Statistics and Programming in R

David Stephens and Niall Adams

[d.stephens,n.adams]@imperial.ac.uk

Department of Mathematics, Imperial College London

29/30 September 2005

Extended Example

This example will be used to illustrate the use of the computation and programming techniques, data input/output methods, graphical output generation we have seen so far.

We will

- change to a working directory
- use an R script
- automate some data input
- loop through operations
- generate and export graphical output
- create a large data-frame with all the data

Confocal Microscopy Data

These data sets are generated by the Centre for Structural Biology, Prof. Paul Freemont's laboratory. They contain the 3d spatial locations of certain objects in the cell nucleus derived from confocal microscopy images.

- Download the data and scripts from

  `stats.ma.ic.ac.uk/~das01/RCourse/AutomatedExample.zip`

- Save in `C:\Temp\RCourse`
- Unzip to form `C:\Temp\RCourse\AutomatedExample`
- Inspect the files in this folder and its subfolders.

The directory contains two scripts

- `01-ReadFull.R`
- `01-ReadImages.R`

These two files perform processing of the data files in in subfolder `Second`

We can open, edit and run the scripts in `R`

R operations

- Double click the R icon on the desktop
- From the *Misc* pulldown menu, de-select the *Buffered output* option that is ticked by default.
- From the *File* pulldown menu, select the *Change dir ...* option to bring up a dialog box. Replace the text with `C:\Temp\RCourse\AutomatedExample`.
- From the *File* pulldown menu, select the *Open Script ...* option to bring up a file selection dialog box.
- Select `01-ReadImages.R`
- **Try not to alter this script at first !**

R Functions

User-defined functions can be used in R. The main function definition syntax is

```
functionname <- function (args) {
   computation
   return(result)
}
```

where *args* is a set of arguments.

A function is called as follows

```
functionname(args)
```

Example

Here is a small function to evaluate the function

$$f(x) = \alpha_1 \exp\{-\lambda_1 x\} + \alpha_2 \exp\{-(\lambda_1 + \lambda_2)x\}$$

at any value of $x > 0$, for user-supplied parameters $(\alpha_1, \alpha_2, \lambda_1, \lambda_2)$.

The function needs to have the arguments

$$x, \alpha_1, \alpha_2, \lambda_1, \lambda_2$$

supplied (in some form) and return the function value

my.function

```
my.function<-function(x,al1,al2,lam1,lam2) {
  y<-al1*exp(-lam1*x)+al2*exp(-(lam1+lam2)*x)
  return(y)
}
x<-seq(from=0,to=10,length=101)
fx<-my.function(x,1.0,2.0,0.1,0.2)
plot(x,fx,type="l",ylim=range(0,4))
```

Alternative `my.function`

Supply the parameters as a vector $\theta = (\alpha_1, \alpha_2, \lambda_1, \lambda_2)$.

