Problem 1

Within *mixtureMean.RData*, there are two test cases (A with large number of components K; and B with small K) consisting of a vector of μ values, a list of weights and a list of IDs that map the weights to the corresponding components in the mean vector.

(a)

One line code using *sapply()* that will calculate the weighted mean $\sum_{k=1}^{m_i} w_{i,k}\mu_{ID_{i,k}}$.

```
#### Problem Set 3 1. mixtureMean
library(rbenchmark)
rm(list = ls(all = TRUE))  # remove all objects
load("mixtureMean.RData")  # import data
```

```
# (a) original data storage
mixmeanA <- sapply(1:length(IDsA), function(i) {
    return(sum(muA[IDsA[[i]]] * wgtsA[[i]]))
})  # sapply test case A
mixmeanB <- sapply(1:length(IDsB), function(i) {
    return(sum(muB[IDsB[[i]]] * wgtsB[[i]]))
})  # sapply test case B
```

(b)

I set up the objects under case A as two matrices, one storing the μ values used for calcuation and one storing the corresponding weights for those mean values. The size of the these two matrices are the same, which is $n \times max(m_i)$, i.e. the number of observations times the maximum number of components per observation.

```
# (b) data setup for A: K=1000 table out as matrix storing the
# mu[ids]/wgts in cols for each observation row
idnum <- length(IDsA)
idlen <- sapply(1:idnum, function(i) {
    return(length(IDsA[[i]]))
})
maxmi <- max(idlen)
muidA <- matrix(as.numeric(NA), nr = idnum, nc = maxmi)
for (i in 1:idnum) {
    muidA[i, 1:idlen[i]] <- muA[IDsA[[i]]]
}
wtidA <- matrix(as.numeric(NA), nr = idnum, nc = maxmi)
for (i in 1:idnum) {
    wtidA[i, 1:idlen[i]] <- wgtsA[[i]]
}
mixmeanA2 <- rowSums(muidA * wtidA, na.rm = TRUE)
```

(c)

The set up of data objects under case B is even simpler. As $K = 10$ which is small, I just use a $n \times K$ matrix to store all the weights for the observations of the mean vector, leaving the untouched components with weight 0.

```
# (c) data setup for B: K=10 small K can allow us to store all the IDs as
# truth table for each u
idnum <- length(IDsB)
munum <- length(muB)  # K is small
wtidB <- matrix(0, nr = munum, nc = idnum)
for (i in 1:idnum) {
    tmpwt <- rep(0, munum)
    tmpwt[IDsB[[i]]] <- wgtsB[[i]]
    wtidB[, i] <- tmpwt
}
mixmeanB2 <- colSums(muB * wtidB)
```

(d)

The comparison for the two test cases are shown below using benchmarking functions.

```
# (d) efficiency comparison
# (d) efficiency comparison
benchmark(A1 = {mixmeanA <- sapply(1:length(IDsA),
                           function(i){ return( sum(muA[IDsA[[i]]]*wgtsA[[i]]) ) })},
          A2 = {mixmeanA2 <- rowSums(muidA*wtidA, na.rm = TRUE)},
          replications = 5)

##   test replications elapsed relative user.self sys.self user.child
## 1   A1            5    5.22    65.25      5.18        0         NA
## 2   A2            5    0.08     1.00      0.07        0         NA
##   sys.child
## 1        NA
## 2        NA

all.equal(mixmeanA, mixmeanA2)

## [1] TRUE

benchmark(B1 = {mixmeanB <- sapply(1:length(IDsB),
                           function(i){ return( sum(muB[IDsB[[i]]]*wgtsB[[i]]) ) })},
          B2 = {mixmeanB2 <- colSums(muB*wtidB)},
          replications = 5)

##   test replications elapsed relative user.self sys.self user.child
## 1   B1            5    5.06    126.5      4.99        0         NA
## 2   B2            5    0.04      1.0      0.05        0         NA
##   sys.child
## 1        NA
## 2        NA

all.equal(mixmeanB, mixmeanB2)

## [1] TRUE
```

We can see that there is about 50X to 100X speed-up compared to the original case, which shows the advantages of re-arranging the data objects.

