Plotting in R

Sam Wilks

To start

We will cover a range of topics from the very basics up until the relatively sophisticated. To help give some context I've labelled sections according to importance:

- Basic plotting: Important ***cough*** examinable ***cough*** basics..
- Extended plotting: Getting a bit more advanced, might pop up a little in the exam.
- Advanced plotting: Good to know, but won't be examined.

The basics

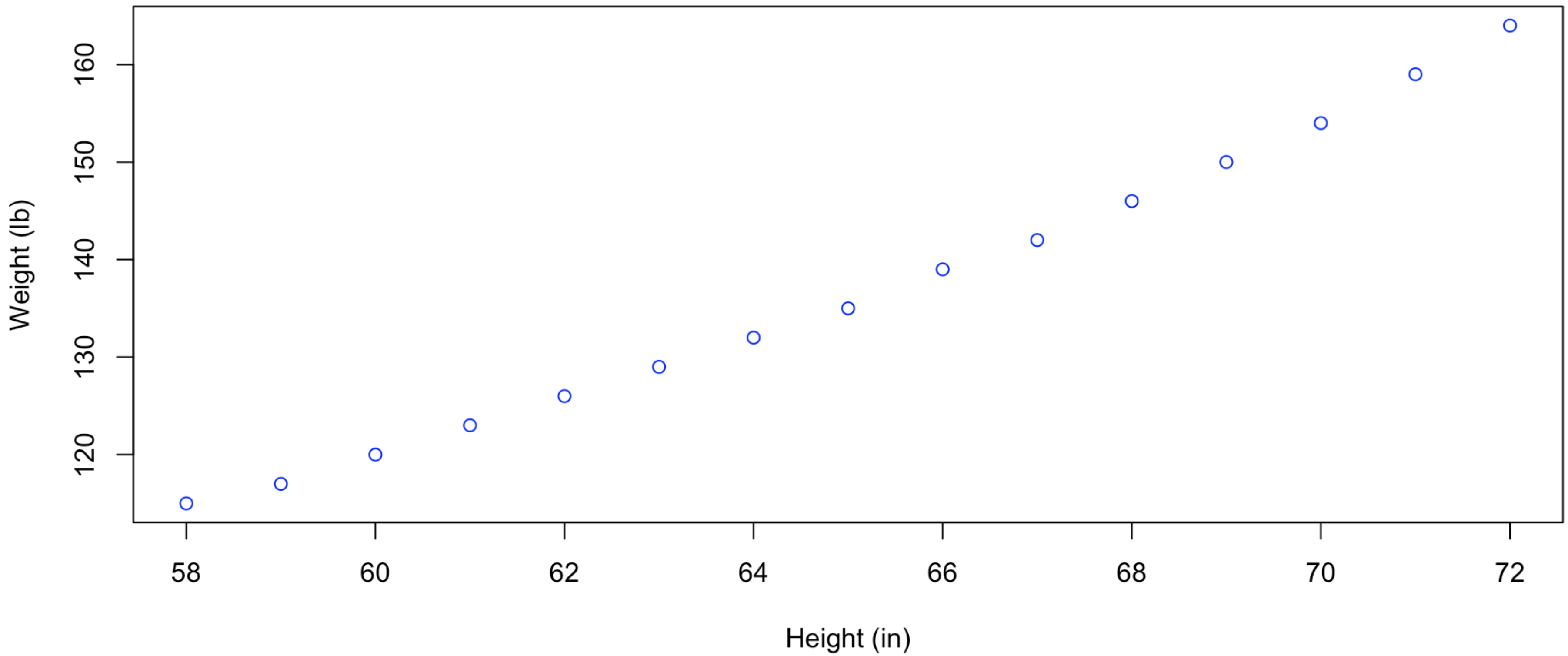
Basic plotting functions

There are several types of basic plotting functions in R, which one you want to use depends on what you want to visualise…

Visualising the relationship between two variables?

The scatterplot function `plot()`..

