Matrix Transformations and Decompositions

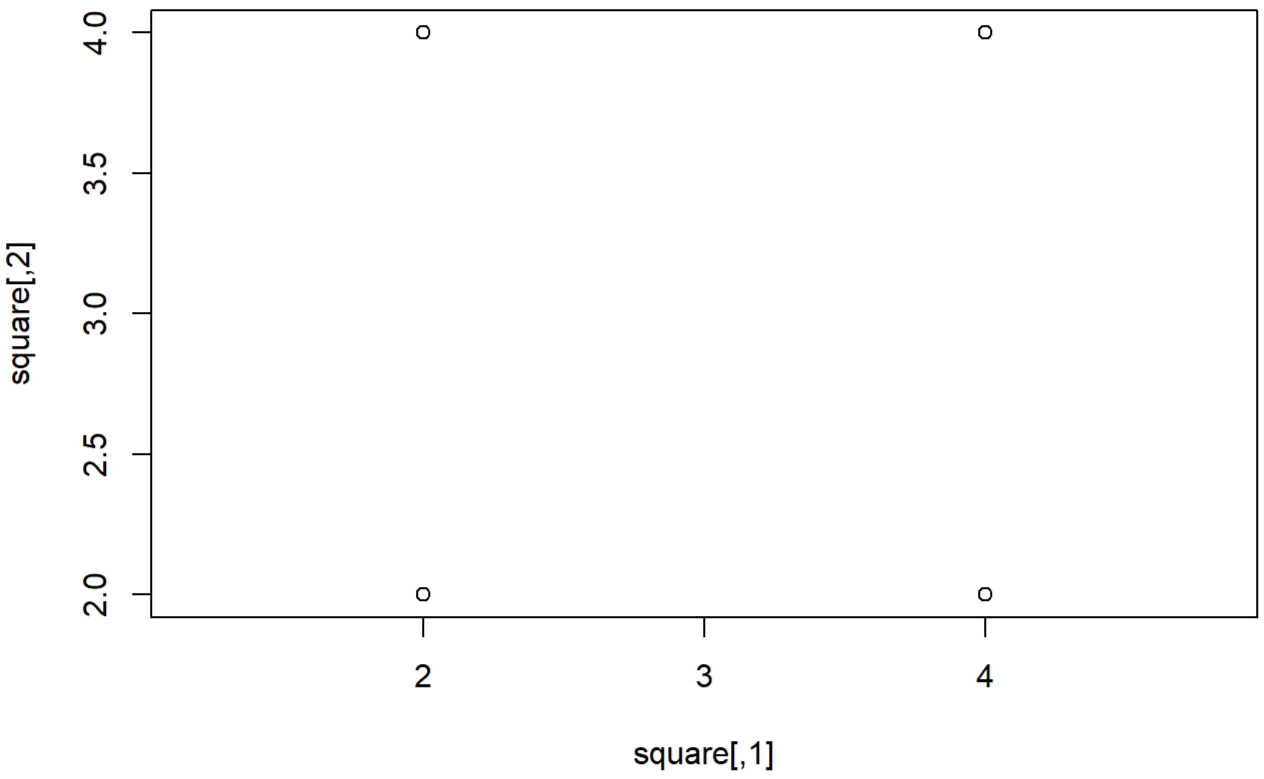
John Ferrara

2025-02-21

~~~
# Geometric Transformation of Shapes Using Matrix Multiplication

## Context
In computer graphics and data visualization, geometric transformations are fundamental. These
transformations, such as translation, scaling, rotation, and reflection, can be applied to sh
apes to manipulate their appearance.

## Task
Create a simple shape (like a square or triangle) using point plots in R. Implement R code to
apply different transformations (scaling, rotation, reflection) to the shape by left multiply
ing a transformation matrix by each of the point vectors. Demonstrate these transformations t
hrough animated plots.

### Create a Shape
Define a simple shape (e.g., a square) using a set of point coordinates.



