

HW4

2024-09-29

Homework 4

Due 5 October

The total points on this homework is 175. Out of these, 5 points are reserved for clarity of presentation, punctuation and commenting with respect to the code.

The homework for this week are mainly exercises in using the apply() function, and its benefits in many cases where the problem can be “reduced” (which may mean, expanded for some cases) to operations at the margins of an array. Some of the problems are for a bigger dataset that is more cumbersome to notice operations on, so you are advised to try it out first on a small array (say of dimension 3x4x5) and then moving to the given problem once you are confident of your approach to solving the problem.

Q1

The file fbp-img.dat on Canvas is a 128×128 matrix containing the results of an image of fluorodeoxyglucose (FDG-18) radio-tracer intake reconstructed using Positron Emission Tomography (PET).

(a):

Read in the data as a matrix.

```
fbpImg <- scan("C:/Users/samue/OneDrive/Desktop/Iowa_State_PS/STAT 5790/PS/PS4/fbp-img.dat") |>
  matrix(nrow = 128, ncol = 128)
```

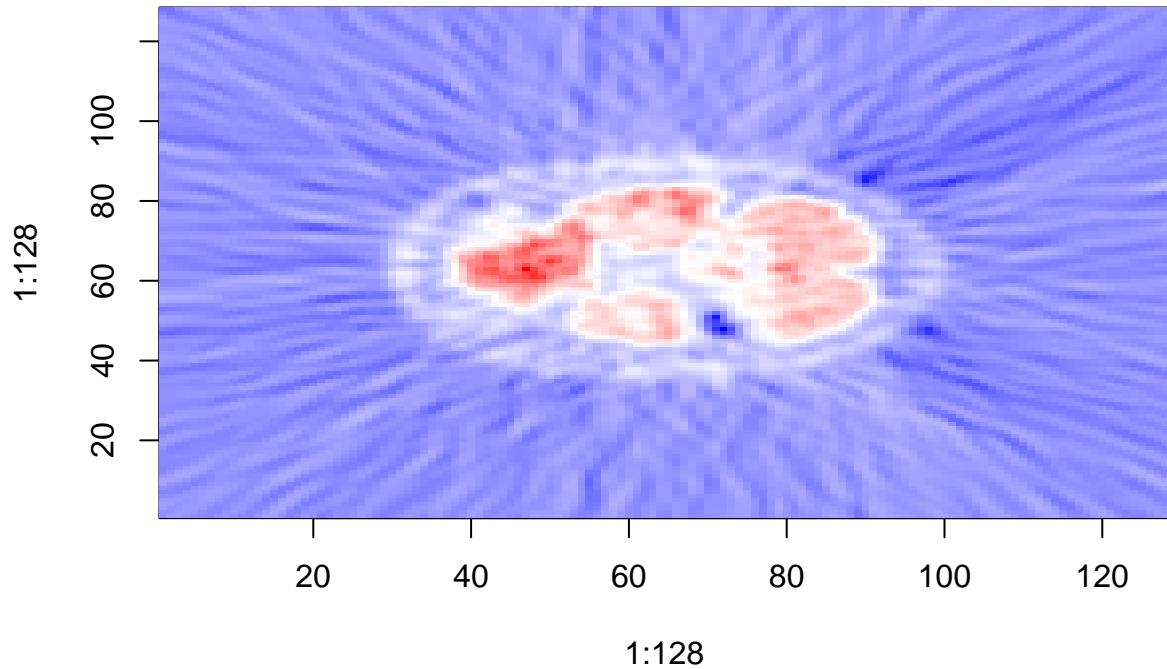
(b):

Use the image() function in R with your choice of color map to image the data.

```
require(wesanderson)

## Loading required package: wesanderson

image(1:128,
      1:128,
      fbpImg[, 128:1],
      # col=colors::diverge_hsv(16)
      col=colors::diverge_hsv(256)
      )
```



```
# wes_palette(n = 256, name = "AsteroidCity3", type = "continuous")
# colorspace::diverge_hsv(256)
```

(c):

We will now (albeit, somewhat crudely) compress the data in the following two ways:

- i. Obtain the range of values in the matrix, subdividing into 16 bins. Bin the values in the matrix and replace each value with the mid-point of each bin. Image these new binned matrix values.

```
rangefbpImg <- range(fbpImg)

# 16 bins implies 17 total lengths
breaks <- seq(to = min(rangefbpImg),
                 from = max(rangefbpImg),
                 length.out = 17)

# Calculate the midpoint of each bin
midpoints <- (breaks[-1] + breaks[-length(breaks)]) / 2

# Bin matrix using values derived
# replace each bin with its midpoint
# gives index values
binnedfbpImg <- cut(fbpImg,
```

```

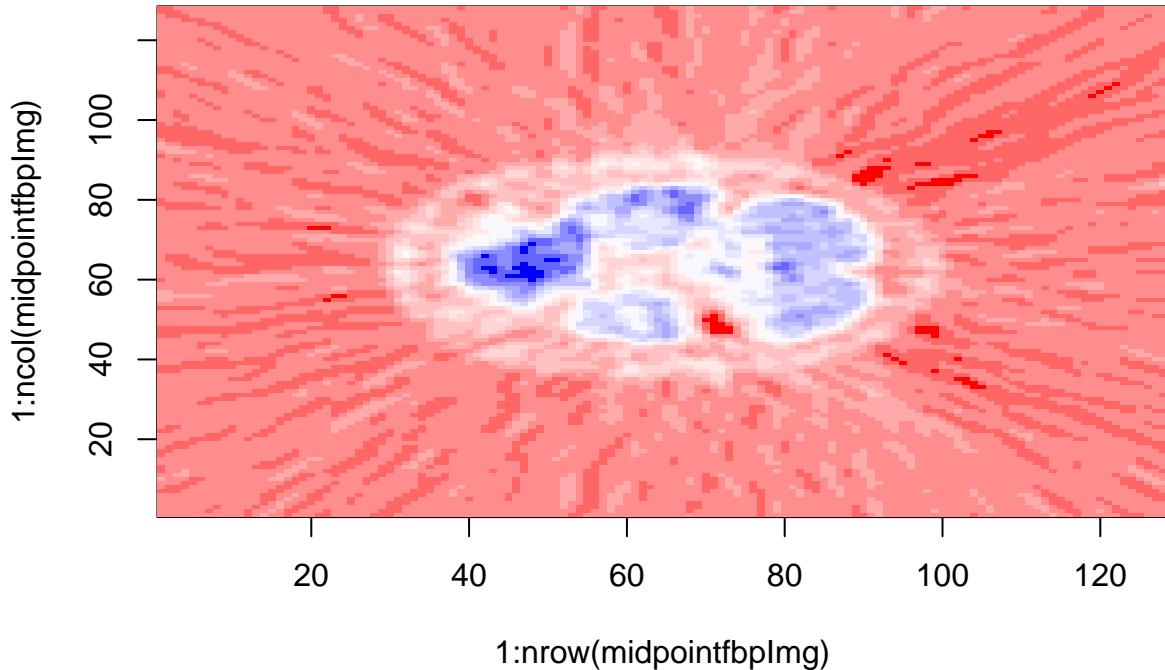
    breaks = breaks,
    labels = FALSE,
    include.lowest = TRUE)

# Replace binned values with corresponding midpoints
midpointMat <- midpoints[binnedfbpImg]

# Reshape the vector back into a matrix
midpointfbpImg <- matrix(midpointMat,
                           nrow = nrow(fbpImg),
                           ncol = ncol(fbpImg))

image(1:nrow(midpointfbpImg),
      1:ncol(midpointfbpImg),
      midpointfbpImg[, 128:1],
      # col=colorspace::diverge_hsv(16)
      # col=hcl.colors(32)
      col=colorspace::diverge_hsv(256)
      )

```



```

# wes_palette(n = 256, name = "AsteroidCity3", type = "continuous")
# colorspace::diverge_hsv(256)

```

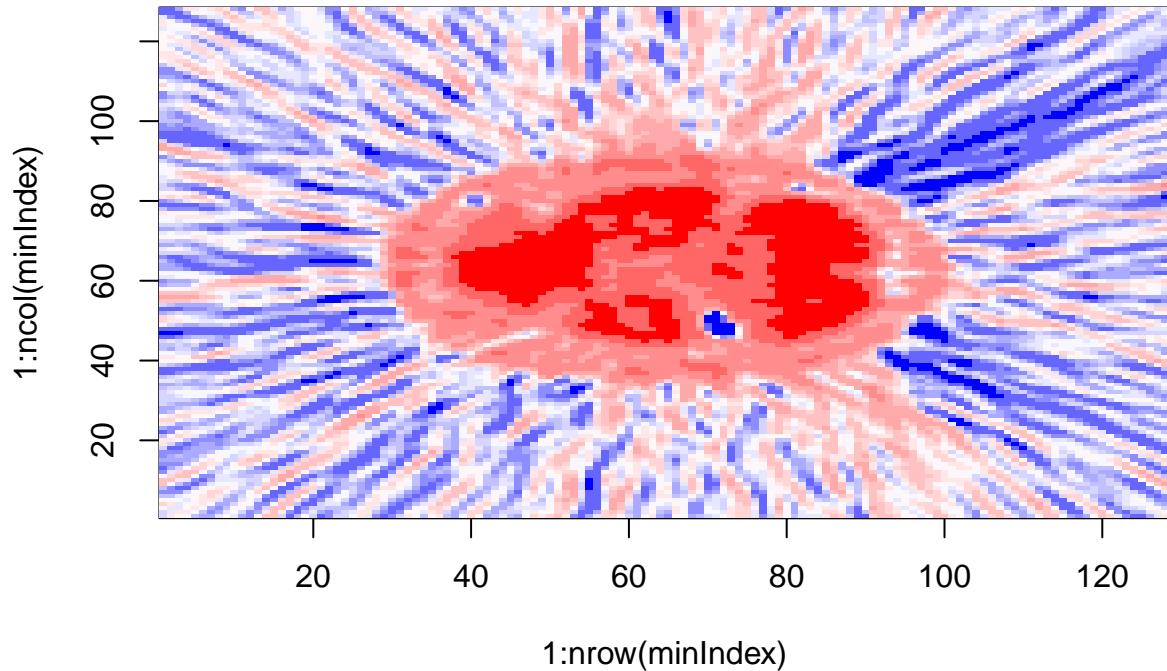
- ii. In this second case, we will group according to the 16 equally-spaced quantile bins, which can be obtained using the `quantile()` function in R with appropriate arguments. Use these obtained quantile bins to group

the data, replacing each entry in the matrix with its mid-point. Image these new binned matrix values.

```
# 16 bins implies 17 total lengths
qtFbpImg <- quantile(fbpImg,
                      probs = seq(0, 1, length = 17)
)
midQtFbpImg <- (qtFbpImg[-1] + qtFbpImg[-length(qtFbpImg)])/2

# stack the matrix 16 times on top of each other
arrFbpImg <- array(fbpImg,
                     dim = c(dim(fbpImg),
                             length(midQtFbpImg)
                     )
)
# dim(arrFbpImg)
midArr <- array(rep(midQtFbpImg,
                      each = prod(dim(fbpImg))
                     ),
                  dim = dim(arrFbpImg)
)
# dim(midArr)
sqDiffArr <- (arrFbpImg - midArr) ^ 2

minIndex <- apply(X = sqDiffArr,
                    MARGIN = c(1, 2),
                    FUN = which.min)
# dim(minIndex)
image(1:nrow(minIndex),
      1:ncol(minIndex),
      minIndex[, 128:1],
      # col=colors::diverge_hsv(16)
      # col=hcl.colors(32)
      col=colors::diverge_hsv(256)
)
```



```
# col=wes_palette(n = 256, name = "AsteroidCity3", type = "continuous")
# colorspace::diverge_hsv(256)
```

iii. Comment, and discuss similarities and differences on the differences in the two images thus obtained with the original image.