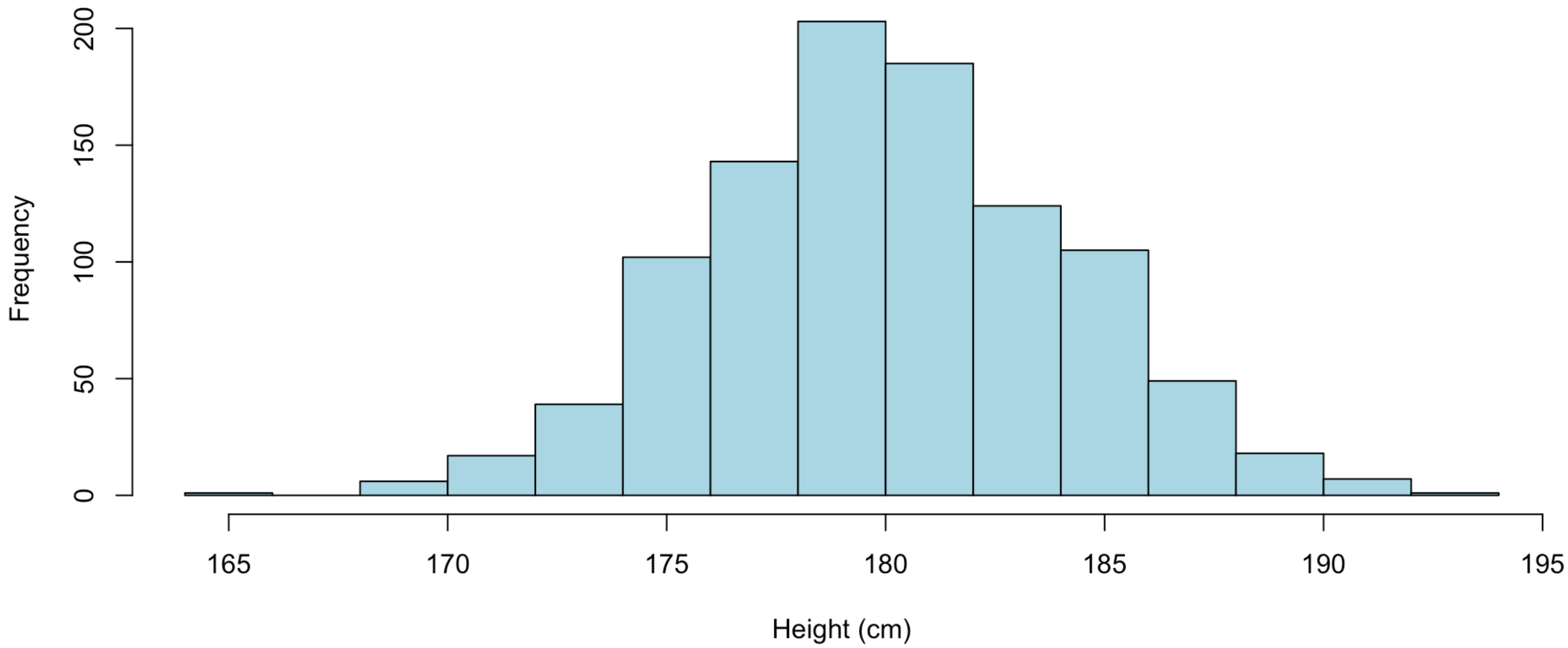
```
plot(x = your_x_values, y = your_y_values)
```



Visualising the distribution of data?

The histogram function `hist()`..

