Common R

Dom Owens

15/10/2019

R is an interpreted language (as opposed to a compiled language, such as C or PASCAL) which is converted to machine language (low-level) line by line.

In consideration of speed, we avoid For loops and instead use **vectorisation**, which applies the same operation to every entry in the vector.

We demonstrate vectorised code is faster and better-looking. First, the For loop:

```
#Sum of cos(x) using a for loop
fsum1 = function(x) {
  fsum = 0
  for (i in 1:length(x)) {
      fsum = fsum + cos(x[i])
  }
  return(fsum)
}
```

Now the same functionality vectorised:

```
#Sum of cos(x) using vectorisation
fsum2 = function(x) sum(cos(x))
```

Compare these in terms of system time:

```
x = 1:1e6 #runif(1e+06, min = -1, max = 1)
system.time(fsum1(x))
```

```
##    user  system elapsed
##   0.107   0.000   0.107
```

```
system.time(fsum2(x))
```

```
##    user  system elapsed
##    0.03    0.00    0.03
```

Matrix operations

Consider nested For loops - we see these perform very badly indeed.

Coin toss example:

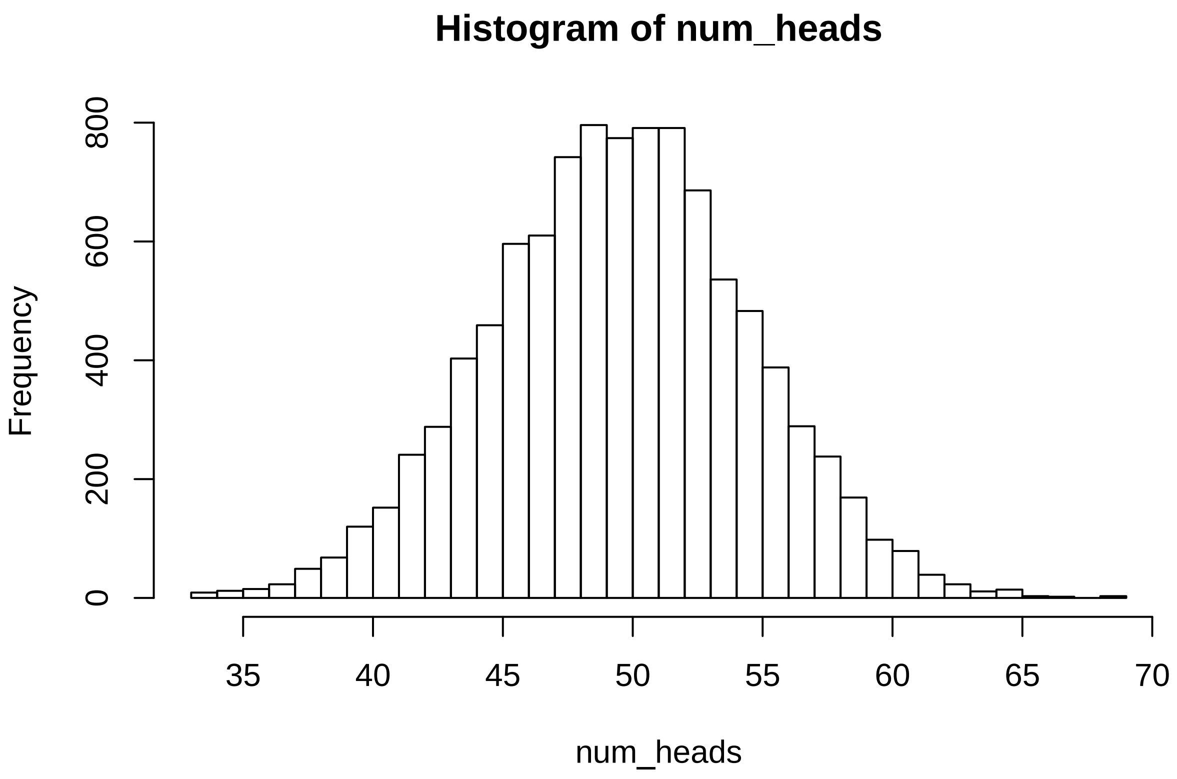
Repeat this n times: Toss a fair coin 100 times and return the observed amount of heads.

