Appendix B: Code Scripts

This appendix contains the necessary code to reproduce the Three Circles results in Section 4.

Please go to https://github.com/JimmyJHickey/EM-for-Transmission-Tomography/tree/clean to find code scripts and data files to generate figures and results for other theta patterns.

```
#################### Construct a Matrix to Represent a Circle ####################

in_circle = function(radius){

  # radius of scan
  # approximately 5% bigger than the radius of the patient cross section
  out_radius = as.integer(radius * 1.05 +1)

  center = (2 * out_radius + 1 + 1) %/% 2

  nrow =  2 * out_radius + 3
  ncol =  2 * out_radius + 3

  # start everything at -1
  in_circ = matrix(-1, nrow = nrow, ncol = ncol)

  # if outside of the scan -1
  # if inside the scan but outside the patient 0
  # if inside the patient random(Uniform(0, 0.25))
  for(x in 1:nrow){
    for(y in 1:ncol){
      # if within the patient radius
      if( sqrt( (x - center -1 )^2 + (y - center-1)^2 ) <= radius ){
        in_circ[x,y] = runif(1, 0, 0.25)
      }
      else if( sqrt( (x - center -1 )^2 + (y - center-1)^2 ) <= out_radius ){
        in_circ[x,y] = 0
      }
    }
  }


  return(in_circ)
}
```

```
############################## Plot a Matrix of Pixels ##############################

plot_matrix = function(in_mat){
  require(ggplot2)

  x = seq(1, nrow(in_mat))
  y = seq(1, ncol(in_mat))

  df = expand.grid(X=x, Y=y)
  df$Z = c(in_mat)

  # doesn't seem to care about those colors at all, but it works
  ggplot(df, aes(X, Y, fill= Z)) +
    geom_tile()+
    scale_fill_continuous(type = "viridis",
                          limits = c(0,4),
                          breaks = c(0, 1, 2, 3, 4),
                          guide_colourbar(nbin = 100),
                          name = "theta")
  }



######################## Create Circles within Cross Section ########################

circle_pattern = function(in_mat, radius, center_x, center_y){
  out_mat = in_mat

  for(x in 1:nrow(out_mat)){
    for(y in 1:ncol(out_mat)){
      # if in the outer circle and the inscribed circle
      if((in_mat[x,y] >0) &&
        (sqrt( (x - center_x)^2 + (y - center_y)^2 ) <= radius )){
          # generate data inversely proportional to
          # distance from center of inscribed circle
           out_mat[x,y] = out_mat[x,y] +  abs(rnorm(1,
                              1/sqrt( 1+(x - center_x)^2 + (y - center_y)^2),
                              1))
      }
    }
  }

  return(out_mat)
}
```

```
####################### Simulate CT Scan X-ray Projections #######################

#ST793 Final Project
#EM algorithm for CT Data
#Eric Yanchenko, Alvin Sheng, and Jimmy Hickey [Copyright]
#November 4, 2020

# calculates column-majorized vector index of a matrix index
calc_index = function(nrow, x, y){
  return(nrow * (y-1) + x)
}



# inverse of calc_index (for testing purposes)
calc_matrix_index <- function(idx, num_row) {

  row_idx <- rep(NA, length(idx))

  col_idx <- rep(NA, length(idx))

  for (i in 1:length(idx)) {

    col_idx[i] <- ceiling(idx[i] / num_row)

    row_idx[i] <- idx[i] - (col_idx[i] - 1) * num_row

  }

  # This order is more intuitive as Cartesian coordinates
  return(cbind(col_idx, row_idx))

}



# returns a logical vector indicating the entries that are -1 and at either end of
the vector
# I use this method of taking out -1's, so I can detect weird cases in which
# -1 is in the middle of the vector instead
neg_ends <- function(hit_thetas) {

  p <- length(hit_thetas)

  nends <- rep(FALSE, length(hit_thetas))

  i <- 1

  while (hit_thetas[i] < 0 && i <= p) {
    nends[i] <- TRUE
    i <- i + 1
  }

  if (i == p) { # if it went through entire vector already
    return(nends)
```

```
  }

  # go from the other side

  other_i <- p

  while(hit_thetas[other_i] < 0 && nends[other_i] == FALSE && other_i >= 1) {
    nends[other_i] <- TRUE
    other_i <- other_i - 1
  }

  return(nends)

}



#Function which generates the observed data for a single theta vector
data_gen <- function(theta, d, l){
  #Theta is the given theta vector used to compute attenutation probs.