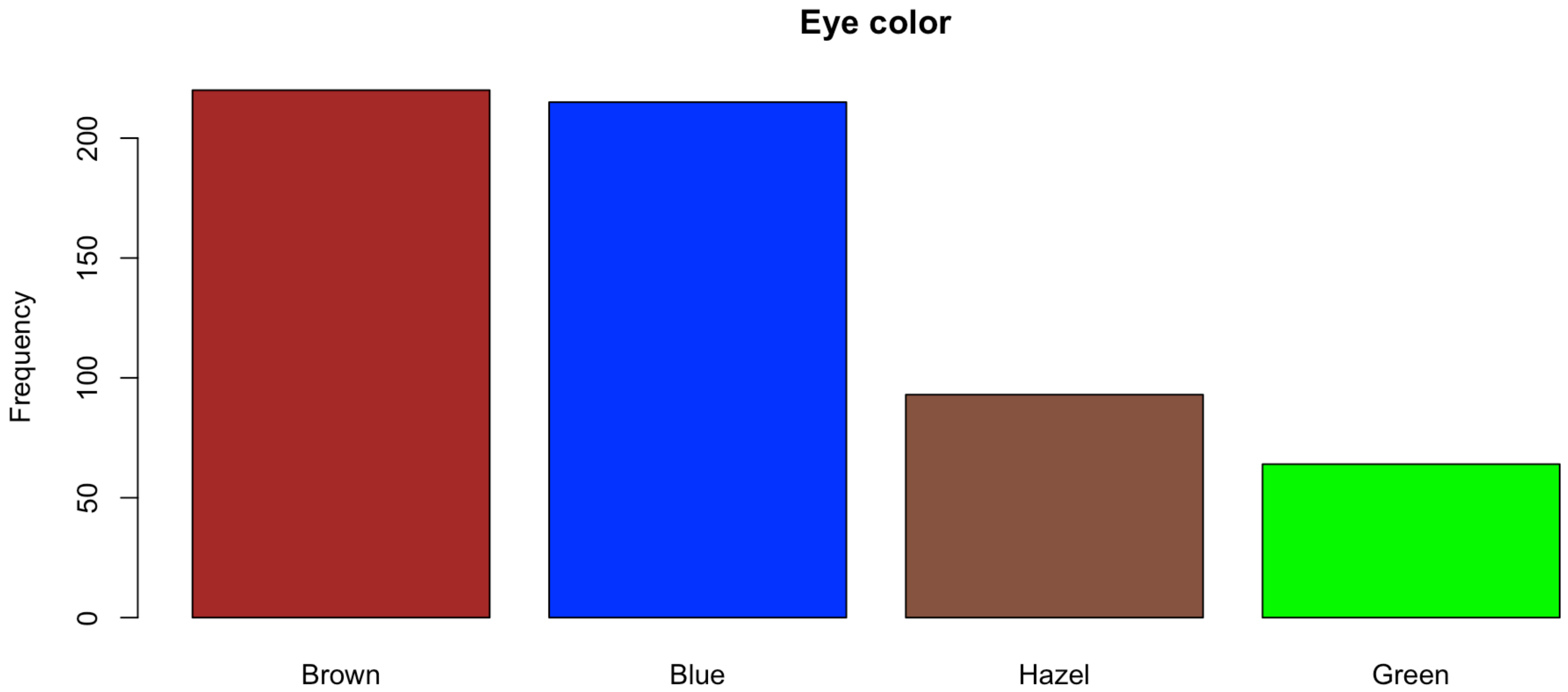
```
hist(your_data)
```



Comparing numbers or proportions between different groups?

The bar plot function `barplot()`..

```
barplot(your_data)
```



And many more…

- `boxplot()`
- `pie()`
- `smoothScatter()`
- `heatmap() # Stats package`
- `vioplot() # vioplot package`

Where to find plotting functions?

- The basic ones are included in the `graphics` package.
- It is a base package, therefore loaded by default so usually* no need to explicitely load it with `library(graphics)`.
- For a complete list of functions use `library(help = "graphics")`. (Although note that some plotting functions are in other base packages like `stats` or provided in packages that must be installed).

*Can you think of a situation where you would want to call to the graphics package explicitly? (i.e. `graphics::plot()`)

High level functions

These are "high level" functions, i.e. they call a lot of other functions underneath but are often quicker to use.

Calling one of them will open a new graphics device/plotting window by default.

Usually the format that you need to supply your data is simple but it will vary between plots, check the **help** pages.

(For example `hist()` just needs your raw data as a vector and will automatically calculate bin sizes for you, but `barplot()` expects a vector of heights for each bar.)

Some side notes..

Although often you use plotting functions to draw a plot, you can also often collect useful outputs from these functions…

```
# Collect the x axis midpoints of each bar into a vector "midpoints".
midpoints <- barplot(height = c(brown=4, blue=2, green=5))

# Collect information on bins used from the hist function.
hist_data <- hist(x = rnorm(1000))
```

Although the `plot()` function is mostly used for scatterplots it can actually do many things depending on the type of data you give it.

```
plot(LifeCycleSavings)        # Paired scatter plots
plot(co2)                     # Time series
plot(density(rnorm(1000)))    # Density plot
plot(Titanic)                 # Mosaic plot
plot(hclust(dist(USArrests))) # Dendrogram
```

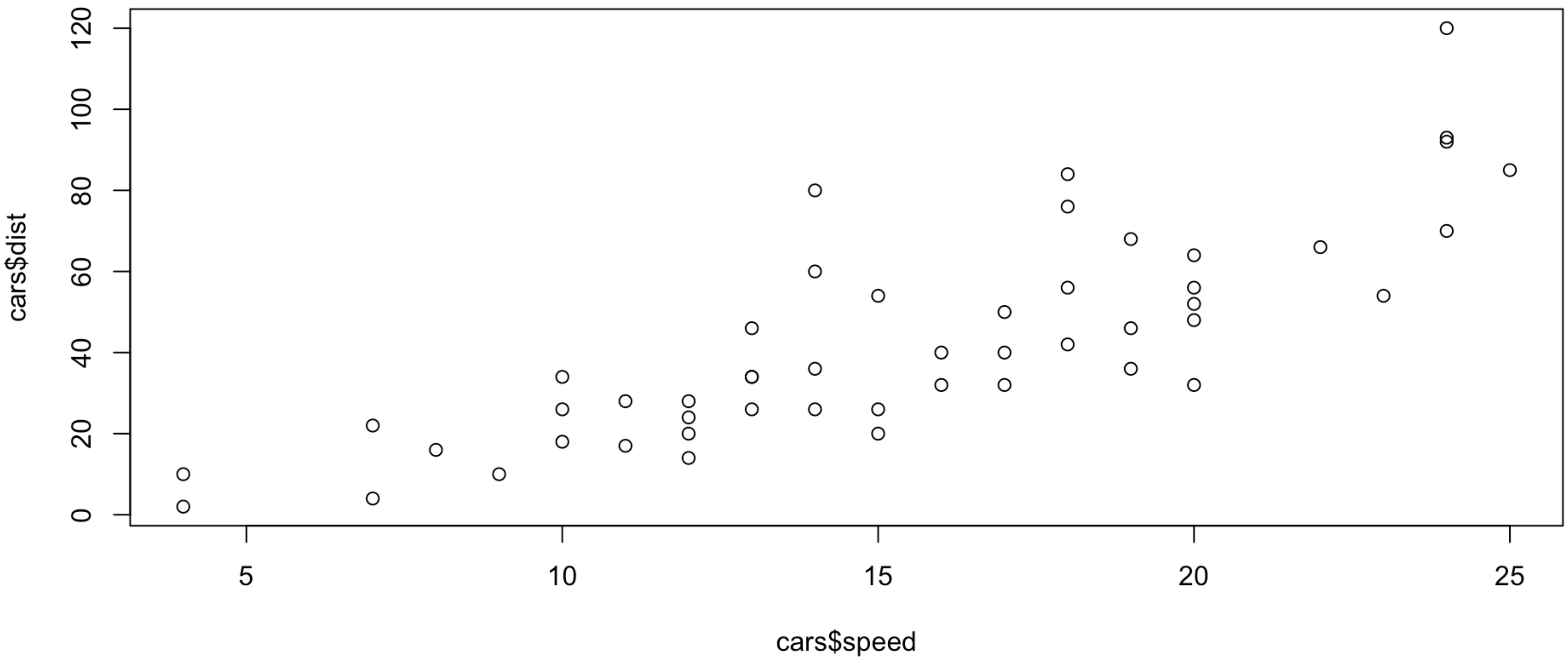
This is why the best help page for a scatter plot is actually found under the less generic `plot.default()` function.

