Elements of R Programming

COSC 405, DATA 405 and DATA 505

Flow control in R

Computer programs often require repeated execution of the same operation, such as when adding a sequence of numbers, and so on.

There are several functions in R that control how many times statements are repeated.

There are also functions which evaluate conditions to decide whether a command should be executed or not.

We will describe the `for()` and `if()` functions here.

Example

We can add the elements of a vector using the `sum()` function, but if we want to add up a sequence of vectors, we might do it with a *for loop*.

Suppose we want to simultaneously add $1 + 2 + 3 + \ldots + 10$ and $1^2 + 2^2 + \cdots + 10^2$.

In other words, we want to add vectors of the form $[i \quad i^2]$, for $i = 1, 2, \ldots, 10$.

We will store the resulting sums in a vector called `sums`.

Example

We will start by assigning [0 0] **to** `sums`**:**

```
sums <- c(0, 0)
```

and sequentially adding vectors [1 1], [4 4], **and so on:**

```
sums <- sums + c(1, 1^2)
sums
```

```
## [1] 1 1
```

```
sums <- sums + c(2, 2^2)
sums
```

```
## [1] 3 5
```

The *for()* function

The command we want to repeatedly execute is of the form

```
sums <- sums + c(n, n^2)
```

where `n` is changing, progressing through the values `1`, `2`, `...`, `10`.

We can use the `:` function to generate these values:

```
1:10
```

```
##  [1]  1  2  3  4  5  6  7  8  9 10
```

and for each of these values, we want to execute the command `sums <- sums + c(n, n^2)`**.**

The *for()* function

The `for()` function allows us to repeat a command a specified number of times.

Syntax:

```
for (n  in values)  command
```

This sequentially sets a variable called `n` equal to each of the elements of `values`.

For each value of `n`, the listed command is executed.

Example

For our example, the necessary code is

```
sums <- c(0, 0) # assign a baseline value to sums
for (n in 1:10)  sums <- sums + c(n, n^2)
```

We can see the result by typing the object name, as usual

```
sums


## [1]  55 385
```

Repeating several commands at a time

If we want to execute several commands at once, we enclose them in curly brackets:

Syntax:

```
for (n  in values)  {
     command 1
     command 2
          ...
}
```

Example - simulation of normal variates

Simulating normal random variables is possible in a variety of ways.

If we add up 12 uniform random variables on $[-.5, .5]$, we can get a sum that follows a close approximation to the standard normal distribution.

We will use a `for()` loop to construct a large vector of such values so that we can draw a histogram and QQ-plot, to verify that we have succeeded in simulating normal random variables.

Example - simulation of normal variates

We can use the `runif()` function to simulate the uniform variates that we will need.

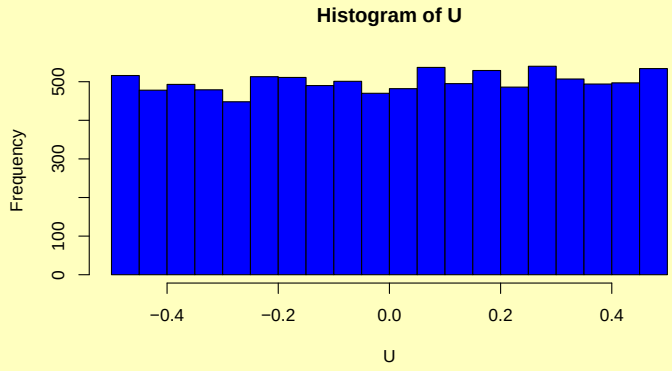
Simulation of uniform variates:

```
N <- 10000
U <- runif(N, min=-.5, max=.5)
```

`U` **contains values in the interval** $[-0.5, 0.5]$**.**

A histogram of the simulated uniform values:

```
hist(U, col="blue")
```



Example - simulation of normal variates

We initially assign 0 to our outcome vector `Z`.

