HW10

2024-11-24

1.

Magnetic Resonance Imaging (MRI) is a noninvasive radiologic technique used to image tissue structure and physiology. The imaging modality works on the principle of characterizing tissues based on the basis of their physical and chemical properties which can be adequately summarized in terms of three physical quantities: the longitudinal or spin-lattice relaxation time (T_1), spin-spin relaxation time (T_2), and the proton density (ρ). These physical quantities are themselves not directly observable but together determine image intensity with respect to photon density. Determination of image contrast is achieved by changing user-defined pulse sequence parameters which modulate the influence of the T_1, T_2, and ρ values at a voxel, and hence the photon intensity.

In spin-echo imaging, there are two design parameters – these are the repetition time (TR) and the echo time (TE). The relationship between T_1, T_2, ρ, and the true MR signal, in the spin-echo sequence of MR imaging, denoted by ν, can be expressed in terms of a solution to the Bloch equation, an empirical expression describing nuclear magnetic resonance phenomena. Thus, the true MR signal ($\nu_{i,j,k}$) at the image pixel (or matrix element) (i, j) for the jth set of design parameters (TE_k, TR_k) is given by:

$$\nu_{i,j,k} = F(\rho_{ij}, T_{1ij}, T_{2ij}; \text{TE}_k, \text{TR}_k) = \rho_{ij} \exp\left(-\frac{\text{TE}_k}{T_{2ij}}\right)\left\{1 - \exp\left(-\frac{\text{TR}_k}{T_{1ij}}\right)\right\}.$$

Different pairs of (TE, TR) values can be used to highlight contrasts between different tissue types. In a clinical context, this means that tuning the design parameters at different settings can be used to highlight various tissue types, providing a method for tissue classification in individuals. Therefore, in principle at least, a radiologist can acquire images for a range of values and then select from these images to optimize the contrasts between different tissue types. However, the optimal control parameters are patient- and/or application-specific and generally not known in advance. A systematic exploration of plausible (TE, TR) pairs is impractical because of constraints in time and expense and also because of patient motion and image registration issues. So, therefore a few (typically no more than three) are acquired. Once obtained, the radiologist could estimate the (ρ, T_1, T_2) at the (i, j) voxel and use these estimates to synthetically predict images (Digital Object Identifier 10.1109/TMI.2009.2039487) at other (unobserved) (TE, TR) values.

The file 2dMRI-SE-phantom.dat contains intensity values of MRI images of a physical phantom (man-made object) acquired at 18 settings (in seconds) of (TE, TR) = $\{0.03, 0.06, 0.04, 0.08, 0.05, 0.10\} \times \{1, 2, 3\}$. For your convenience, these settings are also provided in the file te-tr.dat. The file 2dMRI-SE-phantom.dat should be read in as a vector and further the sequence of images is as follows: the first 256^2 values are for the MR image acquired at the first (TE, TR) setting (i.e., at (0.03, 1.0)), the next 256^2 values are for the MR image acquired at the second (TE, TR) setting (i.e., at (0.06, 1.0)), and so on.

(a)

Read in the dataset and store as a three-dimensional array. [5 points]

```r
nx <- 256
ny <- 256
nz <- 18

data <- scan("2dMRI-SE-phantom.dat")
phantomArr <- array(data, dim = c(nx, ny, nz))
# Check dimensions of phantom
dim(phantomArr)
```

```
## [1] 256 256  18
```

(b)

Display the data as 18 images on one page. Make sure that all the 18 images have the same scale and use a scale of light to dark (in grays) to display the images. (Consider using the "GE" logo that exists on the phantom to make sure that your images are correctly displayed.) [10 points]