``` r
## I intiially created this matrix of a square shape with 4 points.
square <- matrix(c(2,2,4,4,2,4,2,4), nrow=4)
print(square)
~~~

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

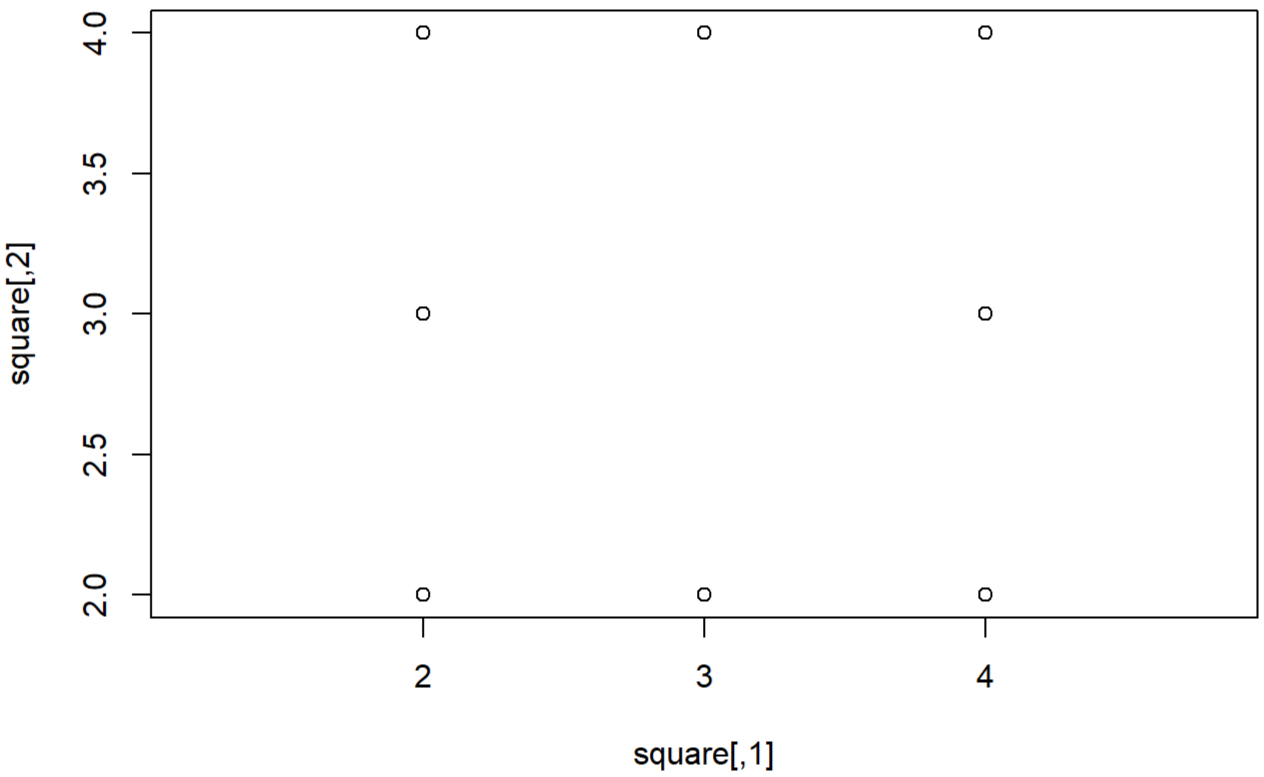
```
plot(square, asp=1)
```



```
## Building out to have more than the corners, so a slightly longer matrix.
square <- matrix(c(2,2,2,3,4,4,4,3,2,3,4,4,4,3,2,2), ncol = 2)
print(square)
```

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

```
plot(square, asp=1)
```



```
### ------------------------------------------- THIS MAY NOT BE NEEDED.
## However, the matrix for this 2-D square is not a square one. It is 2x8. i would need to ma
ke it square in order to use eigenvectors. Which I was unsure of how to do. However, after so
me digging it seems i can use a "covariance" matrix in order to capture the information betwe
en the points in my original matrix. The corvariance matrix will then be square. THis is what
i can use for eigenvalues and eigenvectors.

### Calculating the covariance matrix for my original shape.
### The formula for calculating the covariance matrix is: C = Mt * M

## First we transpose the matrix:

transposed_square = t(square)

## Then we use matrix multiplication:

covar_mtx <- transposed_square %*% square
print(covar_mtx)
```

```
##      [,1] [,2]
## [1,]   78   72
## [2,]   72   78
## Using 2/17 slide the

covar_egval <- eigen(covar_mtx)
covar_egval$values
```

```
## [1] 150   6
```

```
## Now we have a 2 x 2 matrix that is representative of the initial square shape matrix.
###
------------------------------------------------------------------------------------------
```

Apply Transformations

- **Scaling:** Enlarge or shrink the shape.

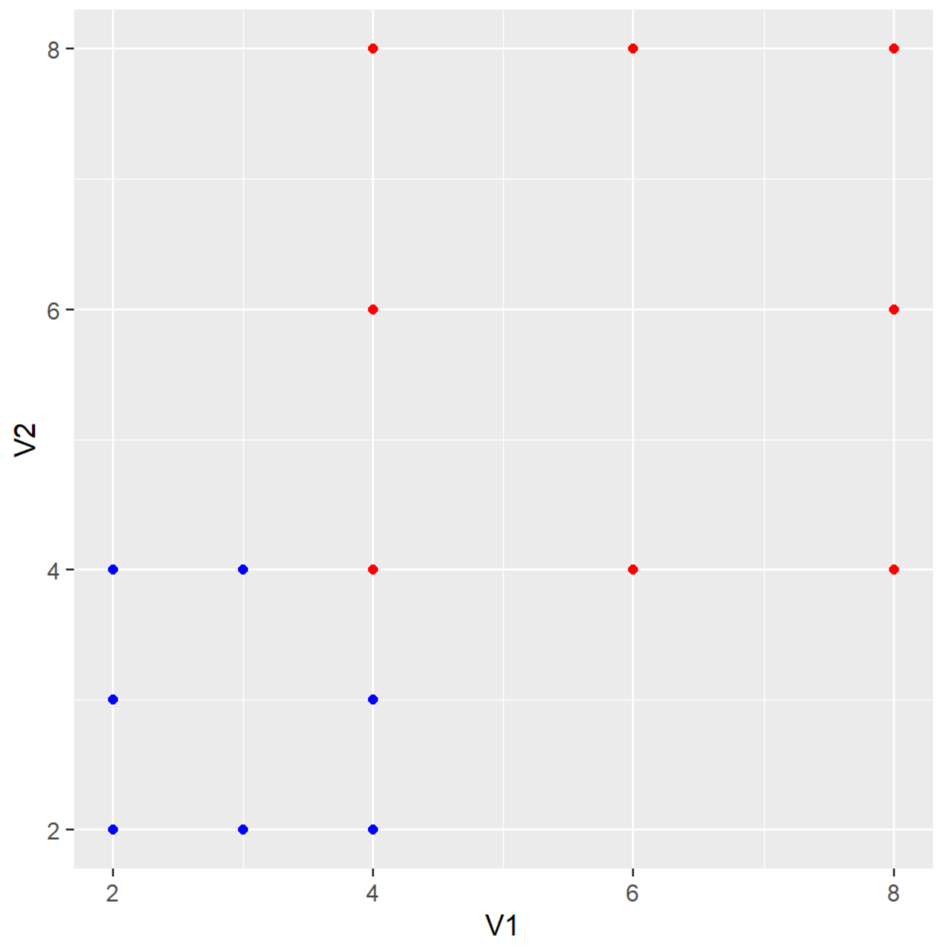
```
### Scaling the Shape Using Matrix Multplication