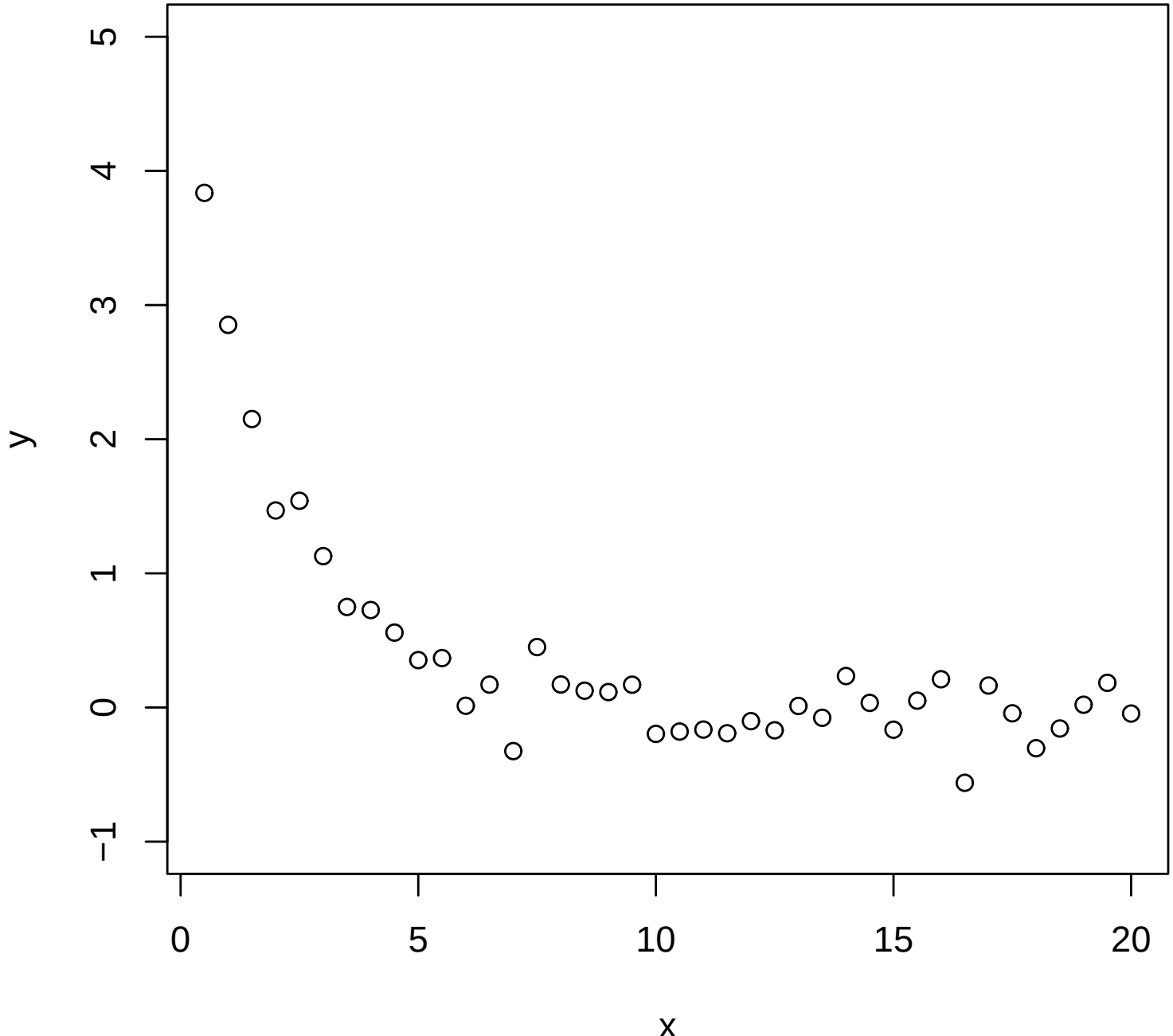
```
my.function.alt<-function(x,th) {

  y<-th[1]*exp(-th[3]*x)
  y<-y+th[2]*exp(-(th[3]+th[4])*x)
  return(y)

}
x<-seq(from=0,to=10,length=101)
theta<-c(1.0,2.0,0.1,0.2)
fx<-my.function.alt(x,theta)
plot(x,fx,type="l",ylim=range(0,4))
```

Exercise

We will simulate some data using the function above and try to recover the parameters

```
set.seed(300905)
theta<-c(3.0,2.0,0.5,0.2)
x<-c(1:40)/2
expected.y<-my.function.alt(x,theta)
y<-expected.y + rnorm(length(x),sd=0.2)
plot(x,y,ylim=range(-1,5))
```

Example Data



Least-squares Fit

We will slightly change the defined function so that the R minimization function `nlm` can be used to find the best fit.

```
my.function.new<-function(th,xvals,yvals){
  fy<-th[1]*exp(-th[3]*xvals)
  fy<-fy+th[2]*exp(-(th[3]+th[4])*xvals)
  ssq<-sum((yvals-fy)^2)
  return(ssq)
}

th.start<-c(3.0,2.0,0.5,0.2)
nlm(f=my.function.new,p=th.start,xvals=x,yvals=y)
```