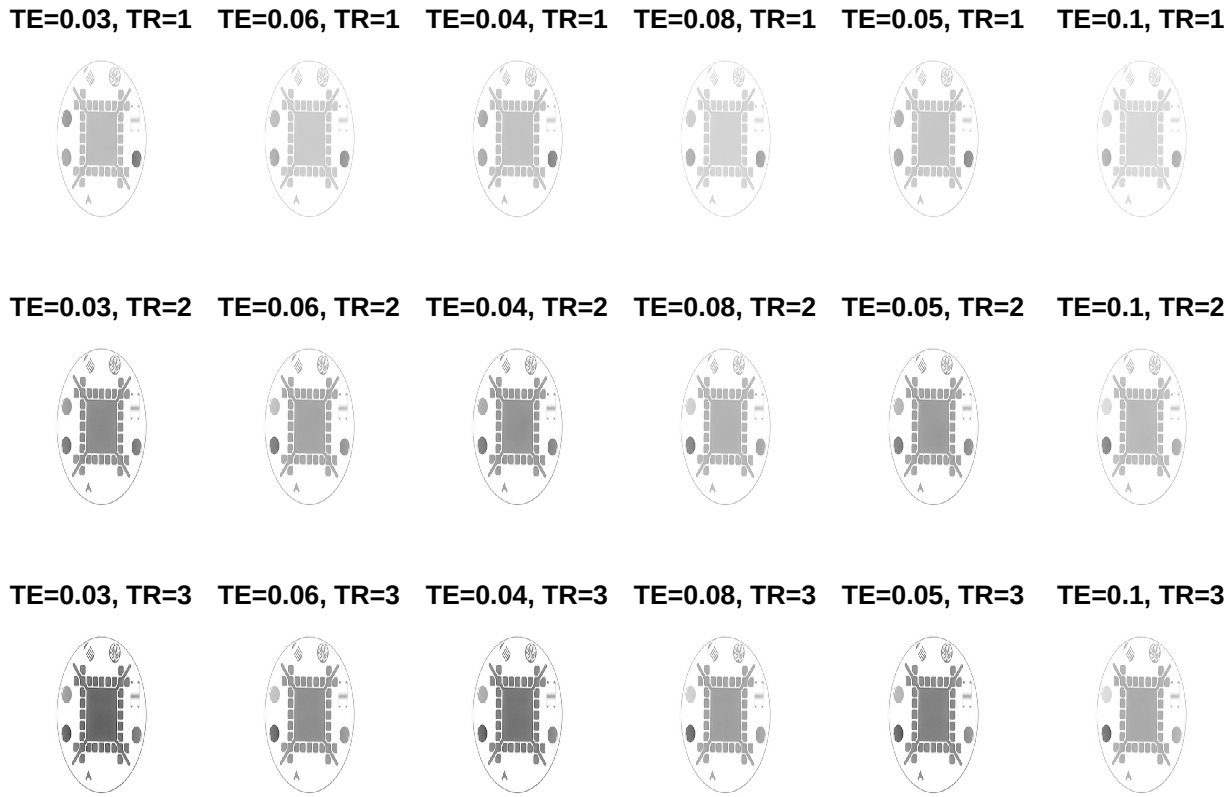
```r
teTrSettings <- read.table("te-tr.dat", header = FALSE)
colnames(teTrSettings) <- c("TE", "TR")

par(mfrow = c(3, 6), mar = c(2, 2, 2, 2))
# need min and max for dimensions
# "pad" empty spaces if they don't meet min, max
globalMin <- min(phantomArr)
globalMax <- max(phantomArr)

# yucky for loop version
# for (i in 1:nz) {
#   image(
#     matrix(phantomArr[, , i], nrow = nx, byrow = TRUE),
#     col = gray.colors(256, start = 1, end = 0),
#     zlim = c(globalMin, globalMax),
#     axes = FALSE,
#     main = paste0("TE=", teTrSettings$TE[i], ", TR=", teTrSettings$TR[i])
#   )
# }

mapply(
  # UDF!
  FUN = function(slice, te, tr) {
    image(
      t(matrix(slice, nrow = nx, byrow = TRUE)[nx:1, ]), # Flip vertically
      col = gray.colors(256, start = 1, end = 0), # Light to dark grayscale
      zlim = c(globalMin, globalMax), # Consistent scaling
      axes = FALSE,
      main = paste0("TE=", te, ", TR=", tr)
    )
  },
  # MoreArgs
  # argumnts to "vectorize" over
  slice = split(phantomArr, rep(1:nz, each = nx * ny)), # Extract slices as list
  te = teTrSettings$TE,
```

```
  tr = teTrSettings$TR
)
```




```
## $`1`
## NULL
##
## $`2`
## NULL
##
## $`3`
## NULL
##
## $`4`
## NULL
##
## $`5`
## NULL
##
## $`6`
## NULL
##
## $`7`
## NULL
##
## $`8`
```

```
## NULL
##
## $`9`
## NULL
##
## $`10`
## NULL
##
## $`11`
## NULL
##
## $`12`
## NULL
##
## $`13`
## NULL
##
## $`14`
## NULL
##
## $`15`
## NULL
##
## $`16`
## NULL
##
## $`17`
## NULL
##
## $`18`
## NULL
```

We see some "GE" logos, so this is a good sign that things are going right.

(c)

Write a function in R that takes in values of (ρ, T_1, T_2) and (TE, TR) and returns the mean observed intensity value for that setting as per Equation 1. [10 points]

```r
# what do you...mean? ;)
meanIntensity <- function(rho, T1, T2, TE, TR) {
  # I'm trying to add more failsafes to my functions
  # and this was a problem I encountered
  if (!all(dim(rho) == dim(T1), dim(T1) == dim(T2))) {
    stop("Inputs are not all the same dimension! Check dim of rho, T1, T2!")
  }
  # if not problematic
  # just a quick and dirty mean
  # signal equation provided, bless
  signal <- rho * exp(-TE / T2) * (1 - exp(-TR / T1))
  mean(signal)
}
```

(d)

Obtain the least-squares (LS) estimates of (ρ, T_1, T_2) given the observed image intensities at the set (TE, TR)s. Our goal therefore is to find (ρ_{ij}, T_{1ij}, T_{2ij}) at the (i,j)th image coordinate (matrix entry) that minimizes

$$\psi(\rho_{ij}, T_{1ij}, T_{2ij}; (\text{TE}_k, \text{TR}_k), Y_{i,j,k}, k = 1, 2, \ldots, 18) = \sum_{k=1}^{18} [Y_{i,j,k} > 0] \left[Y_{i,j,k} - F(\rho_{ij}, T_{1ij}, T_{2ij}; \text{TE}_k, \text{TR}_k)\right]^2 .$$

```
objectiveFun <- function(params, Y, TE, TR) {
  rho <- params[1]
  T1 <- params[2]
  T2 <- params[3]
  # signal formula, again
  # now called predicted
  predicted <- rho * exp(-TE / T2) * (1 - exp(-TR / T1))
  positiveN <- Y > 0
  sum((Y[positiveN] - predicted[positiveN])^2)
}

estimateParams <- function(Y, TE, TR, init = c(1, 1000, 100)) {
  result <- optim(
    par = init,
    fn = objectiveFun,
    Y = Y,
    TE = TE,
    TR = TR,
    # method seemed to work best for this of the options noted in description
    # method = c("Nelder-Mead", "BFGS", "CG", "L-BFGS-B", "SANN", "Brent")
    # take that, brent
    method = "L-BFGS-B",
    lower = c(0, 0, 0)
  )

  # we have multiple things to include in output
  # so storing as a list
  list(rho = result$par[1],
       T1 = result$par[2],
       T2 = result$par[3],
       value = result$value)
}
```

(e)

We will now estimate ($\rho_{ij}, T_{1ij}, T_{2ij}$) for each imaging coordinate given the 18 values.