## Referencing the Slide from the 2/17 Class
scaling_matrix = matrix(c(2,0,0,2),ncol=2)
print(scaling_matrix)
```

```
##      [,1] [,2]
## [1,]    2    0
## [2,]    0    2
```

```
scaled_square <- square %*% scaling_matrix
print(scaled_square)
```

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

```
# Plot both matrices
ggplot() +
  geom_point(data=as.data.frame(square), aes(V1, V2), color="blue") +
  geom_point(data=as.data.frame(scaled_square), aes(V1, V2), color="red")+
  coord_fixed()
```



```
## The shape is 2x scaled, but also moved a bit to the right.
```

- **Rotation:** Rotate the shape by a given angle.

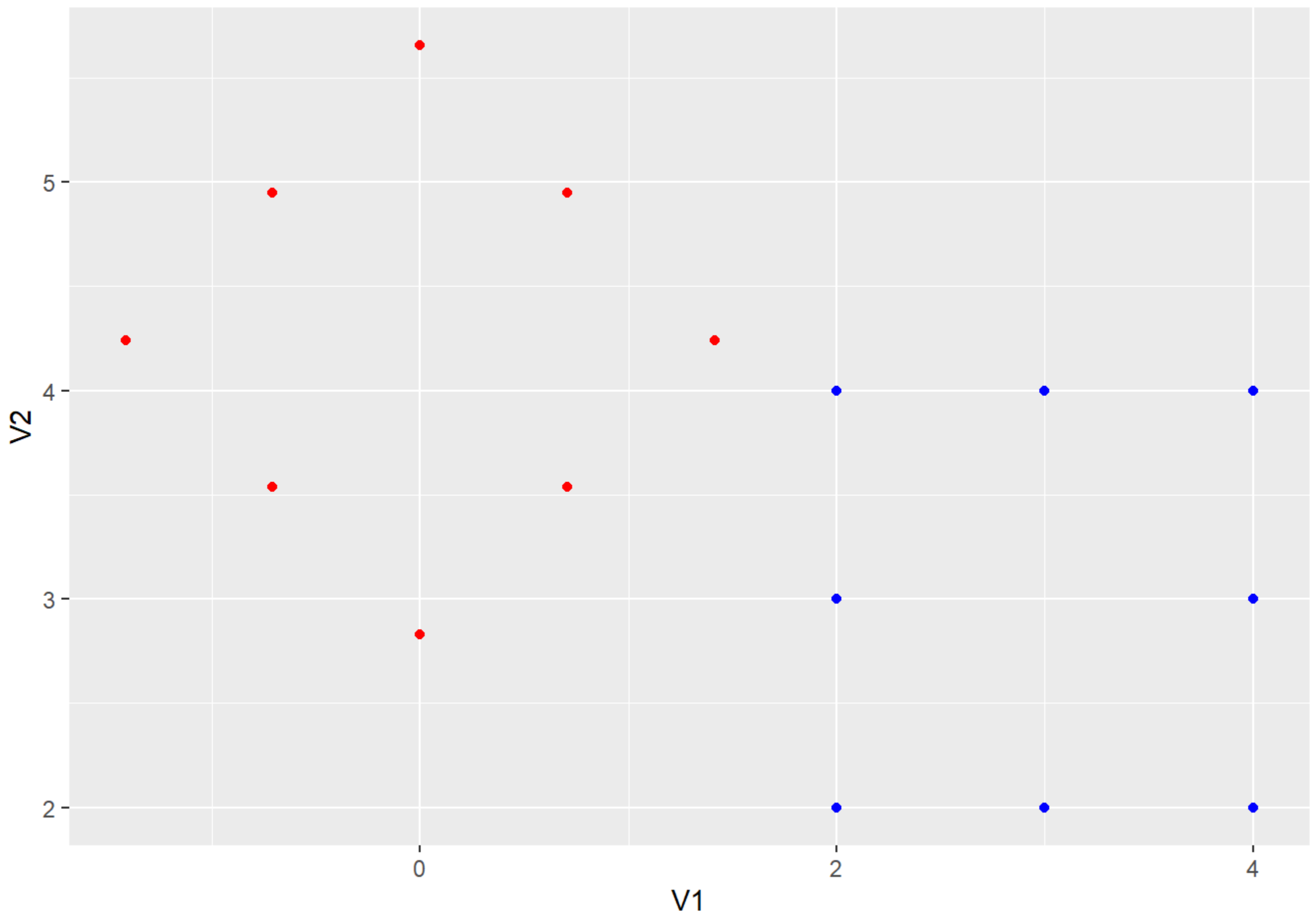
```
### Using the original square shape matrix for rotation.

