STAT0023 Workshop 3: simple programming in **R**

In this workshop you'll start writing your own scripts to produce some simple functions. You will need the slides / handouts from the lecture which explain the key concepts: these can be downloaded from the STAT0023 **Moodle** page, under the 'Course overview and useful materials' tab.
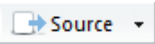
1 Setting up

For this workshop, in addition to the lecture slides / handouts you will need the following, all of which can be downloaded from the 'Week 3' tab of the **Moodle** page:

- These instructions.

- The **R** scripts `ConvertTemp.r`, `MandelbrotFunctions.r` and `MandelbrotDemo.r`. You should save these to the appropriate place on your N: drive, as you created it during Workshop 1 — refer to Section 2 of the notes for Workshop 1 if necessary — and remember to make sure that your web browser doesn't change the filenames when downloading them (Section 1.3 of the Workshop 1 notes).

Having downloaded the files, start up **RStudio** and use the <u>S</u>ession menu to set the working directory appropriately, as usual (refer to your notes from Workshop 1 if necessary).

1.1 Defining and saving functions

In this workshop, you'll get some practice at writing **R** scripts and functions from scratch. To define a function in **R**, the following method is recommended:

- Use the <u>F</u>ile menu in **RStudio** to open a new **R** script (<u>F</u>ile → New <u>F</u>ile → <u>R</u> Script). You will get a blank pane in which to type your function.

- Save the file with a sensible name in your current working directory. For example, in the first exercise below you're asked to write a function called `taylor.sin`. You might choose to save the **R** script for this function in a file called `TaylorSin.r`.

- Click the `Source` button in **RStudio**, to run the script. If your script is correct, you won't see anything on screen but you will now be able to use your function. If it contains syntax errors, **R** will let you know! Sometimes however, the error messages are not directly related to the cause of the problem. At this stage, the cause is usually something like:

  - Mismatched brackets e.g. a missing opening or closing '(' or ')', or an opening '(' that gets closed with a ']', or something like that. If you have used curly brackets '{' and '}' to define blocks of code, make sure these are all balanced. Use the

<u>C</u>ode menu in RStudio to help with this: in particular, if you highlight a sequence of lines and then choose the `Reindent Lines` or `Re`<u>`f`</u>`ormat Code` options, RStudio will try and arrange the code logically according to how you've *actually* defined it (rather than how you *think* you've defined it!). This can help you to spot where the problem lies, because the indentation will look wrong.
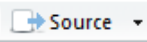
– Mismatched quotes `""`. The syntax highlighting in RStudio is helpful here: make sure that things that are supposed to be inside quotes (like file names) are in an appropriate colour and that this colour has not "escaped" to the rest of your code!

Be aware that the cause of the problem may be on an earlier line than the one where R starts complaining. Also, be aware that if can source your function definition without error, this does *not* mean that your function is correct: it merely means that there is nothing obviously wrong with the basic syntax of your function.

2   Power Series for the Sine Function

Let's start with a simple function. The Taylor expansion of $\sin(x)$ about 0 is

$$\sin(x) = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \dots$$

In order to investigate the accuracy of this power series, create the following function (called `taylor.sin`) using the approach described in Section 1.1 above, and click the [Source ▾] button to load it into R.

```
taylor.sin <- function(x) {
#
# Taylor approximation to sin(x)
#
  x - (x^3)/6 + (x^5)/120
}
```

This function computes the Taylor expansion up to the third term. Now you can use your function. Type the following commands at the Console prompt:

```
> taylor.sin(0)
> taylor.sin(pi/4)
> sin(pi/4)
> taylor.sin(pi/4)-sin(pi/4)
```

The last command is the error between the Taylor approximation and the accurate value for $\sin(\pi/4)$. So far the approximations are reasonable, but as the value of x gets larger, so the approximation gets worse . . .

```
> taylor.sin(pi/2)
> taylor.sin(pi)
> taylor.sin(2*pi)
```

Create a plot to compare the two functions:

```
> theta <- pi*(-200:200) / 150  # Fine grid of values from -4*pi/3 to 4*pi/3
> plot(theta, sin(theta), type="l", col="red") # Exact value
> lines(theta,taylor.sin(theta), col="blue", lty=2) # Approximation
```

Now change your function `taylor.sin` so that the expansion is calculated up to the fourth term. Repeat the above commands to see how much more accurate the method is.

