STAT 210
Applied Statistics and Data Analysis
Problem List 1 - Solutions

Joaquin Ortega

Fall 2021

Exercise 1

Exercise 1

Using the functions `rep` and `seq`, generate the following sequences

1. 10 10 10 10 10 9 9 9 9 8 8 8 7 7 6 5 4 4 3 3 3 2 2 2 2 1 1 1 1 1
2. 1 1 2 3 3 4 5 5 6 7 7 8 9 9 10
3. 100.0000 100.2222 100.4444 100.6667 100.8889 101.1111 101.3333 101.5556 101.7778 102.0000
4. 1.0 1.0 1.0 1.2 1.4 1.4 1.4 1.6 1.8 1.8 1.8 2.0
5. 1 2 3 4 5 2 3 4 5 6 3 4 5 6 7 4 5 6 7 8 5 6 7 8 9

Exercise 1

1.

```
rep(10:1,c(5:1,1:5))
```

```
##  [1] 10 10 10 10 10  9  9  9  9  8  8  8  7  7  6
## [16]  5  4  4  3  3  3  2  2  2  2  1  1  1  1  1
```

2.

```
rep(1:10,rep(c(2,1),5))
```

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

3

```r
seq(100,102,length.out = 10)
```

```
##  [1] 100.0000 100.2222 100.4444 100.6667 100.8889
##  [6] 101.1111 101.3333 101.5556 101.7778 102.0000
```

4

```r
rep(seq(1,2,0.2),rep(c(3,1),3))
```

```
##  [1] 1.0 1.0 1.0 1.2 1.4 1.4 1.4 1.6 1.8 1.8 1.8
## [12] 2.0
```

Exercise 1

5

```
1:5 + rep(0:4, each = 5)
```

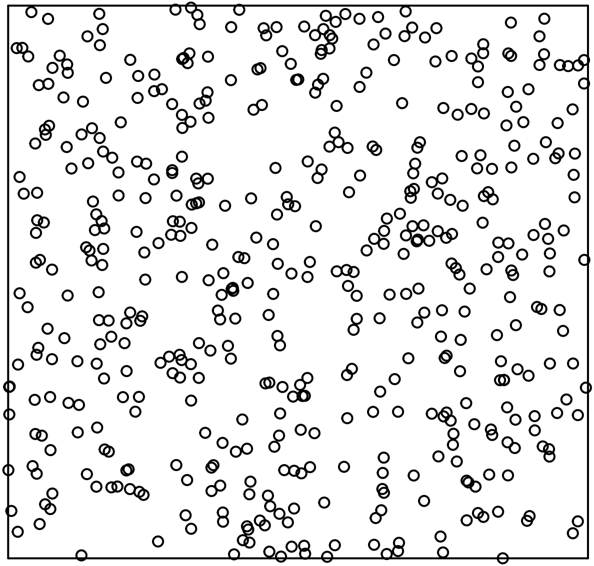
```
##  [1] 1 2 3 4 5 2 3 4 5 6 3 4 5 6 7 4 5 6 7 8 5 6 7
## [24] 8 9
```