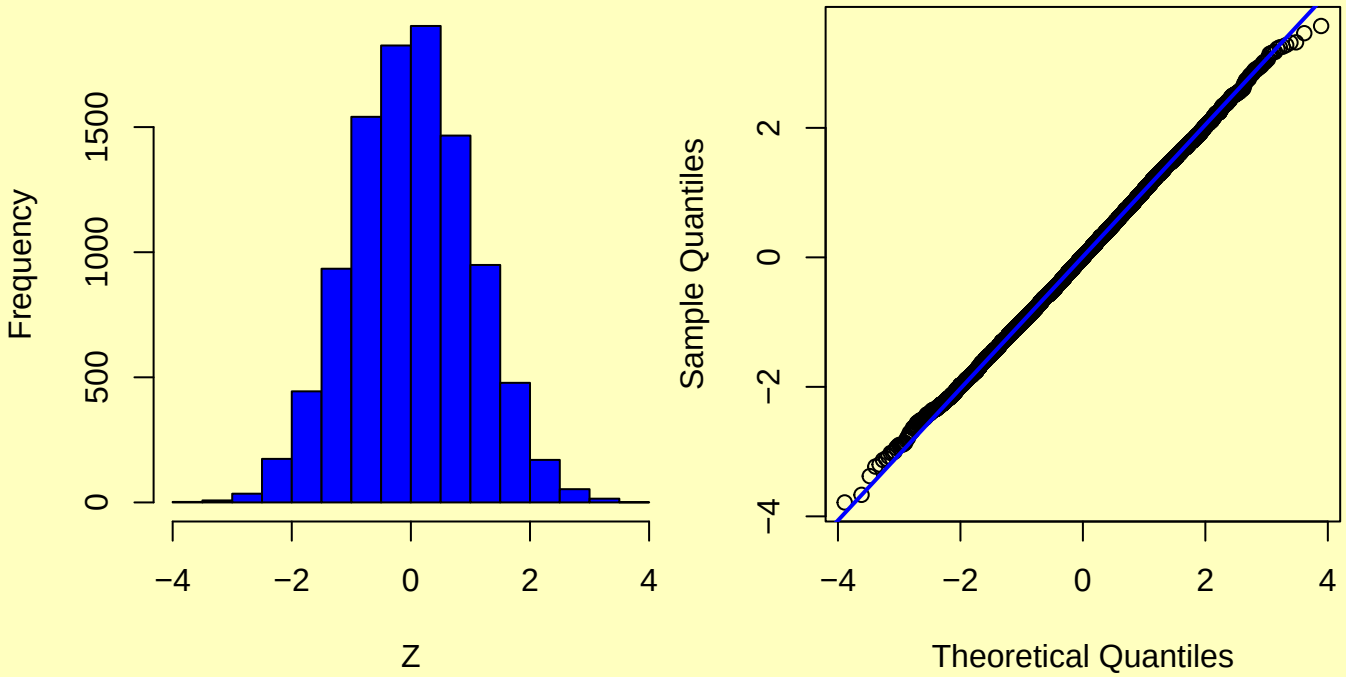
Then we successively add a uniform vector of size $N = 10000$ to `Z`, 12 times.

```
Z <- 0; N <- 10000
for (i in 1:12) {
    U <- runif(N, min=-.5, max=.5)
    Z <- Z + U
}
```

Note that we started with a `Z` which had only one entry, and successively added vectors of size 10000. R automatically changes the length of `Z` to make elementwise addition with `U` possible.

Example - simulation of normal variates

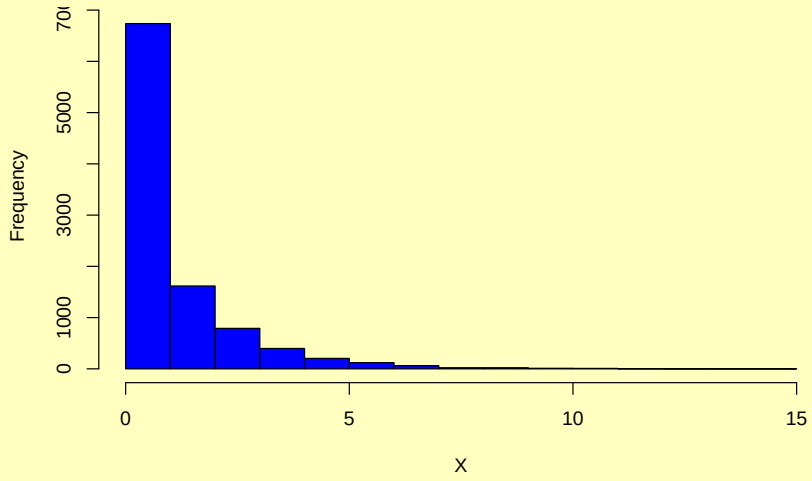
The histogram and QQ-plot of these simulated data are given below, the result of executing the following code.




```
par(mfrow=c(1,2), mar=c(4, 4, .1, .1))
hist(Z)
qqnorm(Z); qqline(Z)
```