  #d is mean of initial Poisson distribution
  #l is the length of each projection through each pixel

  p = length(theta)

  if(p<=0){#If empty theta vector is given, return NA
    return(NA)
  }

  X1 = rpois(1, lambda = d) #Initial number of counts from Poisson dist.

  X = c(X1, rep(0, p)) #Vector to hold the number of counts after each interaction

  for(j in 1:p){
    X[j+1] <- rbinom(1, size = X[j], prob = exp(-theta[j] * l))
    #Generate the number of photons from Binom(X_i, exp(-theta_i * l))
  }

  Y = X[p+1] #Actually observed data (number of counts)

  return(Y)
}


#' Function which generates the observed data for entire theta matrix and arbitrary
projections
#'
#' @param THETA input theta matrix used to compute probs.
#' @param d Poisson mean for initial Poisson generation
#' @param ROW rows which we project on (vector) Enter negative row number to get
opposite direction
#' @param COL columns which we project on (vector) Enter negative col number to get
opposite direction
#' @param reps number of times to run the function
```

```
#' @param rise_vec vector of numerator(s) of the slopes of the parallel projections
(projections perpendicular to them will
#' also be generated)
#' @param run_vec vector of denominator(s) of the slopes of the parallel projections
(projections perpendicular to them will
#' also be generated)
#' Note: One of rise or run is assumed to be one, and the other is assumed to be an
integer >= one.
#' @return estimated theta values
#' @export
data_gen_df <- function(THETA, d, ROW, COL, reps=1, rise_vec = 1, run_vec = 1){

  r = dim(THETA)[1] #Number of rows in THETA
  c = dim(THETA)[2] #Number of cols in THETA

  #Check that ROW and COL projection indices are valid
  if(max(abs(ROW))> r || max(abs(COL)) > c){
    return(simpleError("Row and/or column indices out of range."))
  }

  #Return projection list
  #with d, length of projection within each pixel, indices (in order) beam went
through, counts
  proj.list <- vector("list", 1)

  pl_idx <- 1

  for(a in 1:reps){

    #Run over ROWS first
    for(rr in ROW){
      if(rr > 0){
        y <- data_gen(THETA[rr,THETA[rr,]>=0], d, l = 1)
        #Matrix indices that this beam goes through
        idx = seq(rr, rr+(c-1)*r, r)
        idx = idx[which(THETA[rr,]>=0, arr.ind=TRUE)] #Drop indices with negative
thetas
      }else{ #switch order of theta if row number is negative
        y <- data_gen(rev(THETA[-(rr),THETA[-(rr),]>=0]), d, l = 1)
        #Matrix indices that this beam goes through
        idx = seq(-rr, -rr+(c-1)*r, r)
        idx = rev(idx[which(THETA[-rr,]>=0, arr.ind=TRUE)])
      }

      if (!is.na(y)) {
        add.list <- list(d = d, l = 1, idx = idx, y = y)
        proj.list[[pl_idx]] <- add.list
        pl_idx <- pl_idx + 1
      }

    }

    #Run over COLS second
    for(cc in COL){
      if(cc >0){
```

```
        y<- data_gen(THETA[THETA[,cc]>=0,cc], d, l = 1)
        idx = seq((cc-1)*r+1, cc*r, 1)
        idx = idx[which(THETA[,cc]>=0, arr.ind = TRUE)] # Drop indices with negative
thetas
      }else{
        y <- data_gen(rev(THETA[THETA[,-(cc)]>=0,-(cc)]), d, l = 1)
        idx = seq((-cc-1)*r+1, -cc*r, 1)
        idx = rev(idx[which(THETA[,-cc]>=0, arr.ind=TRUE)])
      }

      if (!is.na(y)) {
        add.list <- list(d = d, l = 1, idx = idx, y = y)
        proj.list[[pl_idx]] <- add.list
        pl_idx <- pl_idx + 1
      }

    }

    #Check that rise_list and run_list have the same lengths
    if(length(rise_vec) != length(run_vec)){
      return(simpleError("rise_vec and run_vec have different numbers of elements"))
    }

    if (length(rise_vec) != 0) {
      for (i in 1:length(rise_vec)) {
        # call a separate function for the angular projections
        angular_proj_list <- angular_proj_list_gen(THETA, d, ROW, COL, rise =
rise_vec[i], run = run_vec[i])
        proj.list <- c(proj.list, angular_proj_list)
        pl_idx + length(angular_proj_list)
      }
    }

  }

  #Return list with d, l, idx, y for each
  return(proj.list)
}



#' Function to generate one set of angular projections, to be used within data_gen_df
#'
#' @param THETA input theta matrix used to compute probs.
