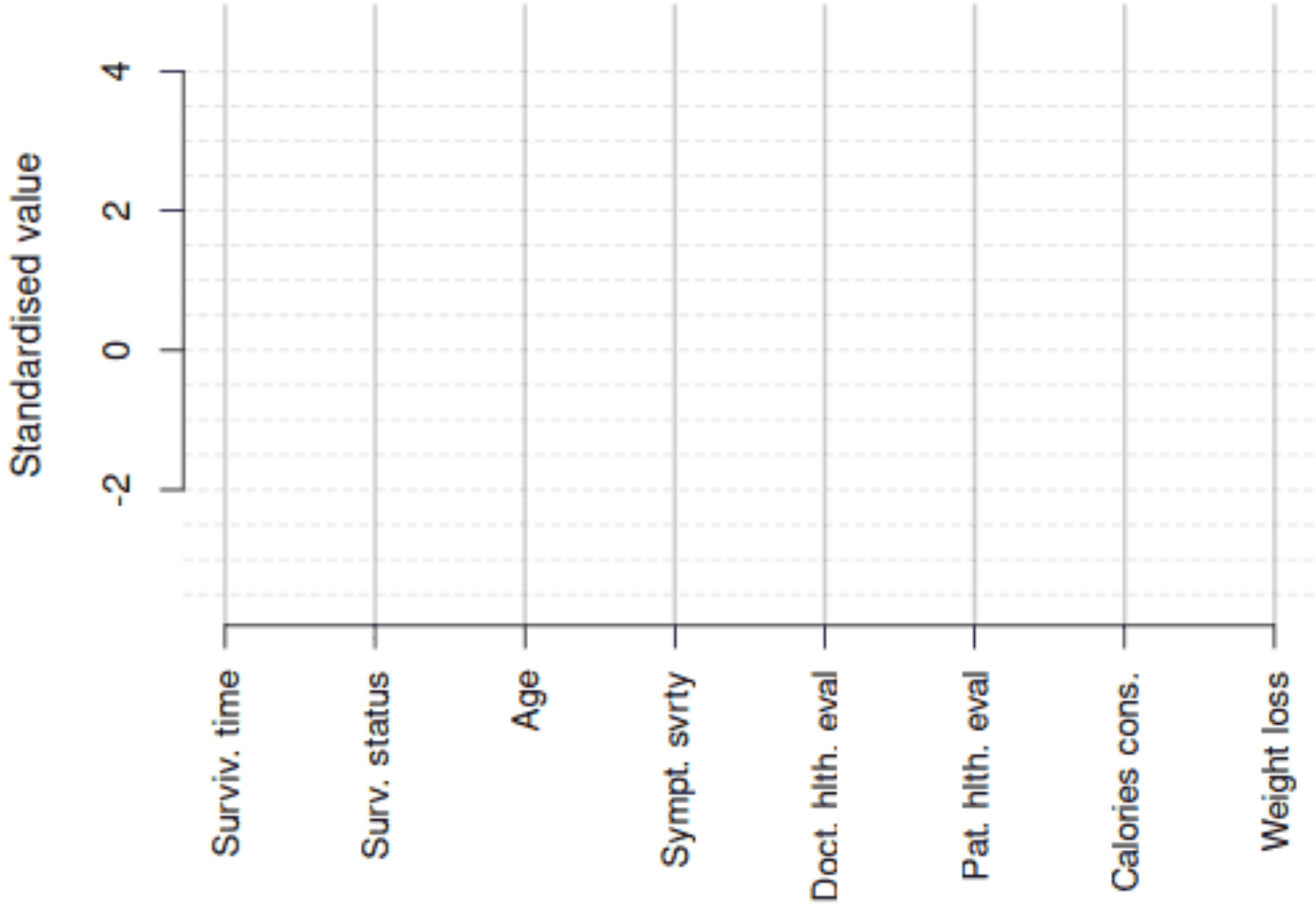
- `scale(x)` returns standardised values: subtracts the mean and divides by the standard deviation
- Since d is a data frame ⇒ a list, `sapply()` calls `lapply()`, computes `scale(x)` for each variable, gets a pure list of vectors as returns, and simplifies the output into a *numeric matrix* (not a data frame)
  - `as.data.frame(lapply(d, scale))` would convert the output list back into a data frame

Paral. coord. solution 1: loops (1/2)

Prepare the canvas and human-readable labels based on the data ranges:

```
par(mar = c(6.5, 4, 0.5, 0.5)) # More space below
yl <- range(d, na.rm = TRUE)
plot(NULL, NULL, bty = "n", xaxt = "n", xlab = "",
  xlim = c(1, ncol(d)), ylim = yl + c(-0.2, 0.2),
  ylab = "Standardised value")
labs <- c("Surviv. time", "Surv. status", "Age",
  "Sympt. svrty", "Doct. hlth. eval", "Pat. hlth. eval",
  "Calories cons.", "Weight loss")
axis(1, 1:ncol(d), labels=labs, cex.axis=0.95, las=2)
abline(v = 1:ncol(d), col = "#00000033", lwd = 2)
abline(h = seq(-4, 4, 0.5), col = "#00000022", lty = 2)
```

Paral. coord. solution 1: loops (1/2)



We are now ready to add lines for every patient.

Paral. coord. solution 1: loops (2/2)

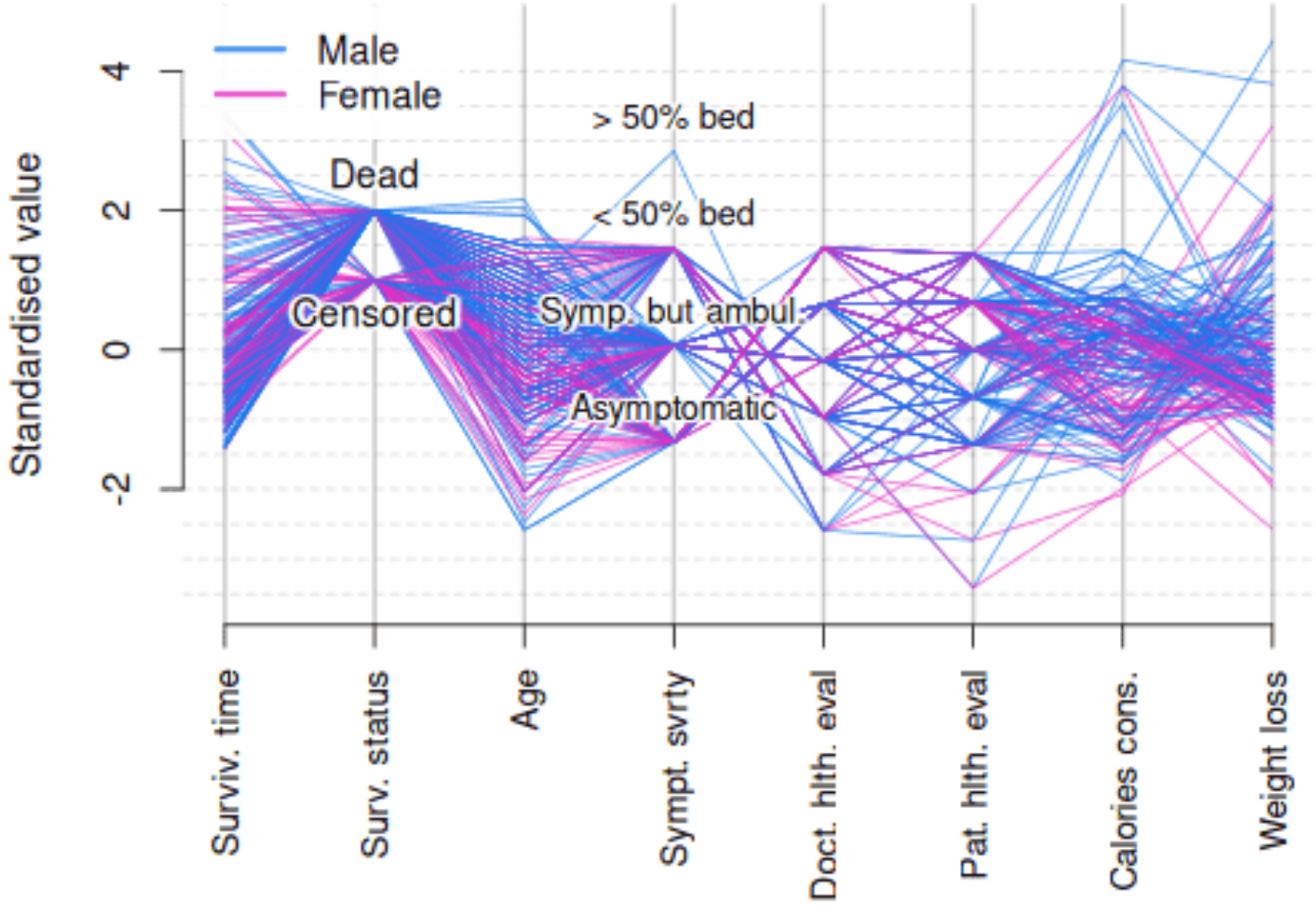
The full plot is produced by just 1 loop over rows:

```
for (i in 1:nrow(d)) lines(d[i, ], col = cs[s[i]])
```

Add the legend and clarifying overlay text:

```
legend("topleft", c("Male", "Female"), lwd = 3,
  col = cs, bg = "#FFFFFFEE", box.col = NA)
l2 <- c("Censored", "Dead")
l4 <- c("Asymptomatic", "Symp. but ambul.",
        "< 50% bed", "> 50% bed")
textWithHalo(rep(2, 2), sort(unique(d[, "status"])),
  labels = l2, pos = c(1, 3), col.halo = "#FFFFFFCC",
  hscale = 0.005, vscale = 0.01)
textWithHalo(rep(4, 2), sort(unique(d[, "ph.ecog"])),
  labels = l4, cex = 0.9, pos=3, col.halo="#FFFFFFCC",
  hscale = 0.005, vscale = 0.01)
```

Paral. coord. solution 1: loops (2/2)



In some fields, such plots are common.

Matrix plots

Writing loops for each line is tedious! Can we plot matrices without loops?

`matplot(x, ...)` plots multiple series for each column of matrix x. It accepts the same styling arguments as `plot()`.

Paral. coord. solution 2: matrix plot

Transpose d and use one command:

```
# Same preparations: we need d and s
matplot(t(d),
  ylim = range(d, na.rm = T) + c(-0.2, 0.2),
  type = "l", lty = 1, col = cs[s],
  bty = "n", xaxt = "n", xlab = "",
  ylab = "Standardised value")
# Same commands for axes, labels, text etc.
```

The plot looks identical to the one from Slide 137!

Plotting multiple stock indices

Plots of multiple times series in the same picture are very popular. The main obstacle to plotting is wildly different value ranges. Solutions:

- Clamp to $[0, 1]$ via $\frac{x - \min x}{\max x - \min x}$
- `scale` by mean and SD
- `scale` by robust versions of mean and SD
  - `median()` and `mean(x, trim = 0.25)` are more robust location measures
  - `mad()` and `IQR()` are more robust dispersion measures

Preparations for multiple stock prices

To prepare multiple stock prices for plotting:

- Download the (adjusted) prices for the selected tickers for the desired date range
- Create the union of all dates (some values may be missing) of length *n*
- Create an `NA` matrix with *n* rows and columns corresponding to the tickers
- For the dates available for each ticker, fill the matching indices with the price values

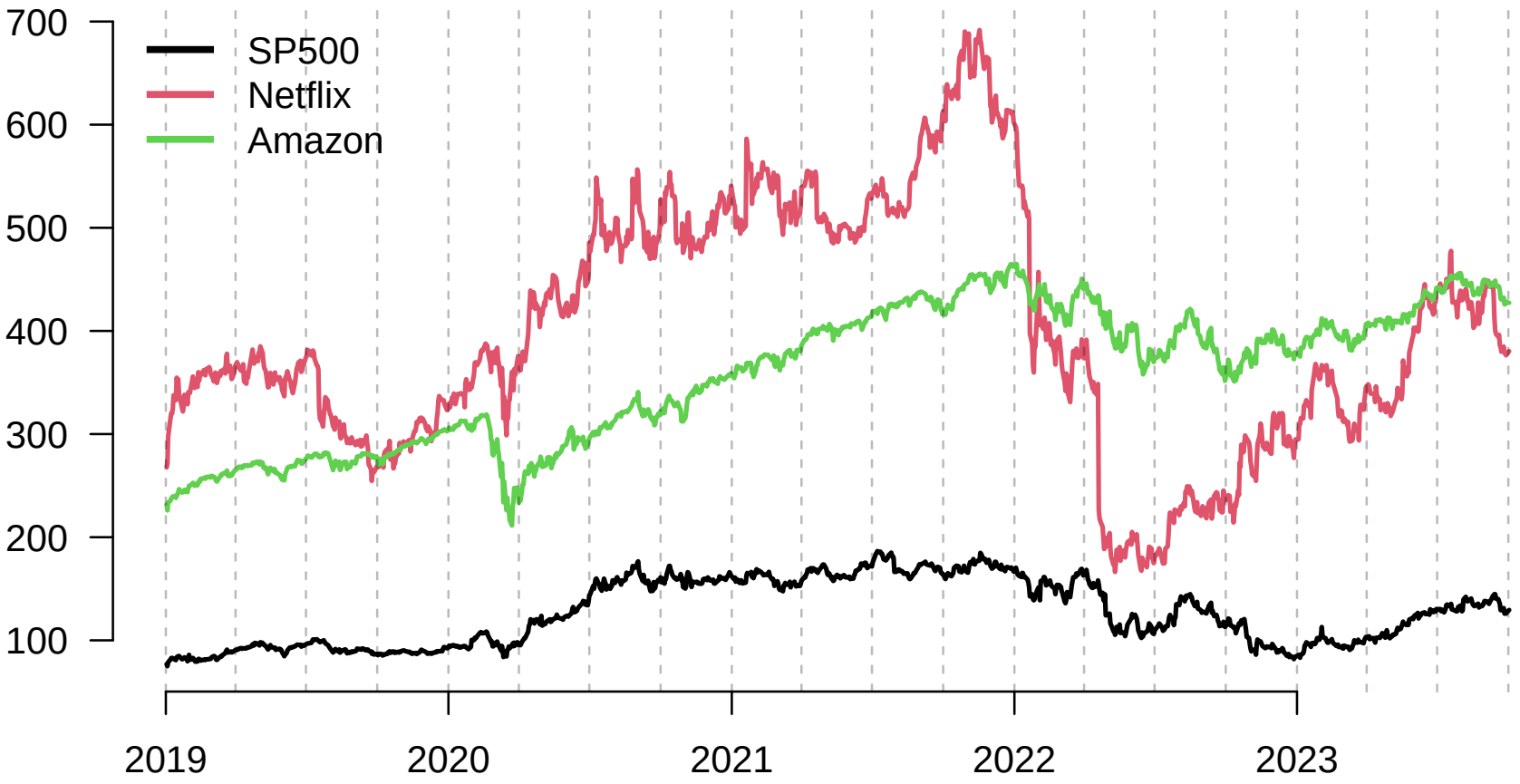
Fetching and plotting tickers

The dates are equal for all stocks – we combine the vectors. Then, all the duty of plotting is done by `matplot()`:

```
library(tidyquant)  # tq_get returns tibbles
tckr <- c("SPY", "NFLX", "AMZN") # Company tickers
d <- as.data.frame(tq_get(tckr, from = "2019-01-01"))
dl <- split(d, d$symbol)
m  <- do.call(cbind, lapply(dl, "[[", "adjusted"))

matplot(dl[[1]]$date, m, type = "l", lty = 1,
  bty = "n", lwd = 2, ylab = "", xlab = "")
abline(v = seq(as.Date("2019-01-01"),
    as.Date("2024-01-01"), by = "3 months"),
  lty = 2, col = "#00000044")
legend("topleft", border.col = "#FFFFFF00", col = 1:3,
  legend = c("SP500", "Netflix", "Amazon"), lwd = 3)
```

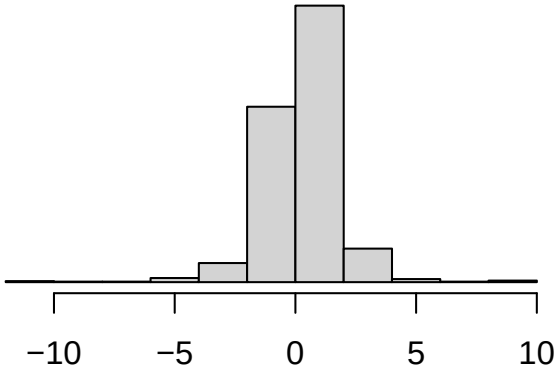
Stock price plot – unscaled



Histograms

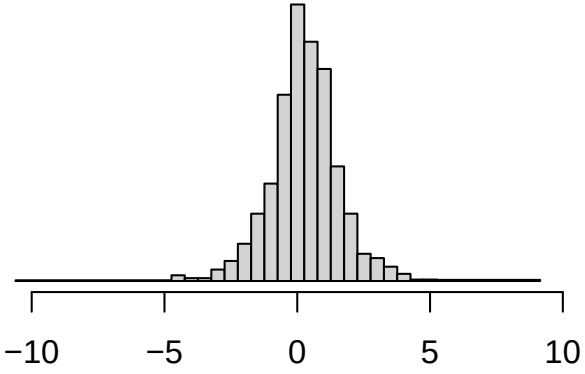
Produce histograms with `hist()`:

```
set.seed(1)
r <- rt(1000, df = 4) + 0.2
hist(r)
```



This distribution should be symmetric around 0.2, but the default breaks make it look skewed. Customise breaks by supplying their exact locations. It is a good idea to centre a histogram around the median.

```
hist(r, breaks = c(min(r),
  (-10:10)/2 + median(r),
  max(r)))
```



Tweaking histograms

- To add counts, use `labels = TRUE`
- To change the bar fill colour, use `col`
  - Set `col = "#00000000"` for transparent bars
- To change the lines colour, use `border`
- Standard `xlim`, `ylim`, `bty` etc. apply

Quirks of histograms

NB. Using `breaks = n` with an integer n does not return the expected result (*n* breaks) because *n* is used merely a 'suggestion' to split the range at 'pretty' (round, or multiples of 2 an 5) values, which cannot be honoured for all *n*.

```
hist(r, breaks = 12)  # Observe no change:
hist(r, breaks = 14)  # Only 12 breaks
```

NB. Border width cannot be changed: `border` is the colour, `lwd` changes *axis* thickness. This is due to the implementation of rectangles in `graphics:::plot.histogram()`:

```
rect(x$breaks[-nB], 0, x$breaks[-1L], y, col = col,
  border = border, angle = angle, density = density, lty = lty)
```

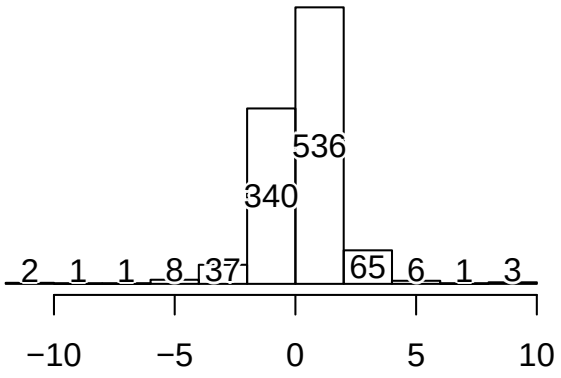
Useful return from hist()

`hist()` does not only plot the data. It returns the information used to construct the plot:

```
h <- hist(r, col = "#00000000")
str(h)  # List of 6:
#> $ breaks  : num [1:12] -12 -10 -8 -6 -4 -2 ...
#> $ counts  : int [1:11] 2 1 1 8 37 340 536  ...
#> $ density : num [1:11] 0.001 0.0005 0.0005 ...
#> $ mids    : num [1:11] -11 -9 -7 -5 -3 -1 1 ...
```

Customise the text position:

```
textWithHalo(x = h$mids,
  y = pmax(max(h$counts)*0.05,
           h$counts/2),
  as.character(h$counts),
  col.halo = "#FFFFFFEE", n = 12)
```



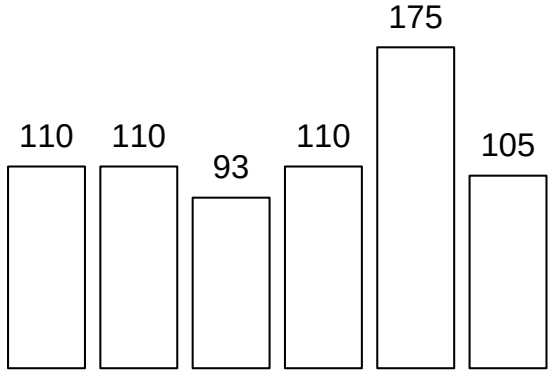
Bar plot

A bar plot is an alternative to scatter plots at regular intervals to compare several numbers.