First, with nested For loops:

```
coin_toss1 <- function(n) {
  output <- rep(0, n) #initialise n vector
  for (i in 1:n) { #repeat n times
    num_heads <- 0
    for (j in 1:100) {
      num_heads <- num_heads + sample(c(0,1), 1) #sum 100 coin tosses
    }
    output[i] <- num_heads #store number of heads
  }
  return(output)
}
```

Now vectorised:

```
coin_toss2 <- function(n) { #simulate and sum n coin tosses
  tosses <- matrix(sample(c(0,1), n*100, replace=TRUE), nrow=n, ncol=100) #form matrix with rows as exp
  return(rowSums(tosses)) #return row sums
}
num_heads <- coin_toss2(10000) #run with n = 10000
hist(num_heads, breaks=50) #plot histogram of experiment outcomes
```



And compare these methods in system time

```
n = 2000
system.time(coin_toss1(n))
```

```
##    user  system elapsed
##   0.876   0.017   0.893
```

```
system.time(coin_toss2(n))
```

```
##    user  system elapsed
##   0.009   0.000   0.009
```

This worsens with higher dimensions of iteration; more For loops leads to a greater divergence in system time. *(Scales quadratically?)*

`apply()` functions

We now look at an alternative method for iterating, the `apply` family. This includes `apply()`, `lapply()` for lists, `sapply()` for simplification *(?)*, `mapply()` for multiple dimensions. These apply the same function on objects within an object.

`apply()` has the syntax `apply(x, margin, fun, ...)` where `x` is an array, `margin` is the vector of subscripts to apply to, and `fun` is the function to be applied.

Consider this matrix example.

```
x <- matrix("n", nrow=5, ncol=4) #5*4 matrix, entries "n"
apply(x, 1, cat) #apply cat (concatenate) function on rows
```

```
## n n n nn n n nn n n nn n n nn n n n
```

```
## NULL
```

```
apply(x, c(1,2), cat) #apply cat (concatenate) function on rows and columns
```

```
## nnnnnnnnnnnnnnnnnnnn
```

```
## NULL
```

`lapply()` has the syntax `lapply(x, fun, ...)` and outputs a list.

```
lapply(rep(1,5), exp) #output list of exp(1), ..., exp(5)
```

```
## [[1]]
## [1] 2.718282
##
## [[2]]
## [1] 2.718282
##
## [[3]]
## [1] 2.718282
##
## [[4]]
## [1] 2.718282
##
## [[5]]
## [1] 2.718282
```

```
x <- list(a=1:10, b=exp(-3:3), c=c(TRUE,FALSE,FALSE,TRUE)) #construct x list
lapply(x, mean) #apply mean function to lists within x
```

```
## $a
## [1] 5.5
##
## $b
## [1] 4.535125
##
## $c
## [1] 0.5
```

```
lapply(x, quantile, probs=(1:50)/50) #apply quantile function to lists
```

```
## $a
##    2%    4%    6%    8%   10%   12%   14%   16%   18%   20%   22%   24%
##  1.18  1.36  1.54  1.72  1.90  2.08  2.26  2.44  2.62  2.80  2.98  3.16
##   26%   28%   30%   32%   34%   36%   38%   40%   42%   44%   46%   48% 
##  3.34  3.52  3.70  3.88  4.06  4.24  4.42  4.60  4.78  4.96  5.14  5.32 
##   50%   52%   54%   56%   58%   60%   62%   64%   66%   68%   70%   72% 
##  5.50  5.68  5.86  6.04  6.22  6.40  6.58  6.76  6.94  7.12  7.30  7.48 
##   74%   76%   78%   80%   82%   84%   86%   88%   90%   92%   94%   96% 
##  7.66  7.84  8.02  8.20  8.38  8.56  8.74  8.92  9.10  9.28  9.46  9.64 
##   98%  100% 
##  9.82 10.00 
## 
## $b
##          2%          4%          6%          8%         10%         12% 
##  0.06005285  0.07031864  0.08058443  0.09085021  0.10111600  0.11138178 
##         14%         16%         18%         20%         22%         24% 
##  0.12164757  0.13191335  0.15393882  0.18184411  0.20974941  0.23765471 
##         26%         28%         30%         32%         34%         36% 
##  0.26556001  0.29346531  0.32137061  0.34927591  0.39316426  0.46901873 
##         38%         40%         42%         44%         46%         48% 
##  0.54487320  0.62072766  0.69658213  0.77243660  0.84829107  0.92414553 
##         50%         52%         54%         56%         58%         60% 
##  1.00000000  1.20619382  1.41238764  1.61858146  1.82477528  2.03096910 
##         62%         64%         66%         68%         70%         72% 
##  2.23716292  2.44335674  2.64955056  3.09194377  3.65243668  4.21292960 
##         74%         76%         78%         80%         82%         84% 
##  4.77342251  5.33391542  5.89440833  6.45490124  7.01539416  7.89691533 
##         86%         88%         90%         92%         94%         96% 
##  9.42049303 10.94407073 12.46764843 13.99122613 15.51480383 17.03838153 
##         98%        100% 
## 18.56195922 20.08553692 
## 
## $c
##   2%   4%   6%   8%  10%  12%  14%  16%  18%  20%  22%  24%  26%  28%  30% 
## 0.00 0.00 0.00 0.00 0.00 0.00 0.00 0.00 0.00 0.00 0.00 0.00 0.00 0.00 0.00 
##  32%  34%  36%  38%  40%  42%  44%  46%  48%  50%  52%  54%  56%  58%  60% 
## 0.00 0.02 0.08 0.14 0.20 0.26 0.32 0.38 0.44 0.50 0.56 0.62 0.68 0.74 0.80 
##  62%  64%  66%  68%  70%  72%  74%  76%  78%  80%  82%  84%  86%  88%  90% 
## 0.86 0.92 0.98 1.00 1.00 1.00 1.00 1.00 1.00 1.00 1.00 1.00 1.00 1.00 1.00 
##  92%  94%  96%  98% 100% 
## 1.00 1.00 1.00 1.00 1.00
```