3   The '...' Argument

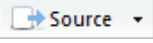In the lecture, the following function was given

```
convert.temp <- function(degrees.F, plot.wanted=TRUE, ...) {
  degrees.C <- (degrees.F - 32) * 5/9
  if (plot.wanted) plot(degrees.F, degrees.C, ...)
  degrees.C
}
```

Open the file `ConvertTemp.r` that you downloaded earlier, by clicking on it in the 'Files' tab of the lower right-hand pane in RStudio. This file contains the `convert.temp` function definition, together with some comments about its use and purpose: these comments follow the general layout of an R help page (you might not like this, but you're just going to have to get used to it!). Specifically, the comments tell you:

- What is the purpose of the function;

- What its arguments are, including *what kinds of objects* they are e.g. that `degrees.F` is a *vector*. This is not just fluffy waffle: it tells the user that if they want to use the function successfully, the input `degrees.F` must be a vector.[1] Almost all of the help pages in the core R package will provide similar information: be aware, and use it! Unfortunately, the help pages for the add-on libraries (the ones that you access using the `library()` command) are of variable quality. You can't complain, though: you get more than you pay for.

- What is the 'value' of the function i.e. what kind of object `dog` would be if you typed `dog <- convert.temp()` with some appropriate arguments.

---

[1]In R, to find out whether an object is a vector you can use the `is.vector()` function which returns either `TRUE` or `FALSE`.

- Any other side effects (in this case, the optional creation of a plot on the current graphics device).

Click the `⤷ Source ▾` button to load this function into R. Then, at the Console prompt, create a vector of Fahrenheit temperatures

```
> fahrtemps <- c(0,20,32,61,70,80,90,100,110,212)
```

Now do the following:

1. Call the function `convert.temp` using these temperatures (i.e. `convert.temp(fahrtemps)`)

2. Call the function again, this time with the additional argument `type="b"`.

3. Add a title by calling the function with yet another additional argument `main="Celsius against Fahrenheit"`.

4. One more time: call the function again, but now so that the points and lines appear in red (you might want to look at the help page for `par()` here — or look at some examples from previous workshops to find out how colours are specified).

Can you see how the argument '...' works and why it is useful?
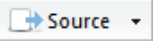
3.1   Adding more detail

You should have the `convert.temp()` function definition open in RStudio. This means that you can edit the function to 'improve' it. So: change the line

```
  if (plot.wanted) plot(degrees.F, degrees.C, ...)
```

to

```
if (plot.wanted) {
  plot(degrees.F, degrees.C, ...)
  if (min(degrees.C) <= 0) {
    abline(h=0, col="blue")
    abline(v=32, col="blue")
  }
  if(max(degrees.C) >= 100){
    abline(h=100, col="red")
    abline(v=212, col="red")
  }
}
```

Reload the function into R (click the `⤷ Source ▾` button ...) and call it again. Can you see what the function `abline` does and how the `if` statements work?
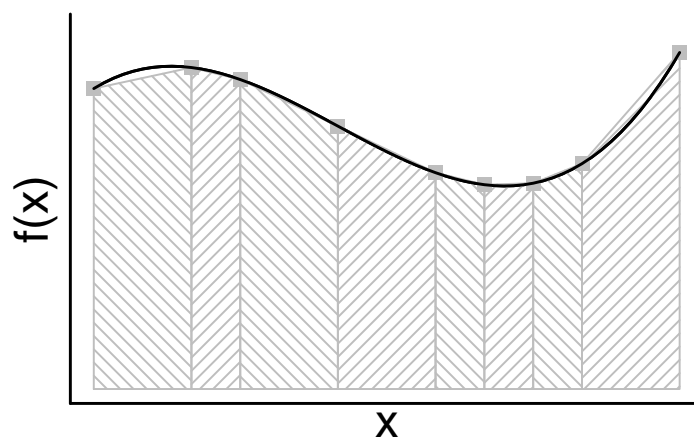
Figure 1: Illustration of the trapezium rule

4   Numerical Quadrature

A common mathematical problem is to integrate a function f between two points i.e. to evaluate $\int_a^b f(x)dx$ (NB we're using the word function in its *mathematical* sense now, rather than the computational sense of the previous section). For many simple functions (e.g. $f(x) = x^2 + 4$), humans can do this using rules learned in school or university, but these rules can't be implemented easily in most computing environments. Moreover, it is common to encounter functions that can't be integrated using standard rules (for example, if you want to find probabilities under the standard normal curve).