#' @param d Poisson mean for initial Poisson generation
#' @param ROW rows which we project on (vector) Enter negative row number to get
opposite direction
#' @param COL columns which we project on (vector) Enter negative col number to get
opposite direction
#' @param rise numerator of the slope of the parallel projections (projections
perpendicular to them will
#' also be generated)
#' @param run denominator of the slope of the parallel projections (projections
perpendicular to them will
#' also be generated)
```

```
#' Note: One of rise or run is assumed to be one, and the other is assumed to be an
integer >= one.
#' @return estimated theta values
#' @export
angular_proj_list_gen <- function(THETA, d, ROW, COL, rise, run) {

  r = dim(THETA)[1] #Number of rows in THETA
  c = dim(THETA)[2] #Number of cols in THETA

  rise_init <- rise
  run_init <- run

  # standardize the slope
  rise <- rise / max(rise_init, run_init)
  run <- run / max(rise_init, run_init)

  # length that each projection traverses across each pixel
  l <- sqrt(rise^2 + run^2)

  #Return projection list
  #with d, length of projection within each pixel, indices (in order) beam went
through, counts
  proj.list <- vector("list", 1)

  pl_idx <- 1



  # 1. Projections with slope (rise/run), starting from left wall
  for(rr in ROW){
    if(rr > 0){

      curr_row = rr
      curr_col = 1

      #Matrix indices that this beam goes through
      idx = c()

      # thetas intersected
      hit_thetas = c()

      # while still in bounds of the scan
      while((1 <= floor(curr_row) && floor(curr_row) <= r) &&
            (1 <= floor(curr_col) && floor(curr_col) <= c)){

        idx = c(idx, calc_index(r, floor(curr_row), floor(curr_col)))
        hit_thetas = c(hit_thetas, THETA[floor(curr_row), floor(curr_col)])
        curr_row = curr_row + rise
        curr_col = curr_col + run

      }
    }
    else{ # switch order of idx and theta if row number is negative
      curr_row = -rr
      curr_col = 1
```

```
    #Matrix indices that this beam goes through
    idx = c()

    # thetas intersected
    hit_thetas = c()

    # while still in bounds of the scan
    while((1 <= floor(curr_row) && floor(curr_row) <= r) &&
          (1 <= floor(curr_col) && floor(curr_col) <= c)){

      idx = c(idx, calc_index(r, floor(curr_row), floor(curr_col)))
      hit_thetas = c(hit_thetas, THETA[floor(curr_row), floor(curr_col)])
      curr_row = curr_row + rise
      curr_col = curr_col + run

    }

    idx <- rev(idx)
    hit_thetas <- rev(hit_thetas)

  }

  idx = idx[!neg_ends(hit_thetas)] #Drop indices with negative thetas
  hit_thetas <- hit_thetas[!neg_ends(hit_thetas)]

  if(length(idx) != 0 && sum(hit_thetas < 0) == 0){
    # if the projection didn't only go through -1's
    # and if there isn't a -1 in the middle of the vector
    y <- data_gen(hit_thetas, d, l = l)
    add.list <- list(d = d, l = l, idx = idx, y = y)
    proj.list[[pl_idx]] <- add.list
    pl_idx <- pl_idx + 1
  }

}

# 2. Projections with slope (-run/rise), starting from left wall
for(rr in ROW){
  if(rr > 0){

    curr_row = rr
    curr_col = 1

    #Matrix indices that this beam goes through
    idx = c()