Results

```
>nlm(f=my.function.new,p=c(3.0,2.0,0.5,0.2),xvals=x,yvals=y)

$minimum [1] 1.415276

$estimate [1] 2.9522862733 1.9203908648 0.5204927933 0.0002269101

$gradient [1] 7.890393e-07 -7.270481e-07 -9.703349e-08 7.016610e-08

$code [1] 2

$iterations [1] 31
```

This means that the line of best fit is obtained when

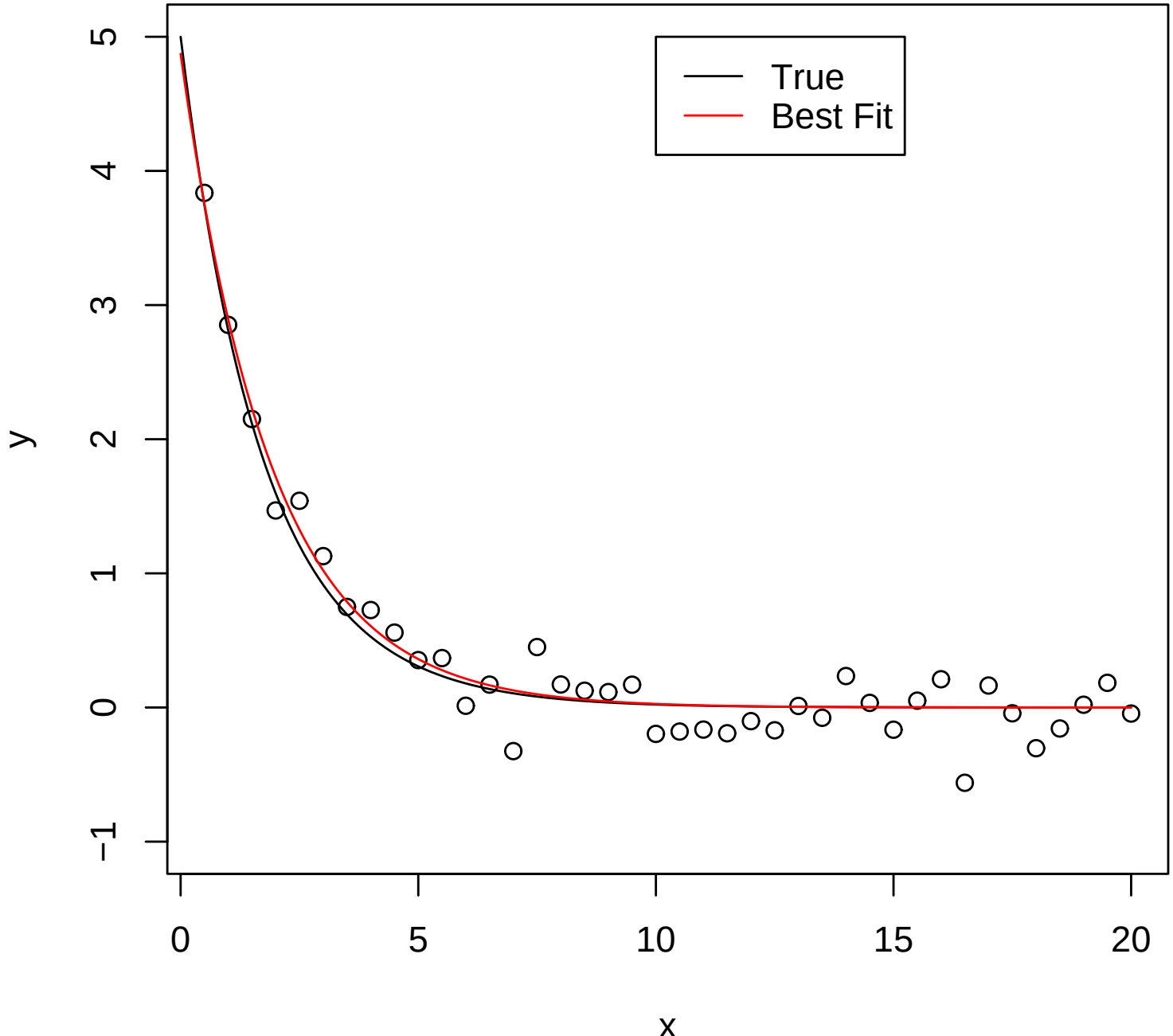
$$\lambda_1 = 2.952 \qquad \lambda_2 = 1.920 \qquad \alpha_1 = 0.520 \qquad \alpha_2 = 0.0002$$