Problem 2

The CSC (compressed sparse column) format for storing sparse matrices has three components:

values: a vector of non-zero entries, stored with column major ordering

rowIndices: a vector of row indices, one for each non-zero entry

colPointers: a vector of entry index at which each column starts (length of $ncol + 1$)

```
#### Problem Set 3 2. CSC matrix
library(Matrix)  # sparse matrix package
library(compiler)
library(rbenchmark)
rm(list = ls(all = TRUE))  # remove all objects
source("cscFromC.R")
```

(a)

I re-write the C-style *makeCSC()* function as below.

```
### (a) R version of makeCSC
makeCSCr <- function(matT) {
    matCSC = list()
    dimInfo <- which(matT != 0, arr.ind = TRUE)  # array indices info (row, col)
    matCSC$values <- matT[dimInfo]  #non-zero matrix entries
    matCSC$rowIndices <- dimInfo[, 1]  # row indices
    matCSC$colPointers <- c(1, cumsum(tabulate(dimInfo[, 2], nbins = ncol(matT))) +
        1)  # cumsum the num of entries in each col
    return(matCSC)
}
m <- matrix(c(1, 0, 0, 7, 0, 2, 0, 0, 0, 0, 0, 0, 0, 0, 0, 4), nr = 4)
makeCSCr(m)

## $values
## [1] 1 7 2 4
##
## $rowIndices
## [1] 1 4 2 4
##
## $colPointers
## [1] 1 3 4 4 5
```

(b)

I used the profiling tool to assess the computation bottleneck of my function, which is shown below. I generated the test matrices using the *makeTestMatrix()* function provided in *cscFromC.R*.

```
m <- makeTestMatrix(2500)
Rprof("makeCSCr.prof", interval = 0.01)
system.time(mr <- makeCSCr(m))

##    user  system elapsed
##    0.69    0.01    0.75
```

```
Rprof(NULL)
summaryRprof("makeCSCr.prof")$by.self

##          self.time self.pct total.time total.pct
## !=            0.52    55.32       0.52     55.32
## which         0.21    22.34       0.73     77.66
## gc            0.20    21.28       0.20     21.28
## makeCSCr      0.01     1.06       0.74     78.72
```

So, an improved version is shown below with built-in packages.

```
### (b) # more efficient after Rprof() and improvement in coding
makeCSCr2 <- function(matT) {
    matCSC = list()
    M <- as(matT, "dgCMatrix")
    matCSC$values <- M@x  #non-zero matrix entries
    matCSC$rowIndices <- M@i + 1  # row indices
    matCSC$colPointers <- M@p + 1  # cumsum the num of entries in each col
    return(matCSC)
}
m <- matrix(c(1, 0, 0, 7, 0, 2, 0, 0, 0, 0, 0, 0, 0, 0, 0, 4), nr = 4)
makeCSCr2(m)

## $values
## [1] 1 7 2 4
##
## $rowIndices
## [1] 1 4 2 4
##
## $colPointers
## [1] 1 3 4 4 5
```

(c)

The speed test for C-style, original R-style, package-based R-style and byte-compiled R is shown below. I have also made a comparison function for checking the accuracy of the functions written in different styles.

```
compMat <- function(m1, m2) {
    # compare the CSC representation of matrix
    flag = all.equal(m1$values, m2$values)
    flag = flag && all.equal(m1$rowIndices, m2$rowIndices)
    flag = flag && all.equal(m1$colPointers, m2$colPointers)
    return(flag)
}
makeCSCrCMP <- cmpfun(makeCSCr)  # byte compiled
############## Test ##################
m <- makeTestMatrix(6400)

system.time(mc <- makeCSC(m))

##    user  system elapsed
##    3.76    0.03    3.79
```

```
system.time(mr <- makeCSCr(m))

##    user  system elapsed
##    1.55    0.17    1.71

compMat(mc, mr)

## [1] TRUE


######## benchmark ##########
benchmark(makeCSC(m), makeCSCr(m), makeCSCr2(m), makeCSCrCMP(m), replications = 2,
    columns = c("test", "replications", "elapsed", "relative", "user.self",
        "sys.self"))

##             test replications elapsed relative user.self sys.self
## 1     makeCSC(m)            2    7.55    3.020      7.53     0.02
## 2    makeCSCr(m)            2    3.08    1.232      2.70     0.35
## 3   makeCSCr2(m)            2    2.50    1.000      1.85     0.63
## 4 makeCSCrCMP(m)            2    2.53    1.012      2.03     0.50
```