- Vector names are used as labels
  - Use either labels or an axis with a faint grid
- Like `hist()`, it returns something – the bar midpoints
- `plot()` customisation arguments apply

This information can be used to add labels:

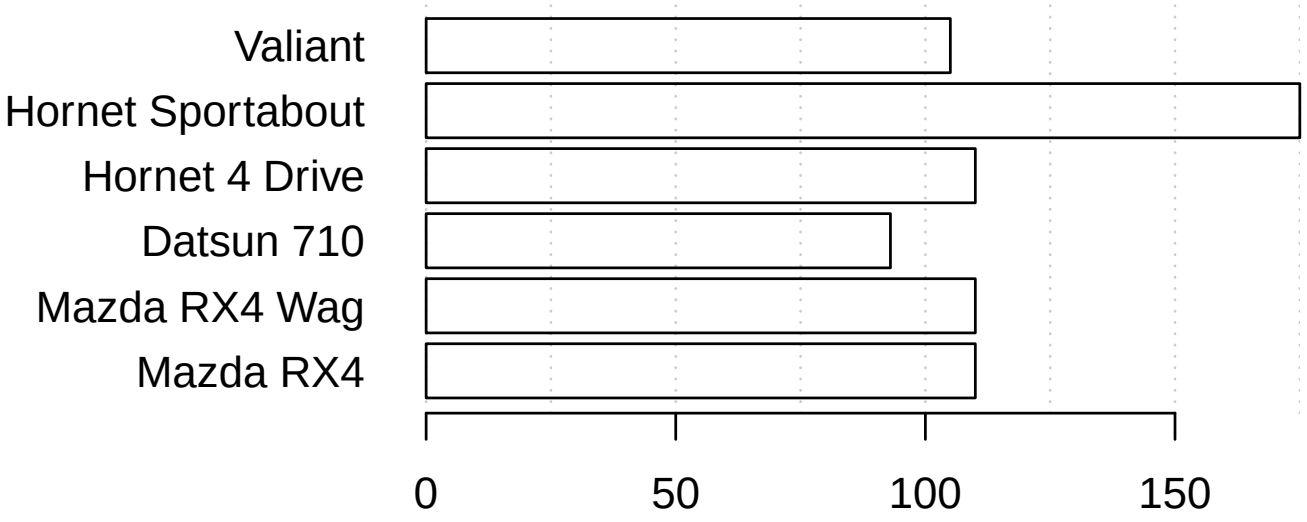
```
y <- mtcars$hp[1:6]
b <- barplot(y, col = "#00000000",
  ylim = c(0, max(y)*1.1))
text(b, y, as.character(y), pos=3)
```



Bar plot orientation

If the names are long, use `horiz = TRUE` to draw the bars horizontally; turn the labels with `las` to avoid neck injuries:

```
names(y) <- rownames(mtcars)[1:6]
par(mar = c(2, 8, 0, 0)+.1) # Wide left margin
barplot(y, col = "#00000000", horiz = T, las = 1)
abline(v=seq(0,175,25), lty=3, col="#00000033")
```

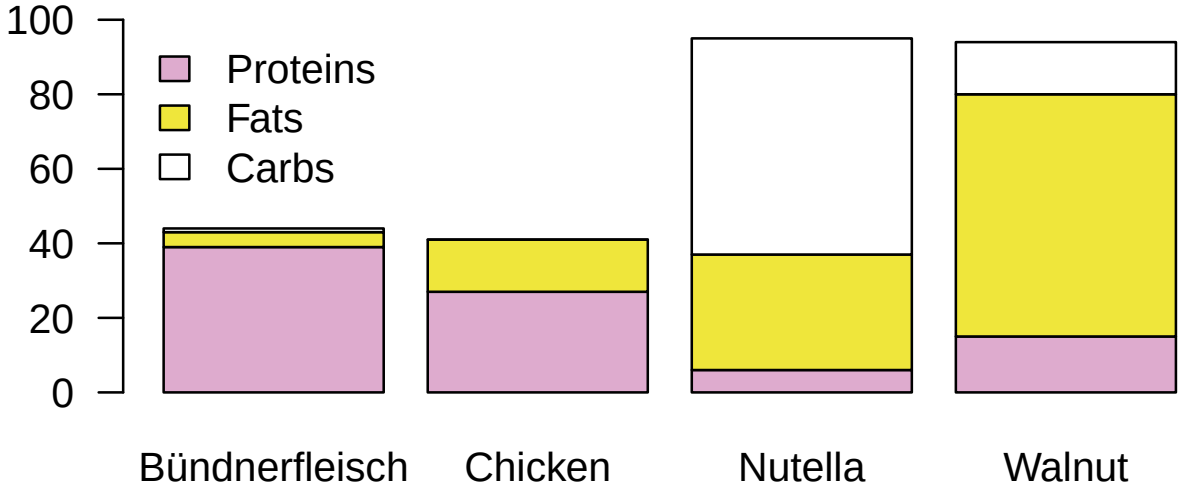


Stacked bar plots

For matrix inputs, `barplot` returns stacked bars (the lengths are given by matrix columns).

```
m <- matrix(c(39,4,1,27,14,0,6,31,58,15,65,14), ncol = 4)
l <- c("Proteins", "Fats", "Carbs")
dimnames(m)<-list(l,c("Bündnerfleisch","Chicken","Nutella","Walnut"))
```

```
cs <- c("#deabce", "#efe63b", "#FFFFFF")
barplot(m, col = cs, ylim = c(0, 100), las = 1)
legend("topleft", l, fill = cs, bty = "n")
```

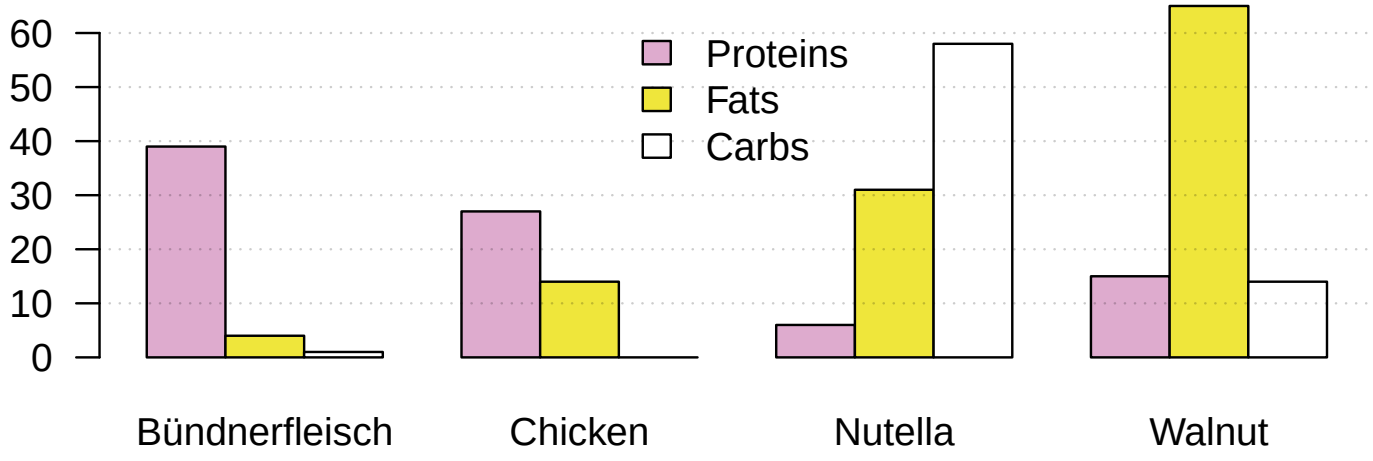


Grouped bar plots

For 'unstacked' grouped bar plots, use `beside = TRUE`.

NB. Wide bar plots with many columns are hard to read without horizontal guides.

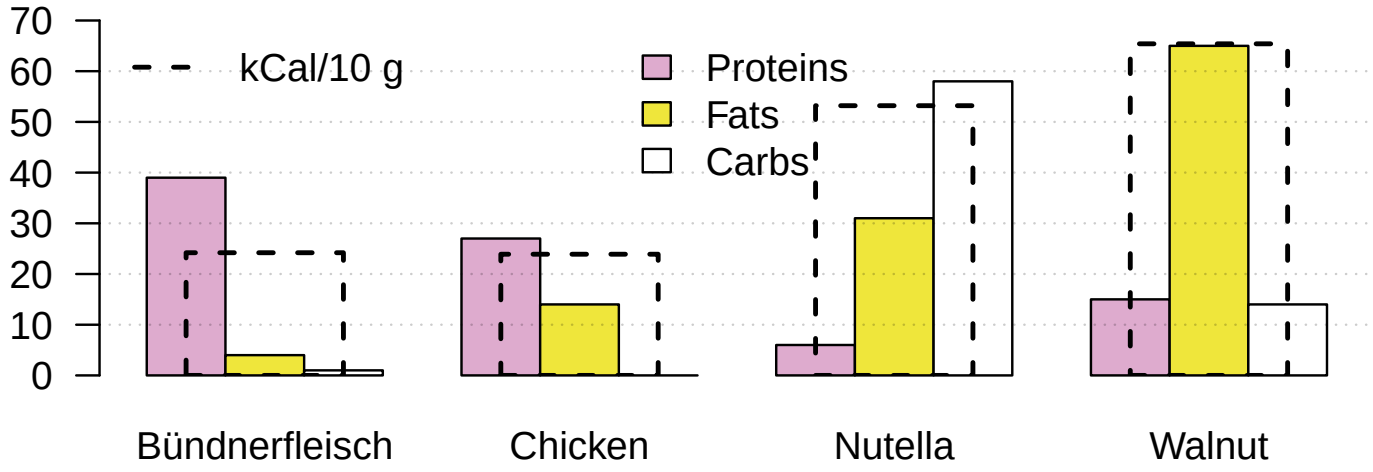
```
barplot(m, col = cs, beside = TRUE, las = 1)
abline(h = (1:6)*10, lty=3, col = "#00000033")
legend("top", rownames(m), fill = cs, bty = "n")
```



Rectangles

Use `rect()` to draw arbitrary rectangles. 4 inputs are needed: bottom left *x*, bottom left *y*, top right *x*, top right *y*. `border` sets the border colour, `col` sets the filling colour.

```
# <... same as the previous plot ...>
cal <- c(242, 239, 532, 654)
rect(b[1, ], rep(0, 4) , b[3, ], cal/10,
  lty = 2, lwd = 2)
legend("topleft", "kCal/10 g", lty=2, lwd=2, bty="n")
```

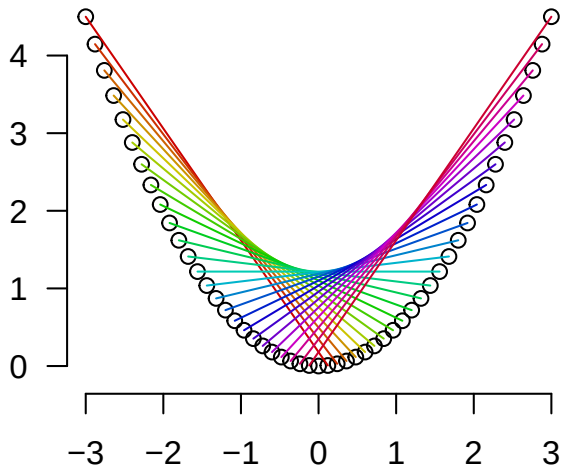


Line segments

To plot multiple straight lines from a sequence of starting points to a sequence of end points, use `segments()`.
4 inputs are needed: start *x*, start *y*, end *x*, end *y*.

Conceptually, `segments()` draws the *diagonals* of the rectangles that would have been drawn by `rect()`.

```
x <- seq(-3, 3, length.out = 51)
y <- x^2/2
plot(x, y, asp = 1)
ii <- 1:(length(x)-26)
segments(x[ii], y[ii],
         x[ii+26], y[ii+26],
  col = rainbow(25, v = 0.8))
```



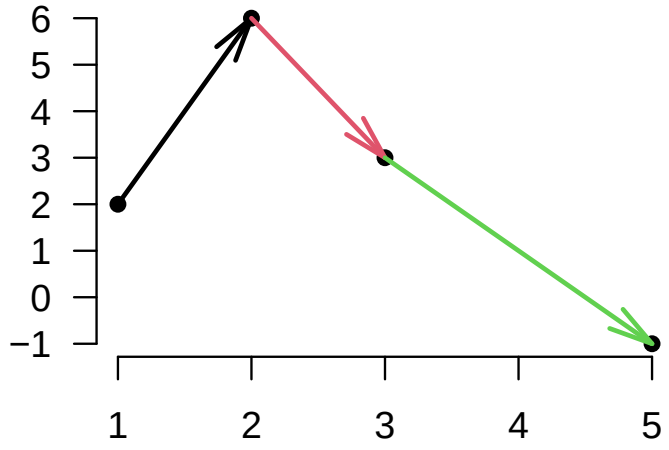
Arrows

`arrows()` takes 4 inputs: start *x*, start *y*, end *x*, end *y* – and draws the same as `segments` plus added arrow heads.

- `code = 1`, `2`, or `3` define if all arrows are facing forwards, backwards, or both ways
- `angle` and `length` define the style of all arrows

NB. `code`, `angle`, and `length` are not vectorised.

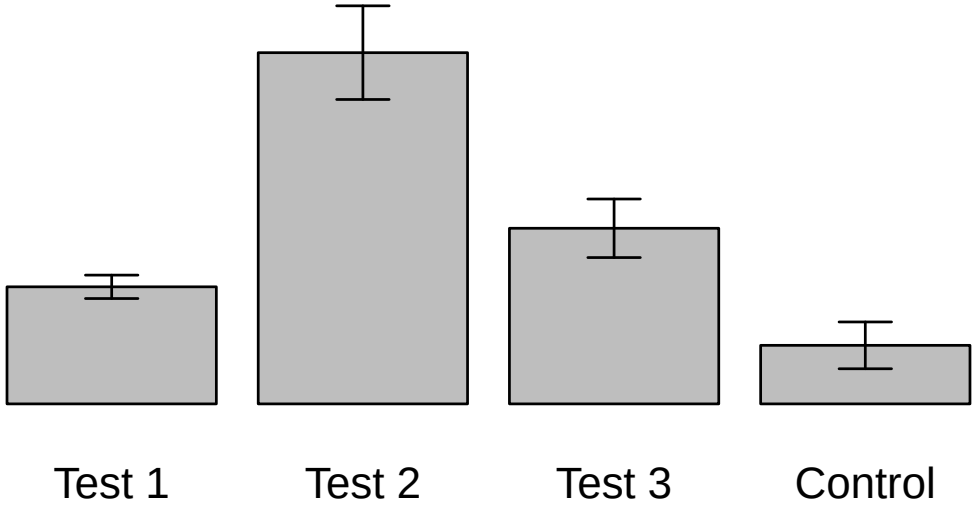
```
x <- c(1, 2, 3, 5)
y <- c(2, 6, 3, -1)
myPlot(x, y, pch = 16)
arrows(x[1:3], y[1:3],
       x[2:4], y[2:4],
  col = 1:3, lwd = 2,
  angle = 15, length = 0.2)
```



Error bars

`angle = 90` straightens the arrows, `code = 3` makes them two-ended, like a capital I. Add error bars in this manner:

```
y <- c(2, 6, 3, 1)
names(y) <- c("Test 1", "Test 2", "Test 3", "Control")
se <- c(0.2, 0.8, 0.5, 0.4)
b <- barplot(y, pch=16, ylim=c(0, 7), yaxt="n")[, 1]
arrows(b, y-se, b, y+se, angle=90, code=3, length=0.1)
```



Arrange multiple plots

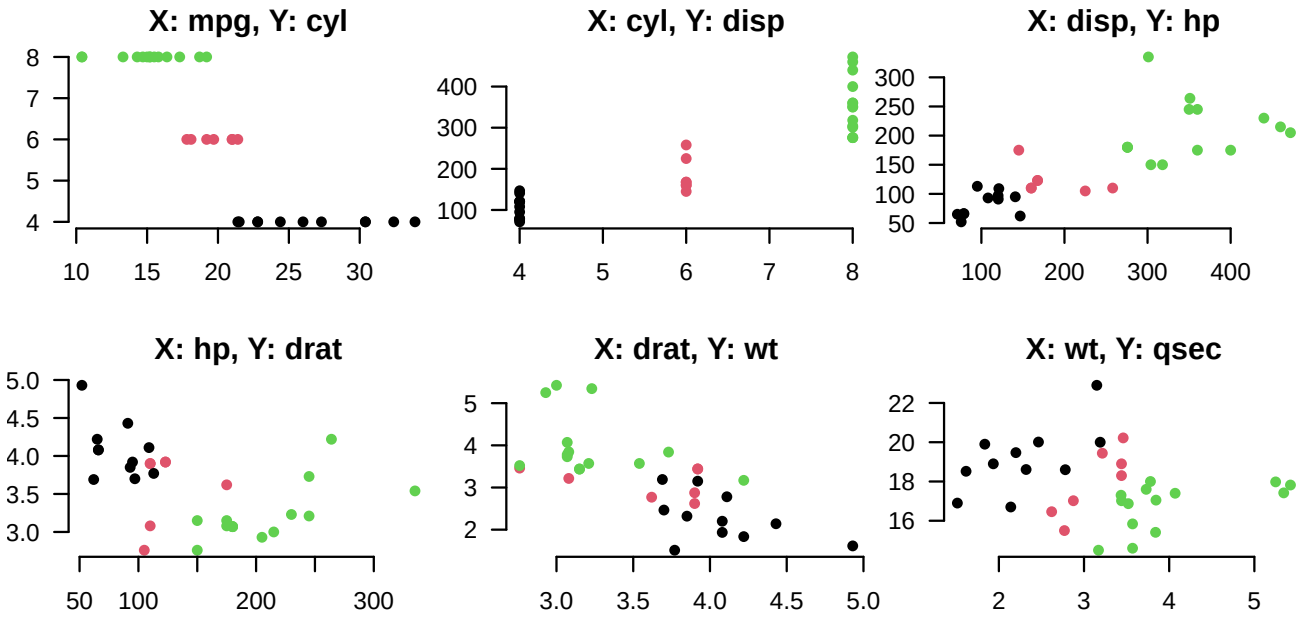
It may be inconvenient to produce and arrange multiple images. Prepare several plots as a matrix with `par(mfrow = c(nRows, nColumns))`.

- When outputting to a device, open the device, call `par(mfrow = ...)`, and then, call the plots
- Each call of `plot()` or anything that calls it (e. g. `barplot()`, `hist()` but not `points()`) starts a new plot in a section of the output device
- Other `par()` options can be changed between plots
- Change back to `par(mfrow = c(1, 1))` after plotting or call `dev.off()` (in interactive mode)

Hint: empty panels remaining? Place the legends there.

Custom matrix plot layout from single PDF

```r
pdf("mfrow.pdf", 6, 3)
par(mfrow = c(2, 3))
vn <- colnames(mtcars)
for (i in 1:6) plot(mtcars[, i:(i+1)],
  col = factor(mtcars$cyl), pch = 16,
  main = paste0("X: ", vn[i], ", Y: ", vn[i+1]))
dev.off()
```



Merging cells in plot matrices

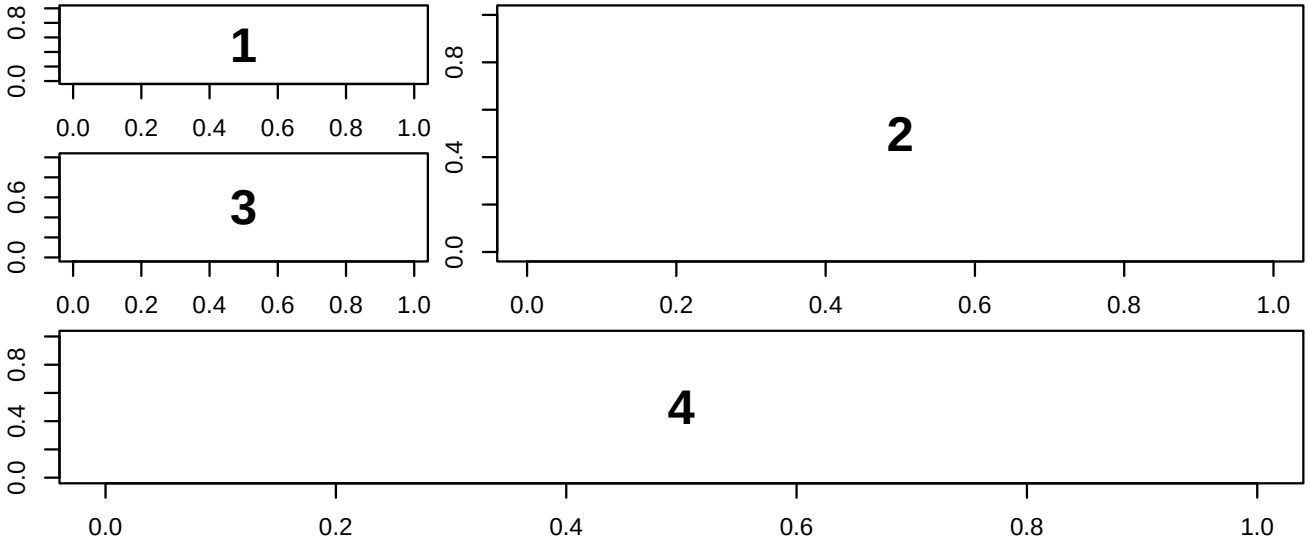
Drawbacks of `par(mfrow = ...)`:

- Fixed equal plot sizes
- Strict grid placement

It is possible to address these issues with `layout()` by defining a table with merged cells and custom row/column widths and heights via a matrix. Use 0 in the matrix to skip layout squares.

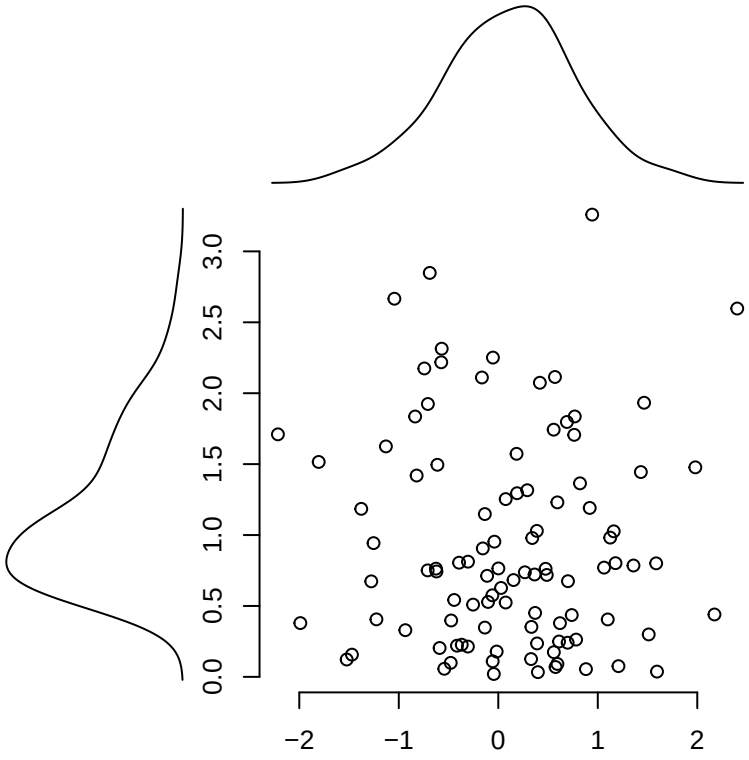
Custom layout example

```r
pdf("s05-layout-1.pdf", 6, 3)
layout(matrix(c(1, 3, 4, 2, 2, 4), ncol = 2),
  widths = c(1, 2), heights = c(1, 1.2, 1.5))
for (i in 1:4) {
  plot(NULL, NULL, xlim = 0:1, ylim = 0:1)
  text(0.5, 0.5, i, cex = 2, font = 2)
}
dev.off()
```

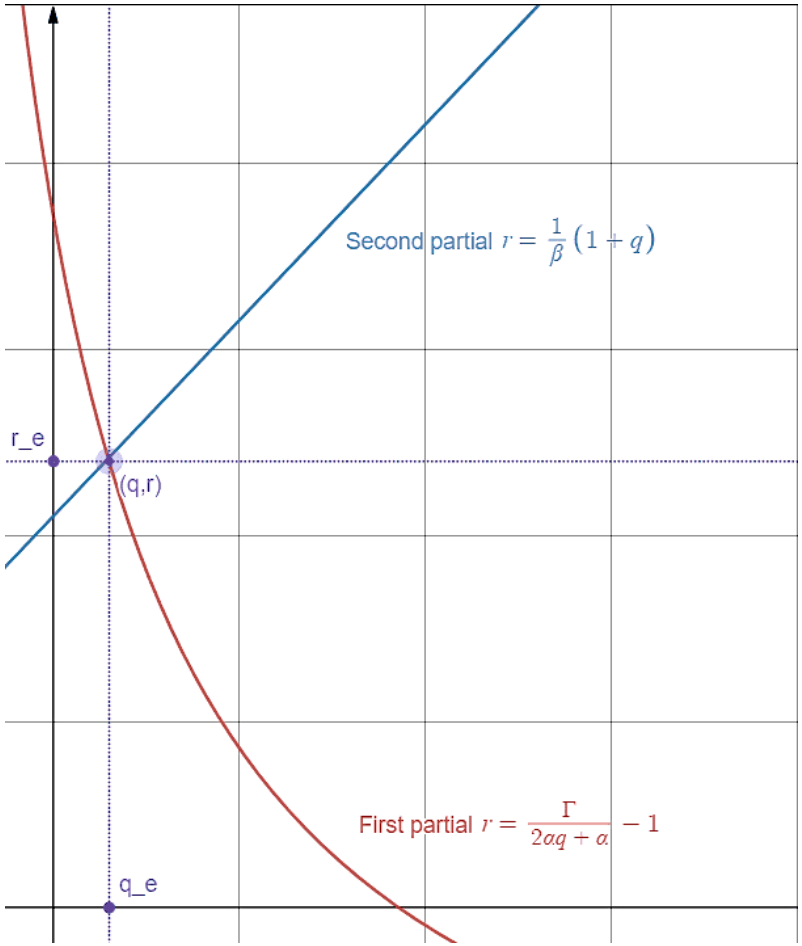


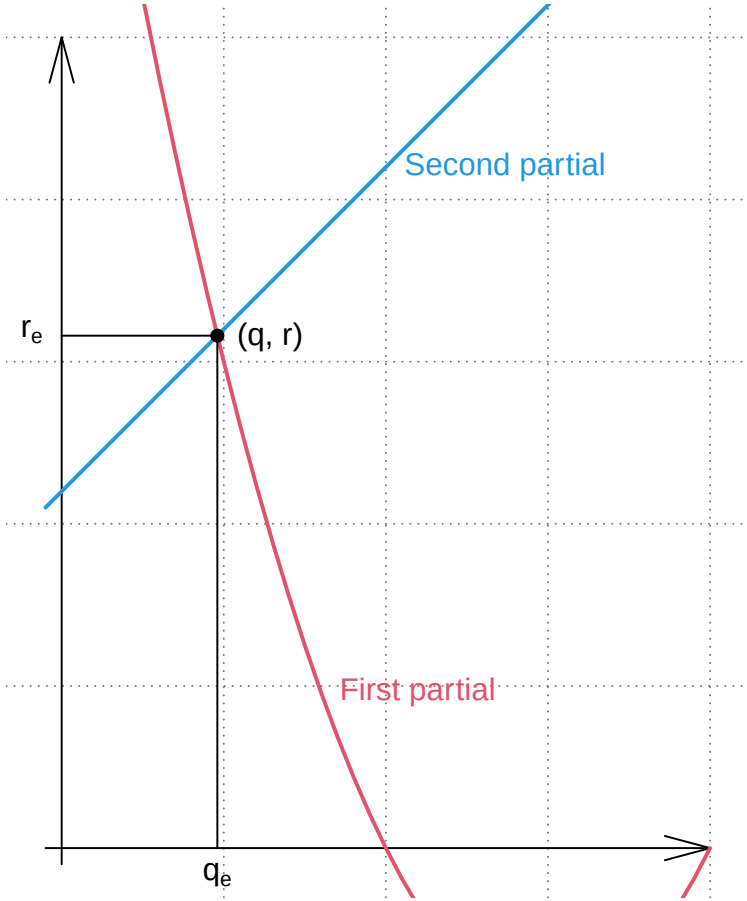
Densities in margins with layout()

```r
set.seed(1)
x <- rnorm(100)
y <- rexp(100)
dx <- density(x)
dy <- density(y)
layout(matrix(c(0, 3, 2, 1),
  ncol = 2), widths = c(1, 3),
  heights = c(1, 3))
par(mar = c(2,2,0,0)+.1)
plot(x, y, bty = "n")
par(mar = c(0,2,0,0)+.1)
plot(dx$x, dx$y, type = "l")
par(mar = c(2,0,0,0)+.1)
plot(-dy$y, dy$x, type = "l")
```



Live demonstration: reproducing a plot





Reproducing code

```
pdf("s05-reproduce.pdf", 4.2, 5)
par(mar = c(1, 1, 0, 0)+.1)
plot(NULL, NULL, xlim = c(-0.1, 4), ylim = c(-0.1, 5), asp = 1,
     bty = "n", xlab = "", ylab = "", yaxt = "n", xaxt = "n")
abline(v = 1:4, h = 1:5, lty = 3, col = "#00000088")
arrows(x0 = c(-0.1, 0), y0 = c(0, -.1),
       x1 = c(4, 0),    y1 = c(0, 5), angle = 15)
f1 <- function(x) 2.2 + 1*x
f2 <- function(x) (x-3)^2-1
xgrid <- seq(-.1, 4, .1)
lines(xgrid, f1(xgrid), lwd = 2, col = 4)
lines(xgrid, f2(xgrid), lwd = 2, col = 2)
# Finding the intersection
sol <- uniroot(\(x) f1(x) - f2(x), interval = c(0, 2))
p <- c(sol$root, f1(sol$root))  # Intersection point
lines(c(p[1], p[1]), c(0, p[2]))
lines(c(0, p[1]), c(p[2], p[2]))
text(p[1], 0, expression(q[e]), pos = 1)
text(0, p[2], expression(r[e]), pos = 2)
text(p[1], p[2], "(q, r)", adj = c(-0.5, 1))
text(1.6, f2(1.6), "First partial", col = 2, pos = 4)
text(2, f1(2), "Second partial", col = 4, pos = 4)
points(p[1], p[2], pch  = 16)
dev.off()
```

Any questions on the common plot types?