## Referencing the slide from 2/17 the rotation matrix is cos and sin of theta.
## If we want to rotate the shape 45 degrees, we need to represent that via theta in radians
first.
## Formula for degress to radians is: radians = degrees*(pi/180)

degrees <- 45
radians_theta <- degrees * (pi/180)
print(radians_theta)
```

```
## [1] 0.7853982
```

```
rotation_matrix <- matrix(c(cos(radians_theta), sin(radians_theta), -sin(radians_theta), cos
(radians_theta)), nrow = 2, byrow = TRUE)

print(rotation_matrix)
```

```
##             [,1]      [,2]
## [1,]  0.7071068 0.7071068
## [2,] -0.7071068 0.7071068
```

```
## Using Matric multiplication to rotate original square matrix
rotated_square_45deg <- square%*%rotation_matrix

# Plot both matrices
ggplot() +
  geom_point(data=as.data.frame(square), aes(V1, V2), color="blue") +
  geom_point(data=as.data.frame(rotated_square_45deg), aes(V1, V2), color="red")+
  coord_fixed()
```



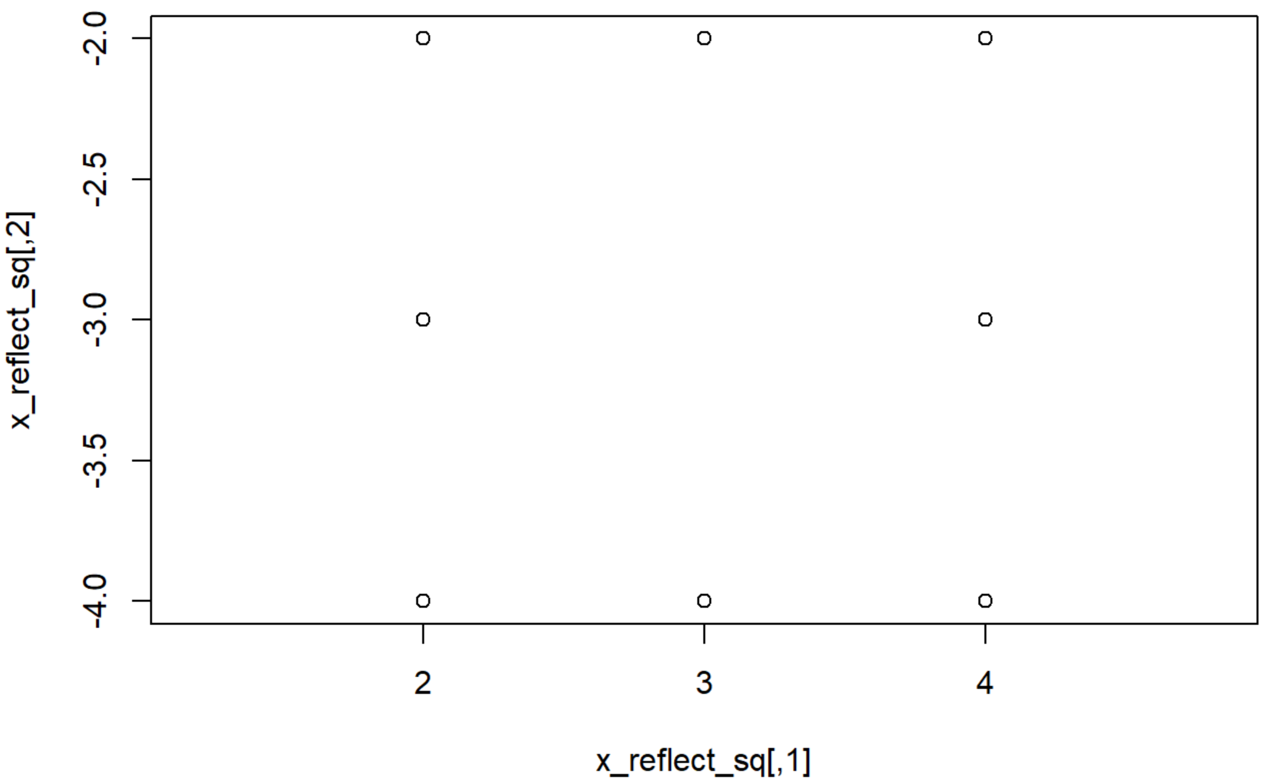
```
## The square was successfully rotated, but also seems to have shifted to the left a bit.
```

- **Reflection:** Reflect the shape across an axis.