An alternative exploits the fact that an integral is just the area under a curve. This area can be calculated approximately by evaluating the function at values $x_0, \ldots, x_n$ to to get $f(x_0), f(x_2), \ldots f(x_n)$, joining up the points with straight lines and calculating the areas of the resulting trapezia as illustrated in Figure 1. This leads to the *trapezium rule* to approximate the integral of f. For equally-spaced x-values, the trapezium rule is

$$\int_{x_0}^{x_n} f(x)dx \approx \frac{h}{2}\left[f(x_0) + 2f(x_1) + \cdots + 2f(x_{n-1}) + f(x_n)\right],$$

where

$$h = \frac{x_n - x_0}{n}.$$

Exercise: we will write an R function called `trapezium`, to approximate an integral using the trapezium rule with equally spaced x values. The R function will have three arguments, as follows:

`v` : a vector containing the values `v[1]`$= f(x_0)$, `v[2]`$= f(x_1), \ldots$`v[n+1]`$= f(x_n)$;

`a` : the lower limit of integration `a`$= x_0$;

b : the upper limit of integration b= x_n.

Use the following code as a template for your function, filling in the gaps where indicated:

```
trapezium <- function(v,a,b) {
 n <- length(v)-1
 h <- ?????      ## as defined above
 intv <- ?????   ## intv is the calculated integral. You may
                 ## need a couple of lines for this, but
                 ## DO NOT USE A LOOP!!! You might want to
                 ## know that the sum() command can be
                 ## used to sum the elements of a vector.
# and return intv
  ????
}
```

We can check the method using a couple of examples for which the answer is known. For example, the value of $\int_0^{10} 1 dx$ can be approximated using trapezium(rep(1,11),0,10) (there are 11 x values, 0,1,2,...,10). How accurate is the answer? Can you explain this?

As another example, we can use the function to evaluate the integral $\int_0^{\pi/2} \sin(x) dx$. In the first instance, this can be done using trapezium(sin((0:6)*pi/12),0,pi/2) (the vector v contains function values from sin(0) to sin($\pi/2$) and the integration limits are defined as a=0 and b=pi/2). Answer the following questions:

1. What value do you get for trapezium(sin((0:6)*pi/12),0,pi/2)?

2. What is the exact answer for this integral? How big is the approximation error from the trapezium rule?

3. It should be clear from Figure 1 that the accuracy of the trapezium rule will improve as the number of evaluation points increases i.e. as n increases (although this comes with an increase in computational cost, obviously). The most human-friendly way of doing this is to change the argument sin((0:6)*pi/12) in the call above to something like sin(seq(0,pi/2,length.out=10)) or sin(seq(0,pi/2,length.out=20)) or sin(seq(0,pi/2,length.out=50)). How fast does your approximation improve as you increase the number of evaluation points? By what fraction does the error decrease as you double the number of evaluation points? Will this increase in accuracy continue forever?

WARNING!!!

You may be tempted to see what happens if you try a *very* large number of evaluation points, such as 1e9. **DO NOT ATTEMPT THIS!!!** The reason is that if you do, R will try to store the evaluation points in a vector containing 10^9 elements. Each element uses 8 bytes of the computer's memory, and the entire vector uses 8 gigabytes. Your computer probably doesn't have this much memory, and it will grind to a halt — possibly for more than an hour.

5 To loop or not to loop?

In the lecture, we discussed the fact that you should avoid loops in R where possible. This section demonstrates the problem.

5.1 How not to calculate a sum

We'll start with a very simple example. Following the procedure described in Section 1.1, define a function called `sillysum()` as follows:

```
sillysum <- function(x) {
  z <- 0
  for (i in 1:x) z <- z+i
  z
}
```

Before you continue: what do you think this function does?
Check that `sillysum(10)` gives the same result as `sum(1:10)`. Then check that `sillysum(1e6)` gives the same results as `sum(as.numeric(1:1e6))` (the 'as.numeric()' is needed to stop R from trying to store the numbers as integers: the largest integer it can store is $2^{31} - 1$). Do you notice the difference in speed? You can quantify this:

```
> system.time(sillysum(1e6))
 [output ...]
> system.time(sum(as.numeric(1:1e6)))
 [output ...]
```

How much slower are the loops?