Interestingly, the colors denoting particular areas of the image are swapped between the two images, i.e. the outside region is predominantly red in the midpoint-based image and is blue with red streaks in the quantile-based image.

While the two images do appear similar, in that they convey the same information of the image being a brain, there are also some differences. One can visualize the outline of the brain and the distinct regions of the brain in greater detail in the quantile-based image (image two), particularly when identifying smaller regions within the brain that are distinct/different from the areas around it; see, for example, the difference in how recognizable the small patch to the top left of the brain is for the quantile-based image compared to the midpoint-based image.

Generally I'd reckon, the quantile-based image (second image) more clearly visualizes distinct areas of the brain compared to the midpoint-based image, particularly with regards to some of the smaller areas whose values differ considerably from the larger regions of the brain.

Q2

Microarray gene expression data. The file, accessible on Canvas at diurnaldatal.csv contains gene expression data on 22,810 genes from Arabidopsis plants exposed to equal periods of light and darkness in the diurnal cycle. Leaves were harvested at eleven time-points, at the start of the experiment (end of the light period) and subsequently after 1, 2, 4, 8 and 12 hours of darkness and light each. Note that there are 23 columns, with the first column representing the gene probeset. Columns 2–12 represent measurements on gene abundance taken at 1, 2, 4, 8 and 12 hours of darkness and light each, while columns 13–23 represent the same for a second replication.

(a):

Read in the dataset. Note that this is a big file, and can take a while, especially on a slow connection.

```
require(readr)

## Loading required package: readr

# I downloaded it locally!
geneDf <- read.csv("C:/Users/samue/OneDrive/Desktop/Iowa_State_PS/STAT 5790/PS/PS4/diurnaldatal.csv")
```

(b):

For each gene, calculate the mean abundance level at each time-point, and store the result in a matrix. One way to achieve this is to create a three-dimensional array or dimension $22,810 \times 11 \times 2$ and to use apply over it with the appropriate function and over the appropriate margins. Note that you are only asked present the commands that you use here. From now on, we will use this dataset averaged over the two replications for each gene.

```
# Exclude first column
geneMat <- as.matrix(geneDf[, -1])

geneArray <- array(geneMat, dim = c(22810, 11, 2))

meanGene <- apply(X = geneArray, MARGIN = c(1, 2), FUN = mean)
dim(meanGene)

## [1] 22810     11

# meanGene
```

(c):

Standardization. Gene data are compared to each other by way of correlations. (Correlation between any two sequences is related, by means of an affine (linear) transformation, to the Euclidean distance between the sequences, after standardization to have mean zero and standard deviation 1).

- i. In pursuance of the objective, use the apply function to calculate the mean of the mean abundance level over all time-points for each gene.

```
meanGeneAllTimes <- apply(X = meanGene, MARGIN = 1, FUN = mean)
length(meanGeneAllTimes)
```

```
## [1] 22810
```

```
# meanGeneAllTimes
```

ii. Set up a matrix of dimension $22,810 \times 11$ of the means of the mean abundance levels, where each row is a replicated version of the first one. Use this to eliminate the mean effect from the matrix stored in the previous part.

```
meanRepGene <- matrix(meanGeneAllTimes, nrow = 22810, ncol = 11, byrow = TRUE)

meanGeneNoMeanEffect <- meanGene - meanRepGene

dim(meanGeneNoMeanEffect)
```

```
## [1] 22810      11
```

iii. Use the apply function again to calculate the standard deviation of each row of the matrix in the part (b) above, and proceed to obtain the scaled measurements on the genes.

```
stdGene <- apply(X = meanGeneNoMeanEffect, MARGIN = 1, FUN = sd)

stdGeneMat <- matrix(rep(stdGene, each = 11),
                      nrow = 22810,
                      ncol = 11,
                      byrow = TRUE)

scaledGenes <- meanGene / stdGeneMat
dim(scaledGenes)
```

```
## [1] 22810      11
```

(d):

The file micromeans.dat contains a 20×11 matrix of measurements. Read in this dataset, and standardize as above.

```
geneMatMicro <- read.table("C:/Users/samue/OneDrive/Desktop/Iowa_State_PS/STAT 5790/PS/PS4/micromeans.dat",
                            as.matrix(nrow = 20,
                                      ncol = 11)

geneArrayMicro <- array(geneMatMicro, dim = c(20, 11, 2))

meanGeneMicro <- apply(X = geneArrayMicro, MARGIN = c(1, 2), FUN = mean)

meanGeneAllTimesMicro <- apply(X = meanGeneMicro, MARGIN = 1, FUN = mean)
```

```

meanRepGeneMicro <- matrix(meanGeneAllTimesMicro, nrow = 20, ncol = 11, byrow = TRUE)

meanGeneNoMeanEffectMicro <- meanGeneMicro - meanRepGeneMicro

stdGeneMicro <- apply(X = meanGeneNoMeanEffectMicro, MARGIN = 1, FUN = sd)

stdGeneMatMicro <- matrix(rep(stdGeneMicro, each = 11),
                           nrow = 20,
                           ncol = 11,
                           byrow = TRUE)

scaledMicro <- meanGeneMicro / stdGeneMatMicro

dim(scaledMicro)

