# Workshop 2: Graphics

The purpose of this exercise is to tour the graphics capabilities of base R.

## Probability distributions

Before we tackle plotting, it will be useful to learn how to draw some random numbers, so that we can quickly create things to plot (and this will also be useful later on when adding stochasticity to simulation models).

R has many built in probability distributions. For each distribution, you can access the density, probability, and quantile functions, or draw randomly from that distribution. For example, for a uniform distribution, `dunif`, `punif`, `qunif` and `runif` accomplish each of these functions respectively. Similarly, for a Normal distribution, you'd use `dnorm`, `pnorm`, `qnorm`, `rnorm`. For example, to draw one random number from a uniform distribution, you could use:

```
runif(n=1)
```

or to draw 5, you could use

```
runif(n=5)
```

To draw 10 random numbers from a normal distribution (with a mean of zero and standard deviation of 1), you could use:

```
rnorm(n=10)
```

To draw 10 random numbers from a normal distribution with a mean of 3 and standard deviation of 2, you could use:

```
rnorm(n=10, mean=3, sd=2)
```

1. Use the `mean` and `sd` function to confirm that, for a large number of draws, the `rnorm` command does, indeed, give you a mean and standard deviation *very close* to the specified values.

## Basic plotting

At the most basic level, you can use the `plot` command to generate a simple plot of some points. For example:

```
plot(x=1:5, y=c(1,5,4,2,3))
```

There are a number of options that you can pass to the `plot` command. For example, compare the above to:

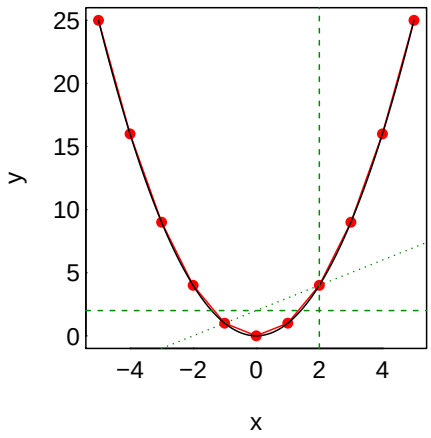
```
plot(x=1:5, y=c(1,5,4,2,3), pch=16, col='red', las=1,
     xlab='Speed', ylab='Distance')
```

Other useful commands for *adding to existing plots* are the `points` and `lines` commands. It can sometimes be useful to create an initially empty plot, and then to add points and lines after. To do this, you also need to specify the $x-$ and $y-$limits of the plot. For example:

```
plot(NA, las=1,
     xlab='Speed', ylab='Distance',
     xlim=c(1,5), ylim=c(0,10))
points(x=1:5, y=c(1,5,4,2,3), pch=16, col='red')l
```

1. Use an `apply` statement of your choice (see previous workshop) to generate a set of values for a parabola and then create a plot of these points.
2. Use the `lines` command to add lines connecting consecutive points.
3. Now try adding a smooth parabola (rather than the straight line segments from the previous step) using the `curve` command. You will need to use the option `add=TRUE` in order to *add* the curve to the existing plot - otherwise `curve` will create its own new plot.
4. Use the `abline` command to add a horizontal, vertical, and sloped line to the plot.

Your final figure should be some variant (e.g., contain all the components) of this figure:




## Arrows and polygons

It is often useful to add measures of uncertainty to plots. Two helpful commands for this are `arrows` and `polygon`.