    # thetas intersected
    hit_thetas = c()

    # while still in bounds of the scan
    while((1 <= floor(curr_row) && floor(curr_row) <= r) &&
          (1 <= ceiling(curr_col) && ceiling(curr_col) <= c)){

      idx = c(idx, calc_index(r, floor(curr_row), ceiling(curr_col)))
```

```
      hit_thetas = c(hit_thetas, THETA[floor(curr_row), ceiling(curr_col)])
      curr_row = curr_row - run
      curr_col = curr_col + rise

    }
  }
  else{ # switch order of idx and theta if row number is negative

    curr_row = -rr
    curr_col = 1

    #Matrix indices that this beam goes through
    idx = c()

    # thetas intersected
    hit_thetas = c()

    # while still in bounds of the scan
    while((1 <= floor(curr_row) && floor(curr_row) <= r) &&
          (1 <= ceiling(curr_col) && ceiling(curr_col) <= c)){

      idx = c(idx, calc_index(r, floor(curr_row), ceiling(curr_col)))
      hit_thetas = c(hit_thetas, THETA[floor(curr_row), ceiling(curr_col)])
      curr_row = curr_row - run
      curr_col = curr_col + rise

    }

    idx <- rev(idx)
    hit_thetas <- rev(hit_thetas)

  }

  idx = idx[!neg_ends(hit_thetas)] #Drop indices with negative thetas
  hit_thetas <- hit_thetas[!neg_ends(hit_thetas)]

  if(length(idx) != 0 && sum(hit_thetas < 0) == 0){
    # if the projection didn't only go through -1's
    # and if there isn't a -1 in the middle of the vector
    y <- data_gen(hit_thetas, d, l = l)
    add.list <- list(d = d, l = l, idx = idx, y = y)
    proj.list[[pl_idx]] <- add.list
    pl_idx <- pl_idx + 1
  }

}

# 3. Projections with slope (rise/run), starting from bottom wall
for(cc in COL[abs(COL) != 1]){ # skipping one to avoid redundancy
  if(cc > 0){

    curr_row = 1
    curr_col = cc

    #Matrix indices that this beam goes through
```

```
    idx = c()

    # thetas intersected
    hit_thetas = c()

    # while still in bounds of the scan
    while((1 <= floor(curr_row) && floor(curr_row) <= r) &&
          (1 <= floor(curr_col) && floor(curr_col) <= c)){

      idx = c(idx, calc_index(r, floor(curr_row), floor(curr_col)))
      hit_thetas = c(hit_thetas, THETA[floor(curr_row), floor(curr_col)])
      curr_row = curr_row + rise
      curr_col = curr_col + run

    }
  }
  else{ # switch order of idx and theta if col number is negative

    curr_row = 1
    curr_col = -cc

    #Matrix indices that this beam goes through
    idx = c()

    # thetas intersected
    hit_thetas = c()

    # while still in bounds of the scan
    while((1 <= floor(curr_row) && floor(curr_row) <= r) &&
          (1 <= floor(curr_col) && floor(curr_col) <= c)){

      idx = c(idx, calc_index(r, floor(curr_row), floor(curr_col)))
      hit_thetas = c(hit_thetas, THETA[floor(curr_row), floor(curr_col)])
      curr_row = curr_row + rise
      curr_col = curr_col + run

    }

    idx <- rev(idx)
    hit_thetas <- rev(hit_thetas)

  }

  idx = idx[!neg_ends(hit_thetas)] #Drop indices with negative thetas
  hit_thetas <- hit_thetas[!neg_ends(hit_thetas)]

  if(length(idx) != 0 && sum(hit_thetas < 0) == 0){
    # if the projection didn't only go through -1's
    # and if there isn't a -1 in the middle of the vector
    y <- data_gen(hit_thetas, d, l = l)
