1 Programming

1.1 File management/code organisation

Organise code into functions. A function is a short section of code that can be reused in various cases in the program. A function usually has *parameters* which define its input, and a *return value* which it passes back to the rest of the program. A good function should be a section of code that does some clearly defined operation. Make a section of code into a function if it is likely that you'll want to run that section of code in a different context. If it is unlikely that you would want to run it outside the current function, don't make it a separate function.

Organising code into functions makes it easier to see what the program is supposed to be doing, and to test where the program is going wrong when there are bugs (and there will be bugs). Functions can often be tested separately, and shown to be working. It is helpful if functions contain as few parameters and return values as possible. (Don't cheat in this aspect by trying to use global variables — see Section 1.4.) Give functions descriptive names so that anyone reading the program can see what it is supposed to be doing.

For longer programs, it can be useful to divide the program into separate files. You can load the code from one file into another file in `R` using the `source` command. There should be a main file that has the main program. Other files can be included using the `source` command. The files that are included using the `source` command should contain only functions — no code should be executed in them. That way, anyone reading your main program will know exactly what the included file is for (because you've used descriptive function names, they don't need to see your function code the know what it's doing). This also has the advantage that the order in which files are included doesn't matter.

1.2 Variable Naming and Comments

It is common either to become distracted for a long while on a new project, then come back to some old code to modify it, or to leave for some job and not give a `*%!$` about the research any more, in which case, making any necessary changes to the program will be the responsibility of your long-suffering supervisor. To make your life easier in the first case, and your supervisor's life easier in the latter, please make it as easy as possible to figure out what your code is doing.

Good comments to include:

- At the start of every function, include comments giving details of the types and meaning of every parameter.

1.3 Making code general

It is very helpful to use variables for *anything* that might ever change. It is almost no work to define a variable for the value of interest and use that variable

every time. However, if in the future you ever want to change the value, you only have to do it once, and if it is in a function, you can make it a parameter.

1.4 Variable scoping & parameter usage

Variable scoping is about in which parts of the code a variable has a certain definition. For example in the following `R` code:

```
setb<-function(){
  b<-4
}
setb()
b
```

We get

```
Error: object 'b' not found
```

This is because the variable `b` is *local* to the function `setb`. This is important. It means that we don't have to worry about the function `setb` interfering with variables already in your program. On the other hand, the reverse scoping works:

```
b<-4
addb<-function(x){
  return(x+b)
}
addb(3)
```

Will output `7`, because the variable `b` is *global*, meaning any part of the program can access it. **NEVER** use this in your code. If your function needs an external variable called `b`, it should be a parameter to your function. (One exception to this rule is functions, which are technically considered variables in `R`, and are usually used as global variables.) Sometimes your function can accidentally refer to a global variable due to a typo (e.g. different capitalisation — if you call the parameter `Lambda` but have a variable in the global namespace called `lambda` then forgetting to capitalise `Lambda` somewhere in the function can lead to `R` accidentally refering to the global variable with unfortunate (and difficult to debug) consequences. A useful tool for dealing with this is the function `codetools::findGlobals`. After defining function `f`, run `codetools::findGlobals(f)`. It will produce a list of all variables used but not defined in `f`. If any of these things are variables (other than constants such as `pi`), you should correct them.

1.5 Variable typing

There are many different types of variables for storing different types of values. In `R`, the main types of variables are the following:

Type	for
numeric	Numbers
logical	True or False
factor	factor variables — variables with a limited number of possible values
ordered	ordered factors — variables whose possible values have an order
character	text values — e.g. filenames, labels
vector	linear array of values of a certain type
list	like a vector but each element can have a different type
data frame	list of vectors of various types with the same length
array	like a vector but indexed by a sequence of numbers
matrix	a two-dimensional array with special operations available e.g. matrix multiplication.
function	a section of `R` code that can be executed

The operations that can be performed on a variable depend on its type, and also on auxiliary attributes such as length of a vector and dimensions of a matrix. Attempting to apply an operation to a variable of the wrong type will usually (if we are lucky) result in an error message, or at least a warning. If we are unlucky, it may just silently produce the wrong result — this is much worse, because when we find out the results are wrong, we don't know what part of the program is wrong.