It doesn't hurt to repeat the warning from the previous section: do *not* be tempted to try even bigger numbers here, because of the memory required to store very large vectors.

5.2 A more challenging example: plotting a Mandelbrot set

The example in this section is considerably more advanced, but it demonstrates clearly the benefits of avoiding loops in R. It's mathematical rather than statistical: it involves the computation of the *Mandelbrot set* (Figure 2). This set is constructed using the function $f(z) = z^2 + c$ where z and c are complex numbers. The function is iterated starting from $z_0 = 0$ to give a sequence $z_1 = f(z_0) = z_0^2 + c$, $z_2 = f(z_1) = z_1^2 + c = (z_0^2 + c)^2 + c$, and so on. The Mandelbrot set is defined as *the set of values $\{c\}$ for which this sequence remains bounded*. Remarkably for such a simple concept, the set has an extremely complex structure — indeed, many of you have probably encountered it before as a well-known example of a *fractal set* (i.e. a set that has structure at all scales).

To draw the Mandelbrot set, the 'obvious' approach is to set up a fine grid of values $\{c = c_x + ic_y\}$ in the complex plane and, for each pair (c_x, c_y), to determine whether the sequence
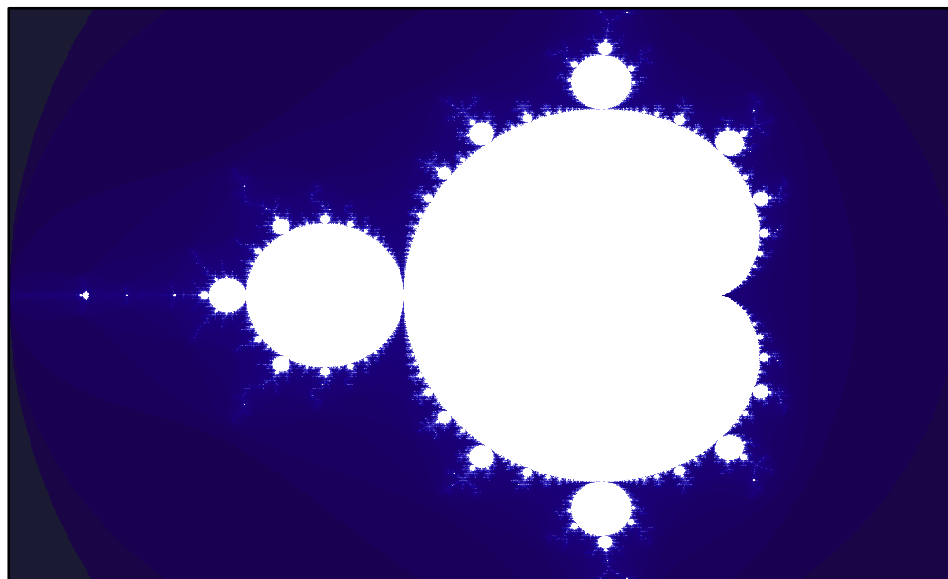
Figure 2: Representation of the Mandelbrot set.

$z_0, z_1, z_2, z_3, \ldots$ remains bounded. Then draw the grid, with each point (c_x, c_y) shaded black (say) if the sequence remains bounded and white otherwise.

This is easier said than done, however! One useful thing to know is that if there is some k with $|z_k| \geq 2$ then the sequence definitely does *not* remain bounded. This means that if you start at $z_0 = 0$ and calculate the sequence $z_0, z_1, z_2, \ldots$ for a particular value of c, and at some stage you produce a value with modulus greater than 2, then this value of c is *not* in the Mandelbrot set. In practice of course, the sequence is infinite so you can't calculate every element. An operational procedure, therefore, is to calculate a *finite* version of the sequence $z_0, z_1, z_2, \ldots, z_K$ where K is some large number: if $|z_K|$ is still less than 2 then your value of c *may* be in the Mandelbrot set, although you can't be sure. This is called the *escape time algorithm*.