We can see that the pakage-based R-style code runs the fastest, that byte-compiled version runs slightly faster than the original one and that C-style code runs the slowest.

(d)

In my R-style *makeCSCr()* function, there is no additional copy of the full matrix or other matrices of that same size. The result is shown below with the *gc()* function.

```
### (d) memory usage
gc()  #initial

##            used  (Mb) gc trigger  (Mb)  max used  (Mb)
## Ncells  1390450  74.3    2251281 120.3   2251281 120.3
## Vcells 26919727 205.4   77314428 589.9 130664802 996.9

m <- makeTestMatrix(2500)
gc()  # with m

##            used  (Mb) gc trigger  (Mb)  max used  (Mb)
## Ncells  1390470  74.3    2251281 120.3   2251281 120.3
## Vcells 30044761 229.3   77314428 589.9 130664802 996.9

system.time(mr <- makeCSCr(m))

##    user  system elapsed
##    0.14    0.00    0.14

gc()  # after mr

##            used  (Mb) gc trigger  (Mb)  max used  (Mb)
## Ncells  1390492  74.3    2251281 120.3   2251281 120.3
## Vcells 30059608 229.4   77314428 589.9 130664802 996.9
```

Problem 3

We would like to set the lower triangle of a matrix to equal the transpose of the upper triangle.

```
#### Problem Set 3 3. lower.triangle(mat) = transpose(upper.triangle(mat))
rm(list = ls(all = TRUE))  # remove all objects
m <- matrix(1:25, nr = 5)  # simple test matrix
```

(a)

The code shown below does not work. As R uses column-major ordering to store matrix entries, when extracted using *upper.tri()*, the ordering is messed up as shown below inline with the diagnosis code.

```
m <- matrix(1:25, nr = 5)  # simple test matrix
m  #showcase

##      [,1] [,2] [,3] [,4] [,5]
## [1,]    1    6   11   16   21
## [2,]    2    7   12   17   22
## [3,]    3    8   13   18   23
## [4,]    4    9   14   19   24
## [5,]    5   10   15   20   25

### (a)
m[lower.tri(m)] = t(m[upper.tri(m)])  # this code does not work
m  #showcase

##      [,1] [,2] [,3] [,4] [,5]
## [1,]    1    6   11   16   21
## [2,]    6    7   12   17   22
## [3,]   11   17   13   18   23
## [4,]   12   18   22   19   24
## [5,]   16   21   23   24   25

# should be row-major conversion here for substitution
m[upper.tri(m)]  # convert to col-major vector

##  [1]  6 11 12 16 17 18 21 22 23 24
```

(b)