In so doing, note that the value is indeterminate if there are no more than two $Y_{i,j,k}$s that are non-zero for any (i,j). For these cases, we will set $\rho_{ij} = 0$, $T_{1ij} = 4.0$, and $T_{2ij} = 0.001$. For the other (i,j)s, use the R function `optim` (note: not `optimize`) to estimate ($\rho_{ij}, T_{1ij}, T_{2ij}$). Restrict ρ to be in $[0, 3000]$, T_1 to be in $[0.01, 4.0]$, and T_2 to be in $[0.001, 2.0]$ by writing your function such that if these values are breached in any case, then the value returned is ∞. The function `optim` also needs initial values: use (1500, 1.5, 0.1) as starting values. Store the estimated ρ, T_1, T_2 values in three 256×256 matrices. [25 points]

I just added a bunch of failsafes because this kept giving me issues...

```
# NOTE:
# This was done when I kept getting errors and a lot of "spinning"
# For the purposes of grading, please see the more concise portion at the bottom
# a number of the debugging/unit testing methods used here ended up being superfluous

# objectiveFun <- function(params, Y, TE, TR) {
#   rho <- params[1]
#   T1 <- params[2]
#   T2 <- params[3]
#   # failsafe 1
#   # for inputs
#   # had issues with convergence
#   if (rho < 0 || rho > 3000 || T1 < 0.01 || T1 > 4.0 || T2 < 0.001 || T2 > 2.0) {
#     return(Inf)
#   }

#   # predictions, signal, same difference
#   predicted <- rho * exp(-TE / T2) * (1 - exp(-TR / T1))
#
#   # failsafe 2
#   # for predicted values
#   if (any(is.na(predicted)) || any(is.nan(predicted)) || any(!is.finite(predicted))) {
#     return(Inf)
#   }
#
#   # computed RSS
#   mask <- Y > 0
#   if (!any(mask)) return(Inf)
#   rss <- sum((Y[mask] - predicted[mask])^2)
#
#   # failsafe 3
#   # for RSS
#   if (is.na(rss) || is.nan(rss) || !is.finite(rss)) {
#     return(Inf)
#   }
#   return(rss)
# }
#
# estimatePixelPar <- function(Y, TE, TR) {
#   # one non-zero Response is ok
#   # but issues if we have more than 1
#   # failsafe
#   if (sum(Y > 0) <= 2) {
#     return(c(0, 4.0, 0.001))
#   }
#
#   # failsafe again,
#   # checking response
#   if (any(is.na(Y)) || any(is.nan(Y)) || any(!is.finite(Y))) {
#     return(c(0, 4.0, 0.001))
#   }
#
```

```
#   # actual optimization procedure
#   result <- tryCatch({
#     optim(
#       par = c(1500, 1.5, 0.1),
#       fn = objectiveFun,
#       Y = Y,
#       TE = TE,
#       TR = TR,
#       method = "L-BFGS-B",
#       lower = c(0, 0.01, 0.001),
#       upper = c(3000, 4.0, 2.0)
#     )
#   }, error = function(e) {
#     # extrra failsafe for optimization
#     return(list(par = c(0, 4.0, 0.001)))
#   })
#
#   # output
#   return(result$par)
# }

# yucky for loop version
# estimateImgParams <- function(imageDat, TE, TR) {
#   nx <- dim(imageDat)[1]
#   ny <- dim(imageDat)[2]
#   nz <- dim(imageDat)[3]
#
#   rhoMat <- matrix(0, nrow = nx, ncol = ny)
#   t1Mat <- matrix(0, nrow = nx, ncol = ny)
#   t2Mat <- matrix(0, nrow = nx, ncol = ny)
#
#   for (i in 1:nx) {
#     for (j in 1:ny) {
#       Y <- imageDat[i, j, ]
#       params <- estimatePixelPar(Y, TE, TR)
#       rhoMat[i, j] <- params[1]
#       t1Mat[i, j] <- params[2]
#       t2Mat[i, j] <- params[3]
#     }
#   }
#   list(rho = rhoMat, T1 = t1Mat, T2 = t2Mat)
# }

# estimateImgPar <- function(imageDat, TE, TR, progress = TRUE) {
#   nx <- dim(imageDat)[1]
#   ny <- dim(imageDat)[2]
#   nz <- dim(imageDat)[3]
#
#   # convert from array to Matrix for operations
#   imageDat2D <- matrix(imageDat, nrow = nx * ny, ncol = nz, byrow = TRUE)
#
#   # estimation on the "pixel"
#   results <- t(apply(imageDat2D, 1, function(Y) {
```

```
#     estimatePixelPar(Y, TE, TR)
#   }))
#
#   # back into format we want, 3 outputs
#   rhoMatrix <- matrix(results[, 1], nrow = nx, ncol = ny)
#   t1Matrix <- matrix(results[, 2], nrow = nx, ncol = ny)
#   t2Matrix <- matrix(results[, 3], nrow = nx, ncol = ny)
#
#   # output as list
#   list(rho = rhoMatrix,
#        T1 = t1Matrix,
#        T2 = t2Matrix)
# }
#
# TE <- teTrSettings$TE
# TR <- teTrSettings$TR
# testDat <- phantomArr[1:10, 1:10, ]
# # testRes <- estimateImgPar(imageDat = testDat, TE = TE, TR = TR)
# results <- estimateImgPar(imageDat = phantomArr,
#                           TE = TE,
#                           TR = TR)
#
# rhoMat <- results$rho
# t1Mat <- results$T1
# t2Mat <- results$T2
```