Example - Squaring a standard normal variate

If Z is a standard normal random, then $X = Z^2$ is called a chi-squared random variable on 1 degree of freedom.



The distribution of a sample of chi-squared random variates on 1 degree of freedom is pictured to the left, the effect of executing the following code:

```
X <- Z^2;   hist(X)
```

Example - nesting `for()` loops

If $Z_1, Z_2, \ldots, Z_k$ are independent standard normal random variables, then the sum of their squares is a chi-squared random variable on k degrees of freedom.

We can use nested `for()` loops to simulate these sums of squared normals.

For example, suppose $k = 7$ as in the following:

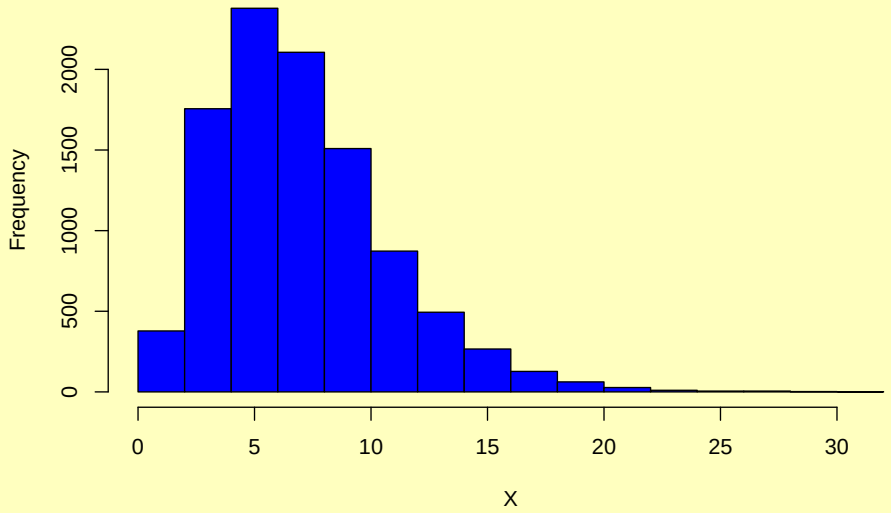
```
X <- 0; k <- 7
for (i in 1:k) {
    Z <- 0
    for (j in 1:12) {
        U <- runif(N, min = -.5, max = .5)
        Z <- Z + U # Z is standard normal
    }
    X <- X + Z^2  # X is chi-squared
}
```

Example - nesting `for()` loops

The histogram to the right shows what a chi-squared distribution on 7 degrees of freedom looks like:

```
hist(X, main="")
```

It is skewed, but not as much as when the number of degrees of freedom is smaller.



The *if()* statement

The `if()` statement allows us to control which statements are executed.

Syntax:

```
if   (condition) {commands |TRUE}
if   (condition) {commands |TRUE} else {commands | FALSE}
```

This statement causes a set of commands to be invoked if `condition` evaluates to `TRUE`.

The `else` part is optional, and provides an alternative set of commands which are to be invoked in case the logical variable is `FALSE`.

Example

```
x <- 3
if (x > 2) y <- 2 * x else y <- 3 * x
```

Since `x > 2` is `TRUE`, `y` is assigned `2 * 3 = 6`. If it hadn't been true, `y` would have been assigned the value of `3 * x`.

The *if()* statement: Caution!

Be careful how you type the `else` statement.