`sapply()` does the same as `lapply()` but attempts to coerce the returned list into a simpler form; for instance, if a list of singleton lists would be returned by `lappy()` then `sapply()` will convert this to a single list of numbers.

```
sapply(x, quantile, probs=(1:50)/50) #return quantiles in single list
```

```
##          a           b    c
## 2%    1.18  0.06005285 0.00
## 4%    1.36  0.07031864 0.00
## 6%    1.54  0.08058443 0.00
## 8%    1.72  0.09085021 0.00
## 10%   1.90  0.10111600 0.00
## 12%   2.08  0.11138178 0.00
## 14%   2.26  0.12164757 0.00
## 16%   2.44  0.13191335 0.00
## 18%   2.62  0.15393882 0.00
## 20%   2.80  0.18184411 0.00
## 22%   2.98  0.20974941 0.00
## 24%   3.16  0.23765471 0.00
## 26%   3.34  0.26556001 0.00
## 28%   3.52  0.29346531 0.00
## 30%   3.70  0.32137061 0.00
## 32%   3.88  0.34927591 0.00
## 34%   4.06  0.39316426 0.02
## 36%   4.24  0.46901873 0.08
## 38%   4.42  0.54487320 0.14
## 40%   4.60  0.62072766 0.20
## 42%   4.78  0.69658213 0.26
## 44%   4.96  0.77243660 0.32
## 46%   5.14  0.84829107 0.38
## 48%   5.32  0.92414553 0.44
## 50%   5.50  1.00000000 0.50
## 52%   5.68  1.20619382 0.56
## 54%   5.86  1.41238764 0.62
## 56%   6.04  1.61858146 0.68
## 58%   6.22  1.82477528 0.74
## 60%   6.40  2.03096910 0.80
## 62%   6.58  2.23716292 0.86
## 64%   6.76  2.44335674 0.92
## 66%   6.94  2.64955056 0.98
## 68%   7.12  3.09194377 1.00
## 70%   7.30  3.65243668 1.00
## 72%   7.48  4.21292960 1.00
## 74%   7.66  4.77342251 1.00
## 76%   7.84  5.33391542 1.00
## 78%   8.02  5.89440833 1.00
## 80%   8.20  6.45490124 1.00
## 82%   8.38  7.01539416 1.00
## 84%   8.56  7.89691533 1.00
## 86%   8.74  9.42049303 1.00
## 88%   8.92 10.94407073 1.00
## 90%   9.10 12.46764843 1.00
## 92%   9.28 13.99122613 1.00
## 94%   9.46 15.51480383 1.00
## 96%   9.64 17.03838153 1.00
## 98%   9.82 18.56195922 1.00
## 100% 10.00 20.08553692 1.00
```