I can't tell if this text will actually above or below the actual code I ran, so here's that note: I removed a number of the failsafe/checks and ultimately went with the following.

```
# part (e)
estimateImageParams <- function(imageDat, TE, TR) {
  nx <- dim(imageDat)[1]
  ny <- dim(imageDat)[2]
  nz <- dim(imageDat)[3]

  imageDat2D <- matrix(imageDat, nrow = nx * ny, ncol = nz, byrow = TRUE)
  lowerBounds <- c(0, 0.01, 0.001)
  upperBounds <- c(3000, 4.0, 2.0)
  init <- c(1500, 1.5, 0.1)

  # function for each pixel
  processPixel <- function(Y) {
    # still need this check
    if (sum(Y > 0) <= 2) {
      # otherwise return a default set of vals
      return(c(0, 4.0, 0.001))
    }
    # Run optimization
    result <- optim(
      par = init,
      fn = function(params) {
        objectiveFun(params, Y, TE, TR)
      },
```

```
      method = "L-BFGS-B",
      lower = lowerBounds,
      upper = upperBounds
    )
    return(result$par)
  }

  # reformat output
  results <- t(apply(imageDat2D, 1, processPixel))

  # extract each "column"
  rhoMat <- matrix(results[, 1], nrow = nx, ncol = ny)
  t1Mat <- matrix(results[, 2], nrow = nx, ncol = ny)
  t2Mat <- matrix(results[, 3], nrow = nx, ncol = ny)

  # return output as list
  list(rho = rhoMat, t1 = t1Mat, t2 = t2Mat)
}

te <- teTrSettings$TE
tr <- teTrSettings$TR
# call and get actual data
results <- estimateImageParams(imageDat = phantomArr,
                               TE = te,
                               TR = tr)

rhoMat <- results$rho
t1Mat <- results$t1
t2Mat <- results$t2
```

(f)

Display the ρ, T_1, and T_2 images estimated as per part (e) above. [5 points]

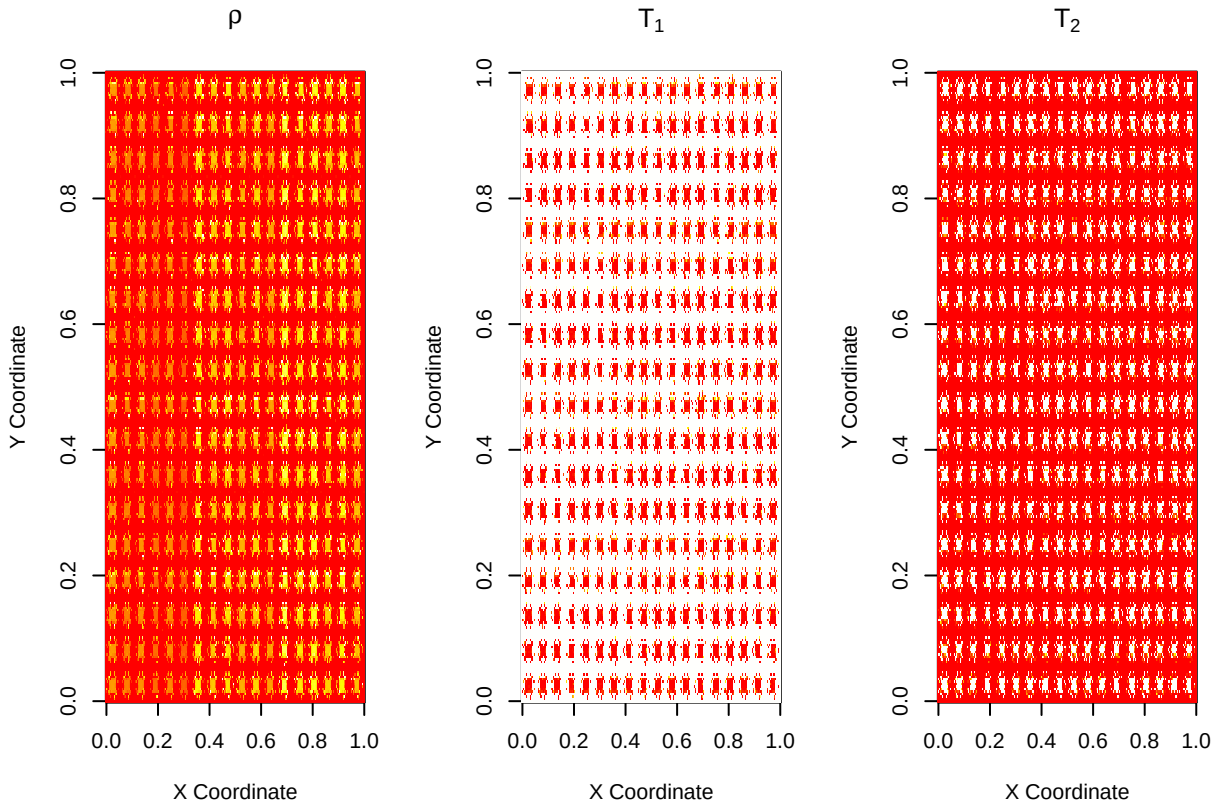
```
displayEstImg <- function(rhoMat, t1Mat, t2Mat) {
  par(mfrow = c(1, 3))

  image(
    t(rhoMat[nrow(rhoMat):1, ]),
    col = heat.colors(256),
    main = expression(rho),
    xlab = "X Coordinate",
    ylab = "Y Coordinate"
  )

  image(
    t(t1Mat[nrow(t1Mat):1, ]),
    col = heat.colors(256),
    main = expression(T[1]),
    xlab = "X Coordinate",
    ylab = "Y Coordinate"
  )
```

```
  image(
    t(t2Mat[nrow(t2Mat):1, ]),
    col = heat.colors(256),
    main = expression(T[2]),
    xlab = "X Coordinate",
    ylab = "Y Coordinate"
  )
}

# this naming convention isn't ideal, I'll admit
displayEstImg(rhoMat = rhoMat,
              t1Mat = t1Mat,
              t2Mat = t2Mat)
```



(g)