```

[1] 20 11

(e):

We will now identify which of these means is closest to each gene. To do so, set up an three-dimensional array (of dimension $22,810 \times 11 \times 20$) with 20 replicated datasets (of the 22,810 diurnal mean abundance levels over 11 time-points). Next, set up replications of the 20 mean vectors into a three-dimensional array (of dimension $22,810 \times 11 \times 20$), with the replications occurring along the first dimension. Using apply and the above, obtain a $22,810 \times 20$ matrix of Euclidean distances for each gene and mean. Finally, obtain the means to which each gene is closest to. Report the frequency distribution, and tabulate the frequencies, and display using a piechart.

```

distancesArr <- sapply(1:22810, function(x) {
  apply(scaledMicro, 1, function(y) {
    sqrt(sum((scaledGenes[x, ] - y)^2))
  })
})

# for (x in 1:22810) {
#   for (y in 1:20) {
#     distancesArr[x, y] <-
#       sqrt(sum(
#         (meanGene[x, ] - geneDfMicro[y, ])^2)
#       )
#   }
# }

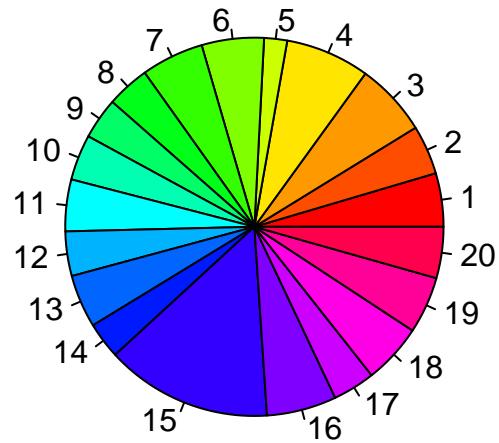
closestMeansIndex <- apply(X = t(distancesArr), MARGIN = 1, FUN = which.min)

frequencyTable <- table(closestMeansIndex)

pie(frequencyTable,
  main = "Frequency Distribution of Genes by Their Closest Means",
  col = rainbow(length(frequencyTable)))

```

Frequency Distribution of Genes by Their Closest Means



Q3

The United States Postal Service has had a long-term project to automating the recognition of handwritten digits for zip codes. The file ziptrain.dat has data on different numbers of specimens for each digit. Each observation is in the form of a 256-dimensional vector of pixel intensities. These form a 16x16 image of pixel intensities for each digit. Each digit is determined to be a specimen of the actual digit given in the file zipdigit.dat. The objective is to clean up the dataset to further distinguish one digit from the other.

(a):

Read in the dataset as a vector. This dataset has 2000 rows and 256 columns. Convert and store the dataset as a three-dimensional array of dimensions $16 \times 16 \times 2000$.

```
ziptrain <- read.table("C:/Users/samue/OneDrive/Desktop/Iowa_State_PS/STAT 5790/PS/PS4/ziptrain.dat")
# zipdigit <- read.table("C:/Users/samue/OneDrive/Desktop/Iowa_State_PS/STAT 5790/PS/PS4/zipdigit.dat")
# dim(ziptrain)
# dim(t(ziptrain))

ziptrain3D <- array(t(ziptrain), dim = c(16, 16, 2000))

dim(ziptrain3D)

## [1] 16 16 2000
```

(b):

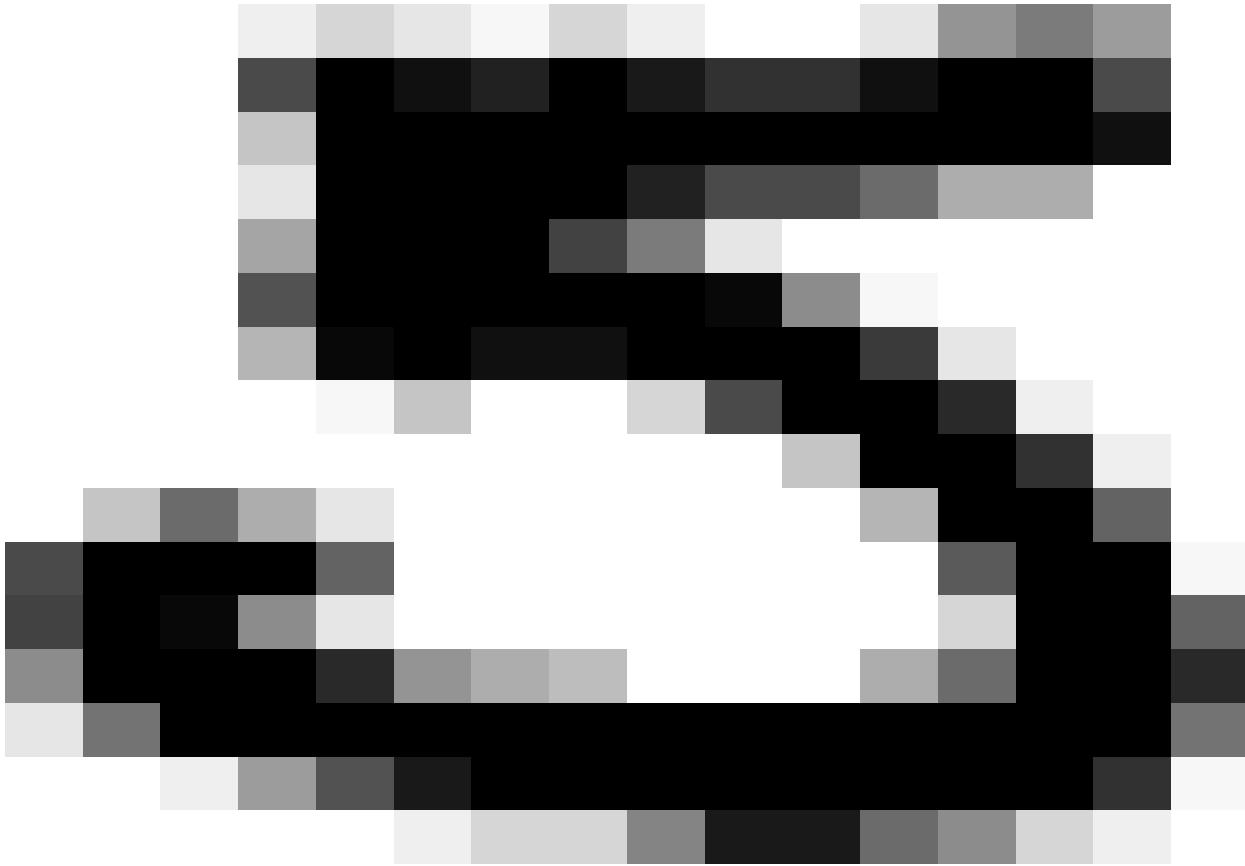
Our objective is to image all the 2000 pictures in a 40×50 display. We will hasten this display process by setting up the stage for using apply.