When using the escape time algorithm, it is common to indicate not just whether or not a grid node is in the set, but instead to draw a 'map' of the first integer k for which $|z_k| > 2$. Thus, Figure 2 shows a part of the complex plane (the axes are omitted because they spoil the view!), with $-2 \leq c_x \leq 1$ and $-1 \leq c_y \leq 1$. This part of the plane has been divided into a regular grid of dimensions 1000×1000, and for each grid node the escape time algorithm has been used with a maximum iteration number of $K = 1000$: at each grid node, the colour indicates the first value of k with $|z_k| > 2$. Dark blue and purple values indicate low values of k, with white indicating $K = 1000$ so that the corresponding points may be in the Mandelbrot set.

In the script `MandelbrotFunctions.r` that you downloaded earlier, there are two functions that implement the escape time algorithm. Open the script and look through the definition of the function `Mandelbrot1()`. This is a direct translation into R of the 'pseudo-code' given on the Wikipedia page at https://en.wikipedia.org/wiki/Mandelbrot_set. You should

not attempt to understand all the details here! Notice the following, however:

- The function has several arguments: `nx` and `ny` to control the numbers of grid nodes in the x and y directions, `xlim` and `ylim` to set the overall ranges of values for the grid, `max.iter` to control the maximum number of iterations (i.e. the value of K in the algorithm description above), `ColourScale` so that the user can specify their own colours for the image. `xlim` and `ylim` have default values corresponding to the image shown in Figure 2; `max.iter` has a default value of 1000; and `ColourScale` has a default value of `NULL`: this is a special value, discussed further below. The '...' argument is also used, because this enables the user to define additional graphics parameters to the `image()` command later on.

  By setting default values for most of the arguments, the function can be called very easily. Try it:

  ```
  > Mandelbrot1(50,50)
  ```

  NB do not try any bigger numbers than 50 here! (you'll see why in a moment). If you do this, after a few seconds you should see a plot which is just about recognisable as a *very* coarse-resolution approximation to Figure 2: it's so coarse (50 grid nodes in each direction) that you can still see the grid structure in the image.

- The `NULL` default value for `ColourScale` is used because the code to set up the default colour scale takes up a couple of lines, and it would be hard to read if if this code were included as part of the argument specification. Instead, the function starts by testing whether `ColourScale` is `NULL` (i.e. the default value): if so, the built-in colour scale is explicitly computed, and otherwise the user-supplied value is used. This is a common way to set non-trivial default values (note that you shouldn't worry about the precise code used to set up the colour scale here — it was produced by trial and error!).

- The main calculations are in a pair of *nested* `for` loops:

  ```
  for (i in 1:nx) {
    for (j in 1:ny) {
      ...
    }
  }
  ```

  The '`i`' loop here deals with one row of the grid at a time; and, for each row, the '`j`' loop moves along the columns. By doing this, we deal with each grid node in turn, which makes the calculations easy to think about.

- For each grid node, some relevant quantities are initialised (`x <- 0` and `y <- 0`, corresponding to the real and imaginary parts of z_0 in the algorithm as described above; and `iter <- 0` to initialise the iteration counter i.e. k in the algorithm description). Then a `while` loop is used:

```
while (x^2 + y^2 < 4 & iter < max.iter) {  # NB escape clause!
  ...
  iter <- iter + 1
}
```

There are *two* conditions in the `while()` statement here: the first is `x^2 + y^2 < 4`, which corresponds to the condition $|z_k| < 2$ in the escape time algorithm (`x` and `y` get continuously updated so that, at each iteration, they represent the real and imaginary parts of z_k). The second condition is `iter < max.iter`: this will be `TRUE` unless the iteration count exceeds the maximum number of iterations allowed. The two conditions are linked by the '`&`' sign which means "and": the entire expression is `TRUE` only if *both* of the individual conditions are `TRUE`, and `FALSE` otherwise. The `while` loop will therefore continue until *either* $|z_k| \geq 2$, *or* the maximum number of iterations is reached (because the iteration count `iter` is incremented on each pass through the loop). The condition `iter < max.iter` is an 'escape clause' ensuring that the loop will always terminate and you don't get stuck on the infinite staircase (if you don't know about this staircase, see the lecture slides / handouts).