```
## This was not in the 2/17 slide so referenced the internet for a "how to". In order to refl
ect a shape over an axis, it is multiplied by a modified version of the identity matrix with
one negative value depending on what axis is being reflected over.
# X Axis Reflection Matrix
x_axis_reflect <- matrix(c(1,0,0,-1),ncol=2)
print(x_axis_reflect)
```

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

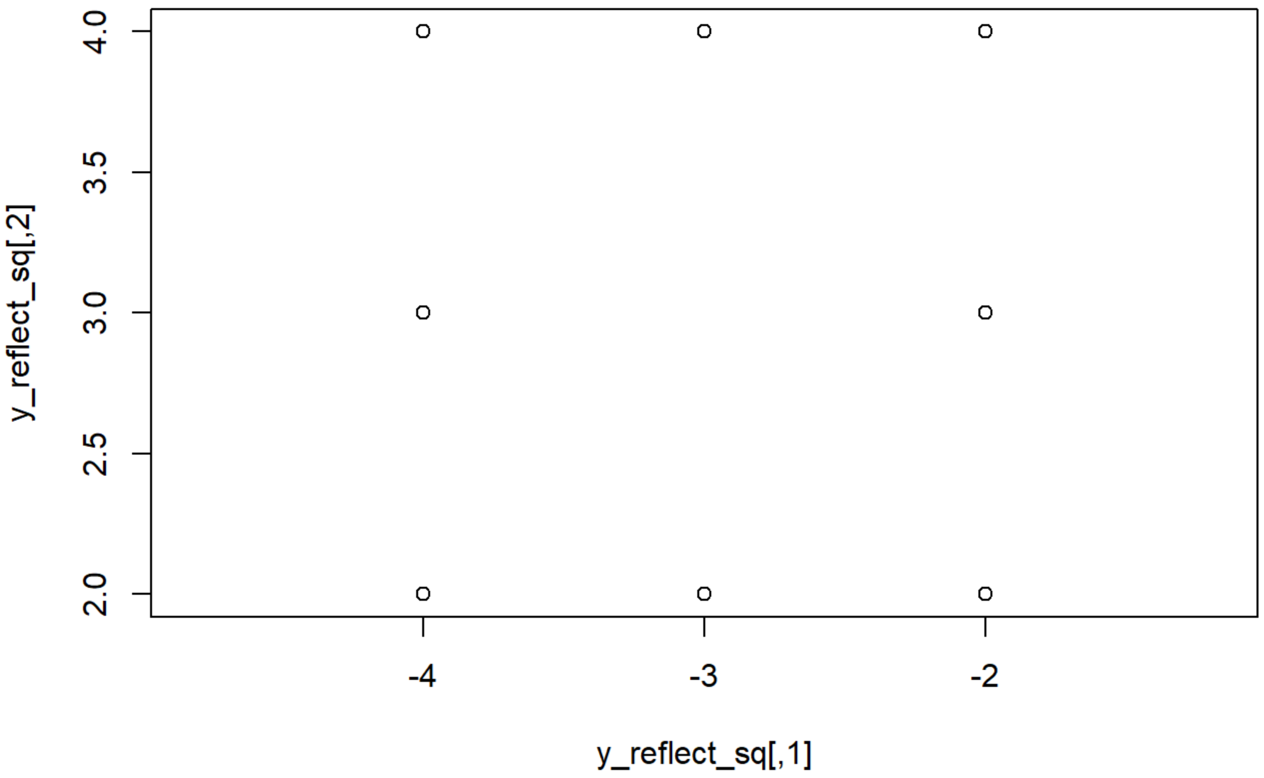
```
## Reflecting Rotated_sqare over the X axis
x_reflect_sq <- square %*%  x_axis_reflect
plot(x_reflect_sq, asp=1)
```



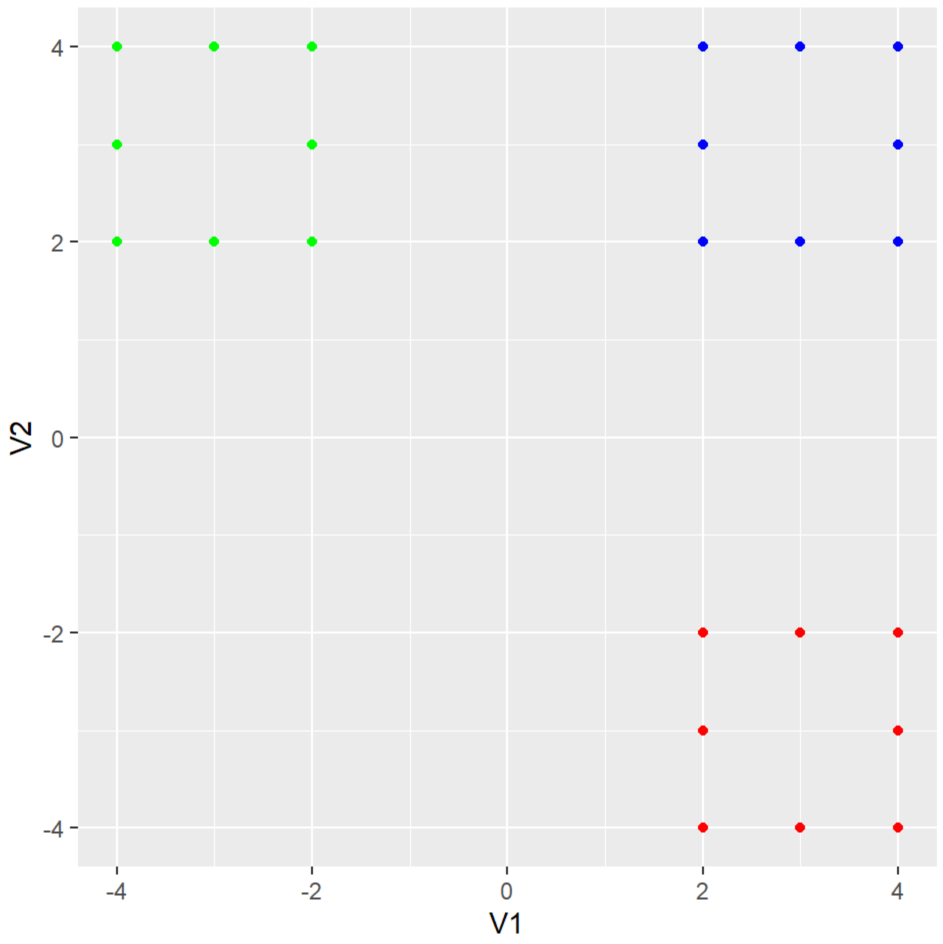
```
# y Axis Reflection Matrix
y_axis_reflect <- matrix(c(-1,0,0,1),ncol=2)
print(y_axis_reflect)
```

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