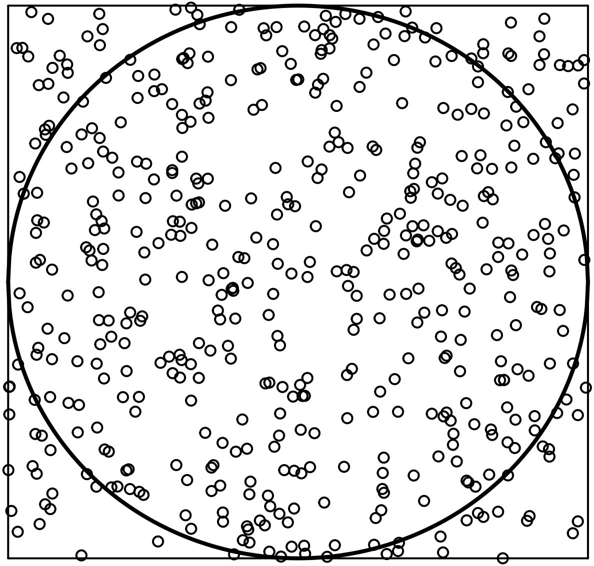
Exercise 2

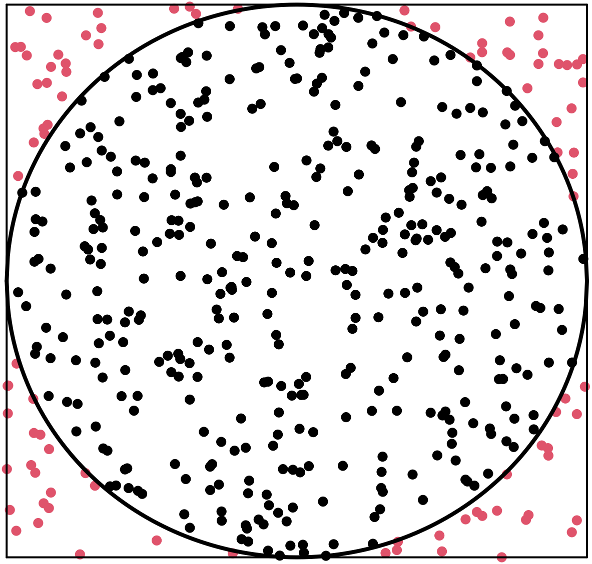
Use the Montecarlo method for estimating π.

The probability that a number generated uniformly at random in the square of sides $[-1, 1]$ falls inside the circle with center the origin and radius equal to 1 is the ratio of the area of the circle over the area of the square.

Exercise 2







$$\begin{aligned} P(\text{Point falls inside the circle}) &= \frac{\text{Area of circle}}{\text{Area of square}} \\ &= \frac{\pi r^2}{(2r)^2} = \frac{\pi}{4} \end{aligned}$$

Montecarlo strategy:

- Simulate a large number of random points in the square.
- Count how many fall inside the circle.
- The proportion of the number that fall inside the circle to the total number of points is an estimate of $\pi/4$:

$$\frac{\pi}{4} \approx \frac{\text{number of points inside the circle}}{\text{total number of points}}$$

Exercise 2

We will generate 10,000 random numbers in the square and count how many of them fall inside the unit circle.

Method 1

Using vectorized operations:

1. Generate 10,000 points with uniform distribution in the square of sides [-1,1].
2. Count how many of them fall inside the unit circle.
3. Calculate the proportion of points that fall inside the unit circle.
4. Calculate the error by subtracting your estimate from π.

```r
x <- runif(10000,-1,1)
y <- runif(10000,-1,1)
z <- x^2+y^2
asum <- sum(as.numeric(z<1))
(piest4 <- 4*asum/10000)
```

```
## [1] 3.1524
```

```r
(error4 = abs(pi-piest4))
```

```
## [1] 0.01080735
```

Loops

A loop is a cycle of operations that are repeated, possibly with changes, according to an index.

In `R` there are three expressions for controlling loops

`for`, `while` and `repeat`.

Vector and matrix operations in `R` are faster and more efficient.

Whenever possible, try to avoid loops and use array operations and functions such as `apply`.

The idea of a `for` loop is that there is a set of indices, `indexset` and for each value of the index in `indexset` a series of commands is executed.

The commands will usually depend on the index value and the process is controlled by the `for` function. The syntax is:

```
for (i in indexset) {R commands}
```

where `indexset` is a vector. For instance:

```
for (i in 1:4) print(i^2)
```

```
## [1] 1
## [1] 4
## [1] 9
## [1] 16
```

If there is only one command in the loop, the curly brackets may be omitted, as in the example above.

The vector `indexset` may be of any mode:

```
transp <- c('car','bus','motorcycle')
for (i in transp) {
  print(paste('I came by',i))
}
```

```
## [1] "I came by car"
## [1] "I came by bus"
## [1] "I came by motorcycle"
```

To write values at the end of a loop, a function like `print` should be used:

```
for (i in 1:4) {
  i
}

for (i in 1:4) {
  print(i)
}
```

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

Observe that in the first loop nothing was printed.

Exercise 2

We will generate 10,000 random numbers in the square and count how many of them fall inside the unit circle.

Method 2

Outside the loop, initialize a variable `s` for storing the sum using the command `s <- 0`.

Using a `for` loop:

1. Generate one point with uniform distribution in the square of sides [-1,1].
2. Determine whether the point falls inside the circle or not. The result should be a logical value: TRUE or FALSE
3. Using the function ‘as.numeric()‘ add this value to ‘s‘
4. Repeat this in a ‘for‘ loop 10,000 times.

Divide the value of `s` obtained using the loop by 10,000. This is the estimate of π.