    add.list <- list(d = d, l = l, idx = idx, y = y)
    proj.list[[pl_idx]] <- add.list
    pl_idx <- pl_idx + 1
  }
```

```
}

# 4. Projections with slope (-run/rise), starting from top wall
for(cc in COL[abs(COL) != 1]){ # skipping one to avoid redundancy
  if(cc > 0){

    curr_row = r
    curr_col = cc

    #Matrix indices that this beam goes through
    idx = c()

    # thetas intersected
    hit_thetas = c()

    # while still in bounds of the scan
    while((1 <= floor(curr_row) && floor(curr_row) <= r) &&
          (1 <= ceiling(curr_col) && ceiling(curr_col) <= c)){

      idx = c(idx, calc_index(r, floor(curr_row), ceiling(curr_col)))
      hit_thetas = c(hit_thetas, THETA[floor(curr_row), ceiling(curr_col)])
      curr_row = curr_row - run
      curr_col = curr_col + rise

    }
  }
  else{ # switch order of idx and theta if col number is negative

    curr_row = r
    curr_col = -cc

    #Matrix indices that this beam goes through
    idx = c()

    # thetas intersected
    hit_thetas = c()

    # while still in bounds of the scan
    while((1 <= floor(curr_row) && floor(curr_row) <= r) &&
          (1 <= ceiling(curr_col) && ceiling(curr_col) <= c)){

      idx = c(idx, calc_index(r, floor(curr_row), ceiling(curr_col)))
      hit_thetas = c(hit_thetas, THETA[floor(curr_row), ceiling(curr_col)])
      curr_row = curr_row - run
      curr_col = curr_col + rise

    }

    idx <- rev(idx)
    hit_thetas <- rev(hit_thetas)

  }

  idx = idx[!neg_ends(hit_thetas)] #Drop indices with negative thetas
  hit_thetas <- hit_thetas[!neg_ends(hit_thetas)]
```

```
    if(length(idx) != 0 && sum(hit_thetas < 0) == 0){
      # if the projection didn't only go through -1's
      # and if there isn't a -1 in the middle of the vector
      y <- data_gen(hit_thetas, d, l = l)
      add.list <- list(d = d, l = l, idx = idx, y = y)
      proj.list[[pl_idx]] <- add.list
      pl_idx <- pl_idx + 1
    }

  }

  return(proj.list)

}
```

```
############### Expectation Maximization for Transmission Tomography ###############


#' Calculates mij for the E step
#'
#' @param proj list of information for this projection
#' @param theta matrix of initial theta values
#' @param j pixel of interest
#' @return mij
#' @export
mij <- function(proj, theta, j) {

  d <- proj$d

  idx <- proj$idx

  y <- proj$y

  l <- proj$l



  if (j %in% proj$idx) {

    k <- which(proj$idx == j)

  } else {

    return(0)

  }



  if (k > 1) {

    idx_sub <- idx[1:(k - 1)]

    theta_sub <- theta[idx_sub]

  } else {

    theta_sub <- 0

  }

  return(d * (exp(-sum(l * theta_sub)) - exp(-sum(l * theta[idx]))) + y)

}



#' Calculates nij for the E step
#'
#' @param proj list of information for this projection
```

```r
#' @param theta matrix of initial theta values
#' @param j pixel of interest
#' @return nij
#' @export
nij <- function(proj, theta, j) {

  d <- proj$d

  idx <- proj$idx

  y <- proj$y

  l <- proj$l



  if (j %in% proj$idx) {

    k <- which(proj$idx == j)

  } else {

    return(0)

  }



  idx_sub <- idx[1:k]

  theta_sub <- theta[idx_sub]



  return(d * (exp(-sum(l * theta_sub)) - exp(-sum(l * theta[idx]))) + y)

}



#' Maximizes Q function for pixel j to estimate theta j