Customising your plot

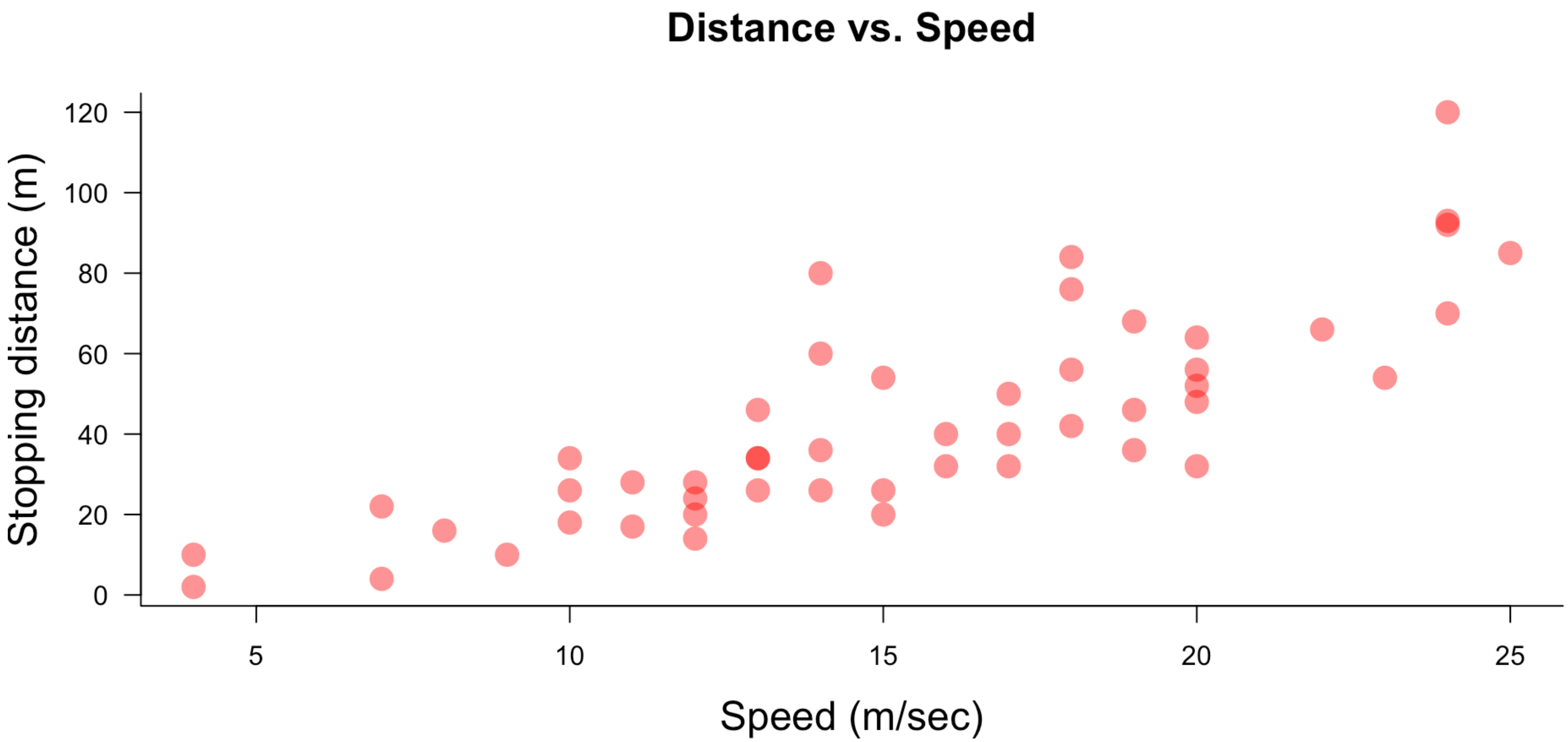
Some things to think about

- **Sizes**: Are the points a good size? Are lines a good width? Are any labels and text clearly visible?
- **Axes**: Are they helpfully labelled? Do the axes ranges make sense? Should they start from 0? Should they *match another plot you want to compare against*? Would it make sense to log them?
- **Colors**: Could you add some color to help interpret the plot or even just to make it look nicer?
- **Clarity**: Does the plot effectively visualise the data? Are there too many points overlapping? Would it be helpful to add some `jitter()`? Are you trying to fit two much on one plot where it would be better to split into several?