- i. If we image the second record, we note that without any changes, the image mirrors the digit “5”. Fix these by reversing the appropriate dimension of the array. Also note that the white space around the axes is wasted space and should be removed. To substantially reduce the white space, consider par(mar = rep(0.05, 4)). Further, the axes conveys no information so consider its elimination, using axes = F in the call to the image() function. Finally, we may as well use a gray scale here so consider using col = gray(31:0/31) in the image argument. Display the second record as an image, after addressing all these issues.

```
properArray <- ziptrain3D[, 16:1, ]

par(mar = rep(0.05, 4))

image(properArray[,, 2],
      axes = FALSE,
      col = gray(31:0/31)
)
```



ii. Using `par(mfrow = c(40,50))` and a total image size of $5.2'' \times 6.5''$, image all the 2000 digits using `apply` and the experience you gained in part i. above.

```
par(mfrow = c(40, 50),
  mar = rep(0.05, 4)
  # this pin size works, I guess
  # pin = c(5.2/50, 6.5/40)
  # pin = c(5.2/40, 6.5/50)
  # pin = c(5.2/100, 6.5/80)
  # pin = c(5.2/80, 6.5/100)
  )

apply(X = properArray, MARGIN = 3, FUN = function(x) {
  image(x[, 1:16],
    axes = FALSE,
    col = gray(31:0/31))
})
```

NULL

(c):

We now compute the mean and standard deviation images of the digits. To do so, we convert the array back to a matrix, and calculate the means for the ten digits, convert back to a three-dimensional array. Do exactly that. Also, compute and display the standard deviation images in exactly a similar way.

```
zipMat <- matrix(scan("ziptrain.dat"),
                  nrow = 2000,
                  ncol = 256,
                  byrow = TRUE)

zipdigit <- read.table("C:/Users/samue/OneDrive/Desktop/Iowa_State_PS/STAT 5790/PS/PS4/zipdigit.dat")

meanDigits <- sapply(X = 0:9,
                      function(x) {
                        digitDf <- zipMat[zipdigit == x, ]
                        # Select rows corresponding to the current digit
                        colMeans(digitDf)
                      }
                    )

# meanDigitsArr <- array(t(meanDigits), dim = c(16, 16, 10))
```

```

meanDigitsArr <- array(meanDigits, dim = c(16, 16, 10))
par(mfrow = c(1, 10), mar = rep(0.05, 4))

# for (x in 0:9) {
#   image(meanDigitsArr[,,x+1],
#         axes = FALSE,
#         col = gray(31:0/31))
# }

# mean images
apply(X = meanDigitsArr, MARGIN = 3, FUN = function(x) {
  image(x[, 16:1],
        axes = FALSE,
        col = gray(31:0/31))
})

```



```

## NULL

stdDigits <- sapply(X = 0:9,
                     FUN = function(x) {
  digitDf <- zipMat[zipdigit == x, ]
  apply(X = digitDf, MARGIN = 2, FUN = sd)
})

```

```

# stdDigitsArr <- array(t(stdDigits), dim = c(16, 16, 10))
stdDigitsArr <- array(stdDigits, dim = c(16, 16, 10))

par(mfrow = c(1, 10), mar = rep(0.05, 4))

# for (x in 0:9) {
#   image(stdDigitsArr[,,x+1],
#         axes = FALSE,
#         col = gray(31:0/31))
# }

# std dev images
apply(X = stdDigitsArr, MARGIN = 3, FUN = function(x) {
  image(x[, 16:1],
        axes = FALSE,
        col = gray(31:0/31))
})

```



```
## NULL
```

(d):

Our final act will involve cleaning up the dataset. To do so, remove the mean digit effect from each record. Then, perform a singular value decomposition on the 2000×256 matrix of deviations Y (that is the records