`mapply()` extends `sapply()` to multiple dimensions. Has the syntax `mapply(fun, ...)` and the `...` denotes arguments to the function.

```
mapply(function(x, y) seq_len(x) + y, c(a=1, b=2, c=3), c(A=10, B=0, C=-10)) #return list according to
```

```
## $a
## [1] 11
##
## $b
## [1] 1 2
##
## $c
## [1] -9 -8 -7
```

How do these compare to For loops in terms of speed? We use x^x elementwise on $(1, ..., 1e6)$ with `sapply` and For loops.

```r
ftest1 <- function(x) x^x
x <- 1:1e6 #runif(1e+06, min = -1, max = 1)
ftest1_apply <- function(x) sapply(x, ftest1) # with sapply
ftest1_loop <- function(x) { #as for loop
  y <- rep(NA, length(x))
  for (i in 1:length(x)) y[i] <- ftest1(x[i])
  y
}
system.time(ftest1_apply(x)) #sapply time
```

```
##    user  system elapsed
##   1.776   0.020   1.796
```

```r
system.time(ftest1_loop(x)) #loop time
```

```
##    user  system elapsed
##   0.499   0.004   0.503
```

```r
system.time(ftest1(x)) #vectorised time
```

```
##    user  system elapsed
##   0.063   0.000   0.064
```

So the apply function takes about 3 times as long as the for loop function, which itself takes around 8 times as long as the vector function.

Map, reduce, filter

`Map` is a wrapper for `mapply` which doesn't simplify the result (often returning a list of lists). The function given as input is applied to each entry of the input object, and a list is returned. This can be used with lambda notation, a one line function.

```r
x <- 1:5
y <- Map({function(a) a*a}, x) #square entries of x, using Map
y
```

```
## [[1]]
## [1] 1
##
## [[2]]
## [1] 4
##
## [[3]]
## [1] 9
##
## [[4]]
## [1] 16
##
## [[5]]
## [1] 25
```

`Reduce` acts similarly to `Map`, applying a function succesively to pairs in the vector/list and returning a single number.

```r
Reduce(function(a,b)a*b, x) #multiply all entries in x
```

```
## [1] 120
```

`Filter` processes a vector to keep only those entries which meet a specific condition, giving a possibly shorter vector as a result.

```r
Filter(function(x) x>2, x) #return entries of x greater than 2
```

```
## [1] 3 4 5
```

```r
Filter(function(x)x-1, x) #remove entries equal to 1
```

```
## [1] 2 3 4 5
```

Parallelisation

Why use parallel programming? Typically, R will use one core of the CPU. Larger jobs can be parallelised to make use of more than one core or more than one processor, allowing faster processing. Problems may also be *memory-bound*, requiring more RAM than is available; *I/O bound*, taking too long to read and write to the permanent memory; or *network bound*, taking too long to transfer over a connection.

Speed gains are not linear - n-fold increases in processes do not cause n-fold gains in time.

Using `mclapply()`

Often, computations will depend on previous results, preventing vectorisation.

```r
library(parallel)
num_cores <- detectCores() #how many cores does this machine have?
num_cores
```

```
## [1] 8
```

