UNIT IV

Debugging in **R**

A grammatically correct program may give us incorrect results due to logical errors. In case, if such errors (i.e. bugs) occur, we need to find out why and where they occur so that you can fix them. The procedure to identify and fix bugs is called “debugging”.

There are a number of ‘R’ debug functions, such as:

traceback()

debug()

browser()

trace()

recover()

1. traceback()

If our code has already crashed and we want to know where the offending line is, then try traceback(). This will (sometimes) show whereabouts in the code of the problem occurred.

When an R function fails, an error is printed to the screen. Immediately after the error, you can call traceback() to see in which function the error occurred. The traceback() function prints the list of functions that were called before the error occurred. The functions are printed in reverse order.

```
f<-function(x) {
r<-x-g(x)
r
}


g<-function(y) {
r<-y*h(y)
r
}

h<-function(z) {
r<-log(z)
if(r<10)
r^2
else
r^3
}
```

```
> f(-1)
Error in if (r < 10) r^2 else r^3 : missing value where TRUE/FALSE needed
In addition: Warning message:
```

```
In log(z) : NaNs produced

> traceback()
3: h(y)
2: g(x)
1: f(-1)
```

2. debug()

The function debug() in R allows the user to step through the execution of a function, line by line. At any point, we can print out values of variables or produce a graph of the results within the function. While debugging, we can simply type "c" to continue to the end of the current section of code.

traceback() does not tell us where the error occurred in the function.

In order to know which line causes the error, we will have to step through the function using debug().

3. browser()

The R debug function browser() stops the execution of a function until the user allows it to continue. This is useful if we don't want to step through the complete code, line-by-line, but we want it to stop at a certain point so we can check out what is going on.

Inserting a call to the browser() in a function will pause the execution of a function at the point where the browser() is called.

Similar to using debug() except we can control where execution gets paused.

Ex

```
h<-function(z) {
  browser() ## a break point inserted here
  r<-log(z)
  if(r<10)
    r^2
  else
    r^3
}
```

4. trace()

Calling trace() on a function allows the user to insert bits of code into a function.

5. recover()

When we are debugging a function, *recover()* allows us to check variables in upper-level functions.

- We can use recover() as an error handler, set using options() (e.g.options(error=recover)).
- When a function throws an error, execution is halted at the point of failure. We can browse the function calls and examine the environment to find the source of the problem.

Simulations

Simulations are a powerful statistical tool. Simulation techniques allow us to carry out statistical inference in complex models, estimate quantities that we can cannot calculate analytically or even to predict under different scenarios the outcome of some scenario such as an epidemic outbreak.