Building up from a simple example..



..to a more complex example



Getting help

Some options can be found directly in the `plot()` help page, or (more helpfully) following the links to the help page for `plot.default`:

```
?plot
?plot.default

plot(x = cars$speed,
     y = cars$dist,
     main = "Speed vs distance",      # A plot title
     xlab = "Speed (m/sec)",          # A label for the x axis
     ylab = "Stopping distance (m)") # A label for the y axis
```

But the options listed directly in the function help pages are sometimes limited…

Controlling graphics with par()

`par` is the function used to set and query graphical parameters used for plotting (`?par`). Depending on the function, many parameters from par can be passed directly into it:

```
plot(x = cars$speed,
     y = cars$dist,
     main = "Speed vs distance",
     xlab = "Speed (m/sec)",
     ylab = "Stopping distance (m)",
     pch = 16,                 # Set point type (see ?pch)
     cex = 2,                  # Set size of points
     cex.lab = 1.5,            # Set size of axis labels
     cex.main = 1.5,           # Set size of title
     las = 1,                  # Set orientation of axis labels
     col = "red",              # Set colour of points
     bty = "l")                # Set type of axis box
```

Setting margin sizes

Some parameters listed in `?par` cannot be passed to functions but must always be set directly with a call to the par function before plotting, for example `mar` for setting the plot margins.

Confusingly, the following code will run fine but **won't change the margins**!

```
plot(cars, mar = c(5,5,2,2)) # Wrong!
```

You have to do:

```
# _First_ set the plot margins (in inches) with a call to par
par(mar = c(5,5,2,2))

# _Now_ you can do the plot...
plot(cars)
```

Querying par settings

The default (or current setting) for any parameter can be queried using the parameter name in quotation marks:

```
# Query the current setting for the plot margins
par("mar")
```

```
## [1] 5.1 4.1 4.1 2.1
```

Some common graphical parameters

```
cex                          # Change point size
cex.main, cex.lab, cex.axis  # Change title/axis title/label size
lwd                          # Change line width
lty                          # Change line type (solid, dotted..)
pch                          # Change point type
col                          # Change plotting color
las                          # Change orientation of axis labels
```

Bonus things

Transparent colours can be created using the function `rgb()`. eg. red with an opacity of 0.5 would be:

```
rgb(1,0,0,0.5)
```

Multiple different colours can be achieved using one of the R color palettes for example `rainbow()`, or you can define your own palette easily with the `colorRampPalette()` function.

```
n_points <- 100
x_data   <- seq(from = 0, to = 2*pi, length.out = n_points)

plot(x = x_data,
     y = sin(x_data),
     cex = 2, pch = 16,
     col = rainbow(n = n_points))
```

See Also

Often a very useful section of the help pages is **See Also**. It points you in the direction of similar or related functions - if you're doing something more complex it's worth taking a look, there may be a more appropriate built in function to help you do it.

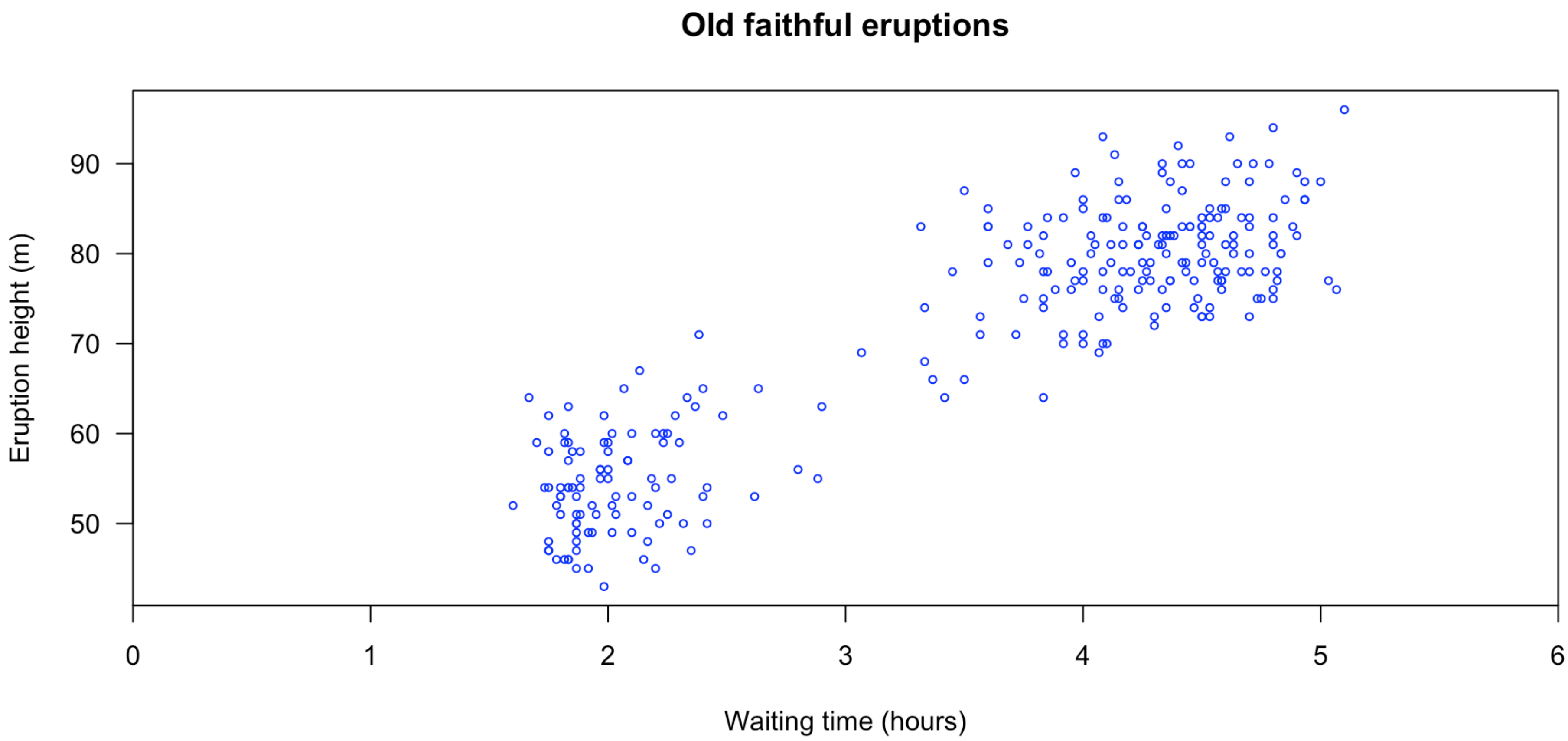
For example under See Also for the plot() function it says:

"For thousands of points, consider using `smoothScatter()` instead of `plot()`."

Under See Also for smoothScatter it leads you to `densCols()`, another very useful function for plotting scatterplots with many points.

Exercise 1

Reproduce this plot using the data from the `faithful` dataset:



Exercise 1 - solution

```
plot(x    = faithful$eruptions,
     y    = faithful$waiting,
     xlab = "Waiting time (hours)",   # Set x label
     ylab = "Eruption height (m)",    # Set y label
     main = "Old faithful eruptions", # Set title
     xlim = c(0,6),                   # Specify x limits
     xaxs = "i",                      # Don't pad x limits
     col  = "blue",                   # Set colour of points
     cex  = 0.6,                      # Decrease size of points
     las  = 1)                        # Set axis labels horizontal
```

Controlling axis limits

Axis limits are controlled with `xlim` and `ylim`. They require a vector of length 2, giving the lower and upper limit.

Make sure you understand what's going wrong or right with the following attempts to set the x axis limit…

```
xlim = faithful$waiting        # Wrong
xlim = 0:6                     # Wrong
xlim = (0,6)                   # Wrong
xlim = c(0,6)                  # Correct
xlim = range(0:6)              # Correct
xlim = range(faithful$waiting) # Correct
```

By default R pads the x and y axis, to suppress this and use exactly the limits you set with no padding, set `xaxs = "i"` or `yaxs = "i"`.

Adding to plots

Some principles

R uses "Pen and paper" plotting (you build up plots incrementally but can't remove things once they've been plotted).

If you know you want to add extra things to a plot later, you need to make sure you leave room for them when you first set up the plot, by increasing `xlim` or `ylim` for example.

Adding to an existing plot

Several functions are provided for adding elements to an already existing plot:

```
lines()   # Add lines
points()  # Add points
rect()    # Add a rectangle
polygon() # Add a more complex polygon
text()    # Add text
mtext()   # Add margin text
title()   # Add a title and/or axis labels
box()     # Draw a box around the plotting region
axis()    # Add axes
```

Unlike `plot()` and `hist()` etc, these functions won't open a new plotting space by default.

A simple example drawing confidence intervals

```r
# Define x and y variables
dist  <- cars$dist
speed <- cars$speed

# Do the initial plot
plot(x = speed, y = dist,
     main = "Stopping distance against speed",
     xlab = "Speed (m/s)",
     ylab = "Stopping distance (m)")

# Fit linear model
cars_model <- lm(dist~speed)

# Get fit and confidence intervals
x_vals   <- seq(from=min(speed), to=max(speed), length.out = 100)
conf_int <- predict(cars_model, newdata = data.frame(speed = x_vals), interval = "c")

# Plot fit line
lines(x = x_vals,
      y = conf_int[,1],
      lwd = 2)

# Plot confidence intervals
lines(x = x_vals,
      y = conf_int[,2],
      lty = 2)

lines(x = x_vals,
      y = conf_int[,3],
      lty = 2)
```

An example with polygon

```r
# Plot some random points
x_data <- runif(100)
y_data <- runif(100)

plot(x = x_data,
     y = y_data,
     xlim = c(-0.2,1.2),
     ylim = c(-0.2,1.2))

# Find indices of points forming the convex hull using chull()
chull_data <- chull(x_data, y_data)

# Plot a polygon using the convex hull coordinates
polygon(x = x_data[chull_data],
        y = y_data[chull_data],
        border = "red",
        col = rgb(1,0,0,0.2))
```

Other ways to add to a plot

Some functions have an argument that determines whether they plot a new plot or add to an existing plot.