#'
#' @param theta_j theta for pixel j to optimize for
#' @param proj_list list of projections
#' @param theta matrix of initial theta values
#' @param j pixel of interest
#' @return q function value for theta j
#' @export
q_fun_j <- function(thetaj, proj_list, theta, j) {

  num_proj <- length(proj_list)

  val <- 0

  for (i in 1:num_proj) {
```

```
    l <- proj_list[[i]]$l

    m_exp <- mij(proj_list[[i]], theta, j)

    n_exp <- nij(proj_list[[i]], theta, j)

    val <- val - n_exp * l + (m_exp - n_exp) * l / (exp(l * thetaj) - 1)

  }

  return(val)

}



#' Detects whether there are any projections with y = 0 in proj_list
#'
#' @param proj_list list of projections
#' @return TRUE/FALSE depending if a zero was detected or not
#' @export
y_zero <- function(proj_list) {

  has_zero <- FALSE

  for (proj in proj_list) {

    if (proj$y == 0) {
      has_zero <- TRUE
    }

  }

  return(has_zero)

}



# To bypass the theta = zero error
max_q_fun_j <- function(interval, proj_list, theta, j) {

  if (theta[j] == 0) {
    return(0)
  }

  e <- try(
    # if I need more efficiency, calculate nij and mij outside of function
    theta_est <- uniroot(q_fun_j, interval, proj_list, theta, j, extendInt =
"downX")$root,
    silent = TRUE
  )
```

```
  if (class(e) == "try-error") {
    return(0)
  } else {
    return(theta_est)
  }

}



#' EM algorithm for transmission tomography
#'
#' @param proj_list list of projections
#' @param theta matrix of initial theta values. Assumes the -1's in the matrix refer
to dead space to
#' be ignored
#' @param tol tolerance used for the stopping rule
#' @return estimated theta values
#' @export
em_alg <- function(proj_list, theta, tol) {

  if(y_zero(proj_list)){
    return(simpleError("There's a projection with zero photons observed."))
  }

  theta_est <- matrix(-1, nrow = nrow(theta), ncol = ncol(theta))

  # indices of theta_est that are nonnegative
  nonneg_idx <- which(theta >= 0)

  ctr <- 0

  diff <- Inf

  while (diff > tol & ctr <= 100000) {

    for (j in nonneg_idx) {

      theta_est[j] <- max_q_fun_j(interval = c(0.0000001, 10), proj_list, theta, j)

    }

    diff <- sum((theta_est - theta)^2)

    theta <- theta_est

    ctr <- ctr + 1

  }

  return(list(theta_est = theta_est, ctr = ctr))

}
```

```
################### Construct True Three Circles Theta Matrices ####################

dir.create("true_theta")

# radius 3

radius = 3

set.seed(316)

in_circ = in_circle(radius)

# add two circles
circle_theta1  = circle_pattern(in_circ, 1, 4, 7)
circle_theta2  = circle_pattern(circle_theta1, 1, 8, 8)
true_theta  = circle_pattern(circle_theta2, 1, 7, 4)
plot_matrix(true_theta)

# check the maximum thetas
summary(as.vector(true_theta))

save(true_theta, file = "true_theta/three_circles_rad3.RData")

# EM algorithm, 10 seconds




# radius 5

radius = 5

set.seed(339)

in_circ = in_circle(radius)
plot_matrix(in_circ)

# add two circles
circle_theta1  = circle_pattern(in_circ, 2, 10, 11)
circle_theta2  = circle_pattern(circle_theta1, 2, 5, 8)
true_theta  = circle_pattern(circle_theta2, 1, 11, 6)
plot_matrix(true_theta)

# check the maximum thetas
summary(as.vector(true_theta))