Many programming languages (e.g. `c`, `c++`, `java`) have *static typing*, namely before a variable is used the intended type is declared, so that any attempt to use it as a different type can be detected and fixed. Other languages use *dynamic typing*, where the type of a variable is determined whenever it is assigned, and can be changed in future. `R` is even worse at variable typing than most of these languages — see Section 2.1. Sometimes a type can be converted to another type. There are two cases of this — implicit conversion and explicit conversion. Implicit conversion happens when it is "obvious" to the computer that this conversion should be performed. For example, a logical vector will be implicitly converted to values 1 for True and 0 for False if it is used in an arithmetic expression. The program does not need to indicate that this should be done. Explicit conversion uses an explict function to tell `R` to convert the variable to the desired type. For example applying the `as.matrix` function to a data frame all of whose variables happen to be numeric, will turn it into a matrix.

1.6 Random Number Generation

Simple rule: if writing a single script, write

```
set.seed(12345)
```

as the first line of your script. **NEVER** call set.seed again in your program. Do this only on the main script, never in a file that gets sourced, never in a function. If you write your code into a package, **NEVER** set the seed in the final package.

For some explanation, the seed determines the sequence of random numbers generated in your program. (Note: this only works if your program runs in series, don't bother setting seeds for parallel exectution.) By setting a seed, we

get the same random numbers every time we run the program. This is helpful if we want to debug some issue, since we can ensure the issue happens every time. Without setting the seed, the issue may only occur sometimes, making it hard to tell whether we have fixed it.

1.7 Algorithmic Complexity

Complexity refers to how much resources (mostly time) are needed for the program to complete running. It is generally important to consider the complexity, to ensure that results are likely to finish running in a reasonable time. It is also important for debugging — if the program takes a long time to run, we need to know at what stage to assume that it is failing to finish due to a bug. If the complexity analysis is particularly difficult, we can try the method empirically by seeing how long it takes on smaller datasets, and how quickly the time required increases as size increases.

One common needless efficiency sink is appending elements to vectors and lists. Often appending an element to a list is very inefficient, because the list will often have to be copied to a new area in memory to make space to append elements. Therefore, if you know how long the list should be don't do this. Instead of

```
v<-NULL
for(i in 1:100000){
  v<-c(v,i^3+4)
}
```

use

```
v<-rep(0,100000)
for(i in 1:100000){
  v[i]<-i^3+4
}
```

[Actually, in this example, you should use the vector operation (see Section 2.2):

```
v<-(seq_len(100000))^3+4
```

since this is even quicker and easier to change — for example the number of elements in `v` only occurs once, so you can't accidentally only assign the first 100,000 entries of a vector of length 1,000,000.]

This has a number of benefits. First, the program will run quicker because there is no need to keep copying the vector around. Secondly, if `v` is too large and the machine will run out of memory, it will do so immediately, allowing you to find a better way. In the first version, the program will run for a long time before giving an error and throwing away all the values you've calculated. Thirdly, it is more obvious that `v` is the correct length and indexed correctly using the second method — The first method can lead to difficult to solve bugs with the indexing. For example, if we forget to initialise `v` to `NULL` before the loop, the values will be appended

1.8 Machine Numbers

Computers **cannot** store numbers. Numeric variables are fixed length binary sequences designed to approximate numbers. The "rules of arithmetic" are not in any way like those for numbers. This means that just because your method might work perfectly for numbers, it doesn't have to work at all on a computer. More specifically, a computer stores a real number as three parts: the sign, the mantissa, and the exponent. The exponent gives the position (in base 2) of the leading 1. The mantissa gives the first significant digits of the number. The sign indicates whether the number is negative. For example, if we want to store $\frac{1}{3}$ using one sign digit, 4 digits for the mantissa and three digits for the exponent, it would look like this:

$$\underbrace{0}_{\text{positive}} \qquad \overbrace{01011}^{\substack{\frac{4}{3} \text{ in binary without the} \\ \text{leading 1. The final digit} \\ \text{has been rounded up.}}} \qquad \underbrace{110}_{\text{exponent}=-2}$$

There are a few details not discussed here that aren't very important for our purposes about how the sign and the exponent are coded. Also the order of these parts and details of the encoding (e.g. some machines write binary numbers right-to-left) vary between machines. The key point is that the computer stores the number in such a format. (Of course the computer would use more binary digits (bits) for both the mantissa and exponent than we do here as an illustration). Suppose now we subtract $\frac{3}{16}$ from this. The computer writes $\frac{3}{16}$ as

$$\underbrace{0}_{\text{positive}} \qquad \overbrace{10000}^{\substack{\frac{3}{2} \text{ in binary without the} \\ \text{leading 1.}}} \qquad \underbrace{101}_{\text{exponent}=-3}$$

To subtract $\frac{3}{16}$, we first write them both at exponent -2 inserting the leading 1 for both numbers:

$$\begin{array}{l} 101011 \\ 0110000 \\ 0100110 \end{array}$$

So the answer is 1.00110×2^{-3}. With real numbers, the result should be $\frac{7}{48}$.