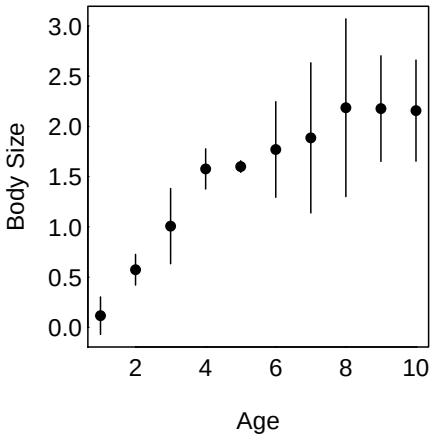
Using the above probability commands, let's generate some random data. Suppose we have 10 estimates of body size for individuals of age 1 through 10 (and that body size grows logarithmically with age):

```
age <- 1:10
body.size <- rnorm(10, mean=log(age), sd=0.1)
```

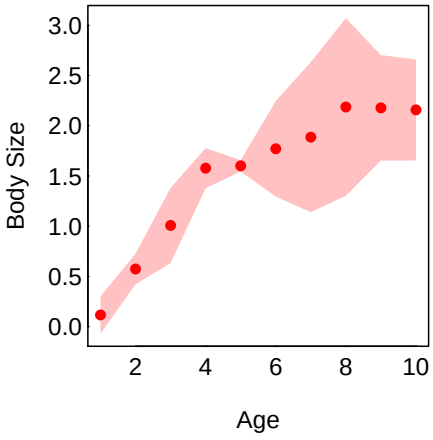
Note that, by passing in a vector of values in for the `mean` argument in the `rnorm` function, each random draw will be drawn with a different mean. In this case, body sizes will, on average, get larger for each age. Now, let's suppose these are estimates and there is some uncertainty surrounding these estimates (I'm just making up "data" here...):

```
uncertainty <- runif(10)
```

1. Plot these data, and use the `arrows` command to add error bars that show the uncertainty. Your figure should look something like this:




2. Now try creating a shaded polygon, using the `polygon` command to show the uncertainty, rather than error bars (or show both!). Your figure should look something like this:



# 2D plots

Often times, it useful to generate 2D plots. The `image` command is useful for that. It requires a matrix of data (which is just a 2D vector) for input and I often find it confusing to get the orientation right. Plotting small simple matrices can help you orient. E.g.,

```
mm <- matrix(c(0,0,1,2), ncol=2)
```

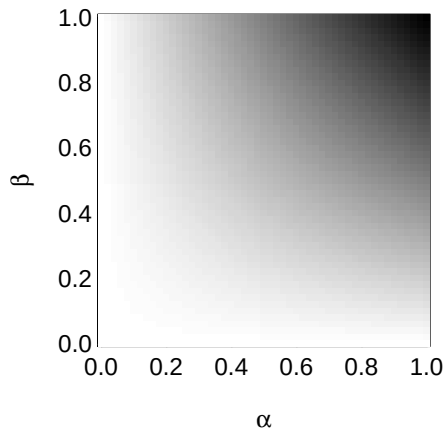
and we can then plot using the default orientation and colour palette

```
image(mm)
```

We can then re-order the matrix so that, in the plot, the cells are in the same location as displayed in the matrix (e.g., bottom left cell is bottom left corner of matrix). And let's change to a grayscale as well.

```
mm.reordered <- t(mm[nrow(mm):1,])
image(mm.reordered, col=gray(10:0/10), las=1)
```

Now, if we were doing something interesting, it could be that we vary two parameters, say $\alpha$ and $\beta$, and measure some response (e.g., population size). I could then present that data in an (much higher resolution) image as:




1. Play around with image, and make something creative! Can you draw a smiley face?

# Vector graphics

Before discussing multi-panel plots and figure margins, it is important to discuss figure formatting. In general, you should always use **vector graphics** formats (e.g., pdf). These graphics do not pixelate as you enlarge them. In general, it is a good habit to write these files directly from within R, rather than to save the figure manually from the console. The latter process means that the configuration of your current R console will affect the dimensions of the saved file (so can change each time you re-size R or Rstudio, for example). To write directly to pdf, for example, you can do the following:

```
pdf('my_figure.pdf', height=3, width=3)
## plotting commands here
```

```
dev.off()
```

This will save a figure of the specified height and width in your working directory with filename my_figure.pdf. The final `dev.off()` command closes the call to the pdf so, without that, each subsequent plot command you execute will continue to write to your open pdf!