The working code is shown below with inline explaination of how it works. The main idea is to extract the extries from the transposed matrix to perserve the row-major ordering of the original matrix.

```
### (b)
m <- matrix(1:25, nr = 5)  # simple test matrix
m  #debug

##      [,1] [,2] [,3] [,4] [,5]
## [1,]    1    6   11   16   21
## [2,]    2    7   12   17   22
## [3,]    3    8   13   18   23
## [4,]    4    9   14   19   24
## [5,]    5   10   15   20   25
```

```
m[upper.tri(m)]  #debug, col-major

##  [1]  6 11 12 16 17 18 21 22 23 24

### after examination, we have to get row-major upper tirangle
t(m)  # transpose of orignal m

##      [,1] [,2] [,3] [,4] [,5]
## [1,]    1    2    3    4    5
## [2,]    6    7    8    9   10
## [3,]   11   12   13   14   15
## [4,]   16   17   18   19   20
## [5,]   21   22   23   24   25

t(m)[lower.tri(m)]  # col-major of t(m) = row-major of m

##  [1]  6 11 16 21 12 17 22 18 23 24

### the working one-line code to do the job
m  #showcase

##      [,1] [,2] [,3] [,4] [,5]
## [1,]    1    6   11   16   21
## [2,]    2    7   12   17   22
## [3,]    3    8   13   18   23
## [4,]    4    9   14   19   24
## [5,]    5   10   15   20   25

m[lower.tri(m)] <- t(m)[lower.tri(m)]  # this code works
m  #showcase

##      [,1] [,2] [,3] [,4] [,5]
## [1,]    1    6   11   16   21
## [2,]    6    7   12   17   22
## [3,]   11   12   13   18   23
## [4,]   16   17   18   19   24
## [5,]   21   22   23   24   25
```

(c)

The *lower.tri<-* replacement function is shown below. The optional *byrow* argument is similar to that in *matrix()*, which let the function to fill the martix with row-major ordering when TRUE.

```
### (c)
`lower.tri<-` <- function(x, byrow = FALSE, value){
    #replacement function for lower.tri

    x <- as.matrix(x)

    if ( length(x[lower.tri(x)]) != length(value) )
      stop("Vector length mismatch!")

    if (!byrow) x[lower.tri(x)] <- value # col-major
    else { # row-major
```

```
        tmpx <- t(x) # no replacement function for t()
        tmpx[upper.tri(tmpx)] <- value
        x <- t(tmpx)
    }
    return(x)
}

########test#########
m <- matrix(1:25, nr = 5) # simple test matrix
m

##      [,1] [,2] [,3] [,4] [,5]
## [1,]    1    6   11   16   21
## [2,]    2    7   12   17   22
## [3,]    3    8   13   18   23
## [4,]    4    9   14   19   24
## [5,]    5   10   15   20   25

lower.tri(m) <- 1:10 ;  m

##      [,1] [,2] [,3] [,4] [,5]
## [1,]    1    6   11   16   21
## [2,]    1    7   12   17   22
## [3,]    2    5   13   18   23
## [4,]    3    6    8   19   24
## [5,]    4    7    9   10   25

lower.tri(m, byrow = TRUE) <- 1:10 ; m

##      [,1] [,2] [,3] [,4] [,5]
## [1,]    1    6   11   16   21
## [2,]    1    7   12   17   22
## [3,]    2    3   13   18   23
## [4,]    4    5    6   19   24
## [5,]    7    8    9   10   25

lower.tri(m) <- c(1,2,3)

## Error:  Vector length mismatch!
```

Problem 4

For the object-oriented coding to generate a Markov chain with a fixed number of states, I choose to use the widely accepted and fast speed S3 class. A simple test case is generated with matrix attributes required for a Markov chain object.

```
#### Problem Set 3 4. Markov Chain (OOP)
library(methods)
rm(list = ls(all = TRUE))  # remove all objects
rnd <- rnorm(50000)  # rnd sample
rnd <- rnd[rnd > 0]  # prob > 0

p = 5  # num of states
x0 <- rep(0, p)  # get length p vector
x0[1] <- 1  # initialize with starting from state 1
xm <- matrix(sample(rnd, p * p), nr = p)  # get p-by-p matrix of rnds
xm <- xm/rowSums(xm)  # normalize rows

drawState <- function(p, pr) {
    if (length(pr) != p)
        stop("Vector length mismatch!")
    states <- diag(nrow = p, ncol = p)  # state vectors
    id <- sample(1:5, 1, prob = pr)
    return(as.vector(states[id, ]))  # get random states according to prob=pr
}
```

For an object in Markov chain class, it has the initial state vector *init* and two matrices i.e. transition matrix P and chain matrix *chain* which stores the process. In details, the transition matrix P of dimension $p \times p$ gives the transition probability $P[i, j] = Pr(X_n = j | X_{n-1} = i)$.