with the corresponding digit mean effect removed). (Recall that the SVD is $\mathbf{Y} = \mathbf{U} \mathbf{D} \mathbf{V}'$ and is obtained via the svd() function in R.) Replace D with a diagonal matrix \mathbf{D}_k of the first k eigenvalues of D and the remainder as zero. Then let $\mathbf{Y}_k = \mathbf{U} \mathbf{D}_k \mathbf{V}'$. Add back the mean and display the resulting images of all the digits for $k = 25, 50, 75$ in the same manner as (b)ii. Comment on the displays.

```

ziptrain <- matrix(scan("ziptrain.dat"), ncol = 256, byrow = TRUE)
zipdigit <- read.table("C:/Users/samue/OneDrive/Desktop/Iowa_State_PS/STAT 5790/PS/PS4/zipdigit.dat")
# 2000 x 1 vector
ziptrainMeans <- rowMeans(ziptrain)
# length(ziptrainMeans)
ziptrainMeansRep <- array(rep(stdDigits, 256), dim = c(2000, 256))
# dim(ziptrainMeansRep)

# 2000 x 256
# difference matrix
# original - mean effect
meanDiffMat <- ziptrain - ziptrainMeansRep
dim(meanDiffMat)

## [1] 2000 256

svdDiff <- svd(meanDiffMat)
svdDiffD <- svdDiff$d

kList <- list(
  diag(c(svdDiffD[1:25], rep(0, 231))),
  diag(c(svdDiffD[1:50], rep(0, 206))),
  diag(c(svdDiffD[1:75], rep(0, 181)))
)

results <- lapply(X = kList, FUN = function(k) {
  kMat <- svdDiff$u %*% k %*% t(svdDiff$v)
  kPlusMeanDiff <- kMat + ziptrainMeansRep
  kPlusMeanDiff
})
)

newK1 <- array(t(results[[1]]), dim = c(16, 16, 2000))
newK2 <- array(t(results[[2]]), dim = c(16, 16, 2000))
newK3 <- array(t(results[[3]]), dim = c(16, 16, 2000))

par(mfrow = c(40, 50),
  mar = rep(0.05, 4)
  # this pin size works, I guess
  # pin = c(5.2/80, 6.5/100)
  )

apply(X = newK1, MARGIN = 3, FUN = function(x) {
  image(x[, 16:1],
    axes = FALSE,
    col = gray(31:0/31))
})

```

583031017011179801497487379136793274542743274
6821960678004825081208335822081489836701470284
800505009032012970065950191171090807913843514954
6844861023984898560226841027102710927048082129113
7323871012285129790271026502760827110177211905689
181031611755365068800019957000160198601186800768
44940204312121213021302131158027900006829113999115145
85712014645320106060902605591177617202605361156121166
010460256090260521609466473618716090260160562207129109
60902601801606261761604731826090260160562207129109
0261821619136090261761604731826090260160562207129109
823418212128658417599217288750016090260160562207129109
0982652840068432811232812211622062074016090260160562207129109
60930530088300831300940308155027430809300302302478101
112137210380371308511285708911178000520011180037594012
97111823002214021090314601720017002138011770010184100
1362026010101838091097800720065090063000207009109700106100
7020660091865400606090072008000530066010097009109700106100
97020428567188600180200109828000013997110401850083853
45891073815276306010746913232045632666149700671457120
98009109148649550782352698290910720850083853
57194468546200178280368236818405027215632883
119977192141911971419724080660059101724079146654739
01900457006270801720017402964905069101724079146654739
19170191931409143611239110212682014311472411942910192501425
9113911420301239110212682014311472411942910192501425
3036630321263035603993031080303943037119966196031971
19549061780414713872177298670612724172503128612920
720303740588141331343707638138381173720296073708381
050198991994681972101988040319711197111772114047601
701317573170111752317301153721759211706617609360177
0379731691378401117056117311752317300170217268174071
733331701111705611773019733600119804111031971119850197
797001980398619891011961190620287111001973160010054010534
19897899914202171101197316001005401053450064010629
10564108270282203170062260603806729009706012506105061031
0608106087000990099639220097515363149011160608990198101
4200097201902092102971201864214330193361191011192551910
591290113140081413014101190310110319010102190115190

```
#> NULL

par(mfrow = c(40, 50),
  mar = rep(0.05, 4)
# this pin size works, I guess
# pin = c(5.2/80, 6.5/100)
)

apply(X = newK2, MARGIN = 3, FUN = function(x) {
  image(x[, 16:1],
    axes = FALSE,
    col = gray(31:0/31))
})
```

```

654730310170111774801497487374136741377454274137746
63218662087820910230912083308220814489846709708046
800308090380122906593091191176090807913044435169544
68448640239868935660226841027102271095704808721132
732227102285422797027102602760827111017764462911903
11810316117553650680001495700016019868000968009685
4494086431212121802130213233800279000068917799129453
857120146353201066460902605961727611206052161766
01026025609026052160546647361816090261701605220921609
6090261801605226176160473182609146912971028070297102
026118216191360902617619004626180146912971028070297102
8234182121228658912332821728278009388365830328601281
028265282298432832811123292212221116276224019345230
6093093008300830083024030815303743080930030230247811
11213121038037130851185708911980005200112003759402
9741821300221150210902146017200117020213861970018410
136402601018309104980720065090063200090700934406900
70806600918654006090025800230066009010043300912006
97075428567188600180201048280602139711040185023853
4589107338152763860107469132320485386661497006771405
98004907140849550482312698480810720850065370692526
591965468544680096820368114850271571185227111893190
1199771971419119714197260624760164631466547390
019002570062708012002940296405054101240794119384254
141701415314091443014810143114147241194230142501425
0141139147032123912022126821402134214094217011121228
3036630326503560303493030326303430327119661960319713
1951906178041471367217798670627941725037251222020
712032374053815138134370753813838117372029607370838
0501989919968197201988040319711197111197117404176011
70131737317011175231736017832175421706617604305177
637573169157340117056173117252317306170217268174071
7333317011170551722019726001198041970319711198501970
79700198039851989101961906702277110019726001019805
19897989914201802171101972600100540105345034010590
105641088705882037100622600038067900876609183506103
06087060870409004763492004751536314011106049049301
420097401972207971029721018642193041933518649181011
59199011870719101191321910119101192551910419129191
41190131400319130140119031101031901010719015901

```

NULL

```

par(mfrow = c(40, 50),
  mar = rep(0.05, 4)
  # this pin size works, I guess
  # pin = c(5.2/80, 6.5/100)
  )

apply(X = newK3, MARGIN = 3, FUN = function(x) {
  image(x[, 16:1],
    axes = FALSE,
    col = gray(31:0/31))
})