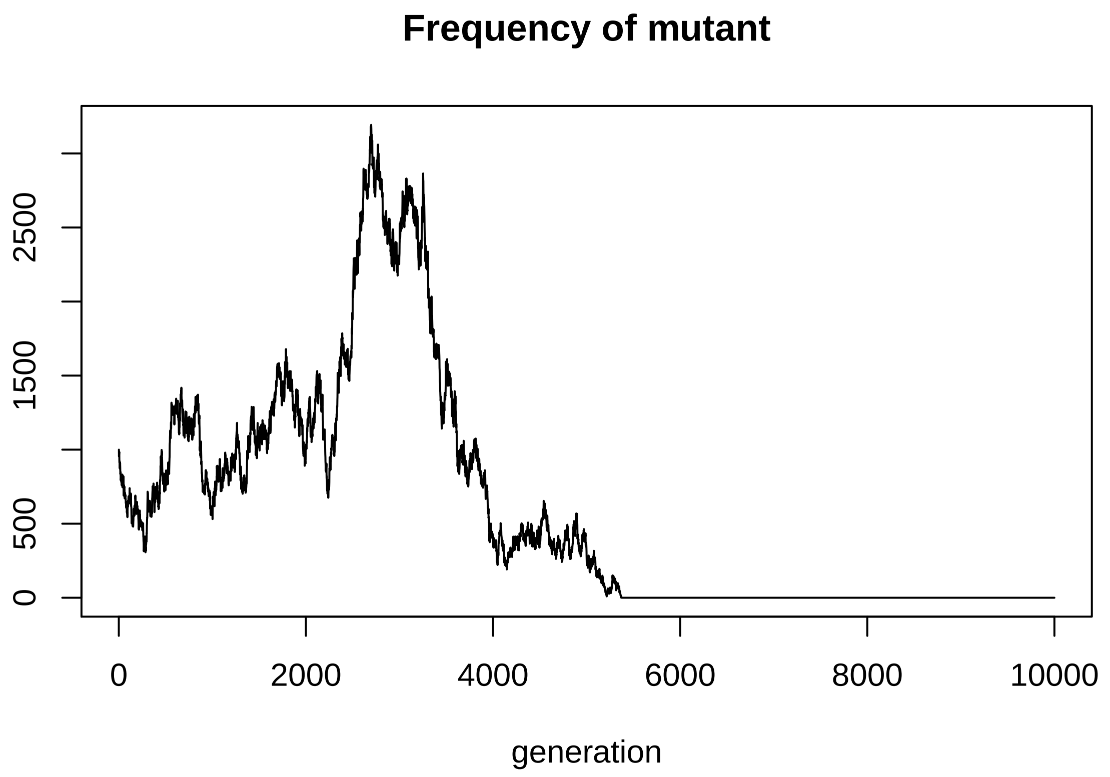
We demonstrate parallelisation on the **Wright-Fisher model**. X_k is a Markov Process such that

$$X_k \sim Binomial(N, \frac{X_{k-1}}{N})$$

Clearly, this is step-dependent, so can't be vectorised.

First, we simulate the model without parallelisation.

```r
simulate_wright_fisher <- function(N, n_gen, init_freq) {
  # population size, number of generation to simulate, and initial frequency of mutant
  counts <- numeric(n_gen)
  counts[1] <- round(N * init_freq)
  for (k in 2:n_gen) {
    counts[k] <- rbinom(1, size=N, prob=counts[k-1]/N)
  }
  return(counts)
}
counts <- simulate_wright_fisher(5000, 10000, 0.2)
plot(counts, type='l', main='Frequency of mutant', xlab='generation', ylab='')
```



We inspect how long this takes.

```
system.time(simulate_wright_fisher(5000, 10000, 0.2)) #how long did this take?
```

```
##    user  system elapsed
##   0.024   0.008   0.032
```

We can, however, parallelise multiple iterations of the same function.

```
library(MASS)
wrapper <- function(init_freq) simulate_wright_fisher(5000, 10, init_freq) #specify model call as funct
init_freqs <- rep(0.2, 3)
res <- mclapply(init_freqs, wrapper, mc.cores=4) #apply model call to initial frequency vector, using 4
res
```

```
## [[1]]
##  [1] 1000  978  932  950  939 1005 1046 1027 1038 1053
##
## [[2]]
##  [1] 1000 1000 1001  973  980  960  966  967  957  984
##
## [[3]]
##  [1] 1000  979  937  906  926  956  927  935  965  953
```

```
wrapper2 <- function(init_freq) tail(simulate_wright_fisher(5000, 1000, init_freq), 1) #return last ent
init_freqs2 <- rep(0.2, 500)
system.time(lapply(init_freqs2, wrapper2)) #how long?
```

```
##    user  system elapsed
##   1.031   0.000   1.031
```

```
system.time(mclapply(init_freqs2, wrapper2, mc.cores=2)) #how long with two cores?
```

```
##    user  system elapsed 
##   0.546   0.064   0.586
```

```
system.time(mclapply(init_freqs2, wrapper2, mc.cores=4)) #how long with four?
```

```
##    user  system elapsed 
##   0.573   0.023   0.383
```

We see the time taken decreases, but at a decreasing rate with respect to the number of cores used.