(a)

The constructor that creates an *Markov* class object utlizes a method *runSteps()* to run the chain for a given number of steps requested by user. The validity is checked inline with the code shown below.

```
### (a) create the Markov S3 class:
### initial state, transition matrix, num of steps, current state and [if needed, chain];

### creating a "runSteps" method for the MarkovS3 class construction
runSteps <- function(object, ...) UseMethod("runSteps") # generic method

runSteps.MarkovS3 <- function(obj) {
    if (obj$n == 0) {
        obj$chain <- matrix(obj$init, nr = 1)
        return (obj)  # no more steps, simple return the initial state
    } else {
        p = length(obj$init)
        obj$chain <- matrix(as.numeric(NA), nr = obj$n + 1, nc = p)
        obj$chain[1, ] <- obj$init
        for (i in 1:obj$n)
            obj$chain[i + 1, ] <- drawState(p, as.vector(obj$chain[i, ] %*% obj$P))
    }
    return(obj)
}
```

```
### constructor for 'MarkovS3' class
MarkovS3 <- function(init = NA, P = NA, n = 0, chain = NA){
    # check validity
    p = length(init) # num of states
    nr = dim(P)[1]; nc = dim(P)[2]; # dim of transition matrix

    if (n < 0)
        stop("Need positive integer steps!")
    if (p < 2)
        stop("Need at least two states for the Markov chain!")
    if (nr != nc)
        stop("The transistion matrix has to be square!")
    if (p != nr)
        stop("Mismatch between state vector length and transition matrix dimension!")
    if (length(init[init < 0]) > 0)
        stop("All state probabilities should be positive!")
    if (length(P[P < 0]) > 0)
        stop("All transition probabilities should be positive!")
    if (sum(init) != 1)
        stop("State probabilities should sum to one!")
    if (!all.equal(rowSums(P), rep(1, p)))
        stop("Transition probabilities for each state should sum to one!")

    # initialize the Markov chain
    obj <- list(init = init, P = P, n = n, chain = NA)
    class(obj) <- 'MarkovS3'

    # construct the Markov chain
    obj <- runSteps(obj)
    return(obj)
}

 ### simple tests
mkc0 <- MarkovS3(x0, xm)
mkc1 <- MarkovS3(x0, xm, n = 1)
mkc1

## Initial state
## [1] 1 0 0 0 0
## Transition matrix
##         [,1]    [,2]   [,3]    [,4]   [,5]
## [1,] 0.07202 0.06330 0.2873 0.10163 0.4757
## [2,] 0.13766 0.40616 0.2335 0.04011 0.1826
## [3,] 0.25238 0.19789 0.1208 0.22651 0.2024
## [4,] 0.05318 0.02573 0.3286 0.43496 0.1575
## [5,] 0.12563 0.08561 0.2353 0.35161 0.2018
## Current step
## [1] 1
## Current state
## [1] 0 1 0 0 0

mkc2 <- MarkovS3(x0, xm, n = 1000)
```

(b)