Summarising and aggregating data

aggregate: apply a function by group

To compute a statistic by group, one can invoke `aggregate()` in one of the two ways:

```
aggregate(formula, data = d, FUN = myFun)
aggregate(d$var1, by = d$var2, FUN = myFun)
```

The first relies on the formula interface (we shall apply formulæ in Session 7). The second one simply specifies the variable(s) to aggregate, the *list* of identifiers to aggregate by, and the function.

One may get by without `dplyr` or `data.table` most of the time because `aggregate()` is so powerful!

One variable, one aggregate statistic

Compute average fuel efficiency by cylinder count.

NB. A single grouping variable must be wrapped into a list.

```
d <- mtcars
a1 <- aggregate(mpg ~ cyl, d, mean)
a2 <- aggregate(d$mpg, list(d$cyl), mean)
aggregate(d$mpg, d$cyl, mean)  # Error
```

The returned data frames a1 and a2 are almost identical; the formula interface preserves names, whereas aggregating an unnamed isolated vector by a list resulted in this:

```
colnames(a1)  # cyl, mpg
colnames(a2)  # Group.1, x
```

Multiple group variables

- Interact the variables in the formula by '+'
- Use `list(var1, var2, ...)` to compute the statistics by every available combination of `var1`, `var2`, ...

Data frames of grouping variables are lists:

```
d <- mtcars
aggregate(mpg ~ cyl + am, mtcars, mean)
aggregate(d$mpg, d[, c("cyl", "am")], mean)
```

```
#> cyl am      mpg
#>   4  0 22.90000
#>   6  0 19.12500
#>   8  0 15.05000
#>   4  1 28.07500
#>   6  1 20.56667
#>   8  1 15.40000
```

Multiple variables, one aggregated statistic

In the formula, write `cbind()` on the left-hand side to compute the same statistic for many variables.

```
aggregate(cbind(mpg, hp) ~ cyl + am,
          mtcars, mean)
aggregate(d[, c("mpg", "hp")],
          d[, c("cyl", "am")], mean)
```

```
#> cyl am      mpg        hp
#>   4  0 22.90000  84.66667
#>   6  0 19.12500 115.25000
#>   8  0 15.05000 194.16667
#>   4  1 28.07500  81.87500
#>   6  1 20.56667 131.66667
#>   8  1 15.40000 299.50000
```

One variable, multiple aggregate statistics

```
meanSD <- \(x) c(mean = mean(x), sd = sd(x))
a1 <- aggregate(mpg ~ cyl, mtcars, meanSD)
a2 <- aggregate(d$mpg, list(d$cyl), meanSD)
str(a1)
```

NB. If the function returns a vector of statistics, the output looks a bit inconvenient – a DF / list with 2 elements!

1. Vector: grouping variable
2. Matrix: all statistics

Fun fact: R supports DFs of DFs, matrices of lists etc.

Convert the result into a non-nested DF for usability:

```
a1 <- data.frame(cyl = a1$cyl, a1$mpg)
a2 <- data.frame(cyl = a2[, 1], a2[, 2])
```

Multiple inputs, multiple aggregate outputs

Nesting for generality: both the names of `FUN` returns and input variable names must be preserved.

```
a1 <- aggregate(cbind(mpg, hp) ~ cyl + am,
                mtcars, meanSD)
a2 <- aggregate(d[, c("mpg", "hp")],
                d[, c("cyl", "am")], meanSD)
```

Check `str(a1)` and un-nest:

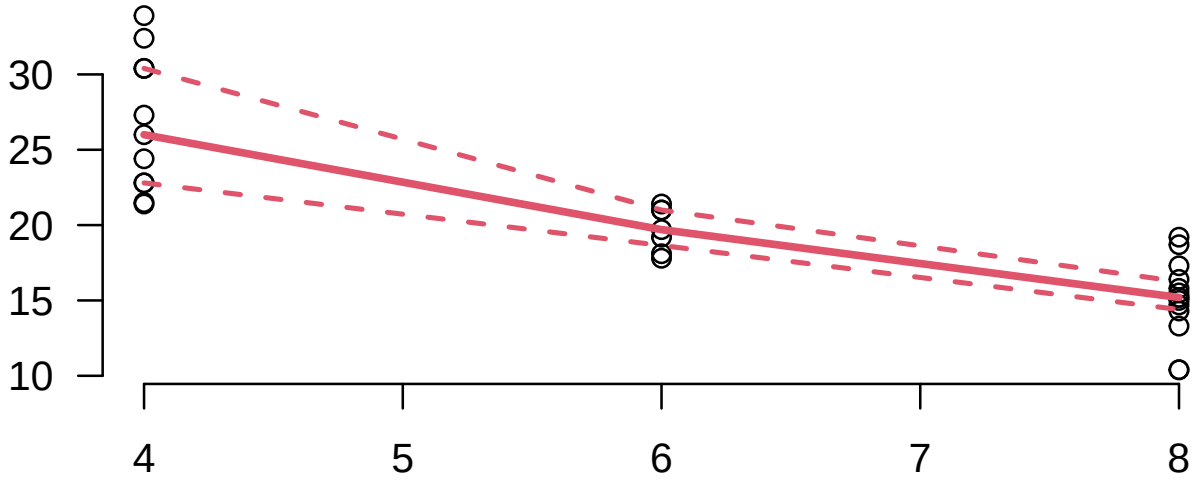
```
a1 <- data.frame(a1[, c("cyl", "am")],
                 mpg = a1$mpg, hp = a1$hp)
a2 <- data.frame(a2[, 1:2],
                 mpg = a2[, 3], hp = a2[, 4])
all.equal(a1, a2)  # TRUE
```

For full automation, use indexing based on the input and output lengths; create names by pasting with ".

Scatterplot with ranges

Plot quartiles of fuel efficiency by cylinder count:

```
s <- function(x) quantile(x, c(0.25, 0.5, 0.75))
a <- aggregate(mpg~cyl, data = mtcars, FUN = s)
a <- data.frame(cyl = a$cyl, a$mpg)
myPlot(mtcars$cyl, mtcars$mpg)
for (i in 2:4) lines(a[, 1], a[, i],
  lwd = 2+(i==3), lty = 2-(i==3), col = 2)
```



Tables

`table()` counts tallies values in vectors by returning a *named vector of counts* (repetitions):

```
x <- rep(c("A", "B", "Q"), times = 2:4)
x  # A A  B B B  Q Q Q Q
table(x)
#> A B Q
#> 2 3 4
```

By default, `NA`s are not tabulated – change the `useNA` argument:

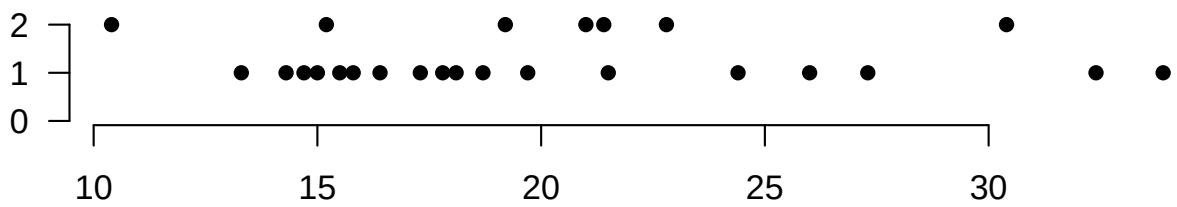
```
table(c(x, NA, NA))  # Same
table(c(x, NA, NA), useNA = "ifany")
#>    A    B    Q <NA>
#>    2    3    4    2
```

Counting with tables

A table of a vector is not really a table – it is a named vector containing the observations counts. To get the original values from the table, convert the names to numeric.

Are there identical fuel efficiency values?

```
x <- table(mtcars$mpg)
as.numeric(names(x))  # Original values
d <- data.frame(mpg = as.numeric(names(x)),
                count = as.numeric(x))
plot(d, ylim = c(0, max(x)))
```



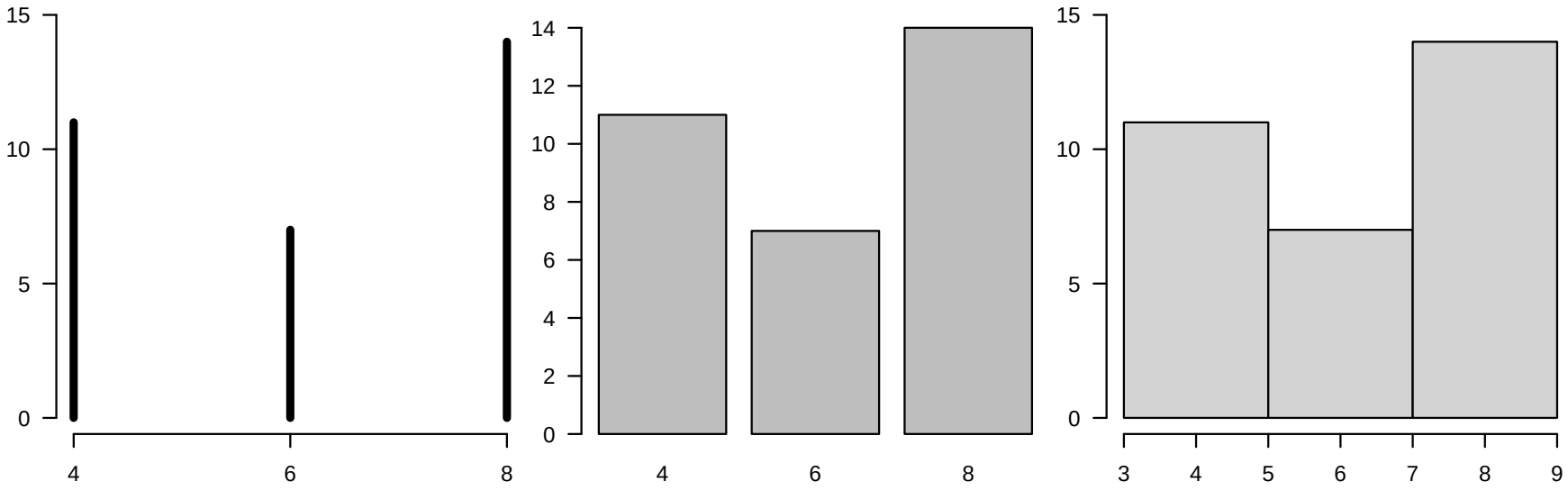
Histograms are count tables

Frequency distributions can be visialised with tables instead of histograms. Let `x <- mtcars$cyl`.

```
plot(table(x),
     lwd=4)
```

```
barplot(
  table(x))
```

```
hist(x, breaks =
  c(3, 5, 7, 9)
```



More generally, a histogram is really `table(cut(...))`.

Box-and-whisker plot

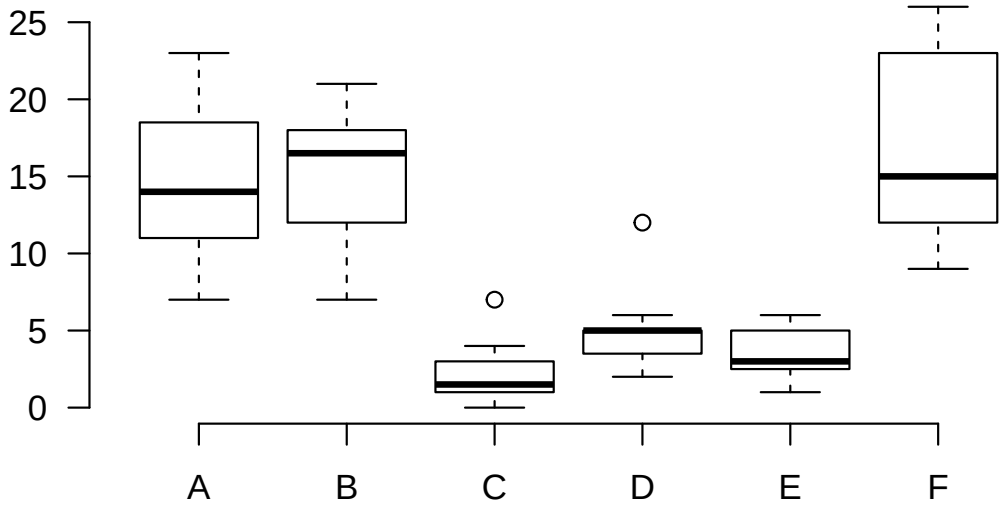
Imagine applying `aggregate()`, computing certain order statistics of a variable by group, and listing all values too far away from the median. Their visualisation is a **box plot**.

- Thick line: median
- Box top and bottom: Q_1 and Q_3
- Whiskers: the most extreme observation within 1.5-IQR distance from the box top and bottom
- Points: observations further away from those box

Application of boxplot()

The `InsectSprays` data set contains information on insect counts in experiments with 6 different insecticides. Apply the same kind of simple formula as in `aggregate()`:

```
boxplot(count ~ spray, InsectSprays,
  frame = FALSE, col = "#00000000")
```



NB. Remove the frame with `frame = FALSE`, not `bty="n"`.

Useful return from boxplot()

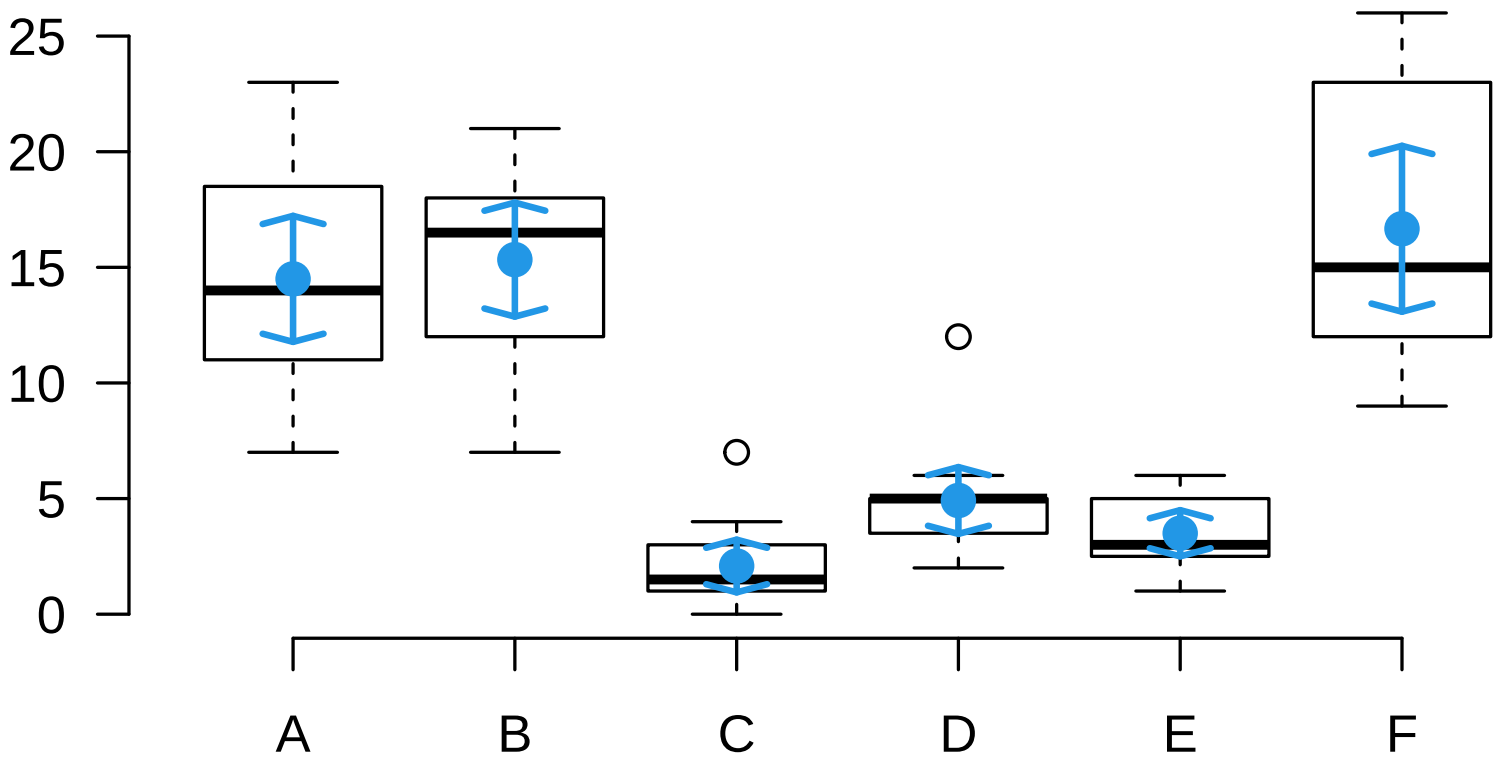
`boxplot()` returns useful coordinates to extend the plot.

Pass the same formula for `boxplot()` and `aggregate()` for comparability. The boxes are centred around 1, 2, ...

Compute the mean and 95% CI = $\bar{X} \pm 2\hat{\sigma}_X/\sqrt{n}$:

```
f <- count ~ spray
meanSD <- \(x) c(mean=mean(x), sd=sd(x), n=length(x))
a <- aggregate(f, InsectSprays, meanSD)
a <- data.frame(count = a[, 1], a[, 2])
b <- boxplot(f, InsectSprays)
points(1:6, a$mean, pch = 16, cex = 1.5, col = 4)
arrows(1:6, a$mean + 2*a$sd/sqrt(a$n),
       1:6, a$mean - 2*a$sd/sqrt(a$n),
       col=4, code=3, angle=75, length=0.1, lwd=2)
```

Box plot with extensions



sweep across array dimensions

Apply statistics across margins with `sweep()`. Use the same dimension indices as with `apply()`: 1 = rows, 2 = columns...

General syntax:

```
sweep(x, MARGIN, STATS, FUN = "-", ...)
```

- x: input array
- `MARGIN`: dimension index
- `STATS`: a vector / matrix of statistics to sweep out
- `FUN`: function that does the transformation: division, subtraction etc.

Application of sweep()

Subtract column means:

```
sweep(mtcars, 2, colMeans(mtcars), "-")
```

Divide matrix rows by row sums so that they add up to to 1:

```
set.seed(1)
x <- sort(rnorm(50)); g <- seq(-5, 5, 0.1)
d <- outer(x, g, \(x, y) dnorm(x-y))
sweep(d, 1, rowSums(d), "/")
# Same as d / rowSums(d) through recycling
# Same as apply(d, 2, \(x) x/sum(x))
```

Generate values uniformly in a 4D cube with sides 3, 4, 2, 6 by rescaling dimensions:

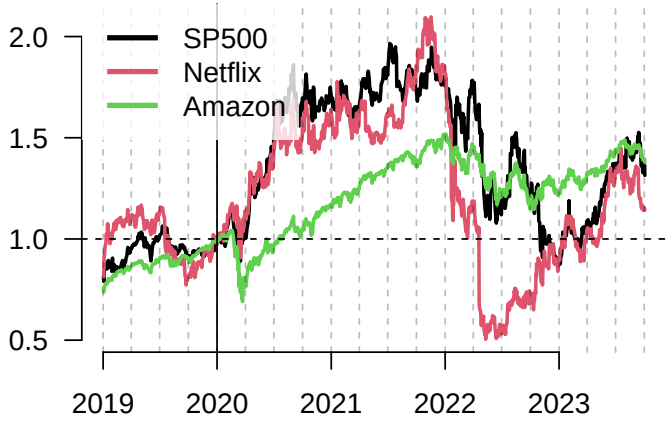
```
set.seed(1)
x <- matrix(runif(1000*4), ncol = 4) # [0, 1]
sweep(x, 2, c(3, 4, 2, 6), "*")
```

Custom series scaling

Often, indices are compared with respect to a reference date (e. g. '2015 = 100%').