`foreach()` and `doparallel()`

`foreach()` parallelises the For loop and returns a list. `%do%` evaluates sequentially, and `%dopar%` evaluates in parallel. This is preferable to `mclapply()` in terms of flexibility.

```
library(foreach)
foreach (i=1:3) %do% {
  i*i
}
```

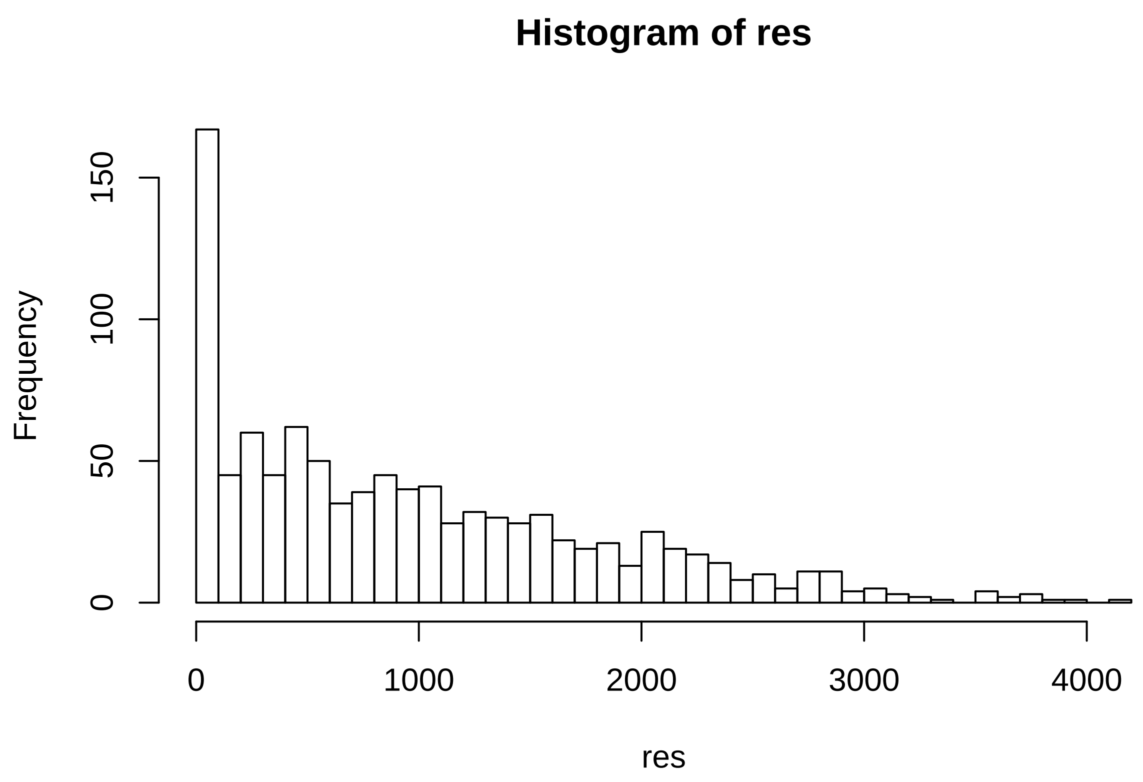
```
## [[1]]
## [1] 1
## 
## [[2]]
## [1] 4
## 
## [[3]]
## [1] 9
```

```
res <- foreach (i=1:2, .combine=rbind) %do% { #run model twice sequentially, combining by row
  simulate_wright_fisher(5000, 10, 0.2)
}
res
```

```
##          [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9] [,10]
## result.1 1000 1039 1063 1044 1023 1044 1039 1019  985  1023
## result.2 1000 1027 1012  999  988  989 1010 1009 1043  1031
```

The `.final` argument determines what should be returned by `foreach()`. To select the terminal number of mutants in the population, we use the following:

```
n_gen <- 1000
res <- foreach (i=1:1000, .combine=rbind, .final=function(x) x[,n_gen]) %do% { #select final entry
  simulate_wright_fisher(5000, n_gen, 0.2)
}
hist(res, breaks=50)
```