Three operators ('+', '-', '[') for the Markov chain class is developed as shown below.

```
#### (b) Markov chain S3 class-specific operators
`+.MarkovS3` <- function(obj, incr) {
    # check validity
    if (incr < 0) stop("Need positive integer steps!")
    if (incr == 0) return (obj)

    p = length(obj$init)
    addchain <- matrix(as.numeric(NA), nr = incr + 1, nc = p)
    addchain[1, ] <- obj$chain[obj$n + 1, ] # current state
    for (i in 1:incr)
        addchain[i + 1, ] <- drawState(p, as.vector(addchain[i, ] %*% obj$P))

    obj$n <- obj$n + as.integer(incr) # add steps
    obj$chain <- rbind(obj$chain, addchain[2:(incr+1), ])
    return(obj)
}

`-.MarkovS3` <- function(obj, decr) {
    # check validity
    if (decr < 0) stop("Need positive integer steps!")
    if (obj$n < decr) stop("Not enough steps of states to remove!")
    if (decr == 0) return (obj)

    obj$n <- obj$n - as.integer(decr) # remove steps
    obj$chain <- obj$chain[1:(obj$n + 1), ]
    if (obj$n == 0)
        obj$chain <- matrix(obj$chain, nr = 1)
    return(obj)
}

`[.MarkovS3` <- function(obj, idVec) {
    # check validity
    if (length(idVec) == 0) stop("No indices for extracting chain states!")
    if (length(idVec[idVec < 1 | idVec > obj$n+1])) stop("Indices out of bound!")

    return(obj$chain[idVec, ]) # init state is 1st
}

# show only the chain
mkc1$chain

##      [,1] [,2] [,3] [,4] [,5]
## [1,]    1    0    0    0    0
## [2,]    0    1    0    0    0

(mkc1 + 1)$chain

##      [,1] [,2] [,3] [,4] [,5]
## [1,]    1    0    0    0    0
## [2,]    0    1    0    0    0
## [3,]    0    0    1    0    0
```

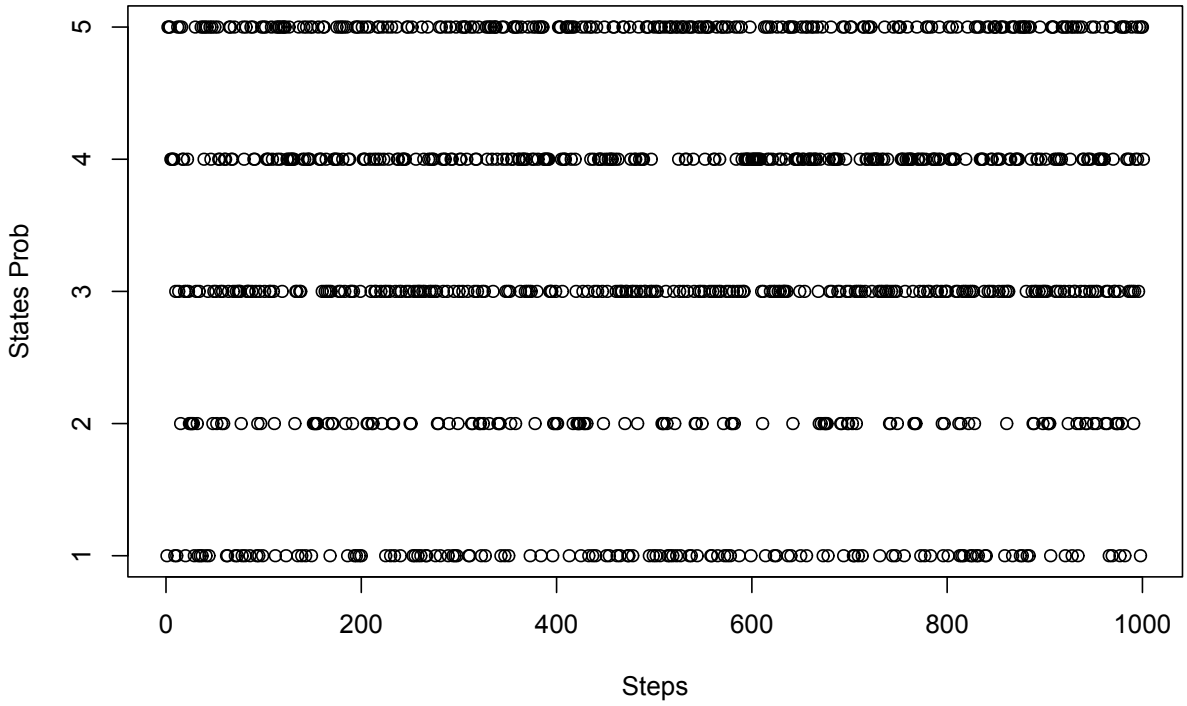
```
(mkc1 - 1)$chain

##      [,1] [,2] [,3] [,4] [,5]
## [1,]    1    0    0    0    0

mkc2[10:15]

##      [,1] [,2] [,3] [,4] [,5]
## [1,]    0    0    1    0    0
## [2,]    1    0    0    0    0
## [3,]    0    0    0    0    1
## [4,]    0    0    1    0    0
## [5,]    0    0    0    0    1
## [6,]    0    1    0    0    0
```