Typing it as

```
if  (condition) {commands when TRUE}
else {commands when FALSE}
```

may produce an error, because R will execute the first line before you have time to enter the second.

If these two lines appear within a block of commands in curly brackets, they won't trigger an error, because R will collect all the lines before it starts to act on any of them.

The *if()* statement: Caution!

To avoid this kind of difficulty, use the form

```
if   (condition) {
     commands when TRUE
     } else {
     commands when FALSE
}
```

The *if()* statement: Another Warning

R also allows numerical values to be used as the value of `condition`.

These are converted to logical values using the rule that zero becomes `FALSE`, and any other value becomes `TRUE`.

Missing values are not allowed for the condition, and will trigger an error.

Functions

As we have seen, R calculations are carried out by functions, and graphs are produced by functions.

The usual composition of a function is

- **a header that includes the word `function` and an argument list (which might be empty)**

- **a body which includes a set of statements enclosed in curly brackets `{}`.**

Function names should be chosen to describe the action of the function. For example, `median()` computes medians, and `boxplot()` produces box plots.

Example

We will write a function to approximately simulate standard normal random variables. An appropriate header for the function could be:

```
rStdNorm <- function(n)
```

Note that this function will take `n` as an input. The output should be that number of standard normal variates.

At some point in the body of the function there is normally a statement like `return(Z)` which specifies the output value of the function. If there is no `return()` statement, then the value of the last statement executed is returned.

Example

In our standard normal simulator, we will want to return a vector of length `n`. We will use `Z` as the name of this object.

```
rStdNorm <- function(n) {
...
return(Z)
}
```

Example

Using the sum of uniforms concept from the earlier example, we will use a function body of the form:

```
{
    Z <- 0
    for (j in 1:12) {
        U <- runif(n, min = -.5, max = .5)
        Z <- Z + U
    }
    return(Z)
}
```

Example

Putting the header and body together, we have the following function:

```
rStdNorm <- function(n) {
    Z <- 0
    for (j in 1:12) {
        U <- runif(n, min = -.5, max = .5)
        Z <- Z + U
    }
    return(Z)
}
```

Example

A trial with 3 values is executed as follows:

```
rStdNorm(3)


## [1] -0.6548109  1.6222985  0.7795247
```

Functions can take any number of arguments

We can use our new `rStdNorm()` function inside a function which calculates chi-squared random variables on k degrees of freedom.

Two arguments, `n` and `k` will be needed in this function.

```
rChisq <- function(n, k) {
    X <- 0
    for (i in 1:k) {
        Z <- rStdNorm(n)
        X <- X + Z^2
    }
    return(X)
}
```

Functions can take any number of arguments

A trial with $k = 17$ degrees of freedom, and 2 values is executed as follows:

```
rChisq(2, 17)


## [1] 14.26515 18.01847
```

Use of default arguments

To give the user of a function a hint as to the kind of input that the function is expecting, we may give default values to some arguments.

If the user doesn't specify the value, the default will be used.

Example

We could have used the *header*, i.e. the first line of the function,

```
rChisq <- function(n, k = 1)
```

to indicate that if a user called `rChisq(10)` without specifying `k`, then it should act as though `k = 1`.

Function environment

We conclude our brief discussion of functions with a mention of the function's *environment*.

We won't give a complete description here, but will limit ourselves to the following circular definition: the environment is a reference to the environment in which the function was defined.

This has implications for where objects are that the function can access.

Example

A function `myfun` is created in an environment that does not contain `mydata`:

```
myfun <- function() {
      mymean <- mean(mydata)
      return(mymean)
}
myfun() # execute function

## Error in mean(mydata): object 'mydata' not found
```

Example

Now, consider what happens when `mydata` is in the function's environment:

```
mydata <- rChisq(4, 1)
myfun()  # mydata exists now and mymean exists internally to

## [1] 0.3870882
```

Note, as well, that `mymean` does not exist in the workspace, only locally to myfun:

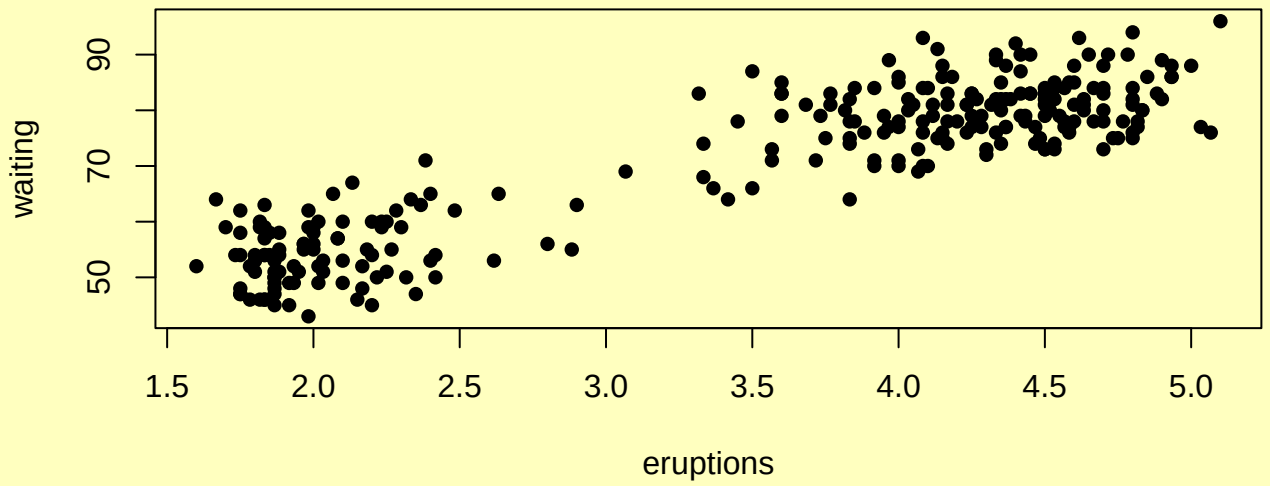
```
mymean

## Error in eval(expr, envir, enclos): object 'mymean'
not found
```

Exercise - smoothing a scatterplot

The `faithful` data set consists of the waiting times until the next eruption of the Old Faithful geyser together with the corresponding eruption times.

The scatterplot for these data can be obtained by typing

```
plot(faithful, pch=16, col="black")
```



Smoothing a scatterplot

A simple way to make predictions from such data is to smooth the scatterplot of the y values that are plotted against the x values.

One way to do this is to use moving averages.

In other words, just take averages of y values that are near each other according to their x values.

Join these averages together to form a curve.

Smoothing a scatterplot

We will construct a function called `smoother()` which outputs a new data frame consisting of a column of equally spaced x values and a column of corresponding local averages, taking the following arguments

- `x`**: the vector of x values**
- `y`**: the vector of y values**
- `x.min`**: a constant which specifies the left boundary of the plotted curve**
- `x.max`**: a constant which specifies the right boundary of the plotted curve**
- `window`**: a constant giving the range of x values used to calculate the moving averages**

Smoothing a scatterplot - function header

The function header:

```
smoother <- function(x, y, x.min, x.max, window) {
```

Smoothing a scatterplot - function output

The output for this function will be a data frame with 2 columns: `x` and `y`, which will correspond to the y-averages and the corresponding x locations where the averages are taken.

Thus, we include a line such as the one at the end of the following body-less function:

```
smoother <- function(x, y, x.min, x.max, window) {
    ...
    data.frame(x = xpoints, y = yaverages)
}
```

Smoothing a scatterplot - function body

We use the `seq()` function to create a sequence of 401 equally spaced x values, starting at `x.min` and ending at `x.max`.

We include a line of code that assigns this sequence to an object called `xpoints`:

```
smoother <- function(x, y, x.min, x.max, window) {
    xpoints <- seq(x.min, x.max, len=401)
    ...
    data.frame(x = xpoints, y = yaverages)
}
```

Smoothing a scatterplot - function body

We use a `for()` loop to calculate the column of corresponding `yaverages`.

To do this, we need to first initialize the `yaverages` object to have the same number of elements as `xpoints`.