For example `boxplot()` has `add` to control if the boxplot should be done in a new plot or **add**ed to an existing plot. The same is true for `hist()`.

```
## An example taken from the boxplot help page...

boxplot(len ~ dose, data = ToothGrowth,
        boxwex = 0.25, at = 1:3 - 0.2,
        subset = supp == "VC", col = "yellow",
        main = "Guinea Pigs' Tooth Growth",
        xlab = "Vitamin C dose mg",
        ylab = "tooth length",
        xlim = c(0.5, 3.5), ylim = c(0, 35), yaxs = "i")

boxplot(len ~ dose, data = ToothGrowth, add = TRUE,
        boxwex = 0.25, at = 1:3 + 0.2,
        subset = supp == "OJ", col = "orange")

legend(2, 9, c("Ascorbic acid", "Orange juice"),
       fill = c("yellow", "orange"))
```

Yet another way to add to a plot..!

Many high level functions like `plot()` check the `par` setting `new` to see if they should start a new plot before plotting. Confusingly, setting `par(new = TRUE)` suppresses the default behaviour and means a new plot **won't** be opened.

I wouldn't recommend this approach but it's good to know that it exists…

```
# Do a scatter plot
plot(x = cars$speed, y = cars$dist)

# Suppress behaviour to start a new plot
par(new=TRUE)

# A new histogram plot now just plots on top!
hist(cars$speed)
```

Redefining the plotting space

Sometimes you have a plot that you want to add something to but using a different scale (usually for the y axis). In these cases you need to redefine the plotting space.

For this you can use the function `plot.window()`.

This redefines the axes of the plots for any **future** functions you call. You won't notice any immediate effect and it won't change what's already been plotted.

A demonstration of plot.window()

```
# Set some data to compare the normal cumulative probability and probability density functions
n_data          <- seq(from=-3, to=3, length.out=1000)
cumulative_prob <- pnorm(n_data)
prob_density    <- dnorm(n_data)

# Plot the cumulative probability first
par(mar = c(5,5,2,5)) # Increase margin size on the right to allow space for extra axis
plot(x    = n_data,
     y    = cumulative_prob,
     xlim = range(n_data),
     ylim = range(cumulative_prob),
     col  = "blue",
     xlab = "n",
     ylab = "Cumulative probability",
     type = "l",
     lwd  = 2, las  = 1, col.axis = "blue")

# Redefine the plotting space using plot.window()
plot.window(xlim = range(n_data),           # We want to keep the same x axis range
            ylim = c(0, max(prob_density))) # But we will increase the y axis range

# Now we can plot the data according to the new axis limits..
lines(x   = n_data,
      y   = prob_density,
      col = "red",
      lwd = 2)

# This is very confusing so we have to remember to put a new y axis and axis label!
axis(side = 4, las = 1, col.axis = "red")
mtext(text = "Probability density", side = 4, line = 3)
```

Plotting a legend

Unsurprisingly the function to plot a legend is `legend()`! It can be positioned either using a keyword ("topleft", "bottomright", etc.) or using exact x and y coordinates.

If you want to plot a legend (or anything else) outside of the plotting space you need to include the argument `xpd = TRUE` (e**xp**an**d** outside plotting space).

```
legend(x      = "topleft",                        # Position with keyword
       legend = c("Probability density",          # Set legend text
                  "Cumulative probability"),
       fill   = c("blue",                         # Set legend colors
                  "red"),
       bty    = "n")                              # Don't draw a box
```

Adding text to a plot

To add text to a plot simply use the `text()` function.

You specify the location on the x and y axis and the text to be plotted.
(remember to set `xpd=TRUE` if you want it to show up outside the plotting space)

A useful argument is also `pos`, for deciding how you want the text to be position relative to the coordinates you gave (below, left, top or above).

Mathematical notation in R plots

Sometimes you want to include mathematical notation in a plot, for this you need the `expression()` function. The way you use it is not immediately obvious but it is generally not too difficult.

The full list of expressions you can call are listed under `?plotmath`. To link text or expressions together you use the "*" operator.

There are also other options such as `bquote` if you are interested.

Some mathematical notation examples

```r
expression(mu*pi*alpha) # Lowercase symbols
expression(Mu*Pi*Alpha) # Uppercase symbols

expression(log[23])      # Subscript
expression(22^{x})       # Superscript

expression(frac(2*x^{alpha}, y[3*beta]))  # Fractions

expression("This is the alpha symbol: "*alpha)  # Including text
```

Under the hood

Building a plot from the ground up

High level functions like `plot()` call a lot of lower level functions to build the plot in one go. In most cases this saves you a lot of typing but sometimes you don't want all of that stuff.