```
## Reflecting Rotated_sqare over the Y axis
y_reflect_sq <- square %*%  y_axis_reflect
plot(y_reflect_sq, asp=1)
```



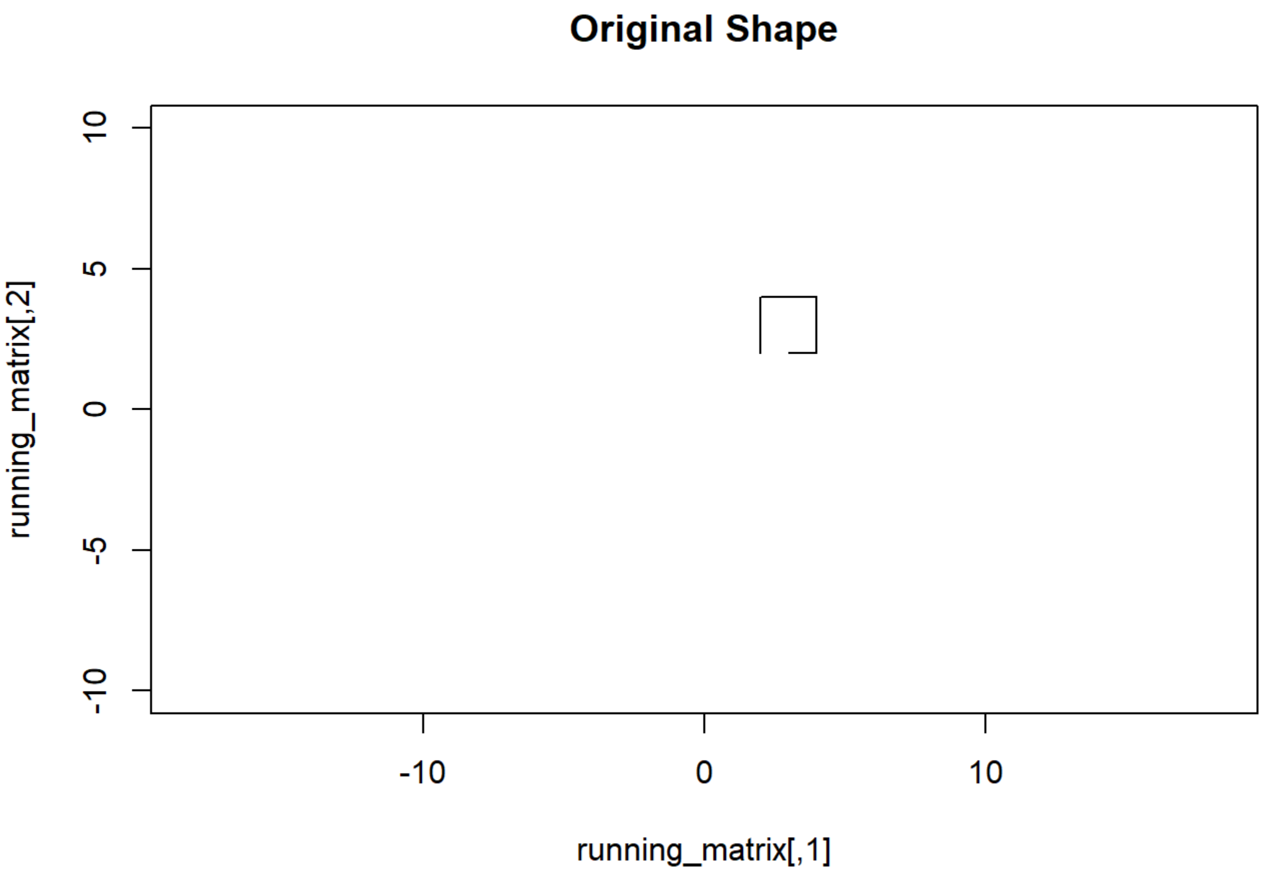
```
# Plot all matrices
ggplot() +
  geom_point(data=as.data.frame(square), aes(V1, V2), color="blue") +
  geom_point(data=as.data.frame(x_reflect_sq), aes(V1, V2), color="red")+
  geom_point(data=as.data.frame(y_reflect_sq), aes(V1, V2), color="green")+