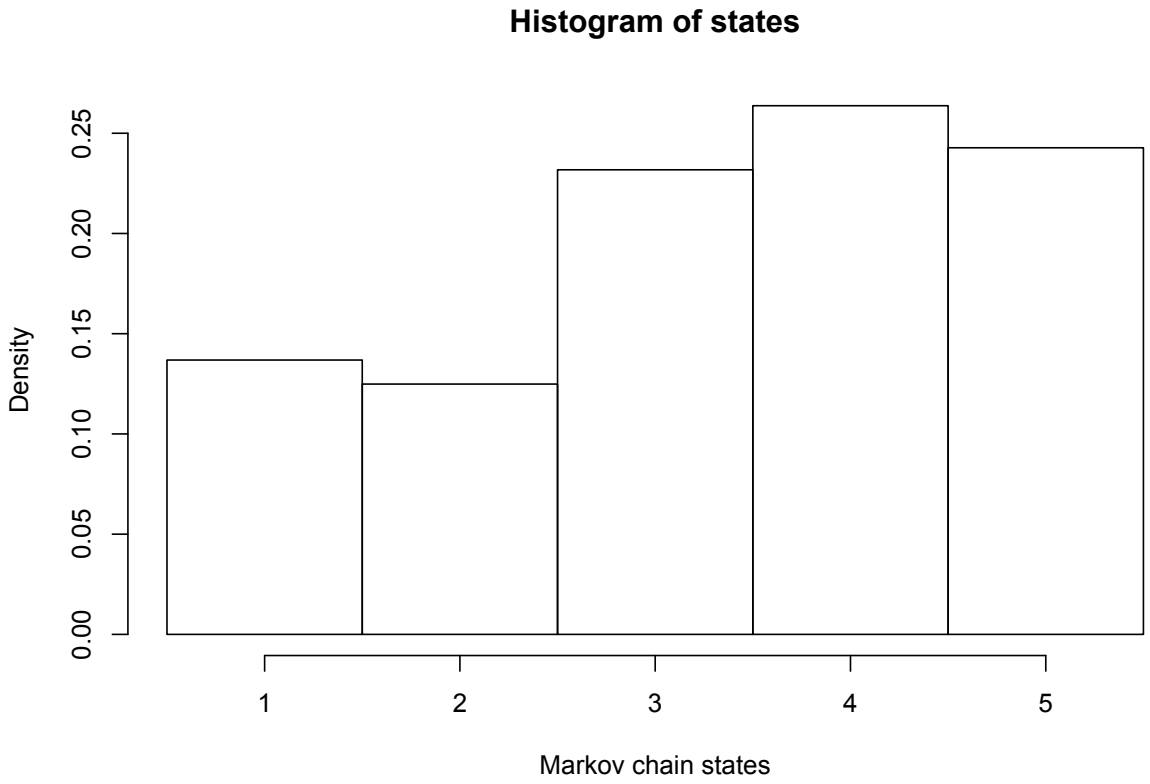
(c)