Recall the matrix of stock prices `m` from Slide 142. Divide every column of the matrix by the beginning-of-2020 price:

```
x <- dl[[1]]$date
i20 <- which(x>="2020-01-01")[1]
m2 <- sweep(m, 2, m[i20, ], "/")
matplot(x, m2, type = "l",
  lty = 1, bty = "n", lwd = 2)
abline(v=as.Date("2020-01-01"))
# <.. Identical commands ..>
```



Smooth conditional average lines

`loess()` depends on two crucial parameters:

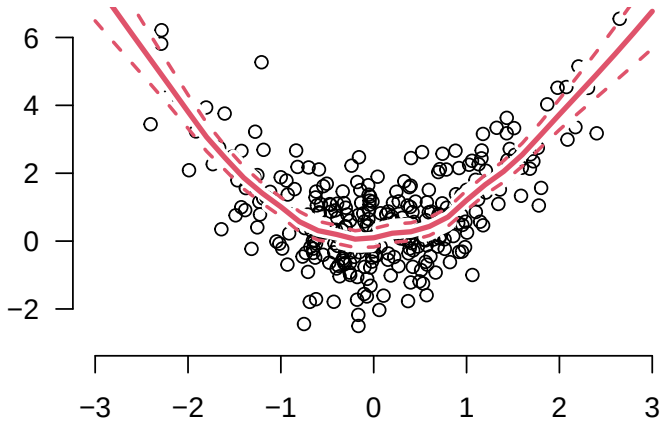
- `span` – the bandwidth for smoothing (values 0.1–0.6 work well)
- `degree` – use 0 for locally constant prediction, 1 for locally linear and 2 for locally quadratic
  - Prediction of bounded variables (dummies, conditional variances etc.) requires `degree = 0` to respect the range
- To predict values close to the boundary, use `control = loess.control(surface = "direct")`
  - Otherwise, **NA**s are possible

Confidence bands

Confidence bands can be plotted around the fitted values where the standard error of prediction is available.

```
set.seed(1); x <- rnorm(300); y <- x^2 + rnorm(300)
l <- loess(y ~ x, span = 0.3, degree = 1,
  control = loess.control(surface = "direct"))
xg <- seq(-3, 3, 0.2) # Grid
p <- predict(l, newdata = xg, se = TRUE)

plot(x, y)
lines(xg, p$fit, col = 2)
lines(xg, p$fit+2*p$se.fit,
  lty = 2, col = 2)
lines(xg, p$fit-2*p$se.fit,
  lty = 2, col = 2)
```

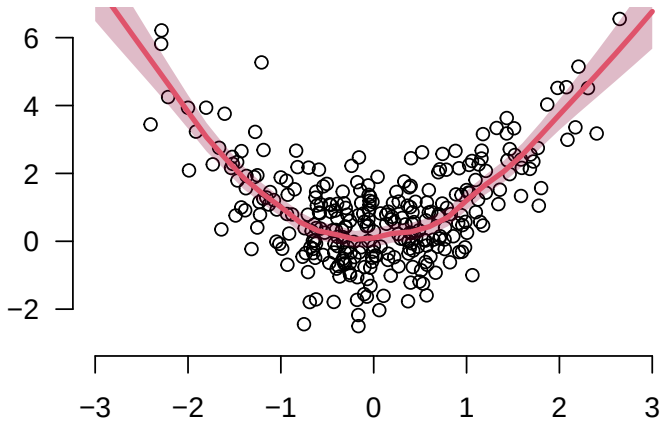


Shading confidence regions

Use `polygon()` to fill the shape between the lines formed by the ends of the point-wise confidence intervals.

To get the shape, start at the first point of the upper CI line, move right, drop to the right end of the lower CI line, and go left (reversing the point order):

```
plot(x, y)
yh <- p$fit+2*p$se.fit
yl <- p$fit-2*p$se.fit
polygon(c(xg, rev(xg)),
  c(yh, rev(yl)),
  col="#88003344", border=NA)
lines(xg, p$fit,
  lwd = 3, col = 2)
```



3D graphics, animations, and video encoding

Palettes from linear colour ramps

To create smooth transitions between multiple colours, use `colorRampPalette()`. It defines a function that accepts a vector of colours as input and returns their interpolation as output.

Skip the green colour in a colour gradient:

```
mycols <- c("#881100", "#BBBB00", "#5500FF")
colFun <- colorRampPalette(mycols)
cs <- colFun(10)
plot(rep(0, 10), col = cs)
```

Custom axis transformation

Some variables exhibit extreme values that extend the plotting ranges and make information in the middle indiscernible.

`plot(..., log = "y")` will not work with negative values.

Create your own unique transformations:

- 'Squish' the data using a bijective function and use the linear scale
- 'Unsquish' the axis labels and place them at appropriate spots of the linear scale
  - Alternative: design a custom non-linear scale with pretty values and 'squish' it

Squishing function requirements

Transformations should not result in artifacts:

- Continuity: transformed data should not have jumps
- Support in the real domain: transform $x \in \mathbb{R}$
- Monotonicity: lower original values $\mapsto$ lower transformed values
  - Invertibility: f^{-1} must exist s. t. $f^{-1}(f(x)) = x$ in its domain
- Lipshitz continuity: $f'(x)$ should be bounded
  - `sqrt(x)` has unbounded derivatives as $x \to 0^+$
- Predictable fixed points: $f(0) = 0$ or $f(1) = 1$ is good
- Oddness: $f(x) = -f(-x)$ to avoid distortions

Squishing function example

A power transformation ($p < 1$) with a shift $a > 0$ to avoid infinite $f'(0)$ looks good:

$$\begin{cases} f(x) & := [(|x| + a)^p - a^p] \cdot \operatorname{sign} x \\ f^{-1}(y) & := [(|y| + a^p)^{1/p} - a] \cdot \operatorname{sign} y \end{cases}$$

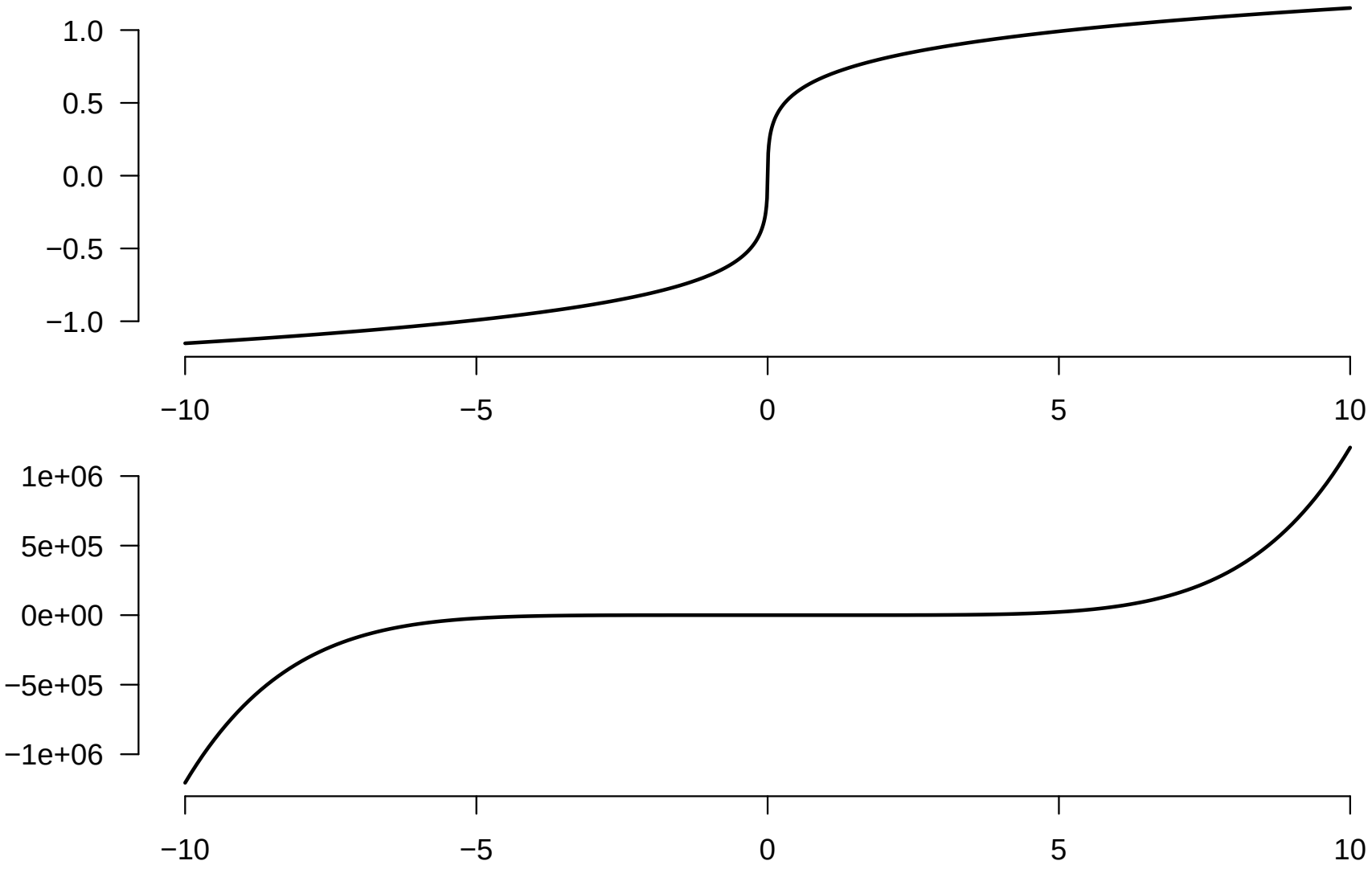
- $f'(x) = p(|x| + a)^{p-1}$ is continuous
- $f(0) = 0$
- $f'(0) \in \mathbb{R}$

Henceforth: $p = 1/6$, $a = 0.001$.

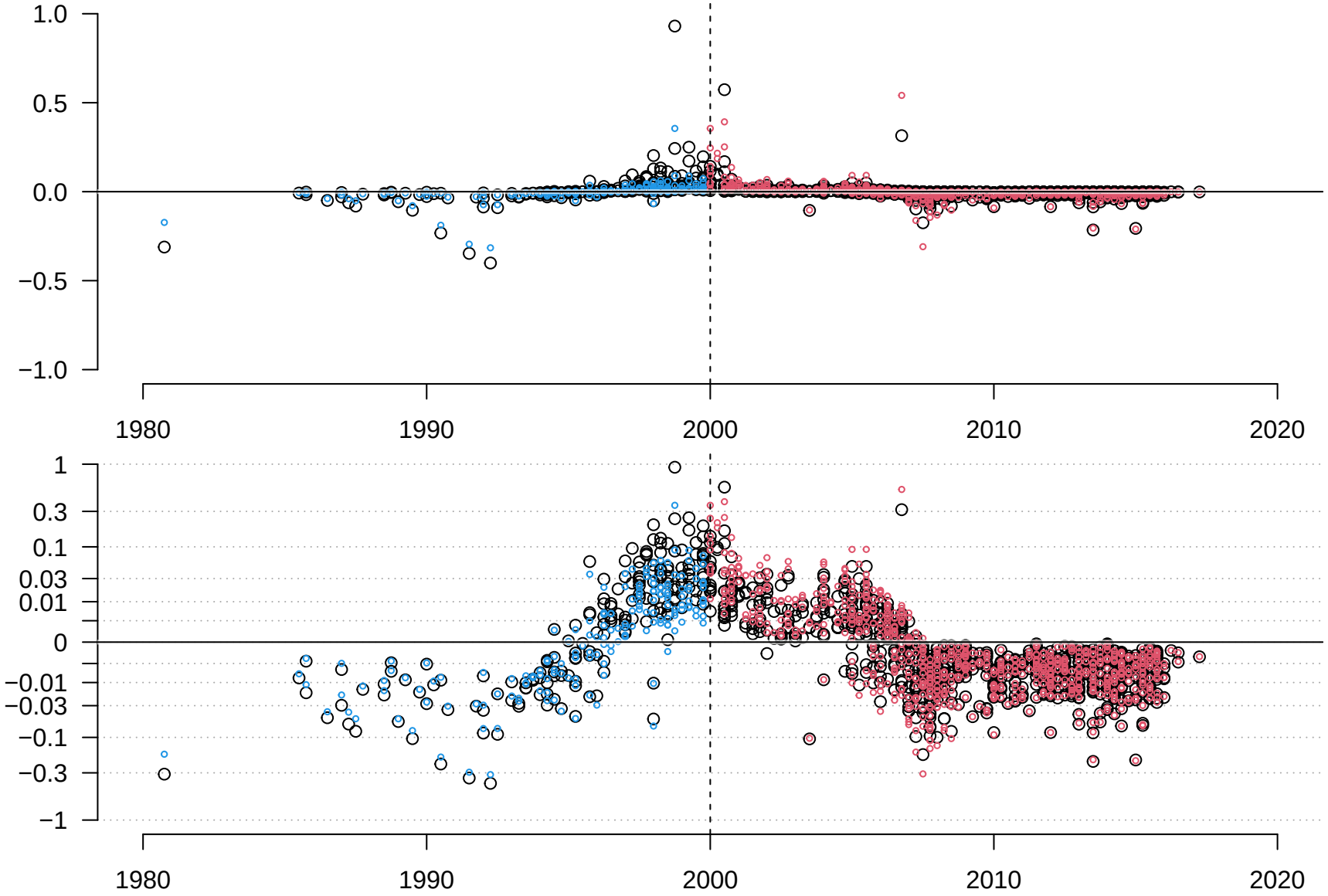
Squishing function definition

```
squish   <- function(x, pow = 1/6, shift = 0.001)
  ((abs(x) + shift)^pow - shift^pow) * sign(x)
unsquish <- function(y, pow = 1/6, shift = 0.001)
  ((abs(y) + shift^pow)^(1/pow) - shift) * sign(y)
```

Squishing function and its inverse



Squishing in practice



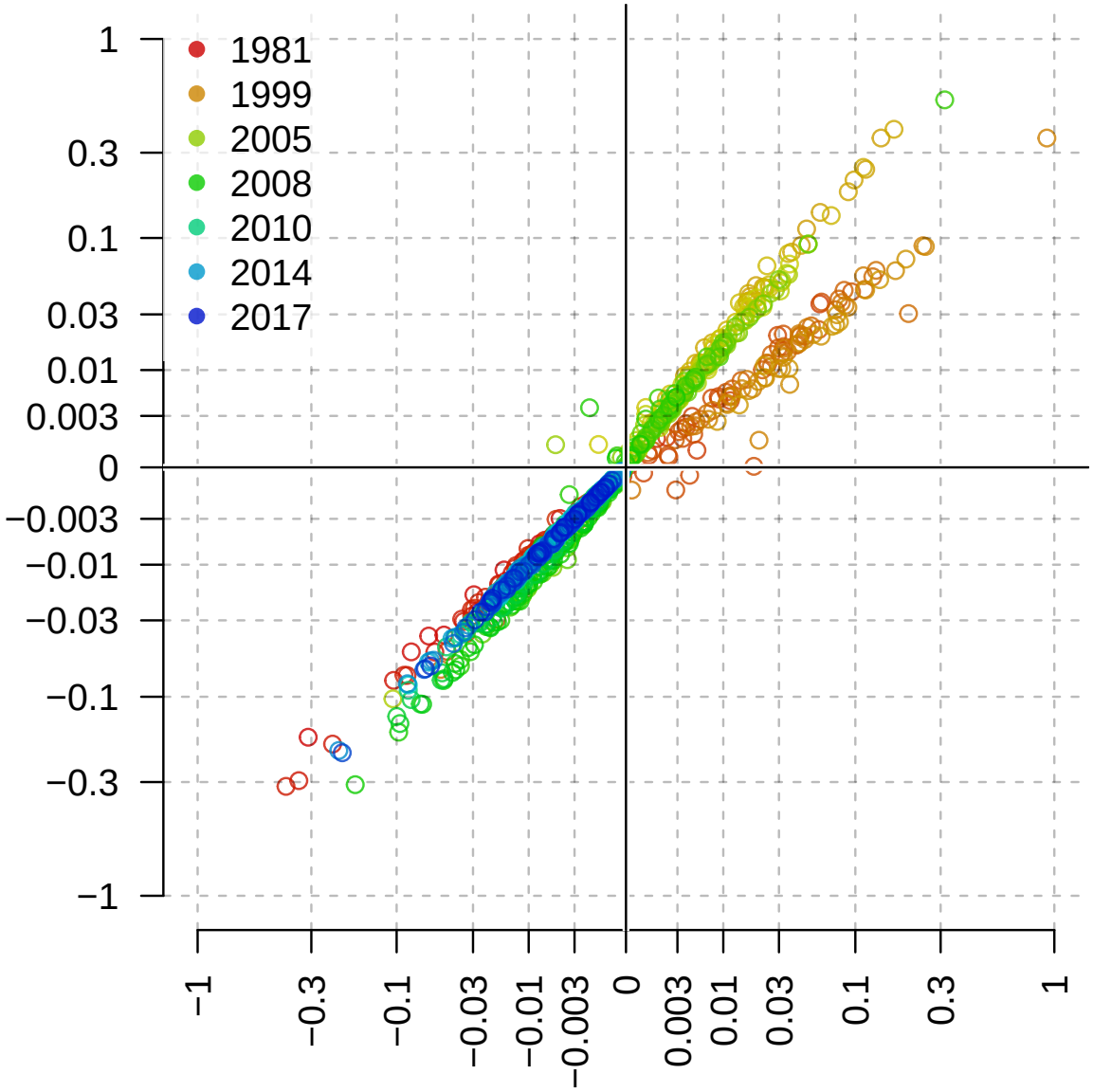
Squishing implementation

```
plot(d$x, d$y1, ylim = c(-1, 1))
points(d$x[d$x<2000],  d$y2[d$x<2000],  col=4, cex=0.5)
points(d$x[d$x>=2000], d$y2[d$x>=2000], col=2, cex=0.5)
```

Squish all vertical coordinates:

```
plot(d$x, squish(d$y1), yaxt = "n",
     ylim = squish(c(-1, 1)))
yax.pos <- c(0.003, 0.01, 0.03, 0.1, 0.3, 1)
yax.pos <- c(-rev(yax.pos), 0, yax.pos)
axis(2, at = squish(yax.pos), labels = yax.pos)
points(d$x[d$x<2000],  squish(d$y2[d$x<2000]),
       col = 4, cex = 0.5)
points(d$x[d$x>=2000], squish(d$y2[d$x>=2000]),
       col = 2, cex = 0.5)
abline(h = squish(yax.pos),
       col = "#00000055", lwd = 1, lty = 3)
```

Squishing both coordinates



X-Y squishing implementation

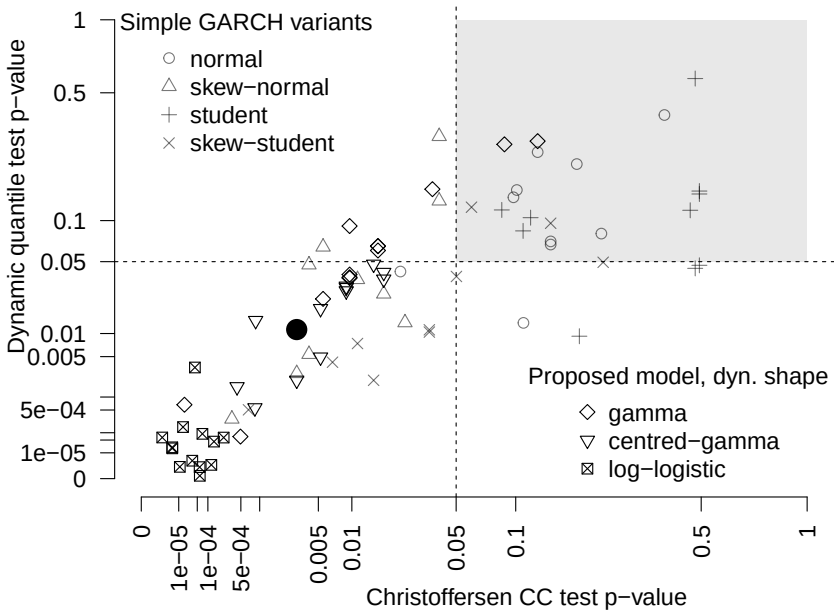
```
yax.pos <- c(0.003, 0.01, 0.03, 0.1, 0.3, 1)
yax.pos <- c(-rev(yax.pos), 0, yax.pos)
cs <- rainbow(nrow(d), end = 0.65, v = 0.8)

plot(squish(d$y1), squish(d$y2), col = cs,
  xlim = squish(c(-1, 1)), ylim = squish(c(-1, 1)),
  xaxt = "n", yaxt = "n", asp = 1)
axis(1, at = squish(yax.pos), labels = yax.pos, las=2)
axis(2, at = squish(yax.pos), labels = yax.pos, las=1)
abline(h = squish(yax.pos), v = squish(yax.pos),
  col = "#00000044", lwd = 1, lty = 2)
abline(h = 0, v = 0, lwd = 1)
ii <- round(quantile(1:nrow(d), 0:6/6))
legend("topleft", legend = round(d$x[ii]),
  col = cs[ii], pch = 16)
```

Squish to help read small values

Plot p-values of two tests and highlight non-rejections ($p = 1/4, a = 0$, source x and y not provided).

```
plot(NULL, NULL, xlim = c(0, 1), ylim = c(0, 1),
  xaxt = "n", yaxt = "n")
ats <- c(0, rep(10^(-5:-1),
  each=2) * c(1,5), 1)
axis(1, at = f(ats),
  labels = ats, las=2)
axis(2, at = f(ats),
  labels = ats, las=1)
abline(h = f(0.05),
  v = f(0.05), lty = 2)
polygon(
  f(c(0.05, 0.05, 1, 1)),
  f(c(0.05, 1, 1, 0.05)),
  col = "#00000018")
points(f(x), f(y))
```



Why animations and 3D?

- Observing the world moving in 3D is the **natural perception** in humans
- Static 2D plots and schemes are unnatural, often confusing, and require special knowledge
- 3D graphics are an excellent way to visualise changes simultaneously in two explanatory variables / parameters
- Animations allow the user to view 3D objects at various angles or to add a new dimension to 2D graphics
  - Indispensable investigation / debugging tool: animated 3D plots = changing 3 parameters simultaneously

Multi-variate tables

`table(x1, x2, x3, ...)` creates a table of counts for all possible combinations of $x_1, x_2, x_3, \ldots$ from their support:

```
set.seed(1)
x1 <- round(rnorm(1000, sd = 2))
x2 <- round(rnorm(1000, sd = 2))
x3 <- round(rnorm(1000, sd = 2))
a <- table(x1, x2, x3)
dim(a)  # 14 15 15
```

Bivariate tables as the foundation

```r
set.seed(1)
x1 <- round(rnorm(1000, sd = 2))
x2 <- round(rnorm(1000, sd = 2))
table(x1, x2)
```

```
    x2
x1   -7 -6 -5 -4 -3 -2 -1  0  1  2  3  4  5  6  7
  -6  0  0  0  0  1  0  1  0  2  0  0  0  0  0  0
  -5  0  0  0  3  0  2  2  0  3  4  2  0  0  0  0
  -4  0  0  1  0  4  5  6  2  3  3  4  1  0  0  0
  -3  0  0  0  1  7  5 16 15  5 10  4  1  1  0  0
  -2  0  0  1  4  5 15 26 21 17 15 10  4  1  0  0
  -1  0  2  1  9 10 25 24 35 32 26 11  6  0  0  0
  0   0  1  4  6 14 24 26 37 22 31 10  9  2  1  0
  1   0  0  1  5 10 22 27 45 28 22 10  2  1  1  0
  2   0  0  3  2  7 14 27 25 17 10  7  3  1  1  0
  3   0  0  0  5  6  4  9 11  5  9  6  3  0  0  0
  4   1  0  0  0  3  3  9  6  9  2  1  0  1  1  1
  5   0  0  1  0  0  1  2  2  2  2  1  0  0  0  0
  6   0  0  0  0  0  0  0  0  0  1  0  0  0  0  0
  8   0  0  0  0  0  0  0  0  0  0  1  0  0  0  0
```

There are many ways to visualise this verbose table.