If we next subtract $\frac{1}{8}$, we get

$$\begin{array}{c} 100110 \\ 100000 \\ 000110 \end{array}$$

which is 1.1000×2^{-6}. Since we only have three digits for the exponent, the exponent is allowed to take values between -3 and 4, so we have an underflow here (meaning our number is closer to zero than the smallest number we can store properly). The computer might therefore actually set this number to zero (some computers can cheat a little to avoid this). Underflows (and overflows, where the number is too big) aren't common problems when the computer has a reasonable number of digits for the exponent. (11 digits gives a range of about 10^{-300} to 10^{300} for numbers, which suffices for most of our purposes). Here, we will assume we have enough digits for the exponent and worry about the mantissa. The correct value for the expression is

$$\frac{1}{3} - \frac{3}{16} - \frac{1}{8} = \frac{1}{48} = 1.0101 \times 2^{-6}$$

We see that our computer's answer is $1.1000 \times 2^{-6} = \frac{3}{128}$. The error is

$$\frac{3}{128} - \frac{1}{48} = \frac{1}{384} = \frac{1}{48} \times \frac{1}{8}$$

So our relative error is 12.5%. This demonstrates a general principle that subtraction of close numbers can destroy *a lot* of accuracy. There is a lot more that can be said on this subject, but for beginning programming, you should be aware of the danger of numerical issues.

1.9 More Advanced Techniques — Functional Programming

A technique that is sometimes useful is that in addition to data, variables in `R` can refer to functions (i.e. sections of `R` code). This means that we can pass them as parameters to other functions. For example, suppose we want to write a function for using the Newton-Raphson method to maximise a function. This relies on knowing the first and second derivatives of the function at arbitrarily chosen values of `x`. We can get around this by passing a function that computes these values to our Newton-Raphson function.

```
NewtonRaphson<-function(start_value,first_derivative,second_derivative){
  x<-start_value
  derivative<-first_derivative(x)
  while(derivative^2>1e-16){
    second_deriv<-second_derivative(x)
```

```
    x<-x-derivative/second_deriv
    derivative<-first_derivative(x)
  }
  return(x)
}
```

[Yes, I know this function isn't very good! It's written to demonstrate a point about using functions as parameters. Please don't actually use this function.] We see that by passing the derivatives as functions, we are able to write a fairly general optimisation function. This sort of technique can often help make code more general.

2 R pitfalls

As far as I can tell, R appears to have been designed with the objective of maximising the probability that if a monkey hits keys at random, the program will run without errors. This unfortunately means that it tries to guess a number of things that the programmer should make explicit.

2.1 Dynamic typing

We already discussed how variables can be of different types for storing different sorts of information. In a lot of programming languages (e.g. `c`, `c++`, `java`) the programmer needs to explicitly specify the type of a variable before using it. This has the advantage of catching errors due to the programmer not knowing the type of some operations. In other programming languages, the type of a variable is defined by the expression that creates it. In R, we have neither of these convenient properties, and the types of variables can vary with different runs of the program. Consider the following program

```
A<-rnorm(36)
dim(A)<-c(6,6)
randomvector<-rnorm(6)<0
v<-1:6
submatrix<-A[randomvector,]
dim(submatrix)[1]
```

A is a 6×6 matrix, and v is a vector of length 6, we select some rows of A at random, then count how many rows the resulting submatrix has. If you try this program a few times, you'll find that it sometimes gives reasonable values and sometimes gives NULL. Why is this? Because while subsetting some rows of a matrix usually gives another matrix, if the number of rows to select happens to be 1, R converts it to a vector instead. The type therefore cannot be predicted from the code, and while the type will usually be a matrix, it may occasionally be a vector.