Our final objective here is to plug in the 18 (TE, TR) values to the ρ, T_1, and T_2 images estimated as per part (e) and to obtain fitted images at the 18 (TE, TR) settings. Use the function you wrote in part (c) along with the estimates obtained in part (e) to obtain the fitted images. Display these images using the same scale as in part 1b above. [15 points]

```
# for loop version
# computeFittedImg <- function(rhoMat, t1Mat, t2Mat, TE, TR) {
#   nx <- nrow(rhoMat)
```

```
#   ny <- ncol(rhoMat)
#   teLength <- length(TE)
#   fittedImg <- array(0, dim = c(nx, ny, teLength))
#
#   for (k in 1:teLength) {
#     fittedImg[, , k] <- rhoMat * exp(-TE[k] / t2Mat) * (1 - exp(-TR[k] / t1Mat))
#   }
#   return(fittedImg)
# }

computeFittedImg <- function(rhoMat = rhoMat,
                             t1Mat = t1Mat,
                             t2Mat = t2Mat,
                             TE = TE,
                             TR = TR) {
  nx <- nrow(rhoMat)
  ny <- ncol(rhoMat)
  teLength <- length(TE)

  teArr <- array(TE, dim = c(nx, ny, teLength))
  trArr <- array(TR, dim = c(nx, ny, teLength))
  rhoArr <- array(rhoMat, dim = c(nx, ny, teLength))
  t1Arr <- array(t1Mat, dim = c(nx, ny, teLength))
  t2Arr <- array(t2Mat, dim = c(nx, ny, teLength))

  fittedImg <- rhoArr * exp(-teArr / t2Arr) * (1 - exp(-trArr / t1Arr))
  fittedImg
}
```

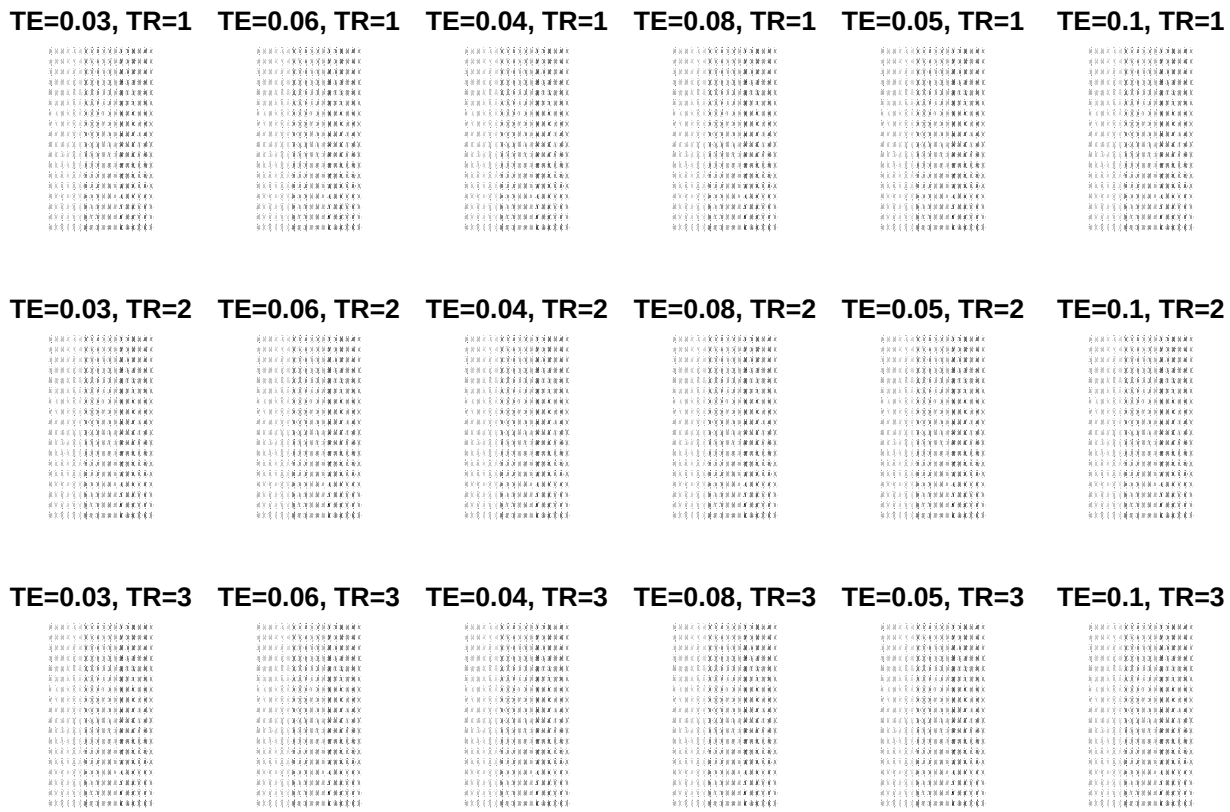
```
fittedImg <- computeFittedImg(rhoMat = rhoMat,
                              t1Mat = t1Mat,
                              t2Mat = t2Mat,
                              TE = teTrSettings$TE,
                              TR = teTrSettings$TR)

# need min and max again to determine output dimensions
globalMin <- min(fittedImg, na.rm = TRUE)
globalMax <- max(fittedImg, na.rm = TRUE)
```

```
# for loop version
# par(mfrow = c(3, 6), mar = c(2, 2, 2, 2))
# # 3 x 6 = 18
# for (k in 1:18) {
#   image(
#     t(fittedImg[, , k][nrow(fittedImg):1, ]),
#     col = gray.colors(256, start = 1, end = 0),
#     zlim = c(globalMin, globalMax),
#     axes = FALSE,
#     main = paste0("TE=", teTrSettings$TE[k], ", TR=", teTrSettings$TR[k])
#   )
# }

par(mfrow = c(3, 6), mar = c(2, 2, 2, 2))
```

```
mapply(
  FUN = function(k, te, tr) {
    imgSlice <- fittedImg[, , k]
    image(
      t(imgSlice[nrow(imgSlice):1, ]),
      col = gray.colors(256, start = 1, end = 0),
      zlim = c(globalMin, globalMax),
      axes = FALSE,
      main = paste0("TE=", te, ", TR=", tr)
    )
  },
  # additional inputs
  k = 1:dim(fittedImg)[3],
  te = teTrSettings$TE,
  tr = teTrSettings$TR
)
```




```
## [[1]]
## NULL
##
## [[2]]
## NULL
##
## [[3]]
## NULL
```

```
##
## [[4]]
## NULL
##
## [[5]]
## NULL
##
## [[6]]
## NULL
##
## [[7]]
## NULL
##
## [[8]]
## NULL
##
## [[9]]
## NULL
##
## [[10]]
## NULL
##
## [[11]]
## NULL
##
## [[12]]
## NULL
##
## [[13]]
## NULL
##
## [[14]]
## NULL
##
## [[15]]
## NULL
##
## [[16]]
## NULL
##
## [[17]]
## NULL
##
## [[18]]
## NULL
```

This is a bit like the "digits" problem from assignments past... Is this really the output we're looking for?