Calculate the error by subtracting your estimate from π.

```r
s <- 0
for (i in 1:10000) {
  x <- runif(1,-1,1)
  y <- runif(1,-1,1)
  s <- s+as.numeric(x^2+y^2 < 1)
}
(piest <- 4*s/10000)
```

```
## [1] 3.142
```

```r
(error = abs(pi - piest))
```

```
## [1] 0.0004073464
```

Functions in R

A function is, simply, a sequence of instructions gathered together to form a new command.

User-defined functions in R have flexibility and capabilities similar to those of other modern programming languages, such as Python or C.

Functions input arguments and output values.

All the variables used in the function definition are internal variables and disappear once the function has been executed.

The use of a function in R is similar to mathematical use. In Mathematics we write $y = f(x)$ and in R

```
y <- function(x)
```

As an example, let's define a function called ppoly that evaluates the polynomial $x^3 - 2x^2$:

```
ppoly <- function (x)
  { return(x^3-(2*x^2)) }
ppoly(2)
```

```
## [1] 0
```

After declaring it, this function can be used like any other R function. It can only be distinguished from resident functions by its location, since it is stored in a different directory.

This function has the same flexibility as any other R function and can be used not only with variables, but also with vectors:

```r
x <- 1:5
ppoly(x/2)
```

```
## [1] -0.375 -1.000 -1.125  0.000  3.125
```

iteratively:

```r
ppoly(ppoly(x))
```

```
## [1]      -3       0     567   30720 410625
```

or with a matrix:

```
(x <- matrix(1:4, nrow =2))
```

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

```
ppoly(x)
```

```
##      [,1] [,2]
## [1,]   -1    9
## [2,]    0   32
```

Functions in `R`

The general syntax for defining a function is as follows

```
name <- function(input variables){
 function instructions
 return(results)
}
```

Expressions in italics must be replaced by valid expressions and names.

input variables are a list of parameters or objects that will be used internally by the function. They are set by the user and may have default values.

function instructions can be any valid `R` instructions, which will be evaluated as `R` executes them, and the results can be values or `R` objects.

Conditional statements

Sometimes, when defining a function, we want to employ different procedures depending on whether a certain condition is satisfied or not. For this we can use the `if else` function:

```
if (condition) {do this}
else {do that}
```

where `condition` must result in a logical value and `do this` and `do that` are sequences of commands.

If `condition` is `TRUE` the commands in `do this` are executed, otherwise those in `do that` are.

```
sevendiv <- function(x){
  if (x%%7==0) {
    print('x is divisible by 7')
  }
  else{
    print('x is not divisible by 7')
  }
}
sevendiv(49)
```

```
## [1] "x is divisible by 7"
```

```
sevendiv(18636)
```

```
## [1] "x is not divisible by 7"
```

However, if we try to evaluate this function on a vector we get a warning:

```
sevendiv(c(123,70))
```

```
## Warning in if (x%%7 == 0) {: the condition has
## length > 1 and only the first element will be used
```

```
## [1] "x is not divisible by 7"
```

The reason is that when `condition` results in a vector of logical values, the `if` function will only use the first component.

ifelse

This is a vector version of the previous function. The syntax is

```
ifelse(condition, expr1, expr2)
```

and the result is a vector with components equal to the result of executing `expr1` for those components for which `condition` is `TRUE` and executing `expr1` for those components for which `condition` is `FALSE`.

```
ifelse((5:8)%%2==0,'even','odd')
```

```
## [1] "odd"  "even" "odd"  "even"
```

```r
s <- 0
for (i in 1:10000) {
  x <- runif(1,-1,1)
  y <- runif(1,-1,1)
  if(x^2+y^2 < 1) s <- s+1
}
(piest <- 4*s/10000)
```

```
## [1] 3.1324
```

```r
(error = abs(pi - piest))
```

```
## [1] 0.009192654
```

Processing time

As an example, let's calculate the maximum of 10 million numbers randomly generated in $[0, 1]$.

We use now the functions `system.time` and `proc.time` which produce vectors of three numbers showing the user, system and total elapsed times for the currently running `R` process. It is the third number that is typically the most useful.

The user time is the CPU time charged for the execution of user instructions of the calling process, the system time is the CPU time charged for execution by the system on behalf of the calling process, and the elapsed time includes other stuff that the computer is doing, unrelated to your R session.

```r
x <- runif(10000000)
(t1 <- system.time(max(x)))
```

```
##    user  system elapsed
##   0.046   0.002   0.050
```

```r
pc <- proc.time()
cmax <- x[1]
for (i in 2:10000000) {
if(x[i]>cmax) cmax <- x[i] }
(t2 <- proc.time()-pc)
```

```
##    user  system elapsed
##   0.196   0.002   0.199
```

```r
t2/t1
```

```
##    user  system elapsed
## 4.26087 1.00000 3.98000
```

As an example of what not to do let us consider a simple exercise.