You shouldn't worry about anything else in the function: there are some tricks to get the colour scale to display sensibly, and to return the matrix result invisibly (you didn't really want to see a 50×50 matrix on your screen when you typed `Mandelbrot1(50,50)`, did you?); but these are not particularly important for this workshop.

That seems all very well: but you may have noticed that it took a few seconds to run the `Mandelbrot1(50,50)` command and the results are nowhere near as impressive as Figure 2. To produce such a high-resolution image, a very fine grid is needed: the grid resolution for Figure 2 is 1000×1000 rather than 50×50. Whatever you do, resist the temptation to deal with this by typing `Mandelbrot1(1000,1000)`! Think about it: a 1000×1000 grid contains 400 times as many nodes as a 50×50 grid, so you can expect the computations to take 400 times as long.[2] On my laptop, the 50×50 computations take around 2.75 seconds: the 1000×1000 computations might take about 18 minutes, therefore.

This is where function `Mandelbrot2()` comes in. It implements exactly the same algorithm as before, but this time there are no `for` loops: the entire grid is updated at each operation, simply by using R's facilities for componentwise multiplication of matrix elements (here representing grid nodes). The key to this is to create additional matrices for the quantities `x` and `y`, as well as for the co-ordinates of the grid nodes themselves; and, additionally, to keep track of which grid nodes have already 'escaped' from the region $|z| < 2$ on an earlier iteration. This 'tracking' is done by creating a matrix called `NotDone`, with elements that are `TRUE` for those grid nodes that have not yet escaped and `FALSE` otherwise. Then, at each iteration, the calculations are only performed on the subset of grid nodes for which `NotDone` is `TRUE` (as always, using the `[]` construction to select the required subset): this ensures that we don't waste computer time doing calculations for nodes that have already escaped and therefore don't need to be considered further. This technique (the use of a logical matrix to restrict the set of elements considered in a calculation) is sometimes called *masking*.

---

[2]This isn't *quite* correct, because the computations for some grid nodes are quicker than others — but it's roughly right.

Don't worry if you don't understand all of this: it is quite advanced programming, and you aren't expected to get to that level in STAT0023! What you *are* expected to do, however, is to appreciate the benefits of avoiding loops. Try it:

```
> Mandelbrot2(50,50)
```

Do you notice the difference in speed compared with `Mandelbrot1`? On my laptop, it runs more than 10 times faster; and it allows Figure 2 to be produced in under 90 seconds.

If you want to relax a bit and entertain yourself, open the file `MandelbrotDemo.r` and run the last few lines:

```
x11(width=10,height=7)
par(mfrow=c(2,2),mar=c(1,1,2,1)) # 2*2 plot array, narrow margins except at top
Mandelbrot2(500,500,xlab="",ylab="",axes=FALSE,main="Complete Mandelbrot set")
box(lwd=2)
for (i in 1:3) {
  cat("Click 2 points on plot to select zoom region\n")
  ZoomRegion <- locator(2)
  rect(ZoomRegion$x[1],ZoomRegion$y[1],ZoomRegion$x[2],ZoomRegion$y[2],border="yellow")
  cat(paste("Working on region (",ZoomRegion$x[1],",",ZoomRegion$x[2],
            ") x (",ZoomRegion$y[1],",",ZoomRegion$y[2],") ...\n",sep=""))
  Mandelbrot2(500,500,xlim=sort(ZoomRegion$x),ylim=sort(ZoomRegion$y),
                    xlab="",ylab="",axes=FALSE,main=paste("Zoom",i))
  box(lwd=2)
}
```

These lines illustrate the following points:

- The inclusion of arguments such as `xlab=""` and `ylab=""` in the `Mandelbrot2()` command: these are interpreted as part of the `...` argument, and hence passed through to the underlying `image()` call.

- The use of a `for` loop to generate a sequence of plots. In this kind of situation a `for` loop is permissible: recall from the lecture that the reason to avoid these loops is that it takes time for R to interpret each line of code before executing it. In this case however, the time taken to *interpret* the code is tiny by comparison with the time taken to *execute* it (because the execution involves a call to `Mandelbrot2()`, which takes a few seconds). Compare with the `for` loops in `Mandelbrot1()`, where each loop only involved a small amount of computation and the interpretation time was therefore substantial by comparison.