2

The dataset MPRAGE0.nii.gz contains the anatomic MR image volume of a healthy female in NIFTI format. The R package oro.nifti is required to read this file, with the function `readNIfTI`. The component `.Data` in the stored object (accessed via the `@` operator) contains the three-dimensional array containing the image voxel values. Display the volume via a three-dimensional contour display such as using the `contour3d` function in the misc3d package. Note that slices do not have the same dimensions (in particular, the vertical axis in the volume has slices that are 1.5 mm apart while the slices on the other two axes are 0.94 mm apart). Account for these differences by scaling the axes appropriately by checking out the recordings from the last virtual asynchronously recorded lecture on 3D graphics. Also test out several viewpoints and display the view that portrays the image the best in your opinion. [30 points]

```
library("oro.nifti")
library("misc3d")
library("rgl")

niftiDat <- readNIfTI("MPRAGEco.nii.gz", reorient = FALSE)
imageVolume <- niftiDat@.Data

dims <- dim(imageVolume)
spacing <- c(0.94, 0.94, 1.5)

x <- seq(0, dims[1] - 1) * spacing[1]
y <- seq(0, dims[2] - 1) * spacing[2]
z <- seq(0, dims[3] - 1) * spacing[3]

contour3d(
  imageVolume,
  x = x, y = y, z = z,
  level = mean(imageVolume),
  alpha = 0.2,
  draw = TRUE
)

# used this for messing around and getting the image I then knit
# I crashed my computer a few times for this
# view3d(theta = 0, phi = 0, zoom = 0.5)
# rglwidget()
# rgl.snapshot("3d_plot_snapshot.png")
```

```
knitr::include_graphics("3d_plot_snapshot.png")
```

My computer is very close. I ended up with a "top down" image as the others just seemed a bit "off", with weird borders around the edge (non brain/head parts), which you still see, though not to a large extent.

Figure 1: CocoMelon

3

The K-means algorithm iteratively partitions a dataset $\{X_1, X_2, \ldots, X_n\}$ into groups G_k for $k = 1, 2, \ldots, K$ and group mean vectors for the observations in that partition by minimizing the objective function:

$$SSW = \min \sum_{i=1}^{n} \sum_{k=1}^{K} I[X_i \in G_k] \|X_i - \mu_k\|^2. \tag{3}$$

Both $I[X_i \in G_k]$ and μ_k are parameters that are estimated by the algorithm which finds locally optimizing solutions in the vicinity of its initialization, therefore it is recommended that the algorithm be run to completion from several starting points. However, this approach is time-consuming so Maitra (2009) (Digital Object Identifier no. 10.1109/TCBB.2007.70244) suggested running the K-means algorithm for one iteration from multiple (M) starting points and then choosing the one that has the lowest values of SSW and then running that to convergence. A further modification would choose the starting points with the m lowest values of SSW and then running K-means to convergence for each of these m values and then choosing the one with the lowest SSW as the optimal partitioning.

The thinking behind this approach is that algorithms with poorer starting values can not hope to recover much and need not be labored over, but rather that the time can be more profitably used by searching over a larger set of initializing values. A further unpublished suggestion, made by former Iowa State University student Wei-Chien Chen (Ph.D., Statistics, 2013) can be modified to run the K-means algorithm not to convergence but for a small number (I) of iterations and then performing the evaluations suggested by Maitra (2009). We will use the R function `kmeans` to easily implement the smorgasbord of suggestions resulting from Maitra (2009) in an embarrassingly parallel setting. [30 points]

(a)

Write a function in which R's `kmeans` function in parallel for M randomly initialized starting points and runs each initialized K-means algorithm for I iterations, chooses the m best solutions and then runs each to convergence, finally choosing the solution with the lowest SSW. Note that the `kmeans` function in R has all the arguments (`iter.max`, `nstart`) that you need and in particular that the function returns the objective function (tot.withinss) as well as the centers (`centers`) and grouping (`cluster` at iteration). [30 points]

```
library(parallel)

optimizedKMeans <- function(data, k, m, i, numBestSolutions) {
  # data, byeah
  # k clusters
  # m random starts
  # i iterations
  # numBestSolutions, yeah

  # set up parallel processing
  cl <- makeCluster(detectCores() - 1)
  clusterExport(
    cl,
    varlist = c("data", "k", "i"),
    envir = environment()
  )

  # k means call, using parallel processing
  initialResults <- parLapply(cl, 1:m, function(x) {
    kmeans(
      x = data,
      centers = k,
      iter.max = i,
      nstart = 1
    )
  })

  # SAVE YOUR RESOURCES MY GOD WHAT HAVE I DONE
  stopCluster(cl)

  # method SSW for determining "best"
  initialSsw <- sapply(initialResults, function(res) res$tot.withinss)

  # identify "best" result
  bestIndices <- order(initialSsw)[1:numBestSolutions]
  bestInitializations <- initialResults[bestIndices]

  # call original kmeans method
  finalResults <- lapply(bestInitializations, function(initRes) {
    kmeans(
      x = data,
      centers = initRes$centers,
      iter.max = 100,
      nstart = 1
    )
  })
```

```
  # get "final" best
  finalSsw <- sapply(finalResults, function(res) res$tot.withinss)
  # extract needed elements from "best"
  bestSolutionIndex <- which.min(finalSsw)
  bestSolution <- finalResults[[bestSolutionIndex]]

  bestSolution
}
```

(b)

K-means color quantization is used in computer graphics to reduce the number of colors in an image without appreciably losing its visual quality. The importance of this process comes from a need to display images on devices that are not completely capable of dealing with multicolor images. It is also used for some image storing standards such as the Graphics Interchange Format (GIF).