Thus, we Include the following line in the function:

```
    yaverages <- numeric(length(xpoints))
```

Smoothing a scatterplot - function body

Next, for each value of i, running from 1 through `xpoints`, we need to determine which elements of the original data vector `x` are close to `xpoints[i]`, so that we can take the average of the corresponding `y` values only.

In other words, we want to determine the `indices` of `x` for which the absolute value of `x - xpoints[i]` is less than the `window` parameter that was specified in the argument to the `smoother()` function.

```
smoother <- function(x, y, x.min, x.max, window) {
    xpoints <- seq(x.min, x.max, len=401)
    yaverages <- numeric(length(xpoints))
    for (i in 1:length(xpoints)) {
        indices <- which(abs(x - xpoints[i]) < window)
    }
    data.frame(x = xpoints, y = yaverages)
}
```

Smoothing a scatterplot - function body

Within the `for()` loop just created, we add a line of code which assigns the average of the values in `y[indices]` to `yaverages[i]`:

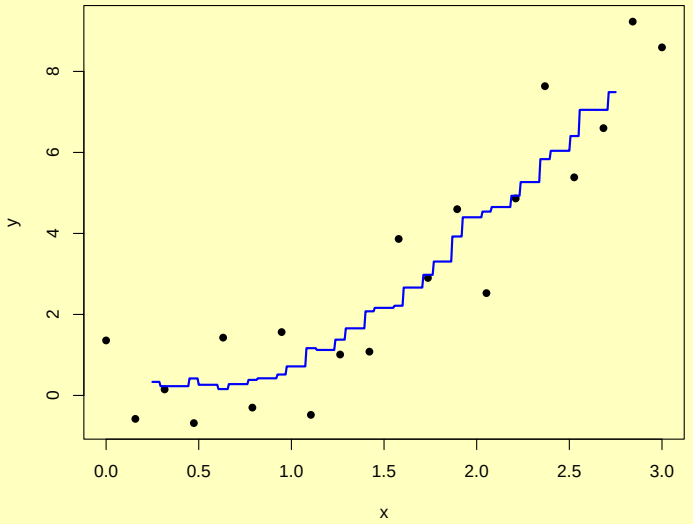
```
smoother <- function(x, y, x.min, x.max, window){
    xpoints <- seq(x.min, x.max, len=401)
    yaverages <- numeric(length(xpoints))
    for (i in 1:length(xpoints)) {
        indices <- which(abs(x - xpoints[i]) < window)
        yaverages[i] <- mean(y[indices])
    }
    data.frame(x = xpoints, y = yaverages)
}
```

Smoothing a scatterplot - testing the function

We should now have a working function, which we can test on artificial data, in this case, a noisy parabola:

For example,

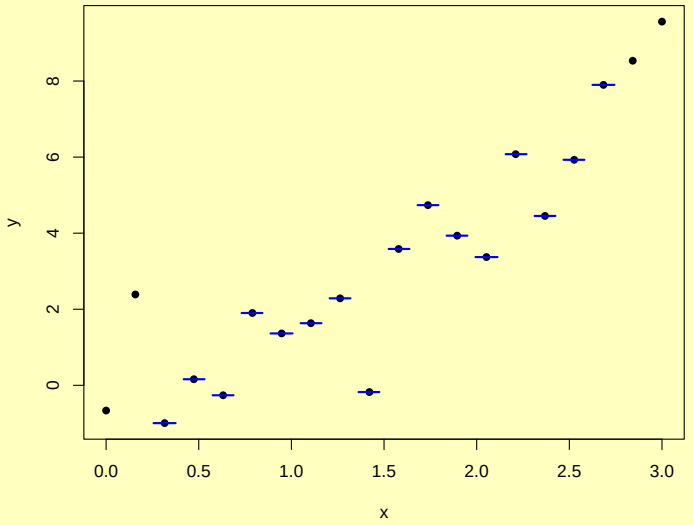
```
x <- seq(0, 3, length=20)
y <- x^2 + rnorm(20)
plot(x, y, pch=16)
lines(smoother(x, y,
    x.min=0.25,
    x.max=2.75,
    window=0.5), lwd=2)
```



Smoothing a scatterplot - testing the function

For example,

```
x <- seq(0, 3, length=20)
y <- x^2 + rnorm(20)
plot(x, y, pch=16)
lines(smoother(x, y,
    x.min=0.25,
    x.max=2.75,
    window=0.06), lwd=2)
```



When `window` is very close to 0, we see missing pieces in the smooth curve. Why?