  coord_fixed()
```



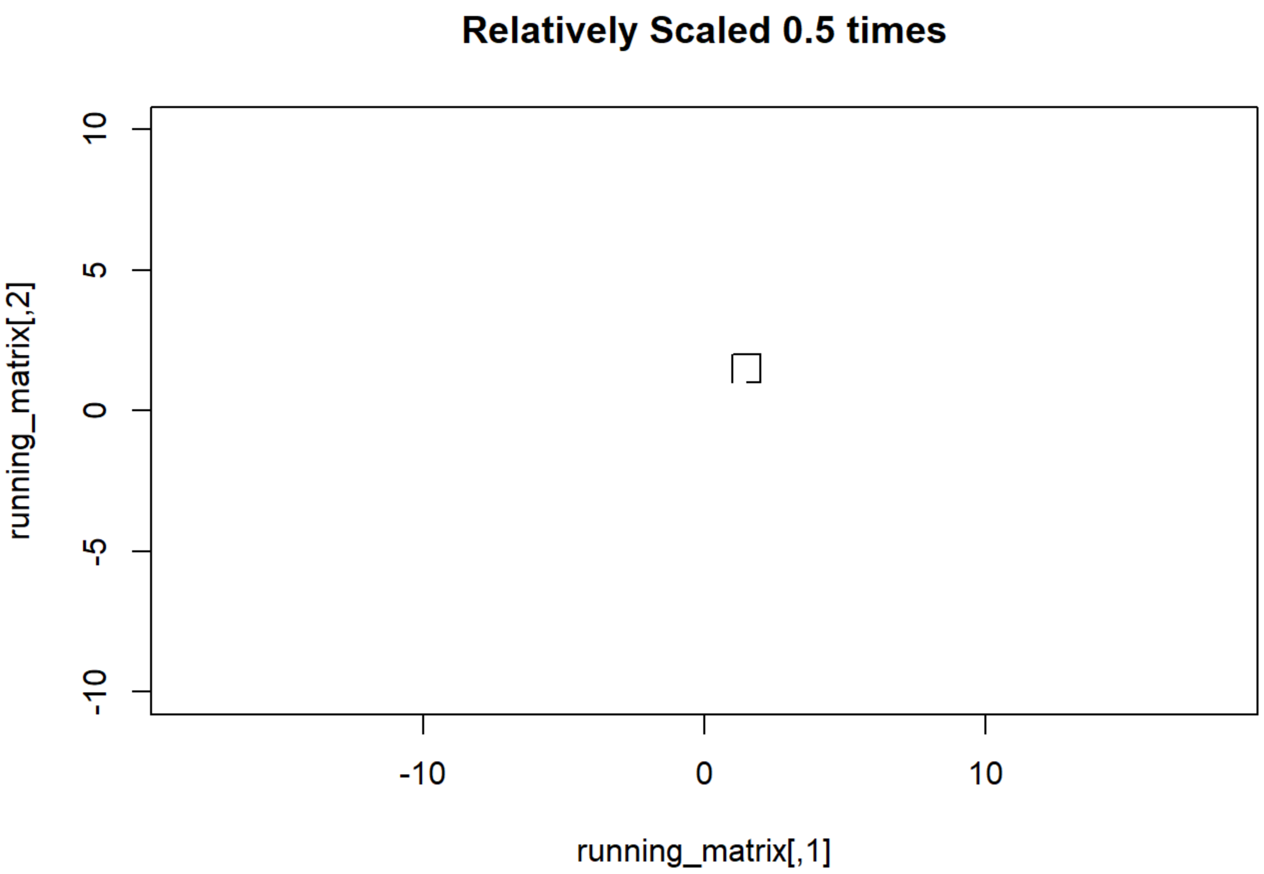
Animate Transformations

Use a loop to incrementally change the transformation matrix and visualize the effect on the shape over time.

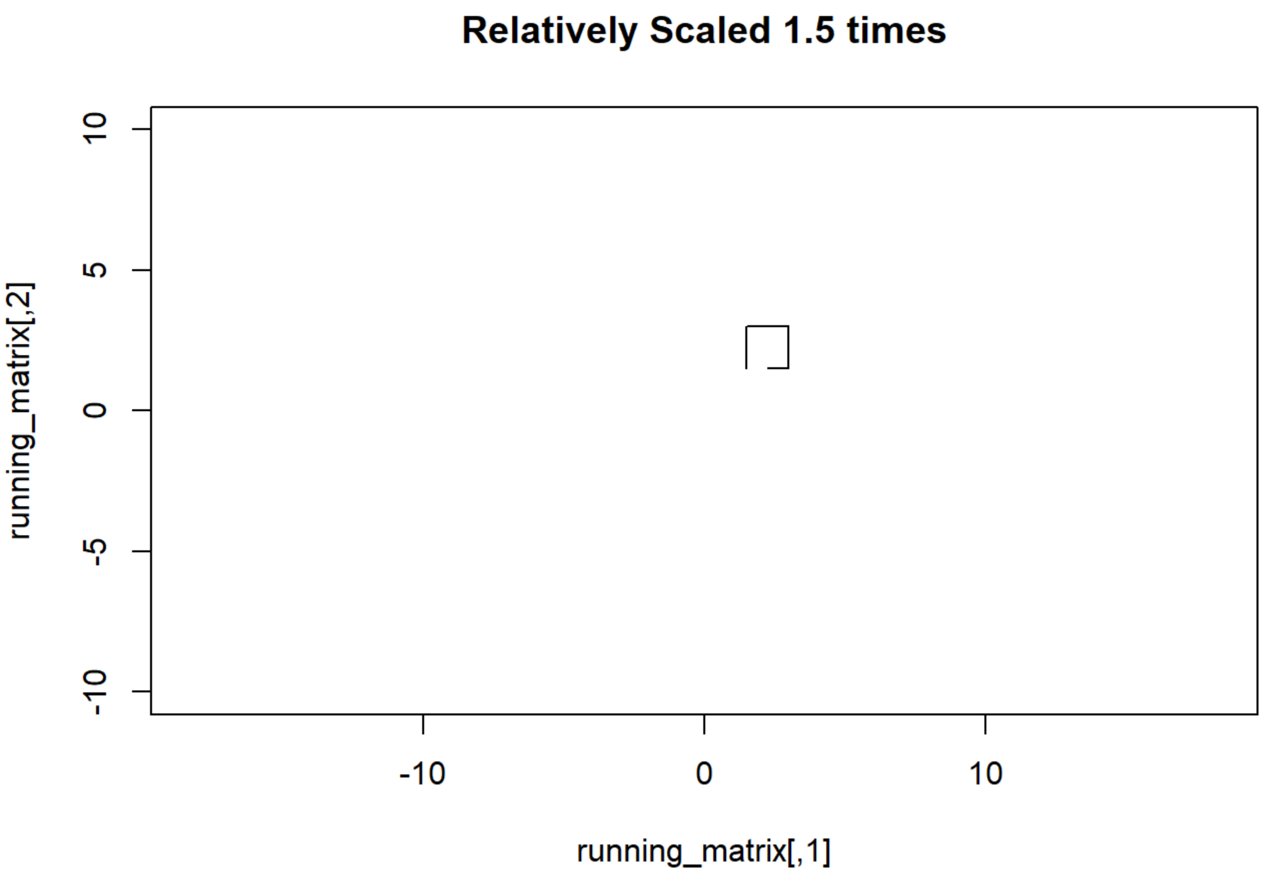
```
## Making a vector to loop through transformation types
looping <- c("original","scale","rotate","reflect")