Best Fit ?

```
my.fit<-nlm(f=my.function.new,p=c(3.0,2.0,0.5,0.2),xvals=x,yvals=y)
param.estimates<-my.fit$estimate
```

```
xv<-c(0:200)/10
true.y<-my.function.alt(xv,theta)
fitted.y<-my.function.alt(xv,param.estimates)
plot(x,y,ylim=range(-1,5))
lines(xv,true.y)
lines(xv,fitted.y,col="red")
legend(10,5,c("True","Best
Fit"),lty=c(1,1),col=c("black","red"))
```

Best Fit !



Extended Example 2

In this example, we will

- simulate a large data set
- automate its analysis
- process and plot the results

The key components will be the use of the function `apply` to a numerical matrix.

Simulated Microarray data

Microarrays are a high-throughput technology for the analysis of the function of genes.

Typically, thousands of genes are processed simultaneously.

Interest lies in distinguishing genes that are “differentially expressed" in two tissue types.

In this experiment we will simulate some appropriate data.

Simulation

```
#Number of genes
ngenes<-7500

#Number of samples
N0<-20
N1<-37

#Select the genes that are differentially expressed
Ndiff<-50
gene.list<-sample(c(1:ngenes),size=Ndiff,rep=F)

#The amount of up-regulation
Up.mean<-2.0
Y0<-matrix(rnorm(N1*ngenes),ncol=N0,nrow=ngenes)
Y1<-matrix(rnorm(N2*ngenes),ncol=N0,nrow=ngenes)
Y1[gene.list,]<-Y1[gene.list,]+Up.mean
Y<-cbind(Y0,Y1)
```

Method I

#One method of analysis

```
date()
test.results<-numeric(ngenes)
for(igene in 1:ngenes){
    y0<-Y0[igene,]
    y1<-Y1[igene,]
    test.igene<-t.test(y1,y0,var.equal=T)
    test.results[igene]<-test.igene$statistic
}
date()
hist(test.results)
```

Method II

Using `apply`

```
my.test<-function(x){
    n0<-x[1]
    n1<-x[2]
    y0<-x[3:(2+n0)]
    y1<-x[(2+n0+1):(2+n0+n1)]
    t.res<-t.test(y1,y0,var.equal=T)
    return(t.res$statistic)
}
tmp.Y<-cbind(rep(N0,ngenes),rep(N1,ngenes),Y)
date()
my.test.results<-apply(tmp.Y,1,my.test)
date()
```

R Programming: Control Structures

To release the power of the programming language, we need to learn about the language constructs the provide *control structures*, that provide the capacity for selection and iteration.

Think back to the `ChickWeight` example: had we required to consider all 4 diet groups, we would have had to write effectively the same piece of code 4 times. An alternative would be to write a new function that conducts the computation for selected data.

First we will look at control structures.

R Programming: Selection

It is often the case that we want a program to take different actions according to the value of a variable. The R language statement `if` provides this functionality. The general format is

```
if (condition)
        true.branch
else
        false.branch
```

We have already seen a variety of logical comparisons that can serve as `condition`. If `condition` evaluates as `TRUE`, then `true.branch` is followed otherwise `false.branch` is followed. If condition evaluates as `NA`, an error occurs.

R Programming: Selection

Example

```
if (x >3)
        {
         y <- 1
         z <- 2
         }
else
         {
         y <- 2
         z <- 1
         }
```

Note the use of curly braces allow us to deliver compound (that is multi-line) statements. Also note the use of indenting to try to clarify structure.

R Programming: Selection

The `else` part of an `if` statement is optional. As regular parts of the `R` language, `if` statements can occur within the branches of `if` statements – that is, they can be nested. For example

```
if (x > 2)

        if (y < 3)
                count <- count + 1
        else
         ...
```

Note `R` is quite fussy about placement of symbols in scripts. For a more elegant alternative to multiply nested `if` statements, use the `switch` function.