save(true_theta, file = "true_theta/three_circles_rad5.RData")

# EM algorithm: 2 minutes




# radius 10
```

```
radius = 10

set.seed(348)

in_circ = in_circle(radius)
plot_matrix(in_circ)

# add two circles
circle_theta1  = circle_pattern(in_circ, 4, 16, 17)
circle_theta2  = circle_pattern(circle_theta1, 3, 8, 11)
true_theta  = circle_pattern(circle_theta2, 2, 18, 9)
plot_matrix(true_theta)

# check the maximum thetas
summary(as.vector(true_theta))

save(true_theta, file = "true_theta/three_circles_rad10.RData")

# EM algorithm: 37.78225 minutes
# 36.80142 minutes 2nd time through. So having extendInt = "downX" doesn't change
things
```

```
##################### Run the Three Circles Simulation Study #####################

dir.create("em_results")
dir.create("em_results/three_circles")

#Set the names for the array
names.radius = c("rad3", "rad5", "rad10")
names.angles = c("none", "deg45", "deg.all")
names.met = c("RMSE", "Spectral", "Iterations")
names.N = c("N1", "N2", "N3", "N4", "N5", "N6", "N7", "N8", "N9", "N10")

names.list = list(names.radius, names.angles, names.met, names.N)

# store the seeds
seed_vec <- rep(NA, length(names.radius) * length(names.angles) * length(names.N))
loop_idx <- 1

#Array to hold all of our results
#Rename to match your circle_theta name
three_circles_results <- array(NaN, dim =  c(3,3,3,10), dimnames = names.list)

#Output is a multi-dimensional array
#Dim 1: Radius. Length=3 for radius=c(3,5,10)
#Dim 2: Angles Length=3 for angles: none, 45 deg and all three
#Dim 3: Metrics. Length=3 for metric = c('RMSE', 'Spectral', 'Iterations')
#Dim 4: Monte Carlo N. length=N=10
#Thus, we will have an 3x5x3x10 array (=450 data points)

#Set the radius and rep numbers to loop over
radius.seq = c(3,5,10)
rise.list = list(c(), c(1), c(1,2,1))
run.list = list(c(), c(1), c(1,1,2))
a = 1
d = 1e9
##----------------------------------------------------------------
##----------------------------------------------------------------
for(radius in radius.seq){
  #Keeps track of progress
  print(paste("radius=",radius))
  #Generating the three_circles_theta matrix. Different for each of us
  load(paste("true_theta/three_circles_rad",radius,".RData",sep=""))
  three_circles_theta  = true_theta

  # iterate over angles
  for(b in 1:3){
    print(paste("angles_ind =",b))
    for(N in 1:10){
      print(paste("N=",N))
      # generating observations
      seed <- as.numeric(ceiling(proc.time()[3]))
      seed_vec[loop_idx] <- seed
      set.seed(seed)
      # vector of boundary rows/columns that aren't solely negative space
      # this code will not be right if there's more than one layer of -1's
      bounds <- 1:nrow(three_circles_theta)
```

```
      #Generate data
      proj_list <- data_gen_df(three_circles_theta, d = d, ROW = bounds,
                               COL = bounds,
                               rise_vec = rise.list[[b]], run_vec = run.list[[b]])

      #Check for no 0 values. If so, regenerate data
      while(y_zero(proj_list)){
        proj_list <- data_gen_df(three_circles_theta, d = d, ROW = bounds,
                                 COL = bounds,
                                 rise_vec = rise.list[[b]], run_vec = run.list[[b]])
      }