If the `window` parameter is too close to 0, there will be no data points close enough to some of the values in `xpoints`, so you will be averaging no data, thus, there is nothing to plot.

Adding an error message - using `if` and `stop()`

To avoid such a problem, we include an error message in the function to tell the user that the `window` parameter is too small.

The `stop()` function provides such a message and aborts execution of the function.

Within the `for` loop to your function, we include the following lines of code:

```
  if (length(indices) < 1) {
        stop("Your choice of window width is too small.")
        } else {
              yaverages[i] <- mean(y[indices])
  }
```

Smoothing a scatterplot

```
smoother <- function(x, y, x.min, x.max, window=1) {
    xpoints <- seq(x.min, x.max, len=401)
    yaverages <- numeric(401)
    for (i in 1:length(xpoints)) {
        indices <- which(abs(x - xpoints[i]) < window)
        if (length(indices) < 1) {
            stop("Your choice of window width is too small.")
            } else {
            yaverages[i] <- mean(y[indices])
        }
    }
    data.frame(x = xpoints, y = yaverages)
}
```

Smoothing a scatterplot

Finally, note that the so-called "smooth" curve is still quite bumpy.

To reduce the bumpiness, we can iterate the smoothing procedure.

In other words, we can repeat the smoothing procedure on the output from `smoother()`, as follows:

```
output1 <- smoother(x, y, 0.25, 2.75,  window = .5)
output2 <- smoother(output1$x, output1$y, 0.25, 2.75,
      window = .25)
```

Observe that the `window` parameter does not have to be the same for each iteration.

Smoothing a scatterplot

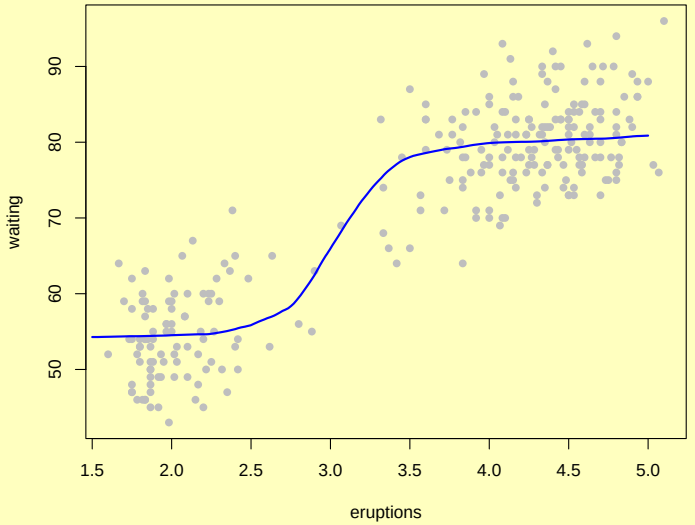
We now construct a new function called `doublesmoother()` which takes the same arguments as `smoother`, but where `window` is now assumed to be a vector with 2 elements.

The output from `doublesmoother()` is again a data frame consisting of `xpoints` and `yaverages` as in `smoother()` but should be the result of the second round of smoothing.

```
doublesmoother <- function(x, y, x.min, x.max, window) {
    output1 <- smoother(x, y, x.min, x.max,  window[1])
    output2 <- smoother(output1$x, output1$y, x.min, x.max,
           window[2])
    output2
}
```

Smoothing the `faithful` scatterplot

Applying the `doublesmoother()` function to the `faithful` data frame, with a `window` parameter of 1 unit for the first level of smoothing and a value of 0.1 unit for the second level, and equally spaced `xpoints` in the interval $[1.5, 5.0]$**:**




```
plot(faithful, pch=16, col="grey")
lines(doublesmoother(faithful$eruptions, faithful$waiting,
      1.5, 5.0, c(1, 0.1)), col="blue", lwd=2)
```