Each pixel in an image is represented in terms of its primary components namely Red, Green, and Blue. Therefore, each pixel has a certain amount of Red, a certain amount of Green, and a certain amount of Blue. This way of representing color is known as RGB format. So, every picture can be presented as a three-dimensional dataset with the number of observations depending on its size: thus an image of size $P \times Q$ pixels is transformed into a dataset with $M = N$ observations. K-means color quantization (Digital Object Identifier: 10.1080/01621459.2011.646935) applies the K-means algorithm, suitably initialized, to such a dataset to yield a palette of K colors for representing the image. We note that K-means represents one of several approaches to color quantization. We use the function written in part 3a to represent an image using 16 colors.

i.

Reading in a TIFF image.
The file provided jubabrinda.tif in the usual places provides a digital file in the Tagged Image File Format (TIFF) of bamboo-handiwork decorations at a street-side exhibit during Kolkata's (formerly, Calcutta) famed (Sharadiyutsab) fall festival. The currently archived R package rtiff can read in this file as follows:

`library(rtiff) juba <- readTIFF(fn="jubabrinda.tif") plot(juba)`

The amount of the primary colors at each pixel can be obtained from the read TIFF file by using: `juba@red`, `juba@green`, `juba@blue`. Note that these are all matrices of the same dimension as the image. (Make sure that you look at the help on `pixmap-class` to get an idea of what the components of `pixmap` are). Note also that these values are all between 0 and 1.

The alternative R package tiff may also be used. To read in the file, we use:

`library(tiff) xx <- readTIFF(source="jubabrinda.tif") img <- as.raster(xx) plot(img)`

The object `xx` is a 3-dimensional array with RGB values in the last dimension. (If it is a 4-dimensional array, it has alpha, that is, transparency values, in the last dimension.)

Read in the TIFF image and convert to a dataset of 375000 3-dimensional observations. [10 points]

```
library("tiff")
xx <- readTIFF(source="jubabrinda.tif")
img <- as.raster(xx)
plot(img)
```

```
imgArr <- readTIFF(source = "jubabrinda.tif")
dataMat <- matrix(imgArr, ncol = 3, byrow = FALSE)

dim(imgArr)
```

```
## [1] 500 750   3
```

```
# head(imgArr)
dim(dataMat)
```

```
## [1] 375000      3
```

```
# head(dataMat)
```

ii.

Use K-means with $K = 16$ and your function in part 3a with $M = 10000$, $I = 3$, and $m = 10$.

(If your resources permit, you may try larger values of M.) Note that this exercise may take up to an hour. Pack the cluster output from your K-means function into a matrix of the size of the image and display using the image function and a categorical palette (e.g., Set2 of RColorBrewer). [20 points]

```
# optimizedKMeans <- function(data, k, m, i, numBestSolutions) {
# data might be image_matrix?
# I MADE A BAILOUT
# I LOVE MY COMPUTER
# result <- optimizedKMeans(
#   data = dataMat,
#   k = 16,
#   m = 10000,
#   i = 3,
#   numBestSolutions = 10
# )
#
# saveRDS(result, file = "optimKMeansDat.rds")
```

The above was run and used for creating a "bailout" object. There were warnings when calling the function, but for brevity the bailout was used so this all knit together in under an hour.

```
result <- readRDS("optimKMeansDat.rds")
```

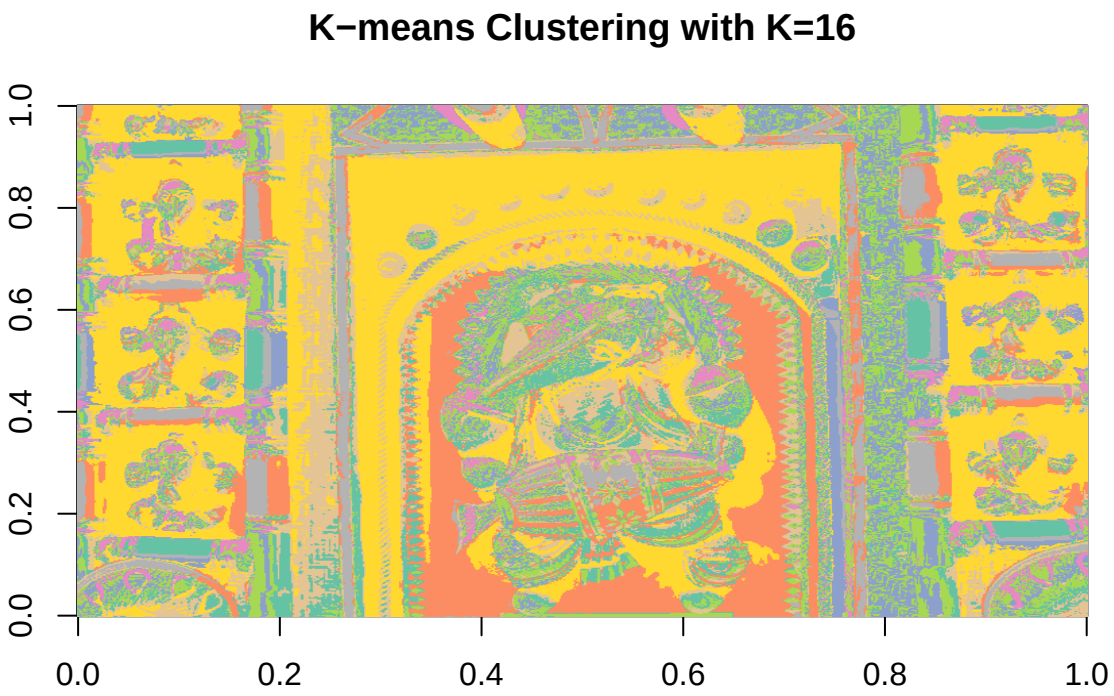
```
imgX <- dim(imgArr)[1]
imgY <- dim(imgArr)[2]
clusterImg <- matrix(result$cluster,
                     nrow = imgX,
                     ncol = imgY)
```

```
library(RColorBrewer)

tImg <- t(clusterImg)
flipImg <- tImg[, ncol(tImg):1]

image(
  flipImg,
  col = brewer.pal(16, "Set2"),
  main = "K-means Clustering with K=16",
  useRaster = TRUE,
  xlab = "", ylab = ""
)
```

```
## Warning in brewer.pal(16, "Set2"): n too large, allowed maximum for palette Set2 is 8
## Returning the palette you asked for with that many colors
```