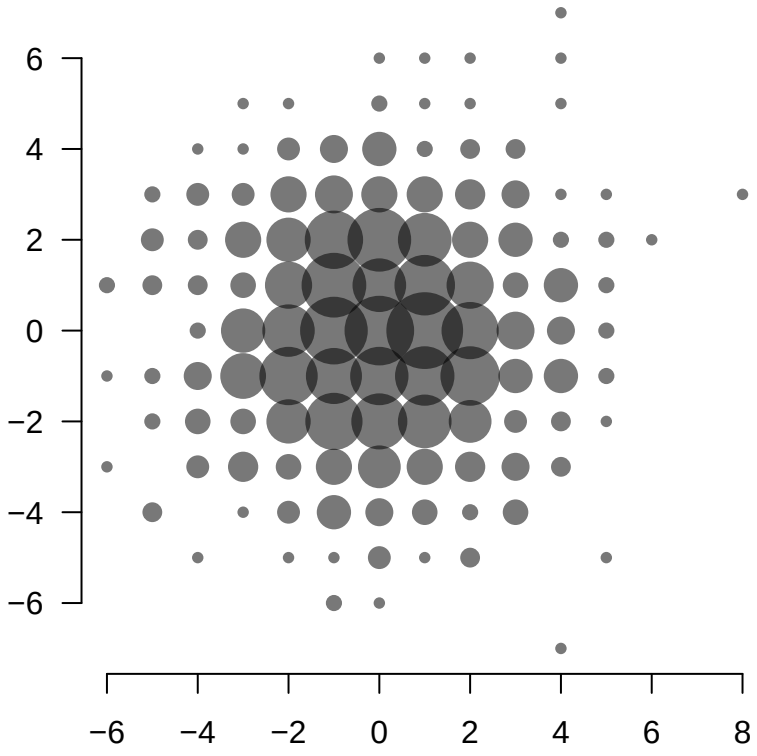
Visualising 2D tables via point size

Extract the coordinates for plotting counts:

```
x <- as.numeric(rownames(a))
y <- as.numeric(colnames(a))
xy <- as.matrix(expand.grid(x, y))
```

Show the number of occurrences as the area of the circle:

```
plot(xy[, 1], xy[, 2],
  pch = 16, asp = 1,
  cex = sqrt(a)*0.8,
  col = "#00000088")
```

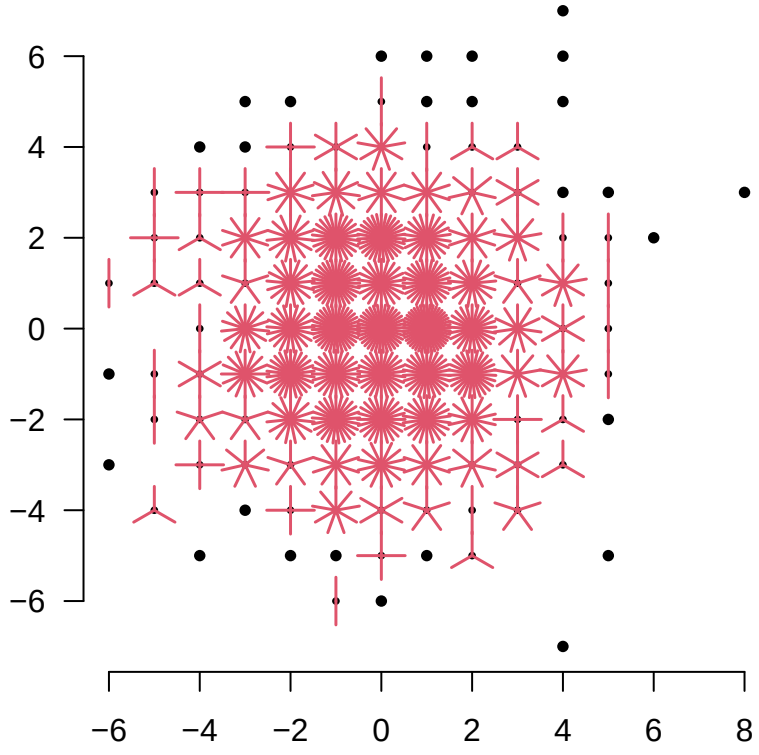


Visualising 2D tables via sunflowers

Points with higher counts are plotted as sunflowers with multiple petals.

Show the number of occurrences as the area of the circle:

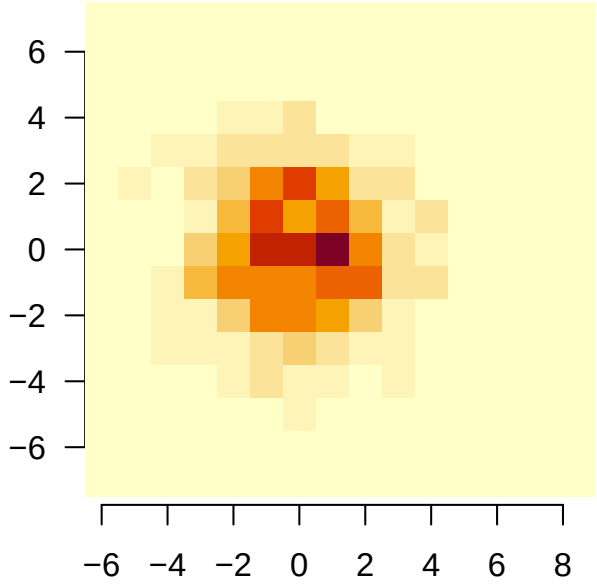
```
sunflowerplot(
  xy[, 1], xy[, 2], a,
  bty = "n", las = 1,
  asp = 1)
```



Visualising 2D tables via image()

Tables are the easiest way to obtain a matrix from data input by counting the occurences:

```
image(x, y, a,
  bty = "n",  # No box
  las = 1,  # Healthy neck
  asp = 1)  # 1:1 ratio
```

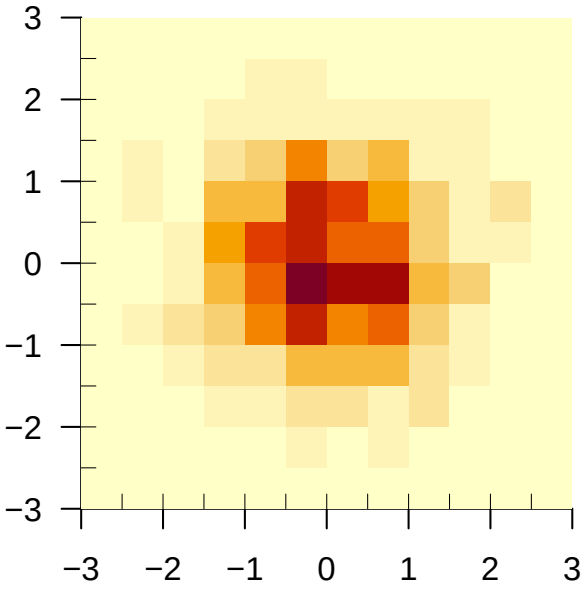


Custom breaks for bivariate histograms

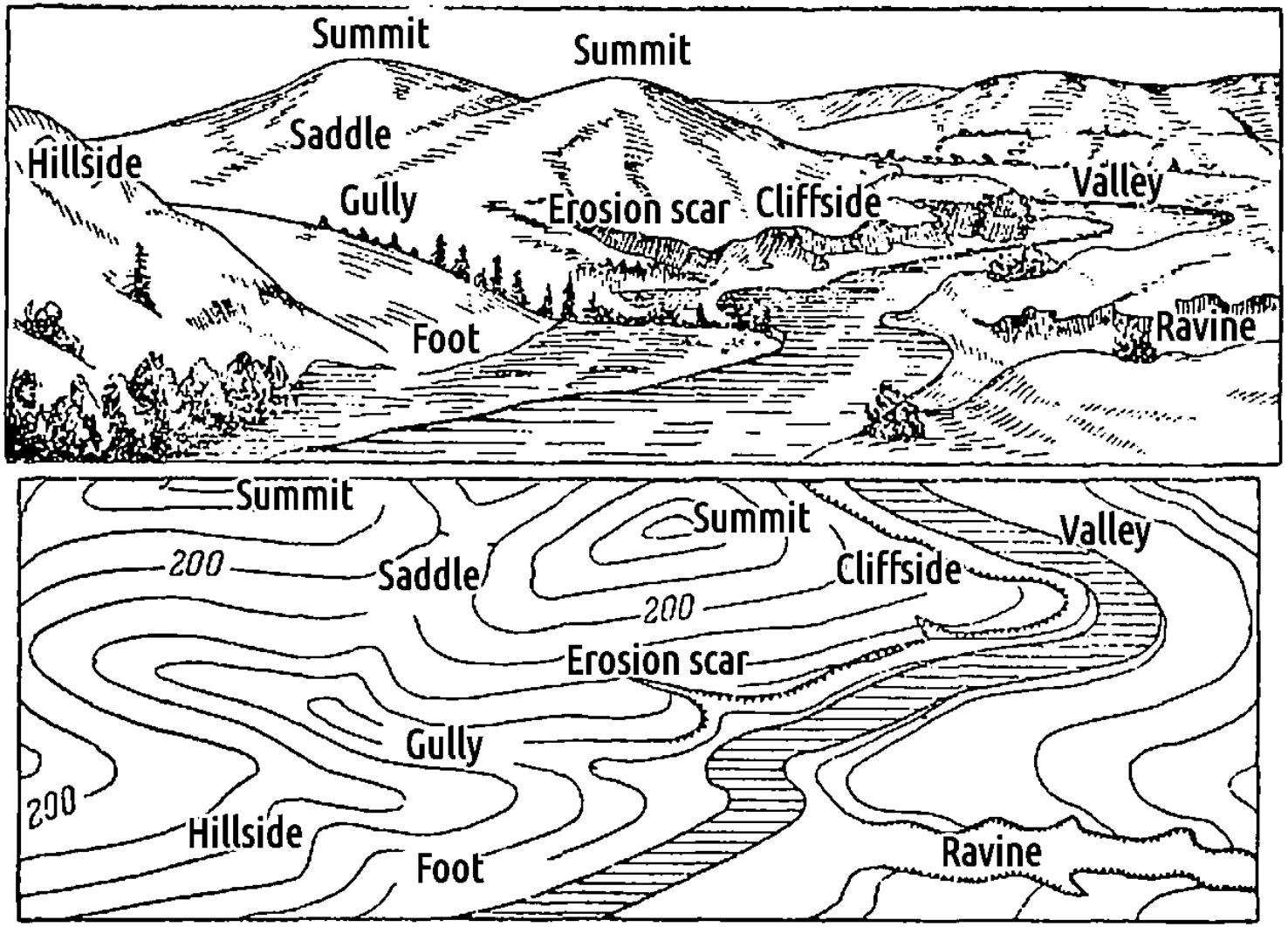
`table()` counts only unique occurences, which is meaningless for continuously distributed variables.

Use `cut()` to 'bin' the data into custom levels:

```
set.seed(1)
x1 <- rnorm(1000)
x2 <- rnorm(1000)
bs <- c(-Inf,seq(-2.5,2.5,0.5),Inf)
x1c <- cut(x1, breaks = bs)
x2c <- cut(x2, breaks = bs)
a <- table(x1c, x2c)
xs <- c(-2.75, bs[2:12]+0.25)
par(mar = c(2, 2, 0.5, 0.5))
image(xs, xs, a,
      bty = "n", las = 1, asp = 1)
rug(bs, side); rug(bs, side = 2)
```



Topographic maps

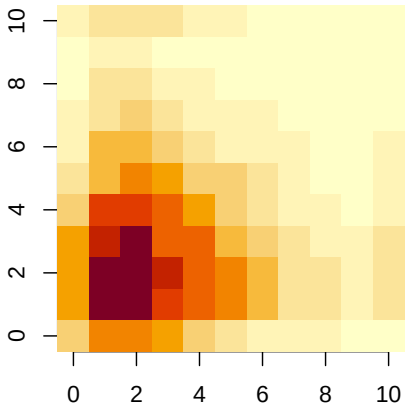


Contour plots

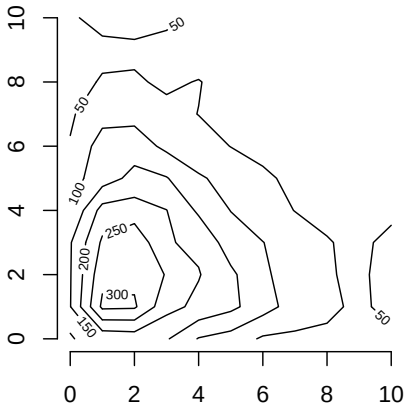
Suppose that a function is sampled on an evaluation grid to obtain a discrete approximation of a 3D object.

`contour()` attempts to draw level lines as shapes determined by equal heights: imagine a person walking around a mountain at the same height level.

```
set.seed(1)
x1 <- rchisq(10000, df = 4)
x2 <- rchisq(10000, df = 4)
xs <- 0:10; bs <- c(xs, Inf)
x1c <- cut(x1, breaks = bs)
x2c <- cut(x2, breaks = bs)
a <- table(x1c, x2c)
```




```
image(
 xs, xs, a)
```




```
contour(
 xs, xs, a)
```

Customise contour plots

Like with histograms, to create a custom number of level lines, do not use the `nlevels = 15` – create the levels explicitly.

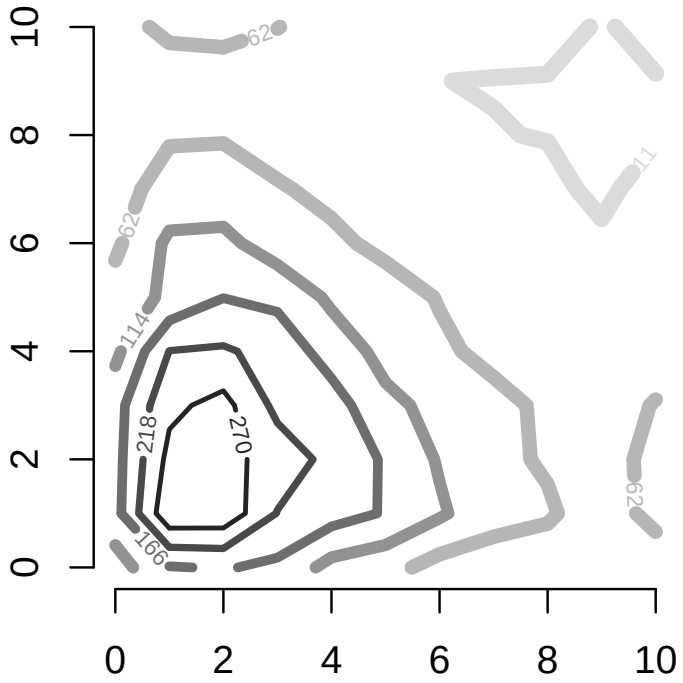
Customise vectorised colours, line widths etc. corresponding to those levels.

Transforming the matrix is the same as applying the inverse transformation to the values (recall squishing).

The results look good if the plot starts at a value slightly greater than the smallest one and ends at a values slightly less than the largest one.

Custom contour plot example

```r
qs <- quantile(unique(a), c(0.05, 0.95))
ls <- round(seq(qs[1], qs[2], length.out = 6))
cs <- gray(6:1/7)
contour(xs, xs, a,
  levels = ls, col = cs, lwd = 6:1+1)
```

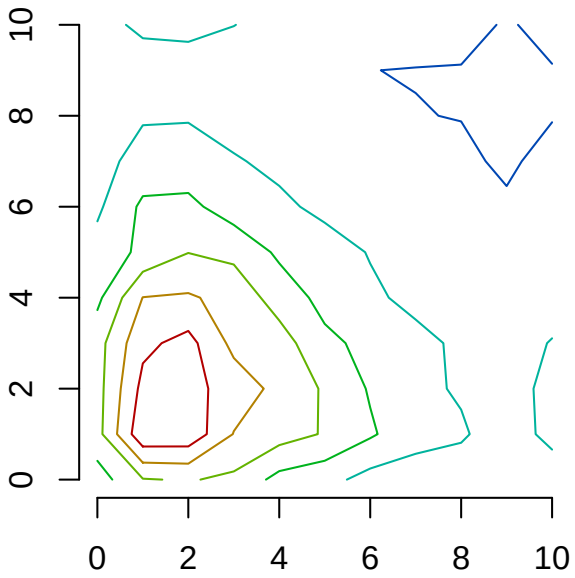


Saving contour lines

In certain cases, the user might want to save the contour lines – e. g. for squishing or 3D perspective transformation

`contourLines()` computes and saves the lines as a list of lists with 3 elements: level, *x* coordinates, *y* coordinates: there can be multiple lines per level.

```
lList <- contourLines(xs, xs, a, levels=ls)
str(lList[[1]])
#> $ level: num 11
#> $ x    : num [1:11] 10 9.33 9 8.54 8 ...
#> $ y    : num [1:11] 7.86 7 6.45 7 7.88 ...
cs <- rev(rainbow(6, end=0.6, v=0.7))
plot(NULL, NULL,
  xlim = c(0, 10), ylim = c(0, 10))
for (i in 1:length(lList)) {
  j <- which(ls == lList[[i]]$level)
  lines(lList[[i]], col = cs[j])
}
```

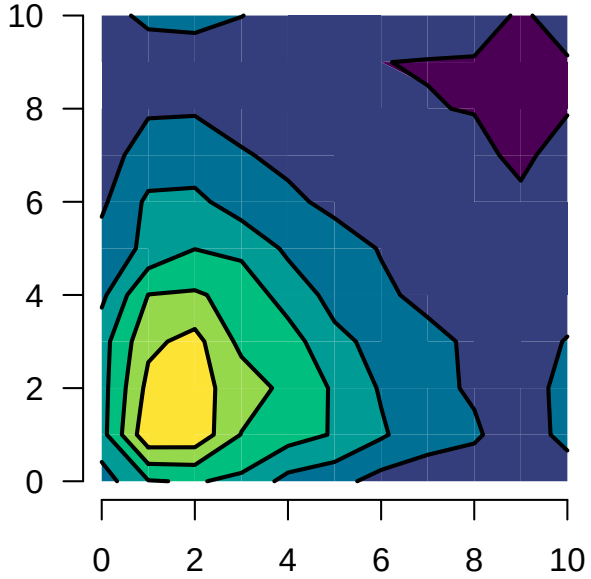


Filled contour plot

`filled.contour()` draws filled topographic maps – it needs the levels incl. max. / min. and +1 colour for the areas.

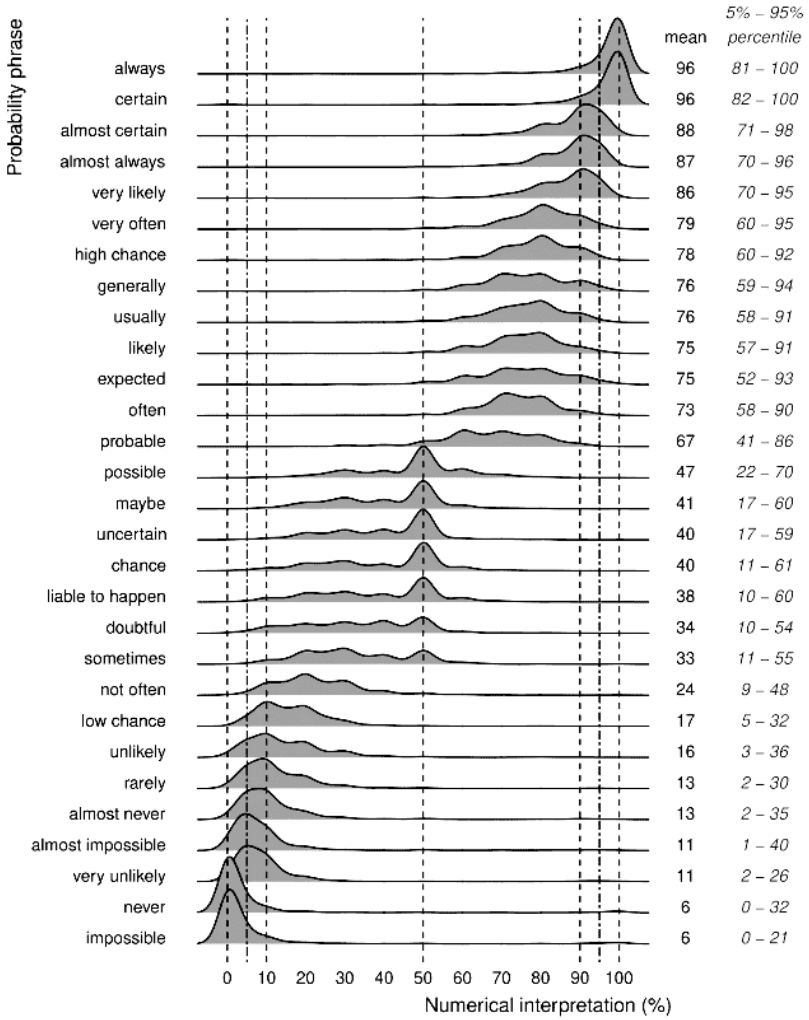
It is not very flexible because it creates a rigid custom `layout()`. The bare-bones command that simply adds polygons is `.filled.contour(x, y, z, levels, cols)`:

```
ls2 <- c(min(a)-1, ls, max(a)+1)
cs2 <- hcl.colors(7)
plot(NULL, NULL,
  xlim = c(0, 10), ylim = c(0, 10))
.filled.contour(xs, xs, a,
  levels = ls2, col = cs2)
sapply(lList, lines, lwd = 2)
```

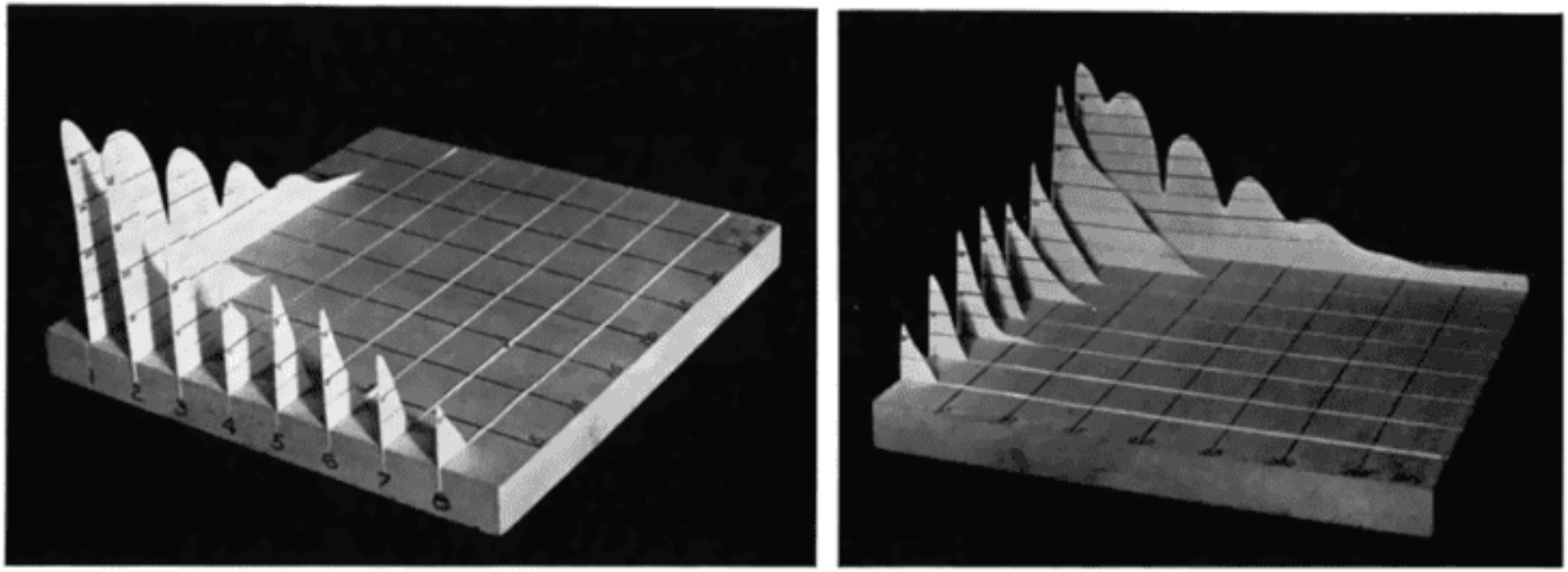


Sometimes, a 2D plot wastes space & time

If a plot is barely visible at full vertical size, it is bad.



3D plots have always been popular



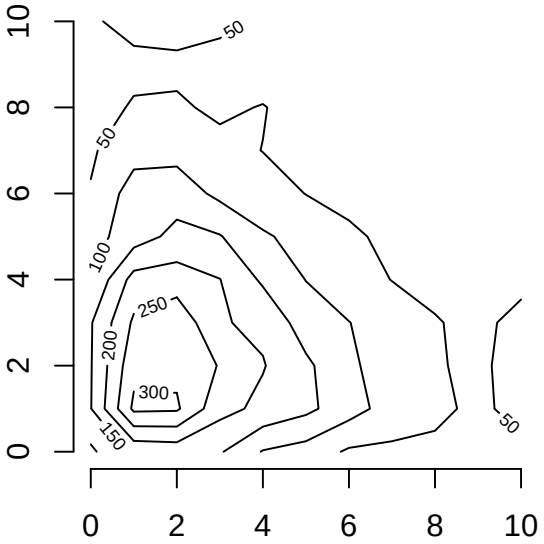
Amplitudes of the partials of a string as functions of time.