- The `locator()` function can be used to select co-ordinates of points on an R graphics device. As each plot is produced therefore, you will be prompted to click two points: these will be taken as the opposite corners of a rectangle, and the next plot will 'zoom in' on that rectangle. This illustrates the usefulness of the `locator()` function as a

way to help you interact with R graphics. There are, apparently, an infinite number of seahorses, nautiluses and other strange creatures hiding in the Mandelbrot set: see if you can find one!

- Functions provide an extremely powerful way to carry out complex tasks with a single command. Thus, to produce a plot for a different region of the complex plane, we don't have to type out all of the `Mandelbrot2` code again: we simply call the command with different values of the `xlim` and `ylim` arguments.

6   Logical operations and comparison of objects

In the previous section we saw the use of the '`&`' symbol to test whether two logical conditions are both `TRUE`. This is an example of a *logical operator*, in the same way that `+`, `-`, `*`, `/` and `^` are arithmetic operators. Like arithmetic operators, '`&`' can be used with vectors. For example:

```
> x <- 1:5
> y <- c(2,-1,1,4,3)
> (x > y) & (y > 0)
[1] FALSE FALSE  TRUE FALSE  TRUE
```

The conditions are combined elementwise, as you might expect.

6.1   The main logical operators

You need to be familiar with the following logical operators:

'`&`': `x & y` is `TRUE` if the corresponding elements of `x` and `y` are both `TRUE`, and `FALSE` otherwise.

'`|`': `x | y` is `TRUE` if *at least one of* the corresponding elements of `x` and `y` is `TRUE`. Think of it as 'or': '`x` or `y` is true'.

'`%in%`': `x %in% y` returns, for each element of `x`, the value `TRUE` if that value appears in the object `y` and `FALSE` otherwise. For example (the `LETTERS` object was introduced in the lecture):

```
> x <- c("L","o","n","d","o","n")
> x %in% LETTERS
[1]  TRUE FALSE FALSE FALSE FALSE FALSE
```

6.2   Further work with logical operations

You also need to be familiar with the following:

- The '!' ('not') symbol swaps round the TRUE and FALSE elements of an object:

  ```
  > !(x %in% LETTERS)
  [1] FALSE  TRUE  TRUE  TRUE  TRUE  TRUE
  ```

- The any() and all() commands can be used to determine whether any, or all, of the elements of a logical object are TRUE.

  ```
  > any(x %in% LETTERS)
  [1] TRUE
  > all(x %in% LETTERS)
  [1] FALSE
  ```

  These commands can be useful in if() and while() conditions (for example, the any(NotDone) condition in the Mandelbrot2 function).

Notice that the conditions in an if() or while() statement must evaluate to a *single* TRUE or FALSE value. It can be very easy to accidentally program a condition that delivers a vector result, in which case you'll get a warning message along the lines of

```
Warning message:
In if (your condition) {your commands} :
  the condition has length > 1 and only the first element will be used
```

6.3 Comparison of objects

When you're using if() or while() statements, or selecting subsets of an object using [] based on some condition, the condition itself often involves the comparison of vectors or matrices. The main comparison operators are:

'==': x==y is TRUE if the corresponding elements of x and y are equal, and FALSE otherwise. **Note**, however, that computers can't do calculations with 100% accuracy, and small rounding errors can lead to unexpected results. For example:

```
> sin(2*pi) == 0
[1] FALSE
> sin(2*pi)
[1] -2.449213e-16
```

To get round this problem, if you need to test for equality of objects that have been produced by some calculation process, it is usually safer to test whether the absolute difference between them is less than some small number.

'!=': x!=y is TRUE if the corresponding elements of x and y are different, and FALSE if they're the same. So, for example, if x and y are vectors of the same length then x!=y is an alternative way of writing !(x==y).

'<', '>': x<y (resp. x>y) return `TRUE` for those elements of x that are less (resp. greater) than the corresponding elements of y, and `FALSE` otherwise. To test whether the absolute difference between x and y is less than some small number — say 10^{-12} — therefore, you can use `abs(x-y) < 1e-12`.

'<=', '>=': as '<' and '>', but now representing 'less / greater than or equal to'. You saw some examples of their use in Section 3.1 above.

7   Moodle quiz

There is just one new Moodle quiz this week. If you have time left in the workshop, try it a few times.

Assignment Project Exam Help

https://tutorcs.com

WeChat: cstutorcs