The resultant figure may look a little funny (e.g., big margins, huge text, compressed plot). You can change the height and width, but you can also change the margins using the `par` command. `par` has a huge number of options for configuring graphics (use `?par` to learn more). A helpful set to start with is:

```
par(oma=c(0.1,0.1,0.4,0.4), mar=c(3, 3, 0.1, 0.1), mgp=c(2,0.2,0))
```

Each plot has a set of inner margins (those that go around each panel) and a set of outer margins (those that surround the set of all panels). `mar` (which stands for "margin") governs the former and `oma` (which stands for "outer margin"), the latter. The best way to learn how these works is trial and error. Try playing around with some of the values in the `par` command below to see what they do:

```
pdf('my_figure.pdf', height=3, width=3)
par(oma=c(0.1,0.1,0.4,0.4), mar=c(3, 3, 0.1, 0.1), mgp=c(2,0.2,0))
plot(1:10, xlab='x-label', ylab='y-label', las=1)
dev.off()
```

# Multi-panel plots

There are a number of ways to make multi-panel plots. One of the most flexible is the `layout` command. `layout` requires a matrix as input, and this matrix specifies the locations of the panels (the location of the 1 corresponds to the first panel, the location of the 2 to the second panel, and so on). For example,

```
layout(matrix(1:2, nrow=1, ncol=2))
plot(1:10, pch=16, col='red')
plot(1:10, pch=16, col='blue')
```

will plot two adjacent panels, whereas,

```
layout(matrix(1:2, nrow=2, ncol=1))
plot(1:10, pch=16, col='red')
plot(1:10, pch=16, col='blue')
```

will plot two stacked panels. You can get *very* creative with your panel configuration by using larger and more complex matrices. For example, have a look at the following matrix:

```
plot.mat <- matrix(c(0,1,0,4,
                     2,2,0,4,
                     0,0,3,4), nrow=3, ncol=4, byrow=TRUE)
```

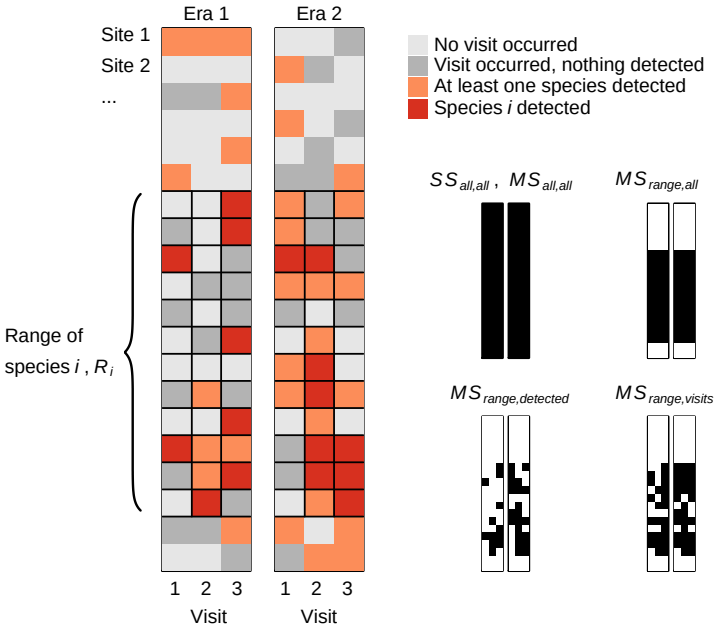
and now, look at the locations of the following panels:

```
layout(plot.mat)
plot(1:10, pch=16, col='red')
plot(1:10, pch=16, col='blue')
plot(1:10, pch=16, col='green4')
plot(1:10, pch=16, col='purple')
```