```

6	5	4	7	3	6	3	1	0	1	7	0	1	1	1	7	7	4	8	0	1	4	9	7	4	8	7	3	7	4	1	3	7	7	4	8	5	4	2	7	4	1	3	7	7	4	8		
6	3	2	0	9	6	6	8	0	8	7	8	2	0	9	1	0	2	2	0	9	1	2	2	9	0	6	5	9	2	0	9	1	9	1	7	6	1	9	7	0	8	0						
8	0	0	3	0	8	0	9	0	3	8	0	1	2	2	9	0	6	5	9	2	0	9	1	1	9	1	7	6	0	8	0	7	9	1	3	6	9	5	4	4								
6	8	4	4	8	6	4	0	2	3	9	8	6	8	9	3	5	6	8	0	2	2	6	8	4	1	0	2	7	1	0	2	2	7	1	0	9	2	7	0	4	8							
7	3	2	2	2	7	1	0	2	2	8	5	4	1	2	7	8	7	0	2	7	1	0	2	6	0	2	7	6	0	8	2	7	1	1	0	1	3	2	9	0	3							
1	1	8	1	0	3	1	6	1	1	7	5	5	3	6	5	0	6	8	8	0	0	1	4	9	5	7	0	0	0	1	6	0	1	9	8	6	0	0	9	6	8	5						
4	4	9	4	0	8	6	4	8	1	2	1	2	1	2	8	0	2	1	3	0	2	1	3	3	8	0	2	7	9	0	0	0	0	6	8	9	1	7	9	1	2	9	4	5	3			
8	5	7	1	8	0	1	4	6	5	3	2	0	1	0	6	6	2	6	0	9	0	2	4	0	5	5	9	6	1	7	7	1	6	1	7	6	6	6										
0	1	0	2	6	0	2	5	6	0	9	0	2	6	0	5	2	1	6	0	5	4	6	4	7	3	6	1	8	0	2	6	0	4	3	5	6	0	5	6	1	8	2	0					
6	0	9	0	2	6	1	8	0	1	6	0	5	2	2	6	1	7	6	0	4	7	3	6	1	8	2	6	0	2	6	0	4	3	5	6	0	5	2	2	6	0	9						
0	2	6	1	8	2	1	6	1	9	1	3	6	0	9	0	2	6	1	9	0	4	6	2	6	1	8	0	1	6	2	8	0	1	0	2	9	7	0	2	7	1	0	2					
8	2	3	4	2	8	2	1	2	2	8	6	5	8	2	1	3	9	2	8	2	1	7	3	8	2	7	8	0	0	7	9	3	2	8	6	0	1	2	8	1								
0	2	8	2	6	5	2	8	2	2	8	4	3	2	8	2	3	2	1	1	2	3	2	2	2	1	1	6	2	7	0	2	2	4	0	1	2	3	4	5	2	3	0						
6	0	9	3	0	0	3	0	0	8	3	0	0	8	3	0	2	4	0	3	0	8	1	5	3	0	2	7	4	3	0	8	0	2	3	0	2	4	7	8	1	1							
1	2	1	3	1	2	1	0	3	1	3	0	3	7	1	3	0	8	5	1	1	8	5	7	0	8	9	1	1	9	8	0	0	5	2	0	0	1	2	0	0	3	7	5	9	4	0	2	
9	7	4	1	8	2	1	3	0	0	2	2	1	5	0	2	1	0	9	0	2	1	4	6	0	1	7	2	0	0	1	9	7	0	0	1	8	4	1	0	0								
1	3	6	4	0	2	6	0	1	0	1	8	3	0	9	1	0	4	9	8	0	7	2	0	0	6	5	0	9	0	0	6	3	0	0	9	0	3	4	4	0	6	9	0					
7	0	8	0	6	6	0	0	9	1	8	6	5	4	0	0	7	5	8	0	0	2	3	0	0	6	6	0	0	9	0	1	0	0	9	3	0	0	6										
9	7	0	7	5	4	2	3	5	5	6	7	1	8	8	6	0	0	1	8	0	2	0	0	4	8	2	0	0	6	6	1	4	9	7	0	0	6	7	1	4	0	5						
4	5	8	9	1	0	7	3	8	1	5	2	4	6	3	0	1	0	7	4	6	9	1	2	3	2	0	4	8	2	0	8	5	0	0	6	5	3	1	0	6	9	2	5	2				
5	7	1	9	6	5	4	6	8	5	4	4	6	8	0	0	9	6	8	0	3	6	8	1	1	4	8	5	0	2	7	8	1	1	8	9	3	1	9	0									
1	1	9	9	7	7	1	9	7	1	9	1	1	9	7	1	9	7	1	9	7	2	6	0	0	6	2	2	4	7	6	3	1	4	6	5	4	7	3	9	0								
0	1	9	0	0	4	5	7	1	0	7	0	0	6	2	7	0	8	3	0	9	4	0	2	1	0	1	7	2	4	5	0	1	2	5	2	5												
1	4	1	7	0	1	4	1	7	0	1	4	0	9	1	4	0	9	1	4	0	9	1	4	0	9	1	4	2	5	0	1	4	2	5	1	4	2	5										
0	1	4	1	1	3	9	1	4	7	0	1	3	2	1	3	2	1	2	0	2	2	1	2	6	8	2	1	4	0	9	2	1	2	2	8	0	1	2	2	8								
3	0	3	6	6	3	0	3	2	6	3	0	3	5	6	0	3	4	9	3	0	3	4	9	3	0	3	4	9	3	0	3	2	7	1	9	6	0	3	1	9	7	1						
1	9	5	1	9	6	1	9	8	0	4	1	9	7	1	3	6	7	2	1	7	7	4	8	6	7	0	6	2	7	2	4	1	7	3	0	3	7	2	5	1	2	2	0					
7	1	2	0	3	2	3	1	7	4	0	5	3	8	1	3	4	3	7	0	7	1	5	3	8	1	3	8	3	8	1	1	7	3	2	0	2	9	6	0	7	3	0	8	3				
0	5	0	1	9	8	9	9	1	9	9	6	8	1	9	7	2	0	1	9	8	8	0	4	0	3	1	9	7	1	1	9	7	1	1	9	7	1	7	6	0	1							
7	0	1	3	1	7	3	7	3	1	7	0	1	1	1	7	5	7	3	1	7	3	4	0	1	7	8	3	7	1	7	5	4	2	1	7	0	6	6	1	7	6	0	1					
6	3	7	5	7	3	1	6	9	1	5	7	3	4	0	1	1	7	0	5	6	1	7	3	0	6	1	7	0	2	1	7	2	6	8	1	7	4	0	7	1								
7	3	3	1	7	0	1	1	7	0	5	5	1	9	7	2	0	1	9	7	2	6	0	0	1	1	9	8	0	4	1	9	7	1	1	9	8	5	0	1	9	7	0						
7	9	7	0	0	1	9	8	0	3	9	8	5	1	9	8	7	1	0	1	9	6	1	9	0	6	7	0	2	7	1	1	0	0	1	9	8	0	5										