R Programming: Selection

It is often useful to have compound conditions with an `if` statement. We can combine conditions with the logical operators && (for AND) and || (for OR). Note these are different to the single character vector operators. For example, in an optimisation problem we may have

```
if (iterations > max.it && abs(error) < tol)
        converged <- T
```

We will sometimes need to use brackets to clarify compound conditions. Note also order of evaluation for %% and ||.

Be careful with conditions. If the condition evaluates to a vector, the first element is used (and could be coerced to logical).

R Programming: Selection

For selection operations on vectors, use the `ifelse` function. The general form of `ifelse` is

```
ifelse(test,true.value,false.value)
```

Here, all the arguments are vectors. `test` is a comparison operation applied to each element of a vector, `true.value` is returned in positions where the comparison is `TRUE`, and `false.value` is returned otherwise. For example

```
ifelse(1:10 < 5,0,1)
```

This should be efficient even for large vectors, and is to be preferred over explicit looping wherever possible. ifelse switch

R Programming: Iteration

We can distinguish two types of iteration construct: count controlled loops, provided by the `for` statement, and variable length loops, provided by the `while` and `repeat` statements.

WARNING: bad use of loops is the most common source of inefficient R code. This is particularly true of nested loops. Always think hard about how use functions like `apply` rather than using a loop. Of course, sometimes it is unavoidable.

R Programming: Iteration

The general format of the `for` statement is

`for (variable in sequence) statement` And note

that `statement` can be compound. `variable` is the counter variable, that will take consecutive values in `sequence`. As a simple example, of something NOT to do, consider the following

```
for(i in 1:length(x))
y[i] <- sin(x[i])
```

In the exercises, you will see just how inefficient this is, compared to using a vectorised function. We can nest `for` loops, but do this with caution. Be careful not to change the vale of the the counter variable.

R Programming: Iteration

The general format of the `while` statement is

```
while (condition) statement
```

Note that a while loop may never execute the statement. The statement is executed repeatedly until `condition` becomes false. In contrast, a `repeat` loop, with general format

```
repeat statement
```

will execute at least once, and continue until it is explicitly interrupted with a `break` statment. In fact, `break` will immediately exit from any loop structure. This can be useful for diagnostic purposes.

R Demo: Bracketing

The function

$$x^2 - 1$$

has a single zero in the interval $(0, 1)$.

A simple approach to finding zeros is *bracketing*, where we find an interval containing the zero, evaluate the function in the middle of the interval, and restrict attention to the half interval containing the zero (as is indicated by the sign of the function at the three points). This process is repeated until the width of the interval is smaller than a specified tolerance.

R Demo: Bracketing

```
# Start values
hi <- 1
lo <- 0
f.hi <- hi*hi-1/2
f.lo <- lo*lo-1/2
# set tolerance
tol <- 1e-9
found <- abs(hi-lo) < tol
# iteration counter

its <- 0
```

R Demo: Bracketing

```
# search
while (!  found){
      mid <- (hi+lo)/2
      f.mid <- mid*mid-1/2
      if (sign(f.mid) == sign(f.hi))
      {
            f.mid <- f.hi
            hi <- mid
             }
      else
            {
            f.mid <- f.lo
            lo <- mid
            }
      its <- its + 1
      if ((its %% 3) ==0)
       cat("Iteration ",its," hi= ",hi," lo= ",lo," mid=", mid,"\ n")
      found <- abs(hi-lo) < tol

}
```

R Demo: Bracketing

Of course, this is not the only way of implementing this procedure. We may have wanted to stop after a certain number of iterations. We could achieve this by modifying the convergence criterion

```
found <- (abs(hi-lo) < tol) && (its <= maxit)
```

Or, by using a `break` statement

```
if (its > 10) break()
```

Note that we could embody this code in a function, and that this function could be made to deal with arbitrary equations.

Be aware that `while` and `repeat` loops may never stop - the condition may not be satisfied.