      # how many nonnegative numbers are in three_circles_theta?
      num_pixel <- sum(three_circles_theta >= 0)

      # copy the true three_circles_theta's negative space, but change the
nonnegative
      # values randomly
      three_circles_theta_init <- three_circles_theta

      three_circles_theta_init[which(three_circles_theta >= 0)] <- runif(num_pixel,
0, 0.1)

      #Run Algorithm
      em_res <- em_alg(proj_list, three_circles_theta_init, .0001)

      save(em_res,
file=paste("em_results/three_circles/three_circles_radius",radius,"_numangles",
b,"_N", N ,".RData",sep=""))

      abs_diff_mat <- abs(em_res$theta_est - three_circles_theta)

      sq_diff_mat <- abs_diff_mat^2

      #plot_matrix(abs_diff_mat)

      #plot_matrix(sq_diff_mat)

      # RMSE:
      three_circles_results[a,b,1,N] <- sqrt(sum(sq_diff_mat) / num_pixel)
      #Spectral Norm
      three_circles_results[a,b,2,N] <-svd(em_res$theta_est -
three_circles_theta)$d[1]
      #Number of iterations
      three_circles_results[a,b,3,N] <-em_res$ctr

      loop_idx <- loop_idx + 1

    }

  }
  a = a+1

}
```

```
save(three_circles_results, file = "three_circles_results.RData")

save(seed_vec, file = "seed_vec_three_circles_results.RData")
```

```
#################### Plot Figures and Results for Three Circles ####################

# plotting figures for three circles, radius 5, 0 angles

load(paste("true_theta/three_circles_rad",5,".RData",sep=""))

load(paste("em_results/three_circles/three_circles_radius",5,"_numangles", 1,"_N",
8 ,".RData",sep=""))

plot_matrix(true_theta)

plot_matrix(em_res$theta_est)

abs_diff_mat <- abs(em_res$theta_est - true_theta)

plot_matrix(abs_diff_mat)



# plotting figures for three circles, radius 5, 2 angles

load(paste("true_theta/three_circles_rad",5,".RData",sep=""))

load(paste("em_results/three_circles/three_circles_radius",5,"_numangles", 2,"_N",
10 ,".RData",sep=""))

plot_matrix(true_theta)

plot_matrix(em_res$theta_est)

abs_diff_mat <- abs(em_res$theta_est - true_theta)

plot_matrix(abs_diff_mat)



# plotting figures for three circles, radius 5, 6 angles

load(paste("true_theta/three_circles_rad",5,".RData",sep=""))

load(paste("em_results/three_circles/three_circles_radius",5,"_numangles", 3,"_N",
7 ,".RData",sep=""))

plot_matrix(true_theta)

plot_matrix(em_res$theta_est)

abs_diff_mat <- abs(em_res$theta_est - true_theta)

plot_matrix(abs_diff_mat)



##################
```

```
# plotting figures for three circles, radius 10, 6 angles

load(paste("true_theta/three_circles_rad",10,".RData",sep=""))

load(paste("em_results/three_circles/three_circles_radius",10,"_numangles", 3,"_N",
7 ,".RData",sep=""))

plot_matrix(true_theta)

plot_matrix(em_res$theta_est)

abs_diff_mat <- abs(em_res$theta_est - true_theta)

plot_matrix(abs_diff_mat)





load("three_circles_results.RData")

#Output is a mulit-dimensional array
#Dim 1: Radius. Length=3 for radius=c(3,5,10)
#Dim 2: Angles Length=3 for angles: none, 45 deg and all three
#Dim 3: Metrics. Length=3 for metric = c('RMSE', 'Spectral', 'Iterations')
#Dim 4: Monte Carlo N. length=N=10
#Thus, we willl have an 3x5x3x10 array (=450 data points)

names.angles = c("none", "deg45", "deg.all")
names.radius = c("rad3", "rad5", "rad10")

names.list = list(names.angles, names.radius)



rmse_mean_mat <- matrix(NA, nrow = 3, ncol = 3, dimnames = names.list)

spec_mean_mat <- matrix(NA, nrow = 3, ncol = 3, dimnames = names.list)

iter_mean_mat <- matrix(NA, nrow = 3, ncol = 3, dimnames = names.list)

rmse_sd_mat <- matrix(NA, nrow = 3, ncol = 3, dimnames = names.list)

spec_sd_mat <- matrix(NA, nrow = 3, ncol = 3, dimnames = names.list)

iter_sd_mat <- matrix(NA, nrow = 3, ncol = 3, dimnames = names.list)

# iterate over angles
for(b in 1:3){

  # iterate over radii
  for(a in 1:3){

    rmse_mean_mat[b, a] <- mean(three_circles_results[a, b, 1, ])
    spec_mean_mat[b, a] <- mean(three_circles_results[a, b, 2, ])
```

```
    iter_mean_mat[b, a] <- mean(three_circles_results[a, b, 3, ])
    rmse_sd_mat[b, a] <- sd(three_circles_results[a, b, 1, ])
    spec_sd_mat[b, a] <- sd(three_circles_results[a, b, 2, ])
    iter_sd_mat[b, a] <- sd(three_circles_results[a, b, 3, ])

  }

}

round(rmse_mean_mat, digits = 2)

round(rmse_sd_mat, digits = 2)
```