Credit: Schuck O. H., Young R. W. Observations on the vibrations of piano strings, Journal of the Acoustical Society of America, 15, 1, 1–11 (1943).

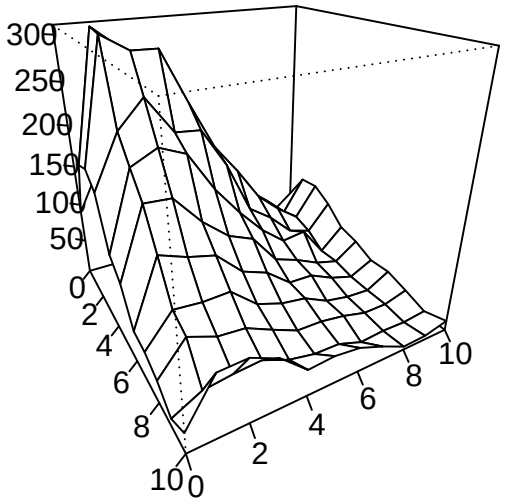
Ingredients for a 3D plot

Making a 3D surface plot is the same as visualising a matrix – treat a grid of values as the surface height.

Substitute `contour()` with `persp()`. Angle of view (degrees): `theta = 60` – horizontal, `phi = 25` – vertical.




```
contour(xs, xs, a)
```




```
persp(xs, xs, a,
  theta=60, phi=25)
```

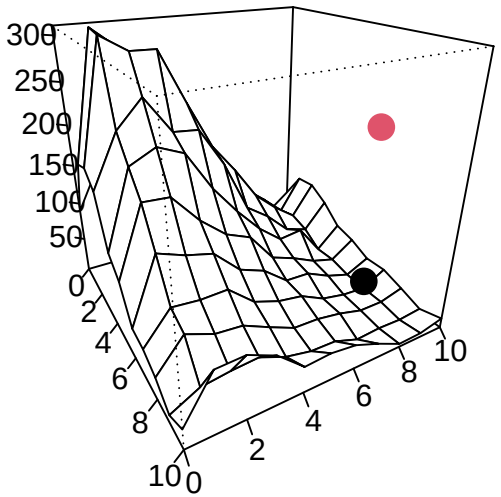
Viewing transformation matrix

`persp()` returns a very important object – the **viewing transformation matrix**.

The VTM is used to transform 3D points into 2D coordinates via `trans3d(x, y, z, VTM)` to added onto the plot via `points()` or `lines()`.

Add points (8, 8, 50) and (10, 6, 200):

```
p <- persp(xs, xs, a,
  theta = 60, phi = 25, ...)
xy <- trans3d(c(8,6), c(8,10),
              c(50,200), p)
points(xy, pch = 16,
  col = 1:2, cex = 2)
```



Adding levels lines to 3D

Recall that the `contourLines()` returns a list of lists with *x*, *y*, and level – but the level is really the *z* coordinate.

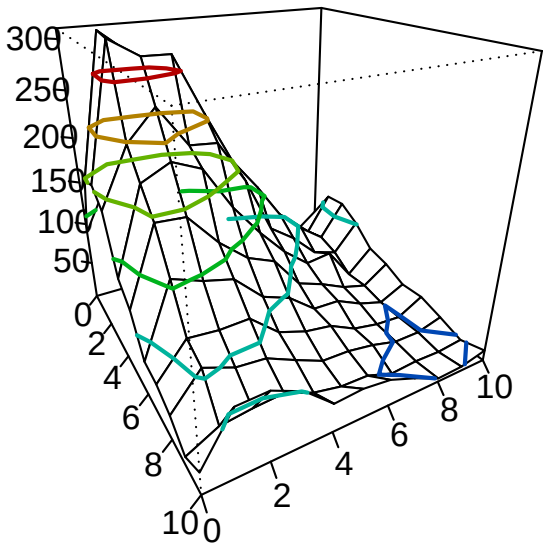
`trans3d(x, y, z, VTM)` requires that *x*, *y*, and *z* be of the same length – this can be achieved by repeating the scalar level as many times as there are points in *x*.

- Compute the points for contour lines
- Initialise a 3D plot, save the viewing transformation matrix
- For each contour line, transform the 3D coordinates and add to the plot
  - Choose appropriate colour and style

3D plot with lines at fixed levels

```
set.seed(1); xs <- 0:10; xb <- c(xs, Inf)
x1 <- rchisq(10000, df = 4); x2 <- rchisq(10000, df = 4)
x1c <- cut(x1, breaks = xb); x2c <- cut(x2, breaks = xb)
a <- table(x1c, x2c)
qs <- quantile(unique(a), c(0.05, 0.95))
ls <- seq(qs[1], qs[2], length.out = 6)
m  <- contourLines(xs, xs, a, levels = ls)
cs <- rev(rainbow(6, end = 0.6, v = 0.7))

p <- persp(xs, xs, a, theta=60, phi=25,
  ticktype = "detailed", xlab = "",
  ylab = "", zlab = "")
for (i in 1:length(m)) {
  z <- m[[i]]$level
  j <- which(ls == z)
  n <- length(m[[i]]$x)
  xy <- trans3d(m[[i]]$x, m[[i]]$y,
                rep(z, n), p)
  lines(xy, col = cs[j], lwd = 2)
}
```



Plotting arbitrary 3D functions

To visualise any 3D function:

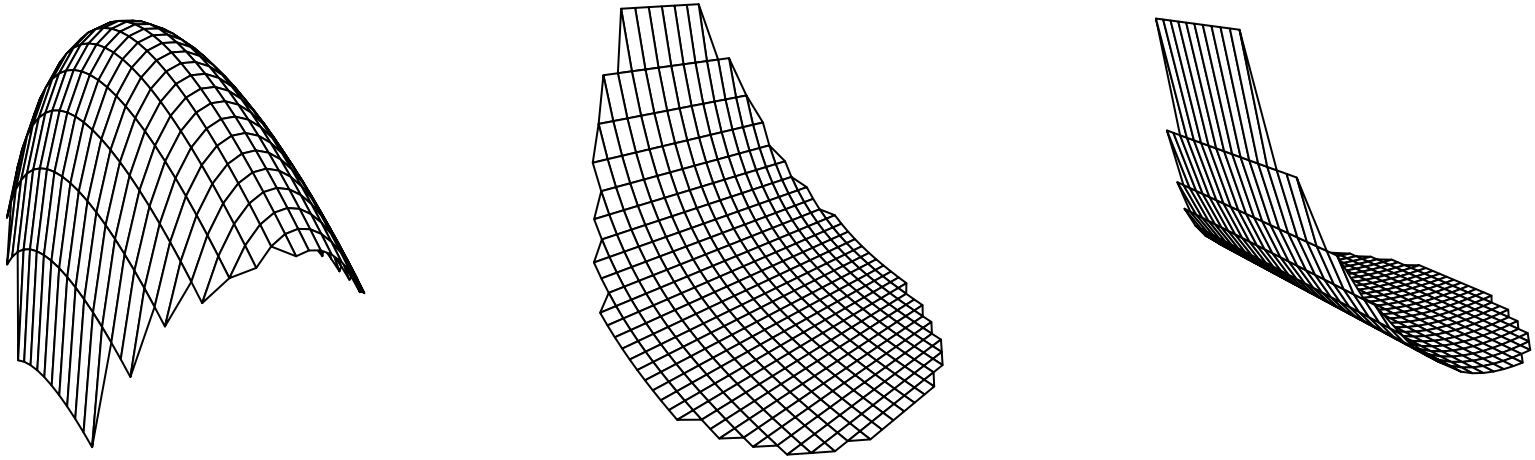
- Create an x grid and a y grid
- For all combinations (x_i, y_j), compute $f(x_i, y_j)$
  - If necessary, assemble the results in a matrix
  - This is where parallelisation shines: compute multiple values in parallel!
- Call `persp()` with the two grids and z matrix

`expand.grid(x1, x2, x3, ...)` returns a DF of all possible combinations of x_1, x_2, x_3, ...

3D function plot example (1/4)

$$f(x, y) \coloneqq \log(x + 1.9) + \log(y + 1.8) - 0.6x - 0.7y$$

$$\nabla f(x, y) = \left[\frac{1}{x + 1.9} - 0.6, \frac{1}{y + 1.8} - 0.7\right]$$



$f(x, y)$ $\nabla_x f(x, y)$ $\nabla_y f(x, y)$

3D function plot example (2/4)

Since $\log x$ may tend to $-\infty$, plot the function in the radius $r = 1.5$ around the origin ($\mathrm{dom} f \coloneqq \|(x\ y)\| < 1.5^2$).

```
f <- \(x) if (sum(x^2)<2.25)
  2 + log(x[1]+1.9) + log(x[2]+1.8)
    - 0.6*x[1] - 0.7*x[2] else NA
fp <- \(x) if (sum(x^2) < 2.25)
  c(1/(x[1]+1.9)-0.6, 1/(x[2]+1.8)-0.7) else c(NA, NA)
nx <- 31; ny <- 21
xseq <- seq(-1.4, 1.4, length.out = nx)
yseq <- seq(-1.4, 1.4, length.out = ny)
xy <- as.matrix(expand.grid(x = xseq, y = yseq))
```

3D function plot example (3/4)

Compute the values of f and ∇f in parallel for all combinations and create the $n_x \times n_y$ matrix of values:

```
library(parallel); ncores <- 4
cl <- makeCluster(ncores) # For all OSs
clusterExport(cl, c("f", "fp", "xy"))
ii <- 1:nrow(xy)
flist  <- parLapply(cl, 1:ii, \(i) f(xy[i,]))
fplist <- parLapply(cl, 1:ii, \(i) fp(xy[i,]))
zmat   <- matrix(unlist(flist), ncol = ny)
zpvecx <- unlist(lapply(fplist, "[", "x"))
zpvecy <- unlist(lapply(fplist, "[", "y"))
zpmatx <- matrix(zpvecx, ncol = ny)
zpmaty <- matrix(zpvecy, ncol = ny)
stopCluster(cl)
```

3D function plot example (4/4)

Create a custom plotting function with ascetic parameters: delete the box, axes, axis labels etc.

```
myPersp <- function(...) persp(...,
  xlab = "", ylab = "", zlab = "",
  asp = 1, theta = 45, phi = 20,
  axes = FALSE, box = FALSE)

par(mar = c(0, 0, 0, 0))
myPersp(xseq, yseq, zmat)
myPersp(xseq, yseq, zpmatx)
myPersp(xseq, yseq, zpmaty)
```

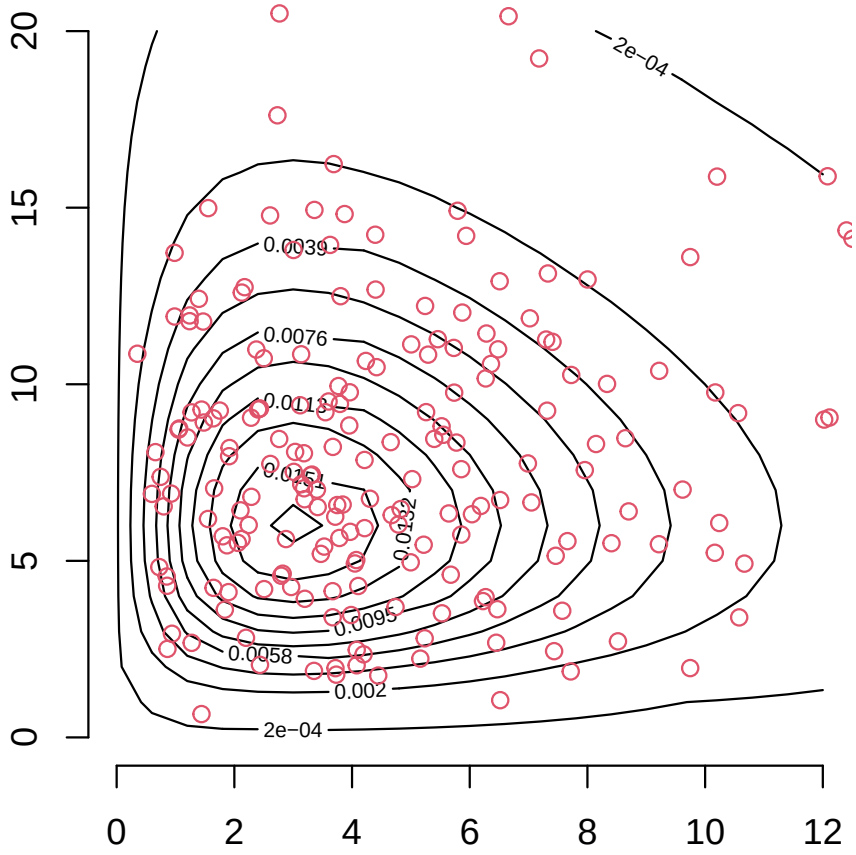
Semi-transparent surfaces

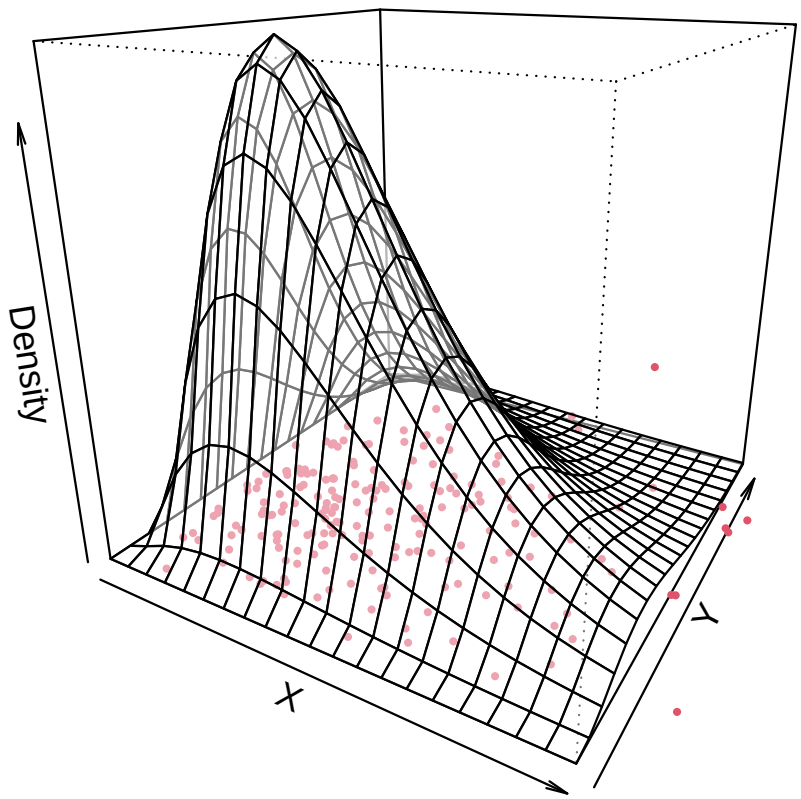
Since the objects are added in the order they are called in, every new addition is super-imposed. As a consequence, there is no proper clipping because of this drawing order.

It is possible to prepare the canvas and change the drawing order manually via a dummy plot.

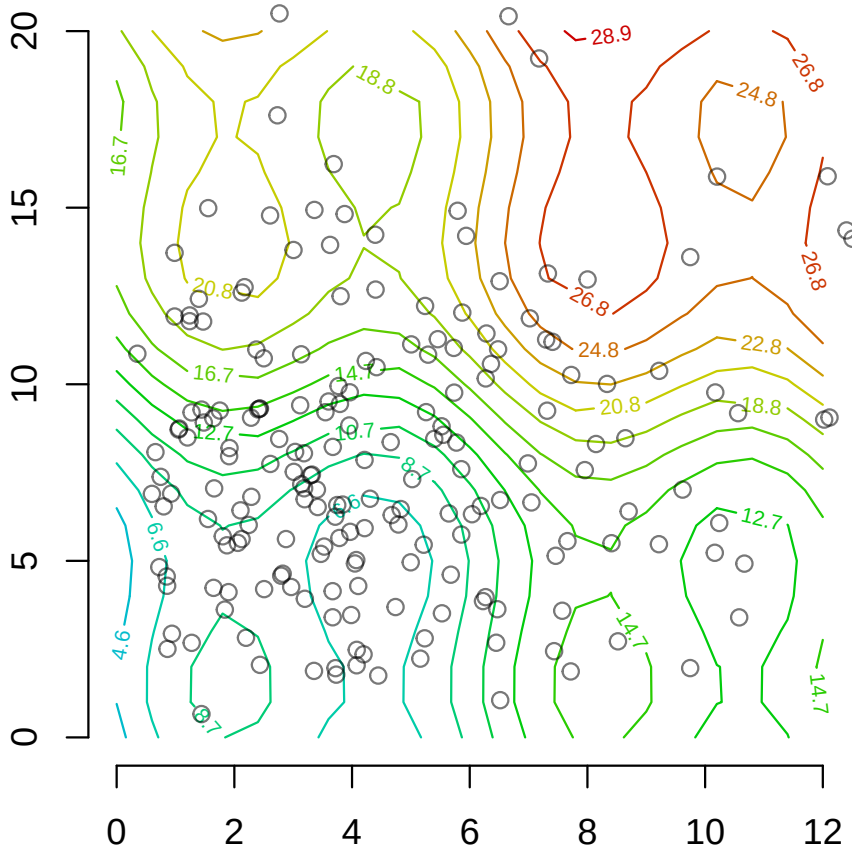
- Create a single fully transparent square and manually set the plotting limits
- Add the points
- Use `par(new = TRUE)` to write over the old plot
- Add the semi-transparent surface with no axes or labels
  - Use the same plotting parameters

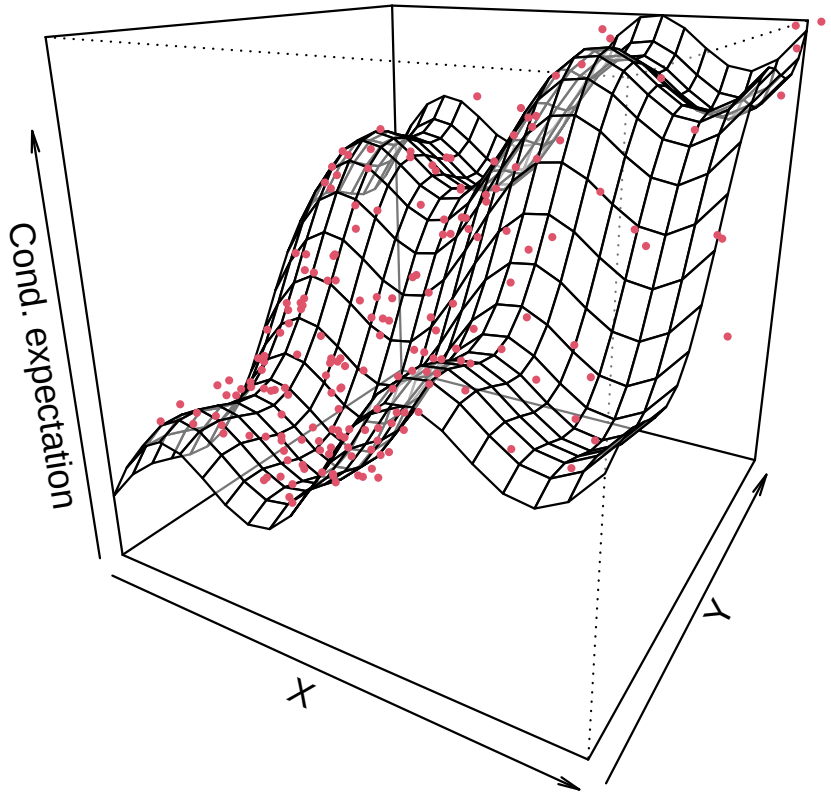
Some height maps are easy to read





Some height maps are not so easy to read





Juxtaposing contour and 3D plots (1/2)

Data used for plotting:

```
f <- \(x, y) dchisq(x, 5) * dchisq(y, 8)
g <- \(x, y) 1 + x + y + 3 * sin(x) + 3 * cos(0.5*y)
set.seed(1)
X <- rchisq(200, 5); Y <- rchisq(200, 8)
XY <- list(x = X, y = Y); Z <- g(X, Y) + rnorm(200)
```

Create the grid and evaluate the functions:

```
ngrid <- 21
Xg <- seq(0, 12, length.out = ngrid)
Yg <- seq(0, 20, length.out = ngrid)
XYg <- as.matrix(expand.grid(Xg, Yg))
fmat <- matrix(f(XYg[, 1], XYg[, 2]), nrow = ngrid)
gmat <- matrix(g(XYg[, 1], XYg[, 2]), nrow = ngrid)
```

Juxtaposing contour and 3D plots (2/2)

```r
par(mfrow = c(1, 2))
ls <- seq(max(fmat)*0.01, max(fmat)*0.98, length.out=10)
contour(Xg, Yg, fmat, levels = ls); points(XY, col = 2)

p <- persp(0:1, 0:1, matrix(rep(0, 4), 2),
  xlim = c(0,12), ylim = c(0,20), zlim = range(0,fmat),
  xlab = "X", ylab = "Y", zlab = "Density",
  theta = 30, phi = 20,
  border = NA, col = "#00000000")  # <-- TRANSPARENCY
points(trans3d(X, Y, rep(0, 200), p),
  pch = 16, col = 2, cex = 0.5)
par(new = TRUE)  # Do not clear the plotting window
persp(Xg, Yg, fmat, xlab="", ylab="", zlab="",
  xlim = c(0,12), ylim = c(0,20), zlim = range(0,fmat),
  theta = 30, phi = 20, col = "#FFFFFF77",
  box = FALSE, axes = FALSE)  # <-- NO BOUNDING BOX
```

Using 3D to save space (1/4)

Recall the example from earlier: plot many densities. It can be done parsimoniously in 3D.