One approach - as we did before - is to pass arguments to `plot()` to change the defaults, or to stop it drawing certain aspects of the plot (for example setting `axes=FALSE` to suppress drawing of axes).

Another approach is to use the lower level functions to build the plot yourself…

Underneath the plot function

The following series of lower level functions reproduces most of the behaviour of the `plot()` function, but at each stage you generally have more control.

```
# Open a new plotting device
plot.new()

# Define the plotting space
plot.window(xlim = range(cars$speed),
            ylim = range(cars$dist))

# Plot the x and y axes
axis(side = 1) # x axis
axis(side = 2) # y axis

# Draw a box around the plotting area
box()

# Label the axes and title
title(main = "Speed vs stopping distance",
      xlab = "Speed",
      ylab = "Stopping distance")

# Plot the data
points(x = cars$speed,
       y = cars$dist)
```

Controlling output

Outputting as different file formats

By default plots are output to the plotting window, but you can output to other ``devices''.

Vector graphics (scalable):

- `pdf()`
- `svg()`

Bitmap graphics (not scalable but can have a smaller file size and be quicker to load):

- `png()`
- `jpeg()`

Outputting syntax

The approach is always the same:

```
# Open the plotting device and set any options
pdf(file   = "~/Desktop/example_plot.pdf",
    width  = 6,
    height = 6)

# Do any plotting that you wish
plot(x = 1:10,
     y = 1:10)

# Close the device (always same command)
dev.off()
```

Choosing the device size

The size you choose for your plotting device will influence how the plot appears. It will be as though you had resized the built in viewer to the size you specify.

A handy value to retrieve from par is "din" - the current width ad height of the plotting device in inches:

```
par("din")
```

Set the width and height of your pdf or other output to this size and it will look similar to how it looked in RStudio.

Tip! If you know you want to output to a specific pdf dimensions, output the plot to the pdf before you start making minor adjustments since things like the legend positions may look quite different.

Plotting problems

If you run into trouble!

```
# Close all plotting devices
graphics.off()
```

Plotting multiple subplots

There are several approaches, I would recommend using `layout()` since it is quite flexible and robust.

Very simply, you set up a matrix that represents the order in which you want to do the subplots:

```
layout_matrix <- matrix(1:4,
                        nrow = 2,
                        byrow = TRUE)
print(layout_matrix)
```

```
##      [,1] [,2]
## [1,]    1    2
## [2,]    3    4
```

Layout example 1

```r
# Make the layout matrix
layout_matrix <- matrix(1:4,
                        nrow = 2,
                        byrow = TRUE)

# Supply it to the layout function
layout(mat = layout_matrix)

# Do your plotting, the plot will move on to the next plotting
# space everytime plot.new() is called
# (and it is called as part of most high level plotting functions)
plot(1:10, main = "Plot1")
plot(1:10, main = "Plot2")
plot(1:10, main = "Plot3")
plot(1:10, main = "Plot4")
```

More complex layouts

Layout can also deal with plots that take up more space than others, any adjacent identical numbers will be merged into one plotting space for example this matrix would create one plot on the left and two on the right:

```
layout_matrix <- matrix(c(1,1,2,3), nrow = 2)
print(layout_matrix)
```

```
##      [,1] [,2]
## [1,]    1    2
## [2,]    1    3
```

Layout example 2

```
# Make the layout matrix
layout_matrix <- matrix(c(1,1,2,3), nrow = 2)

# Supply it to the layout function
layout(mat    = layout_matrix,
       widths = c(1,0.6))

# Do your plotting, this time the first plot will take up twice
# the height of the others
plot(1:10, main = "Plot1")
plot(1:10, main = "Plot2")
plot(1:10, main = "Plot3")
```

Going further

ggplot2

A popular graphics package that aims to reduce the amount of coding required to produce effective graphs.

The syntax can be quite different from normal R plotting though so there is a bit of a learning curve.

One advantage is that the default plots often look a lot nicer! However it does nothing that you can't do using the basic plotting functions.

Interactive plotting

There are several options for adding interactivity to plots, take a look at:

- **manipulate** - A basic package for creating plots with user controls within R studio.
- **Shiny** - A more advanced package that allows making webpage-based plots where user controls are defined.
- **plotly** - A multi-platform library that allows making plots with interactivity such as labels on mouse-over and zooming etc.

3D plotting

- **rgl** The standard package for making 3D interactive plots in R. There are straight-forward 3D equivalents for all the R plotting functions, making usage quite easy but you have to first install X11, a (massively outdated!) windowing environment - now XQuartz on mac.
- **plotly** This also has the functionality to produce interactive 3D plots.
- Packages such as **lattice** and **scatterplot3d** also provide some functions to produce static 3D plots without interactivity.

Further resources

- **Quick-R** Has many useful simple and more advanced examples for plotting in R.
- **Advanced R** A website and book by Hadley Wickham that doesn't cover plotting but has excellent in depth explanations of the different features of R if you really want to understand why things work the way they work!
- **R-Cheat-Sheet** This reference sheet lists most basic commands used for common applications. (cran.r-project.org/doc/contrib/Short-refcard.pdf)