iii.

Replace the cluster indicator for each pixel with the means obtained from your function.

Thus we get a matrix for the red (consisting of 16 distinct values), another matrix for the green (consisting of 16 distinct values), and a third matrix for the blues (again having 16 distinct values). Use `pixmap` to combine these matrices and the function `writeTIFF` to write to a TIFF file your quantized image. Display using R as in part 3(b)i. Comment on how well your quantized image reproduced the original. [30 points]

```
library("pixmap")
library("tiff")

clusterCenters <- result$centers
quantDat <- clusterCenters[result$cluster, ]

imgX <- dim(imgArr)[1]
imgY <- dim(imgArr)[2]

quantImgArr <- array(quantDat,
                     dim = c(imgX, imgY, 3)
                     )

quantImg <- pixmapRGB(data = quantImgArr)
```

```
## Warning in rep(cellres, length = 2): 'x' is NULL so the result will be NULL
```

```r
# second bailout
# saveRDS(quantImg, file = "quantImg.rds")
```

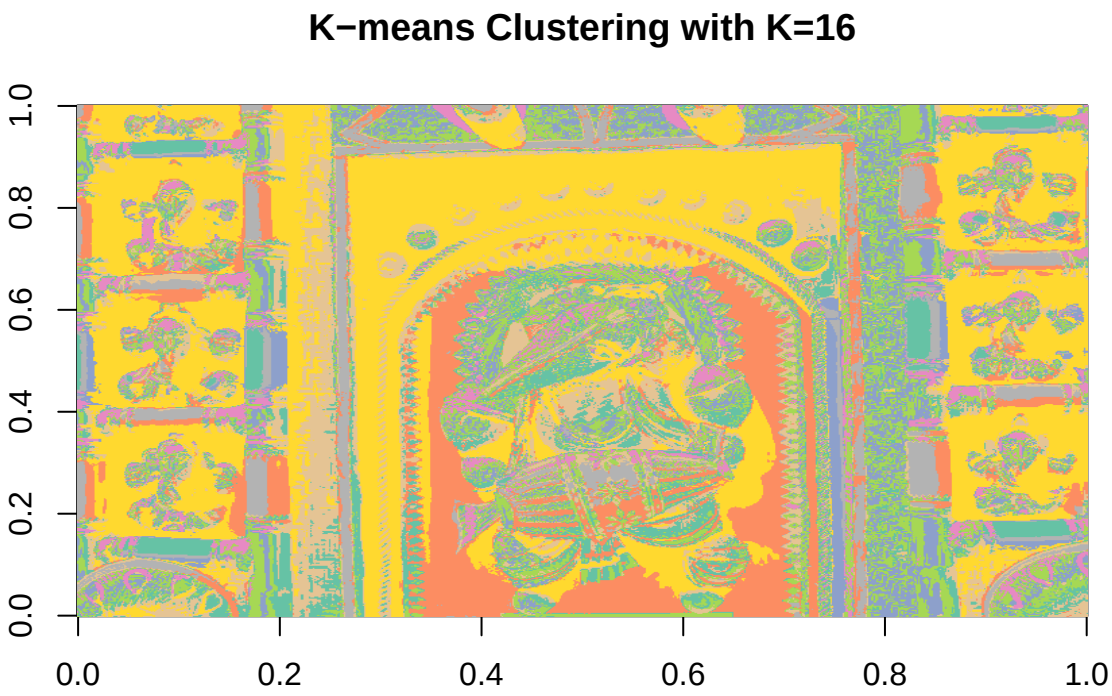
```r
quantImg <- readRDS("quantImg.rds")
```

```r
# Display original image
plot(as.raster(imgArr))
title(main = "Original (JUBA) Image")
```


```r
# K Means
tImg <- t(clusterImg)
flipImg <- tImg[, ncol(tImg):1]

image(
  flipImg,
  col = brewer.pal(16, "Set2"),
  main = "K-means Clustering with K=16",
  useRaster = TRUE,
  xlab = "", ylab = ""
)
```

```
## Warning in brewer.pal(16, "Set2"): n too large, allowed maximum for palette Set2 is 8
## Returning the palette you asked for with that many colors
```




```
library(pixmap)
# Display quantized image
# quantized_raster <- as.raster(quantized_image)
plot(quantImg, main = "K-means Image")
```


```
object.size(imgArr)
```

Fun Fact

```
## 9000224 bytes
```

```
object.size(quantImg)
```

```
## 9002560 bytes
```

```
testImage <- pixmapRGB(data = imgArr)
```

```
## Warning in rep(cellres, length = 2): 'x' is NULL so the result will be NULL
```

```
object.size(testImage)
```

```
## 9002560 bytes
```

Hey, so the amount of data required to locally hold/store the original array and the quantized image are nearly identical (the K-means one is slightly larger). Uh oh!

This is possibly due to storage as objects of different classes though, fingers crossed!