The subsetting operation has an additional logical argument `drop` which should be set to `FALSE`. That is, we should change the line in the above program to

```
submatrix<-A[randomvector,,drop=FALSE]
```

Unfortunately, the default is `drop=TRUE`, which is rarely, if ever, desirable. A lot of packages fail to use the `drop=FALSE` option, resulting in strange bugs. One feature `R` provides to deal with these is a set of functions to check the type of each object. For example `is.matrix(A)` returns `True` if `A` is a matrix, and `False` otherwise. If you find your program seems to have type problems, use of these commands can help to debug.

2.2 Vectorisation

`R` is designed around the concept of vector operations — that is, applying the same operation to a vector of values. For example, is we write

```
> x<-c(1,4,7)
> y<-c(7,2,3)
> x*y
 [1]  7  8 21
```

`R` has applied the multiplication elementwise to the vectors. This is the default behaviour assumed in `R` functions. It is however very easy to get functions where this default behaviour does not produce the desired effect. For example

```
>  f<-function(x){
# x is the mean of v with respect to some weird metric
+      v<-c(1,3,6)
+      M<-cbind(c(3,2,5),c(3,9,2),c(4,2,5))
+      return(t(v-x)%*%M%*%(v-x))
+  }
```

In this function `x` is meant to be a scalar, representing some kind of mean or intercept for the vector `v`. If, however, we accidentally use a vector for `x`, we get a different scalar value, instead of a vector value giving the values of `f` for every `x[i]`. If you are accustomed to using the vectorised form of function calling for other functions, it may be easy to forget that `f` doesn't follow this pattern and accidentally call it with a vector. Worse: even if you remember not to do this explicitly, it may be done implicitly via callback — a lot of `R` functions take a function as argument (see Section 1.9). Some of these functions assume that the function passed as argument vectorises well, and will produce strange behaviour if it does not. For safety, aim to have all your functions vectorise well, and if they shouldn't ever be called with vectors, make them check for vector values of the parameters and give an error if there are any.

2.3 Return values

Functions often calculate a value, which they then pass back to the main program. This is done by the command such as `return(x)` at the end of the function. `R` by default returns the last value calculated in the event that there is no return command at the end of the function. This is a terrible default that offers no benefits whatsoever. The best thing about this is that it makes code slightly harder to read, since it is not explicit what is returned from a function. The really bad thing about it is that if you forget to include a return statement in a function, the program will return something anyway, and if what it returns is wrong, will possibly not give any indication of the problem — just producing wrong answers.

2.4 For loops and the : operator

The `:` operator produces a vector of all values between two numbers. For example `3:7` produces the same vector as `c(3,4,5,6,7)`. This is very useful, particularly with for statements such as

```
for(i in 7:1000000){
  doSomethingUseful(i)
}
```

There are however a couple of pitfalls to watch out for with the `:` operator.

First is precedence — the `:` has relatively high precedence, so for example

```
a<-10
for(i in 1:a+1){
  doSomethingUseful(i)
}
```

interprets the loop statement as `for(i in (1:a)+1)` and so runs the loop starting at `i=2`. To get the effect you want, you need to start the loop `for(i in 1:(a+1))`.

The second pitfall is that many people naturally assume that `for(i in 1:top)` runs for all positive integer values of `i` up to and including `top`. However, it does not always do this. In particular, if `top=0`, then we usually don't want to run any iterations of the loop. For example, in the code

```
for(i in 1:length(v)){
  doSomethingUseful(v[i])
}
```

we are trying to do something to each element of the vector `v`. If `v` happens to be `NULL`, then we don't want to do anything. However, the `:` operator is incorrect — it produces backwards sequences, so `1:0` produces the sequence `c(1,0)`. The code above will first attempt to call

```
doSomethingUseful(v[1])
```

then

```
doSomethingUseful(v[0])
```

which in this case will be fortunate enough to give an error, that we can hopefully decipher and fix.

If you want the sequence of indices of a vector of length `n`, the appropriate code is `seq_len(n)`. If `n` is the length of vector `v`, you can use the code `seq_along(v)` as a shorthand for `seq_len(length(v))`.

3 R CMD BATCH

If you want to run an R command non-interactively, one way you may see suggested is to use `R CMD BATCH`. It is important to note that this command has an unusual and dangerous syntax:

```
R CMD BATCH R_OPTIONS '--args COMMANDLINE_ARGUMENTS' Rscript.R outfile.R
```

What is unusual about this syntax is that most unix programs take arguments in the form `--arg value` separated by the space, which would mean that

```
R CMD BATCH --args 'COMMANDLINE_ARGUMENTS' Rscript.R
```

would be a natural format. `R CMD BATCH` does not follow this syntax. Instead, it will interpret `COMMANDLINE_ARGUMENTS` as the name of your scipt, and `Rscript.R` as the outfile. This will result in your script being overwritten, usually by an error message.