Generate a data set:

```
set.seed(1)
m <- rep(1:20, each = 100)
x <- rnorm(2000)*m^0.25 + m
d <- data.frame(x, m)
```

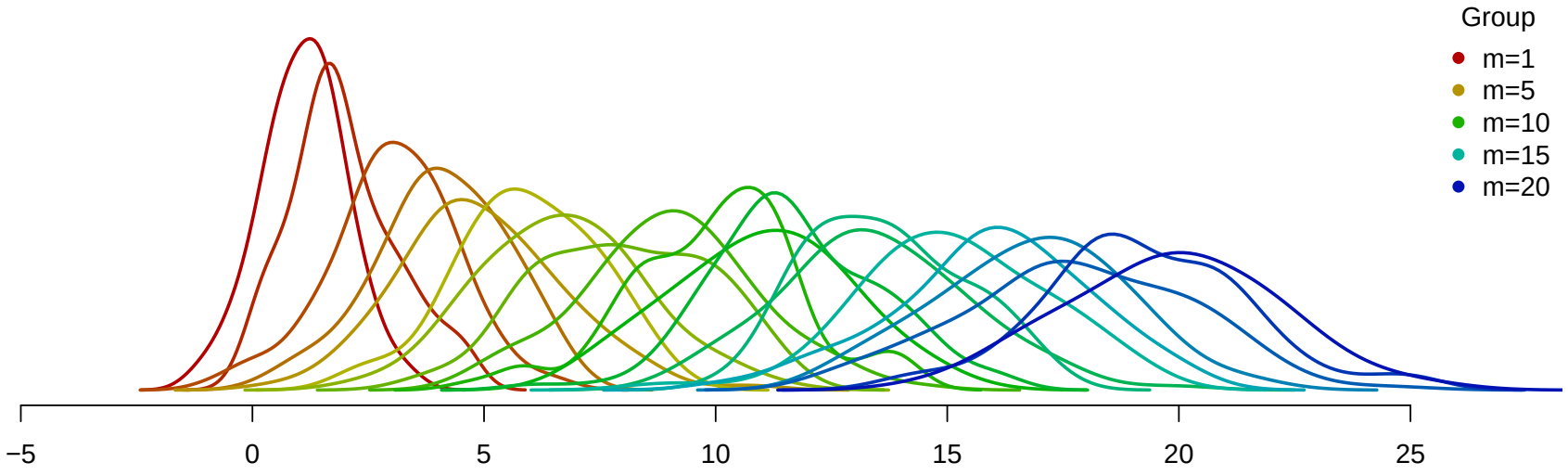
Compute the densitiy of X for each value of m:

```
dl <- lapply(1:20, \(i) density(d$x[d$m == i], bw="SJ"))
```

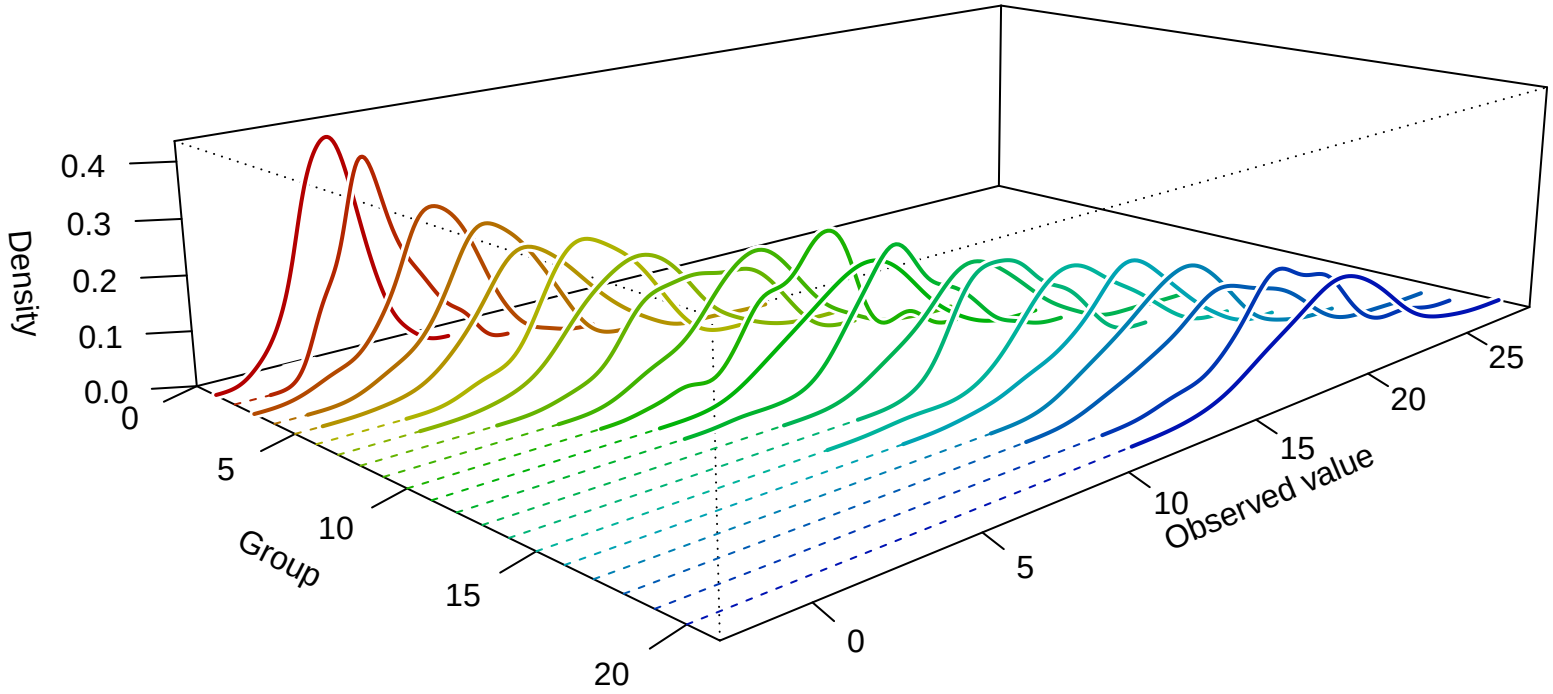
Using 3D to save space (2/4)

Many densities in a single plot look cluttered:

```
cs <- rainbow(20, end = 0.65, v = 0.7)
plot(NULL, NULL, xlim = c(-5, 27), ylim = c(0, 0.45))
lapply(1:20, \(i) lines(dl[[i]], col = cs[i], lwd = 2))
ms <- c(1, 5, 10, 15, 20)
legend("topright", paste0("m=", ms), col = cs[ms],
  pch = 16, bty = "n", title = "Group")
```



Using 3D to save space (3/4)



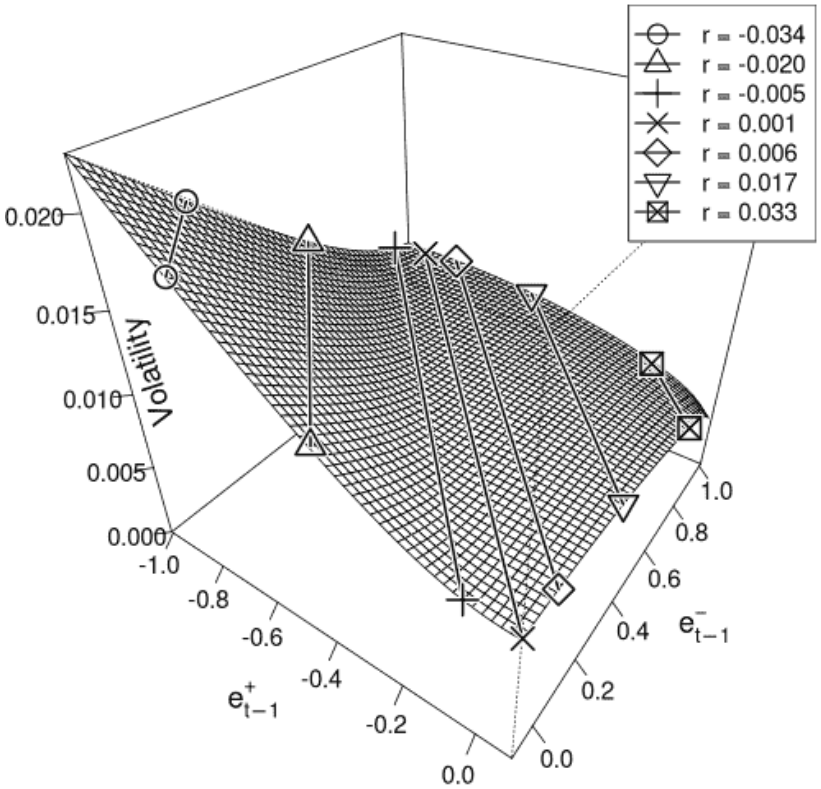
Using 3D to save space (4/4)

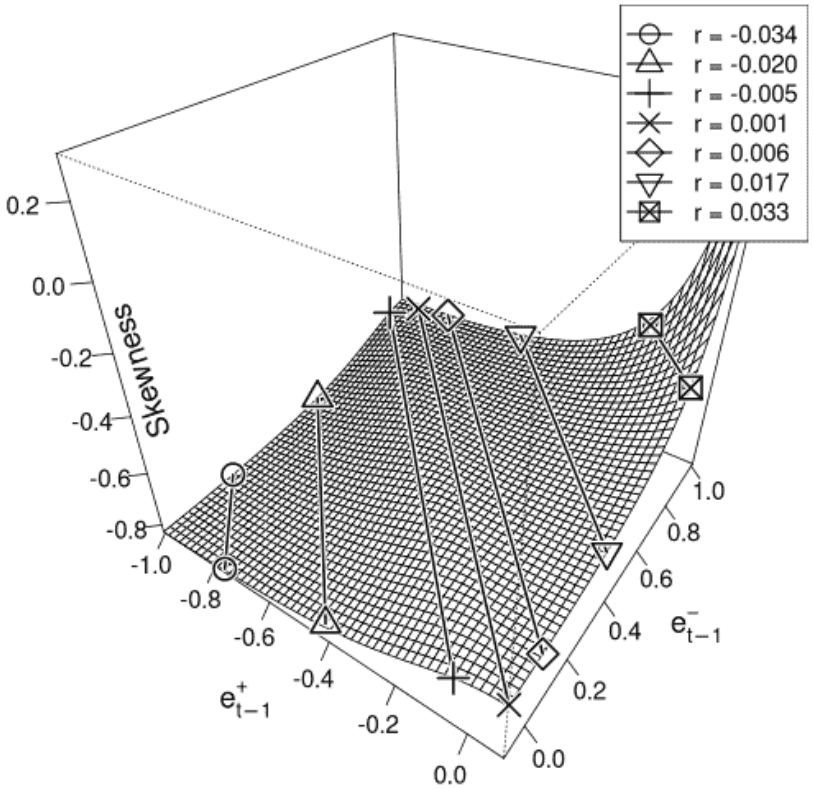
```r
# Empty perspective box with NO SCALING
p <- persp(0:1, 0:1, matrix(rep(0, 4), 2), border = NA,
  xlim = range(d$m) + c(-1, 1),
  ylim = (yl <- range(sapply(dl, "[[", "x"))),
  zlim = range(0, sapply(dl, "[[", "y"))*1.05,
  theta = 45, phi = 15, scale = FALSE, expand = 15,
  xlab = "Group", ylab = "Value", zlab = "Density",
  ticktype = "detailed")

lapply(1:20, \(i) {  # Lines with white halps
  xy <- trans3d(rep(i, length(dl[[i]]$x)),
    (ys <- dl[[i]]$x), dl[[i]]$y, p)
  lines(xy, col = "white", lwd = 5)
  lines(xy, col = cs[i], lwd = 2)
  g <- trans3d(c(i, i), c(yl[1], min(ys)), c(0, 0), p)
  lines(g, lty = 2, col = cs[i])  # Dashed guides
})
```

Optimising 3D plots for journals

Use monochromatic images with large labels and halos for better legibility. Find a good angle!





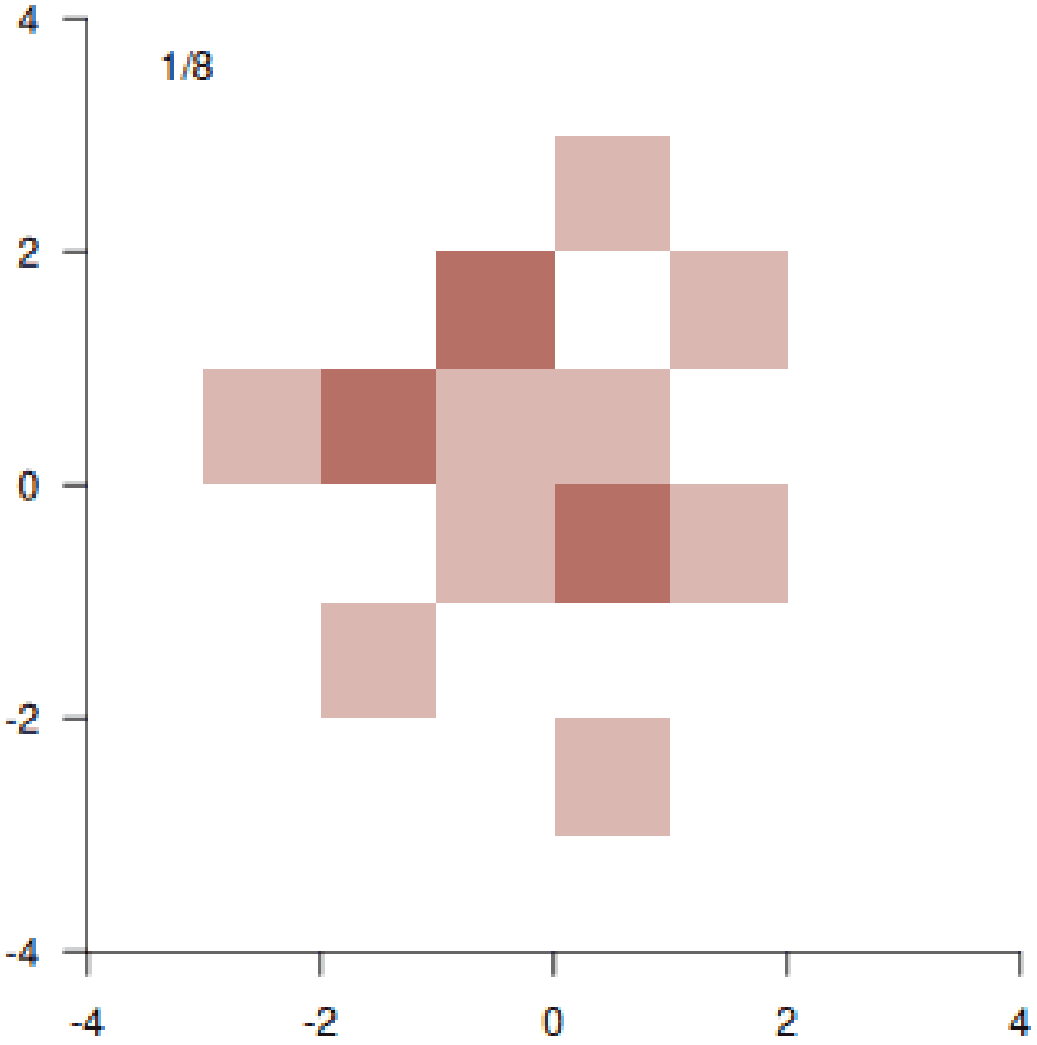
Animations in R

Animations are typically created from a sequence of frame.

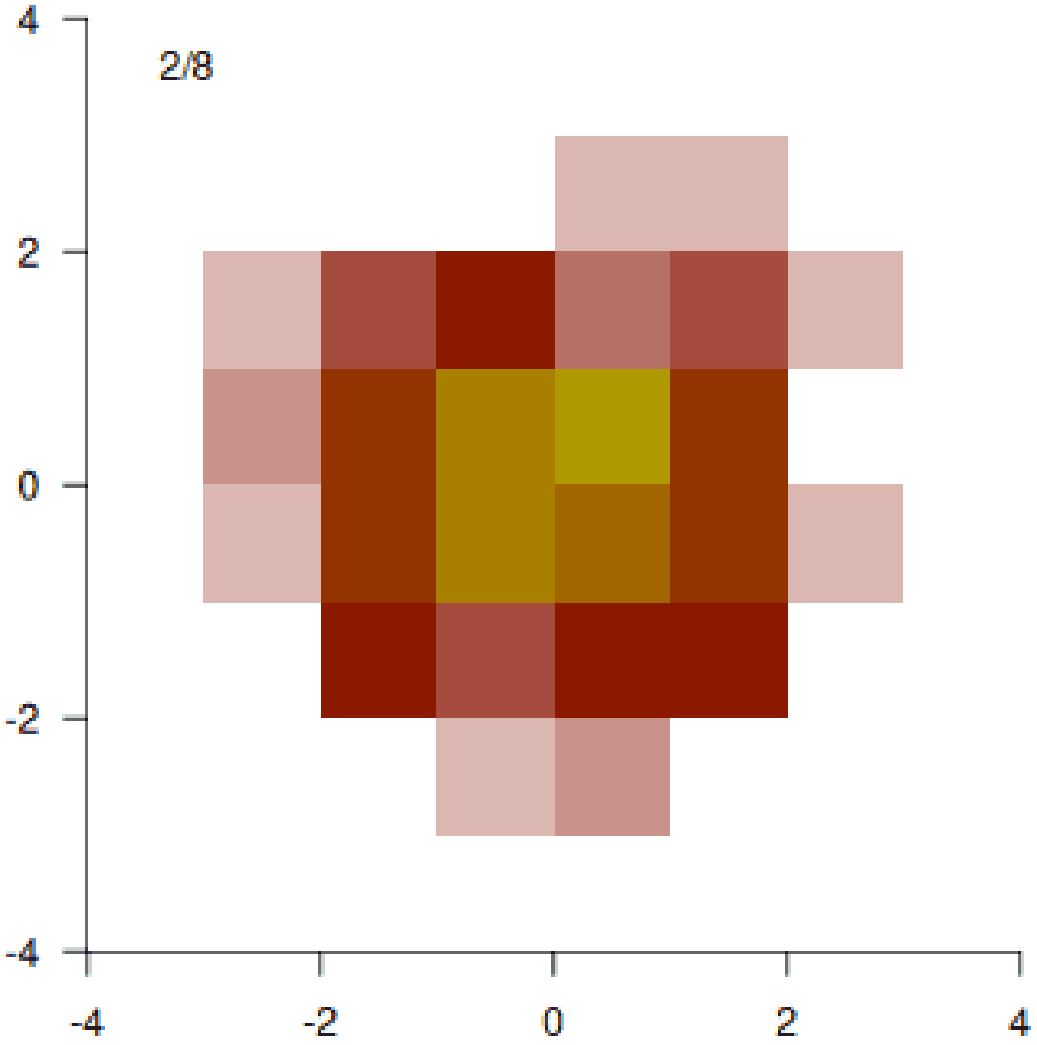
Standard animation workflow:

- Produce *n* plots in a sequence, writing each plot to disk as a **raster** image (PNG, not PDF)
  - `image(..., zlim = c(a, b))` ensures that the colours are consistent (same cut-off in all frames)
- Use an external tool (ImageMagick or FFmpeg) to assemble the results into a GIF or MP4
  - Optionally: crush the GIF size with gifsicle
  - Or simply insert the images into different slides