Let's introduce parallel computation.

```
library(doParallel)
```

```
## Loading required package: iterators
```

```
registerDoParallel(4) # register 4 cores
n_runs <- 500 #500 runs
system.time(foreach (i=1:n_runs, .combine=rbind, .final=function(x) x[,n_gen]) %do% { #how long does th
  simulate_wright_fisher(5000, n_gen, 0.2)
})
```

```
##    user  system elapsed
##   1.264   0.004   1.269
```

```
system.time(foreach (i=1:n_runs, .combine=rbind, .final=function(x) x[,n_gen]) %dopar% {#how long does
  simulate_wright_fisher(5000, n_gen, 0.2)
})
```

```
##    user  system elapsed
##   2.289   0.184   0.542
```

```
t1 <- Sys.time()
res1 <-  foreach (i=1:n_runs, .combine=rbind, .final=function(x) x[,n_gen]) %do% {
  simulate_wright_fisher(5000, n_gen, 0.2)
}
t2 <- Sys.time()
res2 <- foreach (i=1:n_runs, .combine=rbind, .final=function(x) x[,n_gen]) %dopar% {
  simulate_wright_fisher(5000, n_gen, 0.2)
}
t3 <- Sys.time()
t2-t1
```

```
## Time difference of 1.287343 secs
```