4 Debugging

One of the things that annoys me most as a supervisor is when students come to my office and tell me "The program doesn't work." without any further details. This is poor, lazy and inefficient! There are a number of steps that should be performed to diagnose where and when the problem occurred.

4.1 Is the program wrong?

One possible explanation of the problem is that the program works perfectly, but the statistical method it is based on does not. We can often diagnose this by checking intended output conditions. For example, if the program is trying to optimise a function, check the derivatives to see if it has found a local optimum.

4.2 Is there an error message?

Error messages are associated with a particular line in the code. This may not be where the error occurs, but is the first place where the error is detected, and is a good place to start looking. The error message will usually tell us in what way the assumptions on this line are violated. We can then decide whether these assumptions are wrong, or whether something earlier is wrong — for example, perhaps a variable is of the wrong type because of earlier errors. Sometimes error

messages can be cryptic: in this case Google the error message, and you should be able to find a better explanation with probable causes. Finding out where the assumptions were violated involves working backwards. Usually getting the program to output all intermediate values can help to pinpoint where things start to go wrong — use a divide and conquer strategy: test the values at some point between where you know things are right, and where you know things are wrong, then you have reduced the search for problems to a smaller section of code. Test all functions on their own multiple times.

4.3 Does the program run without stopping

This can often happen if the termination condition of a loop is never reached. A difficulty here is that we can't wait forever to see if the program ever stops, then go back and start again. Instead, we need to decide how long to let the program run before deciding that there is a problem. This is related to the expected complexity (see Section 1.7). By analysing the complexity and trying smaller examples, we can see how long we would expect the program to take. For example, if the program completes one simulation in 1 second, then it should complete 100 simulations within a few minutes. If it takes much longer, there is most likely some problem.

This problem usually happens because some loop doesn't terminate, or runs many more times than expected. It can sometimes be tracked by adding commands to print out the current step number of a loop at each iteration. This will usually tell you where the program gets stuck — if it prints out about one number every second, then after printing out 23, it doesn't print out any more numbers for several minutes, it has probably got stuck on the 23rd iteration of the loop. If it keeps printing larger numbers every second, and reaches much larger numbers than we expected, it is probably never reaching the termination condition for the loop.

4.4 Does the program always have a problem

Sometimes a bug causes problems every time. Other times, it only has problems for certain inputs. Try running the program with smaller simpler data or input, or just different values to see if it still has the same problem.

4.5 Try to locate the source of the error

This can be tricky if we don't know exactly what causes the error. Try to test all the parts of the program independently. Add commands to print out important variables and other diagnostics at each stage to find out when things go wrong. If there is an error message, it will usually give you a place to start looking. You know the problem is before the error message. If the program doesn't terminate, get it to print out messages at regular intervals, then the last message will show where it gets stuck. If the final answer seems wrong, print out values in the middle to see where they seem to go wrong.

5 Debugging Checklist

1. If the program gives an error message:
   (a) What are the possible causes of this error message? Use Google to help with this.
   (b) If the error occurs in a third-party function (these have usually been well checked, so it's more likely your parameter values are wrong, but don't assume that just because someone manages to put their package online, they have coded perfectly):
      i. Include the package name in your search.
      ii. Try to directly obtain input to the function that causes the error. Examine this input to see if there is anything obviously wrong with it.
      iii. Check the documentation to ensure your parameters are correct.
      iv. If the parameters seem correct, and there's no mention of the bug online, contact the package maintainer.
   (c) If you think the problem occurs earlier, use a divide and conquer approach to figure out exactly where.
2. If the program just keeps running:
   (a) Think about how long you expect the program to take.
   (b) Try rerunning the program on smaller/simpler data to see if it is quicker and how long it takes.
   (c) Make the program print out its progress while running to see where it gets stuck.
3. If the program gives a result that looks wrong:
   (a) Double-check your expectations — is your expectation wrong?
   (b) Try running the program on simple cases where it is easier to be sure what the correct output is.
   (c) Find ways to test the output is correct — e.g. if the program is trying to optimise a function, check whether the derivatives are zero at the output value.
   (d) Once convinced that the results are wrong, print out intermediate values to determine when results start to go wrong.