Three methods, i.e. *plot()*, *summary()*, *print()* or *show()* that produce nice, formatted output are developed for Markov chain class shown below.

```
 #### (c) Markov chain S3 class-specific functions
plot.MarkovS3 <- function(obj, ...) {
    if (obj$n == 0)
        cat("The current state is the same as the initial state\n", obj$chain[1, ], "\n")
    if (obj$n > 0){
        data <- which(obj$chain > 0, arr.ind = TRUE)
        plot(data[ ,1], data[ ,2], xlab="Steps", ylab="States Prob", ...)
        hist(data[ ,2], c(0:5+0.5), probability = TRUE,
             main = "Histogram of states", xlab = "Markov chain states")
    }
}

summary.MarkovS3 <- function(obj, ...) {
    p <- length(obj$init)
    allPre <- colSums(obj$chain[-(obj$n+1), ]) # previous states counts
    Tsub <- obj$chain[-1, ] - obj$chain[-(obj$n+1), ] # transition records (1:from,-1:to)
    Tadd <- obj$chain[-1, ] + obj$chain[-(obj$n+1), ]
    Tadd[which(Tadd == 1)] <- 0 # retaining the state
    Pemp <- diag(colSums(Tadd)/2/allPre, nrow = p, ncol = p) #retaining states counts, diag
    for (i in 1:p)
        for (j in 1:p)
            if (i != j)
                Pemp[i,j] <- length(which(Tsub[ ,i] == 1 & Tsub[ ,j] == -1))/allPre[i]

    cat("The initial state of the Markov chain is\n"); print(obj$init)
    cat("The transition matrix of this chain is\n"); print(obj$P)
    cat("The current step is", obj$n, ", and the current state is\n");
    print(obj$chain[obj$n+1, ])
    cat("The empirical transition probabilities are \n"); print(Pemp)
    cat("The empirical state probabilities are \n"); print(colMeans(obj$chain))
}
```

```
print.MarkovS3 <- function(obj, ...) {
    cat("Initial state\n"); print(obj$init)
    cat("Transition matrix\n"); print(obj$P)
    cat("Current step\n"); print(obj$n)
    cat("Current state\n"); print(obj$chain[obj$n+1, ])
}

summary(mkc2)

## The initial state of the Markov chain is
## [1] 1 0 0 0 0
## The transition matrix of this chain is
##         [,1]    [,2]   [,3]    [,4]   [,5]
## [1,] 0.07202 0.06330 0.2873 0.10163 0.4757
## [2,] 0.13766 0.40616 0.2335 0.04011 0.1826
## [3,] 0.25238 0.19789 0.1208 0.22651 0.2024
## [4,] 0.05318 0.02573 0.3286 0.43496 0.1575
## [5,] 0.12563 0.08561 0.2353 0.35161 0.2018
## The current step is 1000 , and the current state is
## [1] 0 0 0 1 0
## The empirical transition probabilities are
##         [,1]    [,2]   [,3]    [,4]   [,5]
## [1,] 0.05109 0.13869 0.4380 0.05839 0.3066
## [2,] 0.06400 0.33600 0.3920 0.04000 0.1680
## [3,] 0.17241 0.11638 0.1164 0.37931 0.2155
## [4,] 0.04563 0.02281 0.1939 0.44867 0.2928
## [5,] 0.28807 0.12757 0.1852 0.18107 0.2181
## The empirical state probabilities are
## [1] 0.1369 0.1249 0.2318 0.2637 0.2428

print(mkc1)

## Initial state
## [1] 1 0 0 0 0
## Transition matrix
##         [,1]    [,2]   [,3]    [,4]   [,5]
## [1,] 0.07202 0.06330 0.2873 0.10163 0.4757
## [2,] 0.13766 0.40616 0.2335 0.04011 0.1826
## [3,] 0.25238 0.19789 0.1208 0.22651 0.2024
## [4,] 0.05318 0.02573 0.3286 0.43496 0.1575
## [5,] 0.12563 0.08561 0.2353 0.35161 0.2018
## Current step
## [1] 1
## Current state
## [1] 0 1 0 0 0

plot(mkc0)

## The current state is the same as the initial state
##  1 0 0 0 0

plot(mkc2)
```


States Prob
1
2
3
4
5
0
200
400
600
800
1000
Steps


Histogram of states
Density
0.00
0.05
0.10
0.15
0.20
0.25
1
2
3
4
5
Markov chain states