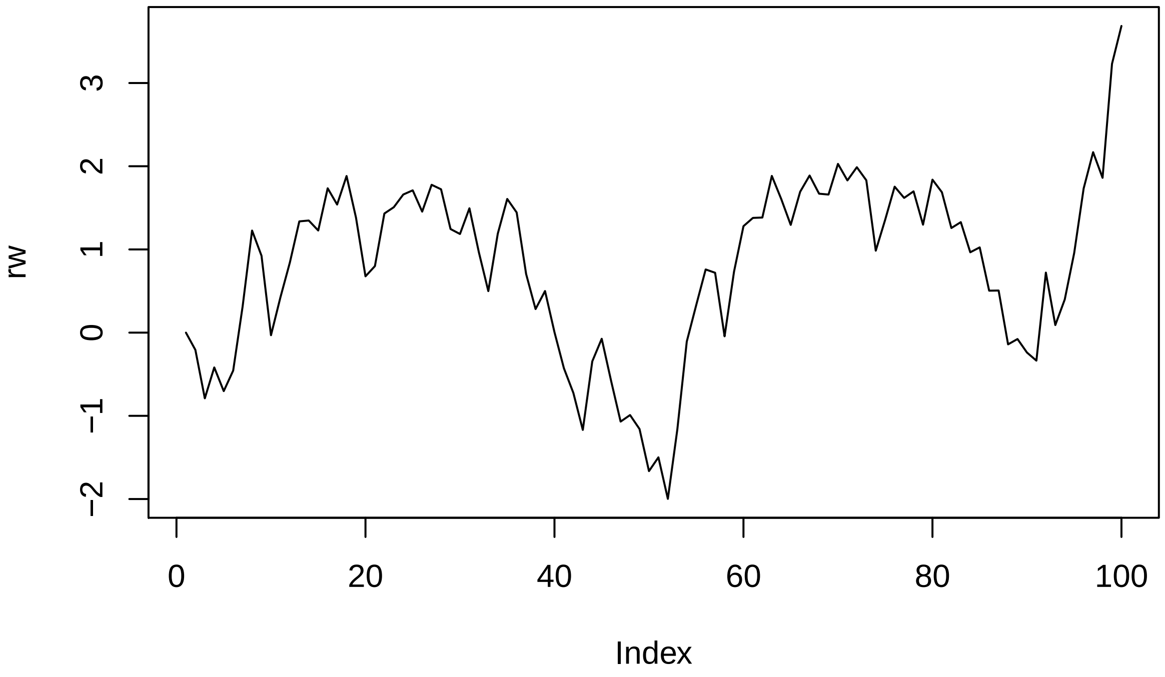
Consider now the damped random walk

$$X_t = \alpha X_{t-1} + Z_t;$$

where

$$Z_t \sim \mathcal{N}(0,1)\ ,\ |\alpha| < 1, \text{ and } X_t = 0$$

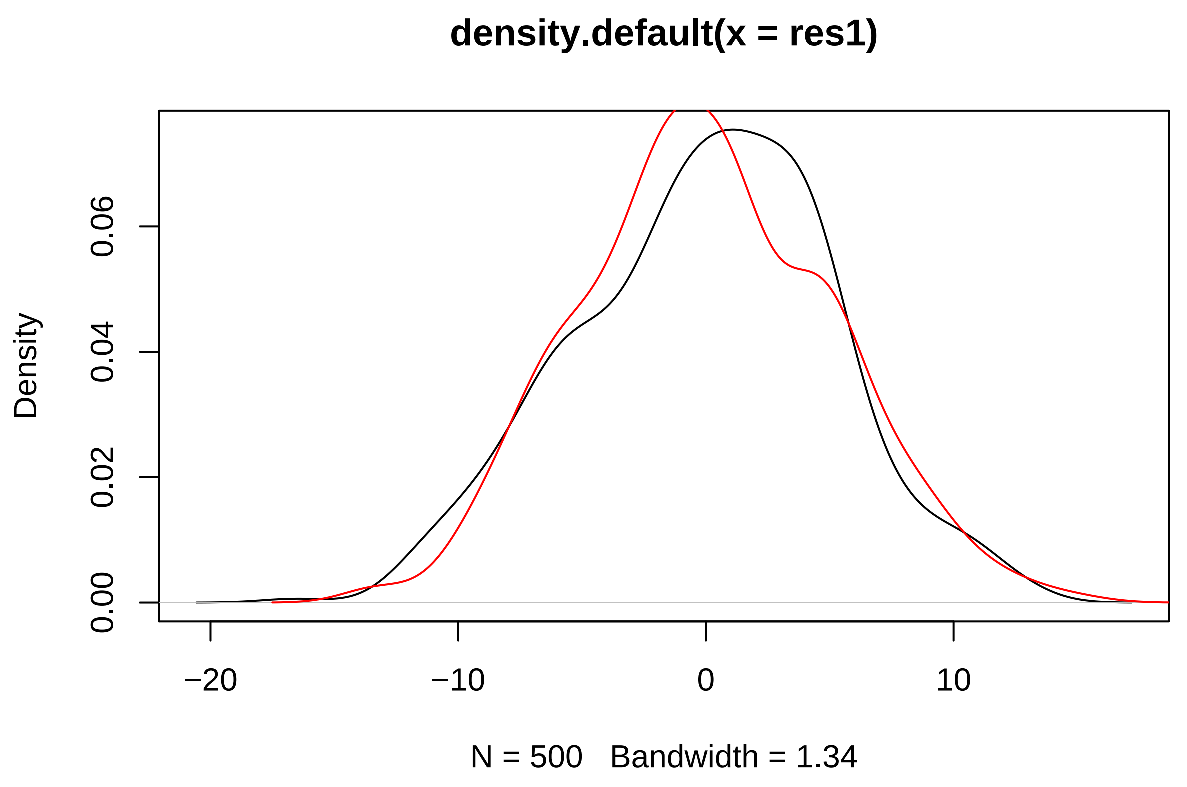
```
simulate_rw <- function(X0, alpha, n){ #create random walk
  X <- rep(X0, n)
  for (i in 2:n) {
    X[i] <- X[i-1] + alpha*rnorm(1, 0,1)
  }
  X
}
rw <- simulate_rw(0,0.5,100)
plot(rw, type = "l")
```



Let's compare parallelisation methods for this.

```
n_gen <- 100
n_runs <- 500
t1 <- Sys.time()
res1 <- foreach (i=1:n_runs, .combine=rbind, .final=function(x) x[,n_gen]) %do% { #run random walk 500
  simulate_rw(0,0.5,n_gen)
}
plot(density(res1))


t2 <- Sys.time()
res2 <- foreach (i=1:n_runs, .combine=rbind, .final=function(x) x[,n_gen]) %dopar% { #run 500 times in
  simulate_rw(0,0.5,n_gen)
}
lines(density(res2), col="red") #plot parallel density in red
```



```
t3 <- Sys.time()
t2-t1
```

```
## Time difference of 0.2769976 secs
```

Note here that the empirical means of both stay around zero, suggesting the series is stationary.

```
# clean up the cluster
stopImplicitCluster()
```