We want to create a vector containing the sequence of integers from 1 to 100,000.

Procedure 1

A quick way to do this is using the `seq` function that is built in `R`. Recall that for integer sequences with unit increment, we can use the colon (:) notation:

```
y <- 1:n
```

Procedure 2

Next, we use a loop, and define a numeric vector with length 100,000 before starting the loop. This is called *allocation*.

```
y <- numeric(100000)
for (i in 1:n) y[i] <- i
```

Procedure 3

Finally, we use a loop again but we do not define in advance the length of the vector we are going to need, and instead we build it up, adding a new component in each iteration. This is called *re-dimensioning*.

```
y <- NULL
for (i in 1:n) y <- c(y,i)
```

Now we execute the three functions and measure the time taken to complete each one.

Processing time

```r
system.time(y <- 1:100000)
```

```
##    user  system elapsed
##       0       0       0
```

```r
system.time({
  y <- numeric(100000)
  for (i in 1:100000) {
  y[i]<-i
}})
```

```
##    user  system elapsed
##   0.004   0.000   0.004
```

```r
system.time({
  y <- NULL
for (i in 1:100000) {
  y <- c(y,i)
}})
```

```
##    user  system elapsed
##  11.718   1.468  13.254
```

Exercise 3

Exercise 3

We will use the data set `mtcars`, that has information regarding fuel consumption and 10 related variables for 32 different car models.

1. Use the function `str` to explore the data set.
2. Using the function `subset`, create a new file named `file1` containing the variables `mpg`, `hp` and `wt`, but only for cars with 6 cylinders or more.
3. Using the functions `apply` and `mean`, calculate the mean value for each of the three variables in `file1`. Store the result in a vector called ‘means‘.
4. Using the function `sweep`, create a new object called `file2` with the data in `file1` after subtracting the means for each variable.
5. Using the function ‘apply‘, verify that the variables in ‘file2‘ have means zero.
6. Using the function `within` create a new columm in `file2` containing a new variable called `par1` calculated as `par1` = 0.8*`area` + 1.2*`peri`.

1. Use the function `str` to explore the data set.

```
str(mtcars)
```

```
## 'data.frame':    32 obs. of  11 variables:
##  $ mpg : num  21 21 22.8 21.4 18.7 18.1 14.3 24.4 22.8 1
##  $ cyl : num  6 6 4 6 8 6 8 4 4 6 ...
##  $ disp: num  160 160 108 258 360 ...
##  $ hp  : num  110 110 93 110 175 105 245 62 95 123 ...
##  $ drat: num  3.9 3.9 3.85 3.08 3.15 2.76 3.21 3.69 3.92
##  $ wt  : num  2.62 2.88 2.32 3.21 3.44 ...
##  $ qsec: num  16.5 17 18.6 19.4 17 ...
##  $ vs  : num  0 0 1 1 0 1 0 1 1 1 ...
##  $ am  : num  1 1 1 0 0 0 0 0 0 0 ...
##  $ gear: num  4 4 4 3 3 3 3 4 4 4 ...
##  $ carb: num  4 4 1 1 2 1 4 2 2 4 ...
```

2. Using the function `subset`, create a new file named `file1` containing the variables `mpg`, `hp` and `wt`, but only for cars with 6 cylinders or more.

```
file1 <- subset(mtcars,cyl >= 6, select = c(mpg, hp, wt))
str(file1)
```

```
## 'data.frame':    21 obs. of  3 variables:
##  $ mpg: num  21 21 21.4 18.7 18.1 14.3 19.2 17.8 16.4 17.3 ...
##  $ hp : num  110 110 110 175 105 245 123 123 180 180 ...
##  $ wt : num  2.62 2.88 3.21 3.44 3.46 ...
```

3. Using the functions `apply` and `mean`, calculate the mean value for each of the three variables in `file1`. Store the result in a vector called `means`.

```
(means <- apply(file1, 2, mean))
```

```
##       mpg        hp        wt
##  16.64762 180.23810   3.70519
```