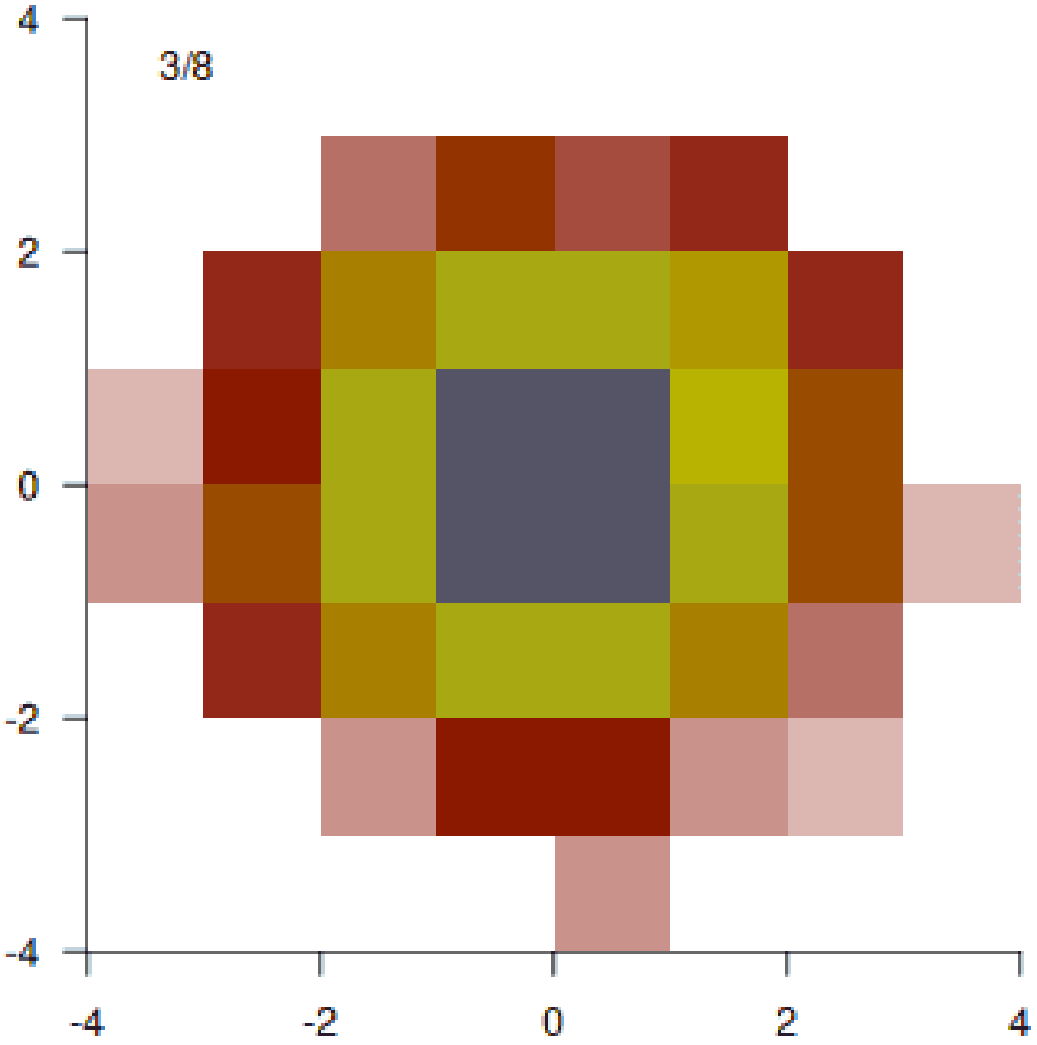
Manual invocation of image sequence



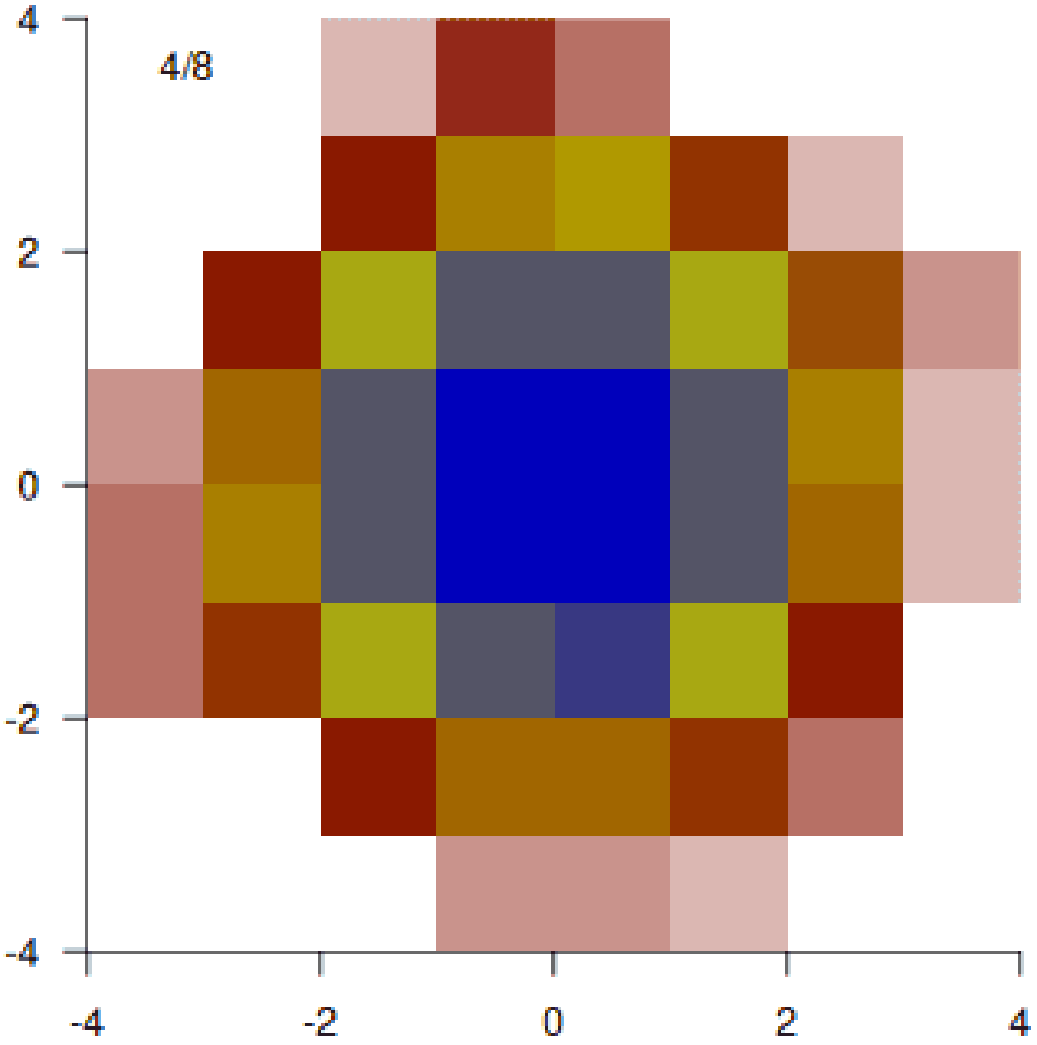
Manual invocation of image sequence



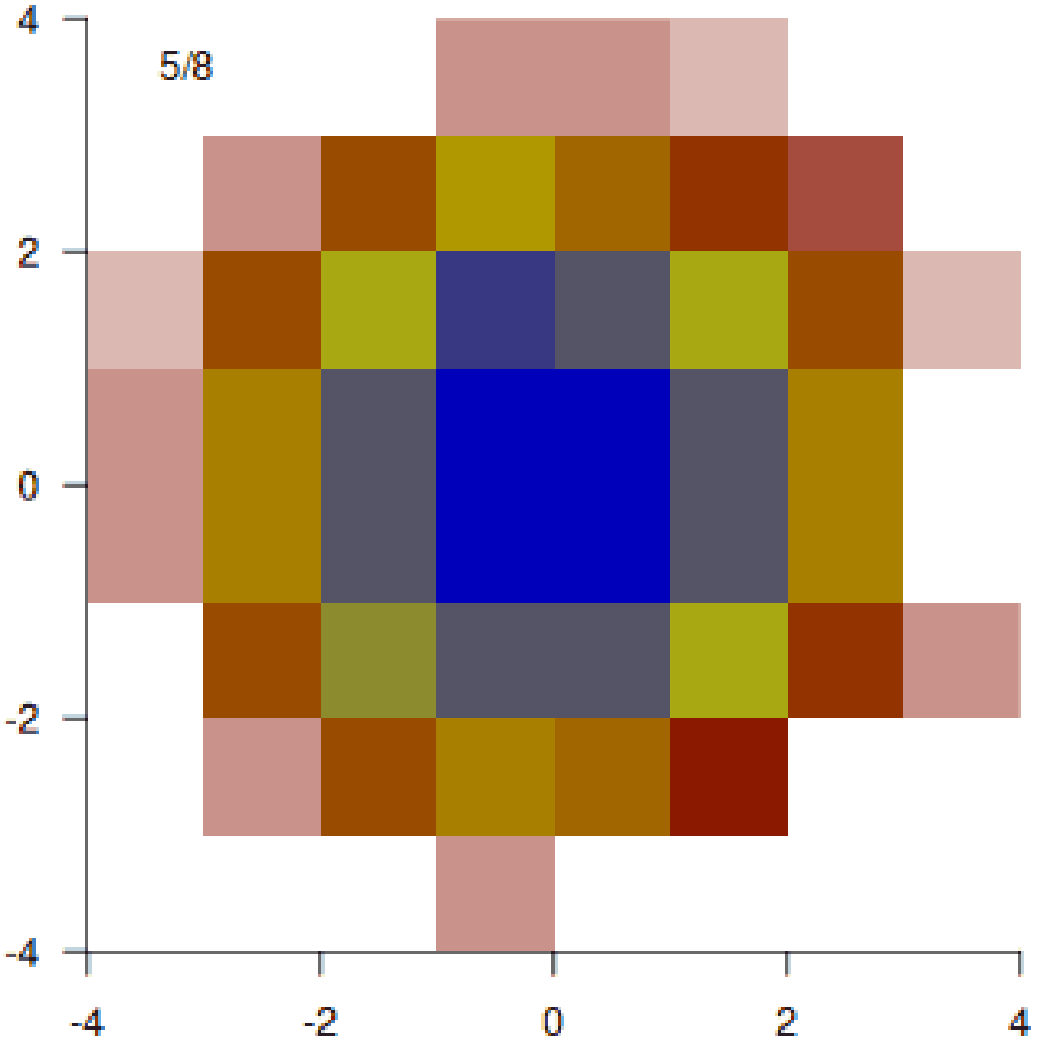
Manual invocation of image sequence



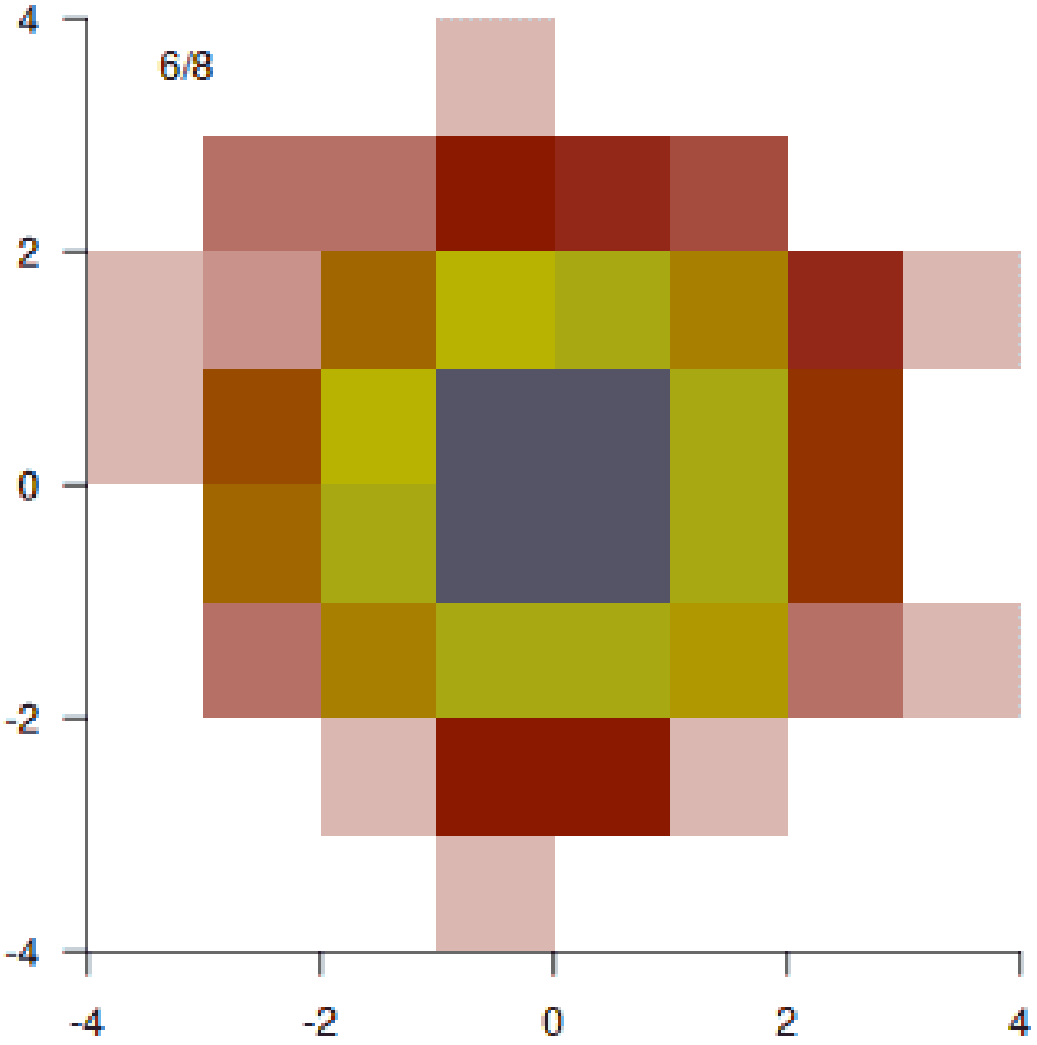
Manual invocation of image sequence



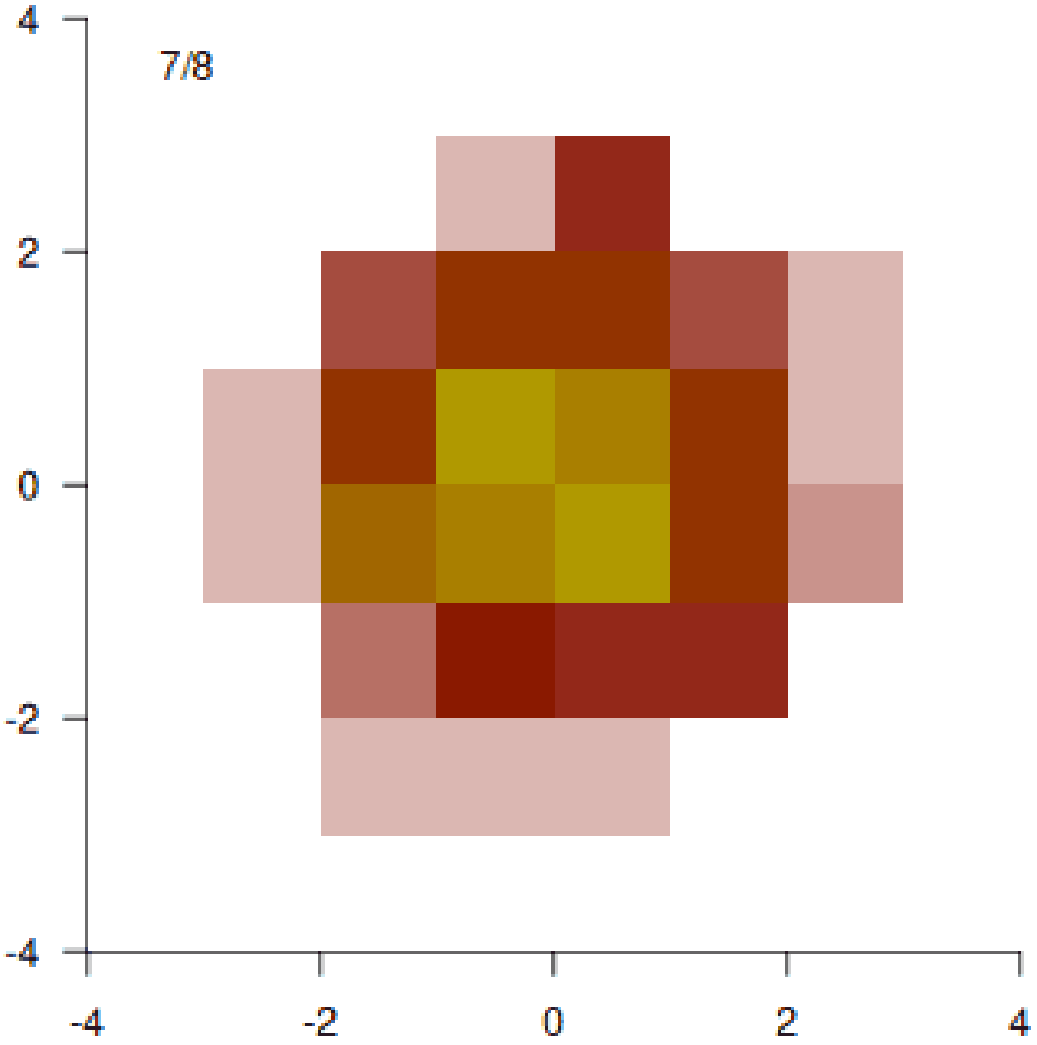
Manual invocation of image sequence



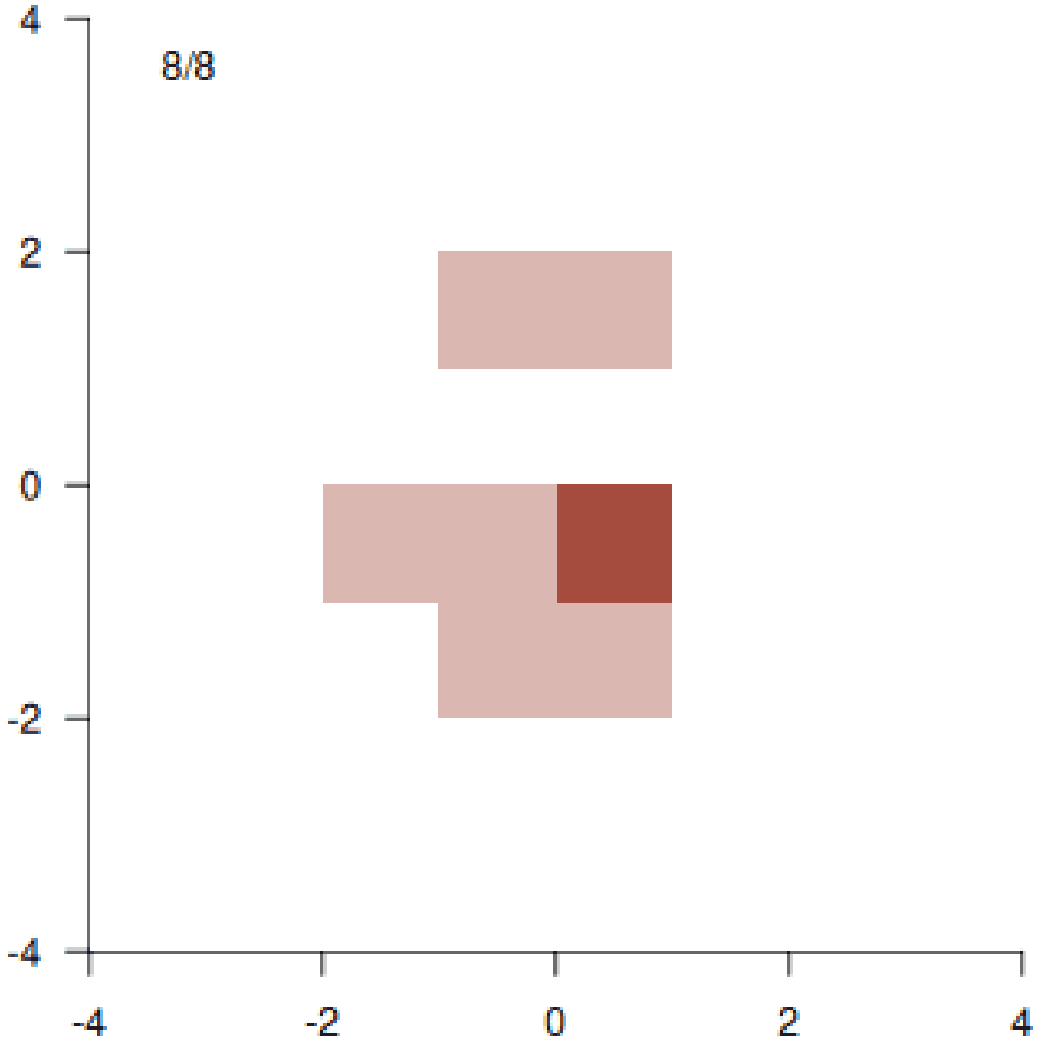
Manual invocation of image sequence



Manual invocation of image sequence



Manual invocation of image sequence



Simplest animation: just images

```r
set.seed(1); x <- matrix(rnorm(30000), ncol = 3)
bs <- c(-Inf, -3:3, Inf)
z <- table(cut(x[, 1], breaks = bs),
  cut(x[, 2], breaks = bs), cut(x[, 3], breaks = bs))
xs <- seq(-3.5, 3.5, 1)
z <- log1p(z)  # Colours on a logarithmic scale
mycols <- c("#FFFFFFAA","#881100","#BBBB00","#0000BB")
cs <- colorRampPalette(mycols)(21)

drawFrame <- function(i) {
  par(mar = c(2, 2, 0.5, 0.5))
  image(xs, xs, z[,,i], zlim = c(0, max(z)),
        col = cs, bty = "n", las = 1, asp = 1)
  legend("topleft", paste0(i,"/",dim(z)[3]), bty="n")
}
for (i in 1:dim(z)[3]) { # i in 1:8
  png(paste0("anim-",i,".png"), 320, 320, type="cairo")
  drawFrame(i)
  dev.off()
}
```

Prerequisites for creating GIFs

The `animation` package has facilities for creating GIFs.

Having a working ImageMagick installation is required.

Check: call `convert` from the command line / terminal / console / PowerShell:

```
convert --version
#> Version: ImageMagick 7.1.1-19 ...
#> Copyright: (C) 1999 ImageMagick ...
#> ...
```

This is the sign that the binary was found in the `PATH` variable.

Using saveGIF()

```
drawFrame <- function(i) {  # Draws the i-th slice of a 3D array
  par(mar = c(2, 2, 0.5, 0.5))
  image(xs, xs, [, , i], zlim = c(0, max(z)),
        col = cs, bty = "n", las = 1, asp = 1)
  legend("topleft", paste0(i, "/", dim(z)[3]), bty = "n")
}
```

Run a loop or – even better – (s)apply a function that draws every frame inside `saveGIF()`.

```
library(animation)
saveGIF(sapply(1:8, drawFrame), "anim.gif",
  interval = 0.25,  # Dey between frames
  ani.width = 320, ani.height = 320,
  ani.dev = \(...) png(..., type = "cairo"))
```

Optional: apply gifsicle (36 → 20 kB):

```
system("gifsicle -O3 --colors 32 anim.gif -o anim.gif")
```

Using FFmpeg

GIF animations may take up a lot of space because the GIF video compression algorithm is highly inefficient.

FFmpeg offers opportunity for better-quality plots:

- Write individual frames as files
  - Preferably to the temporary directory `tempdir()`
- Use the `ffmpeg` command to convert the image sequence input to properly compressed output

Video codecs

FFmpeg comes bundled with many video encoders.

Viceo codec: algorithm to compress and decompress video and audio

Popular codecs:

- AV1 – best compression for given quality, used by Netflix
- H265 – highly efficient; used by newer smartphones
- H264 – old but gold, worse compression ratio but supported virtually everywhere, used in WhatsApp, Telegram etc.

Do not use anything else – for compatibility, encode H264!

FFmpeg example

Create 720 files with a 0.5° angle step:

```
td <- tempdir()
sapply(seq(0, 359.5, 0.5), \(j) {
  fn <- paste0(td, "/f", sprintf("%04d", j*2), ".png")
  png(fn, 320, 320, type = "cairo")
  df3D(i = floor(j / 45)+1, theta = j)
  if (j %% 50 == 0) print(j)
  dev.off()
})
```

Run in console as a single line:

```
ffmpeg -y -framerate 30 -pattern_type glob
  -i '/tmp/<YourValueOfTD!>/f*.png'
  -an -c:v libx264 -pix_fmt yuv420p -crf 25
  -preset slower s05-anim.mp4
```

Calling FFmpeg from within R

FFmpeg options:

- `-an` = **a**udio **n**one
- `-c:v` = **c**odec of **v**ideo
- `-crf` = **c**onstant-**r**ate **f**actor = compression strength
  - Higher = lower quality but smaller files
- `-preset slower` = encode more efficiently

Invoke FFmpeg from R with the explicit temporary directory path (make it a single line):

```
mask <- paste0(td, "/frame*.png")
system(paste0("ffmpeg -y -framerate 30
  -pattern_type glob -i '", mask, "'
  -an -c:v libx264 -crf 25 -pix_fmt yuv420p
  -preset slower s05-anim.mp4"))
```

Trailing-tail plot

Time-series plots of multiple variable often look clearer if the evolution is shown in the context of previous observations: which changes are the most recent and what lead up to them?

Possible uses:

- Phillips curve
- Co-integrating relationships
- Systems of equations (even in 3D)

Example of trailing-tail plot

Steps for trail plotting

- Plot the newest dynamics in bold and dark
- Plot the old dynamics in thin and pale
- Label round years (2000, 2004, ...)
- Mark all years (2000, 2004, ...)

Function for trail plotting (1/2)

```
tailFrame <- function(time, x, y, i = NULL, n = 48, ...) {
  if (is.null(i)) i <- length(x)
  dots <- list(...)
  if (is.null(dots$xlim)) dots$xlim <- range(x, na.rm = TRUE)
  if (is.null(dots$ylim)) dots$ylim <- range(y, na.rm = TRUE)
  dots[c("x", "y")] <- list(NULL, NULL)
  dots$bty <- "n"
  do.call(plot, dots)
  abline(v = pretty(x), h = pretty(y), lty = 2, col = "#00000044")
  inds <- max(i-n, 1):i # These indices are plotted in a bolder and darker line
  m <- length(inds)
  cols <- colorRampPalette(c("#00000033", "#000000FF"), alpha = TRUE)(m) # Darker
  ↪  colours
  wds <- exp(seq(log(1.5), log(4), length.out = m)) # Thicker lines towards the end
  segments(x0 = x[inds[-1]], y0 = y[inds[-1]], x1 = x[inds[-m]], y1 = y[inds[-m]], col =
  ↪  cols[-1], lwd = wds)
  if (i > 1) lines(x[1:i], y[1:i], col = cols[1], lwd = 1) # Lines for old obs: thin and
  ↪  pale

  # Adding features for integer years
  is.int <- abs(time - floor(time)) < 0.0001
  int.time <- round(time[is.int])
  ti <- which(is.int & (time <= floor(time[i]))) # Indices of previous periods (no
  ↪  future)
  cexs <- pchs <- rep(1, length(int.time))
  cexs[int.time %% 4 == 0] <- sqrt(2) # Mark every 4th year with a larger point
  cexs[int.time %% 8 == 0] <- sqrt(3) # Mark every 8th year with an even larger point
  pchs[int.time %% 2 == 0] <- 16 # Mark every 2nd year with a filled circle
  ycols <- rainbow(length(int.time), v = 0.7, end = 0.65) # Different colours for
  ↪  temporal evolution
```

Function for trail plotting (2/2)

```
  # Plot integer year points up until the chosen time moment
  if (length(ti) > 0) {
    s <- length(ti)
    points(x[ti], y[ti], cex = cexs[1:s], col = ycols[1:s], pch = pchs[1:s], lwd = 2)
  }

  # Plot marked year labels up until the chosen time moment
  marked.years <- unique(floor(int.time/4)*4)
  myi <- which(int.time %in% marked.years) # Marked point indices
  ycols <- rainbow(length(int.time), v = 0.7, end = 0.65)

  # These indices = until the chosen time moment
  ryrs <- round(time*4) / 4 # Division by powers of 2 is lossless
  these.myi <- which((ryrs %in% marked.years) & (ryrs <= time[i]))
  if (length(these.myi) > 0) {
    s <- length(these.myi)
    ii <- myi[1:s]
    for (j in 1:length(these.myi))  # Not vectorising to create overlaps
      textWithHalo(x = x[these.myi[j]], y = y[these.myi[j]], labels = int.time[ii[j]],
      ↪  pos = 3, font = 2, col.halo = "#FFFFFFEE", hscale = 0.001, vscale = 0.005)
  }
  return(invisible(NULL))
}
```

Encoding an MP4 of trailing-tail animation

```
d <- read.csv("phillips-france.csv")
d$Date <- seq(1990, by = 0.25, length.out = nrow(d))
d$dCPI <- c(NA, diff(log(d$CPI)))

td <- tempdir()
for (i in 1:nrow(d)) {
  png(paste0(td, "/pc", sprintf("%04d", i), ".png"),
   ↪  1280, 720, pointsize = 24)
  tailFrame(x = d$A, y = d$dCPI*12*100, time =
   ↪  d$Date, i = i, xlab = "Unemployment rate", ylab
   ↪  = "Inflation rate, %")
  if (i %% 25 == 0) print(i)
  dev.off()
}
system(paste0("ffmpeg -y -framerate 4 -pattern_type
 ↪  glob -i '", paste0(td, "/pc*.png"), "' -an -c:v
 ↪  libx264 -pix_fmt yuv420p -crf 25 -preset slower
 ↪  s05-anim-2.mp4"))
```

Any questions on 3D visuals and making films?

Further reading

- How to make figures and presentations that are friendly to colour-blind people (4-page poster only)
- Bongard puzzles
  - 300+ Bongard problems online
- Modern digital image formats
- How JPEG compression works
  - Teddy Tablante (Branch Education)
  - Dr. Mike Pound (Computerphile)
  - Nipun Ramakrishnan (Reducible)
- A parody of an enthusiastic ffmpeg user
- Video formats, codecs, and containers
- Basics of video compression + why video glitches happen (Captain Disillusion, 02:21)

Thank you for your attention!