1	9	8	9	7	9	8	9	9	1	9	9	6	8	1	9	7	2	0	1	8	0	2	1	7	1	0	1	9	7	1	1	0	5	4	0	1	0	5	9	0								
1	0	5	6	4	1	0	8	8	7	0	5	8	8	2	0	3	7	1	0	0	6	2	2	6	0	4	0	3	8	0	6	7	9	0	0	8	7	0	6	0	9	1						
0	6	8	9	7	0	6	0	8	7	0	4	0	9	0	6	0	4	7	6	3	4	9	2	0	0	4	7	5	1	5	3	6	3	1	4	0	1	0	6	0	4	9	0	4	9	3	0	1
4	2	0	0	9	7	2	0	1	9	7	2	0	7	9	1	7	0	2	9	7	2	1	0	1	8	6	4	2	1	9	3	0	1	0	4	9	1	8	0	1								
5	9	1	9	9	0	1	1	8	7	0	4	1	9	1	0	1	1	9	3	2	1	9	1	0	1	1	9	2	5	5	1	9	1	0	4	9	1	2	4	1	9	1						
4	1	1	9	0	1	3	1	4	0	0	3	1	9	1	3	0	1	9	0	3	1	1	0	1	0	3	1	9	0	1	0	7	1	9	0	1	5	9	0									

```
## NULL
```

```
# this code is here in memorium
# for the work that was done
# in vain
# ziptrain <- matrix(scan("ziptrain.dat"), ncol = 256, byrow = TRUE)
# zipdigit <- read.table("C:/Users/samue/OneDrive/Desktop/Iowa_State_PS/STAT 5790/PS/PS4/zipdigit.dat")
#
# meanImages <- array(0, dim = c(256, 10)) # 256 pixels, 10 digits
#
# # for (digit in 0:9) {
#   # meanImages[, digit + 1] <- colMeans(ziptrain_data[zipdigit == digit, ])
# }
#
# meanImages <- sapply(X = 0:9,
#   FUN = function(x) {
#     meanImages[, x + 1] <- colMeans(ziptrain[zipdigit == x, ])
#   }
# )
#
# deviationMat <- ziptrain
#
# # I swear I cannot make this work using sapply
# # this for loop is holding onto my sanity
# for (digit in 0:9) {
```

```

#   meanDigit <- matrix(meanImages[, digit + 1], nrow = 16, ncol = 16)
#   deviationMat[zipdigit == digit, ] <-
#     deviationMat[zipdigit == digit, ] - as.vector(t(meanDigit))
# }
#
# svdDf <- svd(deviationMat)
#
# reconstruct_images <- function(u, d, v, k) {
#   Dk <- diag(d[1:k])
#   Yk <- u[, 1:k] %*% Dk %*% t(v[, 1:k])
#   return(Yk)
# }
#
# k_values <- c(25, 50, 75)
# for (k in k_values) {
#   Yk <- reconstruct_images(svdDf$u, svdDf$d, svdDf$v, k)
#
#   for (digit in 0:9) {
#     meanDigit <- matrix(meanImages[, digit + 1], nrow = 16, ncol = 16)
#     reconstructedImg <- Yk[zipdigit == digit, ] + as.vector(t(meanDigit))
#     image(matrix(reconstructedImg[1, ], nrow = 16, ncol = 16)[, 16:1], axes = FALSE, col = gray(31:0))
#   }
# }

# par(mfrow = c(length(k_values), 10), mar = rep(0.05, 4))
#
# sapply(k_values[1:3],
#        function(k) {
#          Yk <- reconstruct_images(svdDf$u, svdDf$d, svdDf$v, k)
#          sapply(0:9, function(x) {
#            meanDigit <- matrix(meanImages[, x + 1], nrow = 16, ncol = 16)
#            reconstructedImg <- Yk[zipdigit == x, ] + as.vector(t(meanDigit))
#            image(matrix(reconstructedImg[1, ],
#                         nrow = 16,
#                         ncol = 16)[, 16:1],
#                  axes = FALSE,
#                  col = gray(31:0/31))
#          })
#        })
#      )
#    )
#  )

```

Comments

It appears the digits are more clearly delineated for larger values of k , though there is some “noise” surrounding the digits. Generally speaking, the digits displayed above are legible and would typically be understood in terms of which digit they denote, e.g. it is unlikely someone would confuse a “5” for a “6” in the above images, or vice versa.