4. Using the function `sweep`, create a new object called `file2` with the data in `file1` after subtracting the means for each variable.

```
file2 <- sweep(file1,2,means)
head(file2)
```

```
##                         mpg         hp         wt
## Mazda RX4          4.352381 -70.238095 -1.0851905
## Mazda RX4 Wag      4.352381 -70.238095 -0.8301905
## Hornet 4 Drive     4.752381 -70.238095 -0.4901905
## Hornet Sportabout  2.052381  -5.238095 -0.2651905
## Valiant            1.452381 -75.238095 -0.2451905
## Duster 360        -2.347619  64.761905 -0.1351905
```

5. Using the function `apply`, verify that the variables in `file2` have means zero.

```
apply(file2, 2, mean)
```

```
##           mpg            hp            wt
## -1.522592e-15 -5.413659e-15 -2.550870e-16
```

6. Using the function `within` create a new columm in `file2` containing a new variable called `par1` calculated as `par1` $=$ $1.4hp$ - 0.5`wt`.

```
file2 <- within(file2, {par1 = 1.4*hp+ 0.5*wt})
head(file2)
```

```
##                          mpg         hp         wt        par1
## Mazda RX4           4.352381 -70.238095 -1.0851905  -98.875929
## Mazda RX4 Wag       4.352381 -70.238095 -0.8301905  -98.748429
## Hornet 4 Drive      4.752381 -70.238095 -0.4901905  -98.578429
## Hornet Sportabout   2.052381  -5.238095 -0.2651905   -7.465929
## Valiant             1.452381 -75.238095 -0.2451905 -105.455929
## Duster 360         -2.347619  64.761905 -0.1351905   90.599071
```

Exercise 4

Exercise 4

a. Create a vector named `smpl1` with a sample of size 100 from the set of categories 'bad', 'reg', 'norm', 'good', 'exc'. The categories 'bad' and 'exc' should have probability 0.1, 'reg' and 'good' should have probability 0.2, and 'norm', probability 0.4.

b. Create a factor named `fact1` using the vector `smpl1` as input.

c. Create an ordered factor named `fact2` using the vector `smpl1` as input. The levels should be in increasing order.

d. Now, you want to reduce the number of categories to three: 'bad' and 'reg' will now be 'poor', 'norm' will be 'normal' and 'good' and 'exc' will now be 'great'. One easy way to do this is to use the `labels` argument in the function `factor` to rename the levels. Look up the help page for `factor`; there is an example that will show you how to do this. Name the resulting ordered factor `fact3`.

e. Use the function `table` to create tables for the three factors you have made. Comment on the differences.

a. Create a vector named `smpl1` with a sample of size 100 from the set of categories 'bad', 'reg', 'norm', 'good', 'exc'. The categories 'bad' and 'exc' should have probability 0.1, 'reg' and 'good' should have probability 0.2, and 'norm', probability 0.4.

```
smpl1 <- sample(c('bad','reg','norm','good','exc'), 100,
                replace = T,prob = c(.1,.2,.4,.2,.1))
```

ⓑ Create a factor named `fact1` using the vector `smpl1` as input.

```
fact1 <- factor(smpl1)
str(fact1)
```

```
##  Factor w/ 5 levels "bad","exc","good",..: 5 5 1 4 1 4 3 4 4
```

- Create an ordered factor named `fact2` using the vector `smpl1` as input. The levels should be in increasing order.

```
fact2 <- ordered(smpl1, levels = c('bad','reg','norm',
                                    'good','exc'))
str(fact2)
```

```
##  Ord.factor w/ 5 levels "bad"<"reg"<"norm"<..: 2 2 1 3 1 3 4
```

d Now, you want to reduce the number of categories to three: ‘bad’ and ‘reg’ will now be ‘poor’, ‘norm’ will be ‘normal’ and ‘good’ and ‘exc’ will now be ‘great’. One easy way to do this is to use the `labels` argument in the function `factor` to rename the levels. Look up the help page for `factor`; there is an example that will show you how to do this. Name the resulting ordered factor `fact3`.

```
fact3 <- ordered(smpl1,
                 levels = c('bad','reg','norm',
                            'good','exc'),
                 labels = c('poor','poor','normal',
                            'great','great'))
str(fact3)
```

```
##  Ord.factor w/ 3 levels "poor"<"normal"<..: 1 1 1 2 1 2 3 2 2
```

Exercise 4

```
table(fact1)
```

```
## fact1
##  bad  exc good norm  reg
##   10   15   20   33   22
```

```
table(fact2)
```

```
## fact2
##  bad  reg norm good  exc
##   10   22   33   20   15
```

Exercise 4

```r
table(fact3)
```

```
## fact3
##   poor normal  great
##     32     33     35
```