You can even specify heights and widths for the rows and columns of the matrix to further refine your figure. E.g., see below. Note, I've put this inside a pdf command and added a call to par, so that I can control margins. You'll have to open the resultant pdf to see this one.

```
pdf('my_figure.pdf', height=6, width=6)
par(oma=c(0.1,0.1,0.4,0.4), mar=c(3, 3, 0.1, 0.1), mgp=c(2,0.2,0))
layout(plot.mat, widths=c(1,1,1,3), heights=c(1,3,2))
plot(1:10, pch=16, col='red')
plot(1:10, pch=16, col='blue')
plot(1:10, pch=16, col='green4')
plot(1:10, pch=16, col='purple')
dev.off()
```

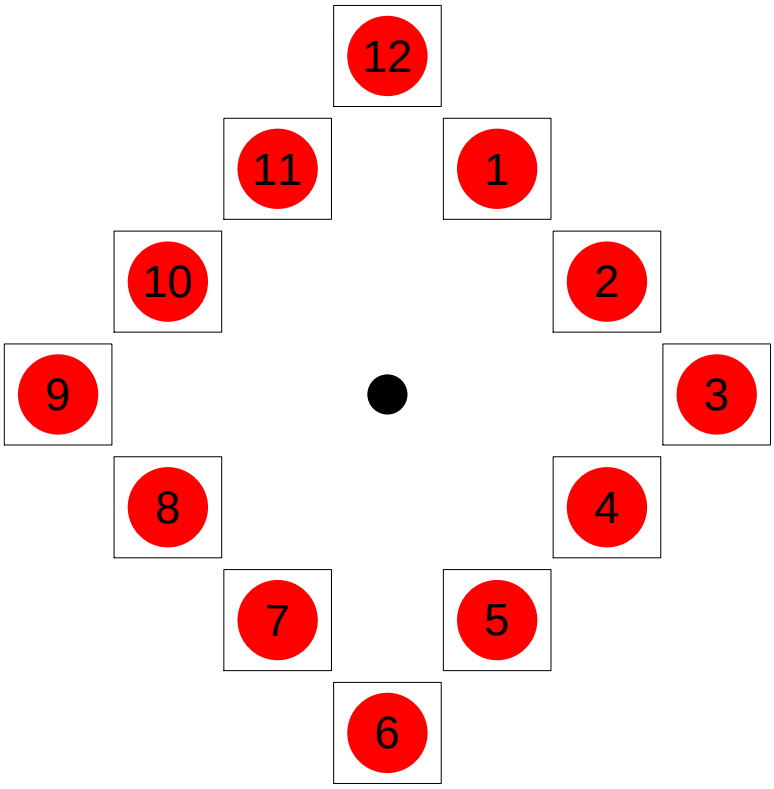
Here's a figure I recently made for a project that used almost entirely syntax we have covered so far (e.g., `image` and `layout`). In addition, `legend` lets you add a legend and `mtext` is useful for adding text in specified locations. Nothing was done outside R.




The layout and par command for this figure are:

```
layout(matrix(c(1,0,2,0,3,4,4,4,4,4,4,4,
                1,0,2,0,3,0,0,0,0,0,0,0,
                1,0,2,0,3,0,5,6,0,7,8,0,
                1,0,2,0,3,0,0,0,0,0,0,0,
                1,0,2,0,3,0,9,10,0,11,12,0),
              5, 12, byrow=TRUE),
       heights=c(1,0.4,1.5,0.2,1.5),
       widths=c(0.1,0,0.25,0.05,0.25,0.3,0.07,0.07,0.3,0.07,0.07,0.3))
par(oma=c(2,4,0,0), mar=c(0.1, 0.1, 1, 0.1),
    mgp=c(2,0.2,0), tcl=0, cex.axis=0.7, cex.main=0.7)
```

1. Using the `layout` command and multiple `plot` commands (or a single plot command wrapped up inside a function!), create a figure of a clock (something similar to the below figure - but feel free to do better):




2. Make a multi-panel figure containing *at least* four different panels (but more if you want), to present whatever you want (fake data, real data, cool patterns, etc). The goal here is to impress me with your base R graphics capabilities by making something beautiful (and not to worry about actual visualization of data). This should be **more** impressive than the clock from above.