## used this to help: https://www.youtube.com/watch?v=LpICOCbmxks

### Presetting the axis ranges for animation
lower <- -10
upper <- 10


for(l in looping){
  if(l == "original"){
    running_matrix <-square
    print(plot(running_matrix, type='l',asp=1,xlim=c(lower,upper), ylim=c(lower,upper),main
="Original Shape"))
    Sys.sleep(1)
  } else if(l =="scale"){
    print("scaled plot")
    for (scl in c(0.5,1.5,2)){
      scaling_matrix = matrix(c(scl,0,0,scl),ncol=2)
      running_matrix<- running_matrix %*% scaling_matrix
      print(plot(running_matrix,type='l', asp=1,xlim=c(lower,upper), ylim=c(lower,upper),main
=paste("Relatively Scaled",scl,"times")))
      Sys.sleep(1)
    }
  } else if(l=="rotate"){
    for (rot in c(45,90,135,180)){
      radians_theta <- rot * (pi/180)
      rotation_matrix <- matrix(c(cos(radians_theta), sin(radians_theta), -sin(radians_thet
a), cos(radians_theta)), nrow = 2, byrow = TRUE)
      running_matrix <- running_matrix%*%rotation_matrix
      print(plot(running_matrix,type='l', asp=1,xlim=c(lower,upper), ylim=c(lower,upper),main
=paste("Relatively Rotated",rot,"degrees")))
      Sys.sleep(1)
      }
  } else {
    for (ref in c("x","y")){
      if (ref == "x") {
        x_axis_reflect <- matrix(c(1,0,0,-1),ncol=2)
        running_matrix <- running_matrix%*%x_axis_reflect
        print(plot(running_matrix, type='l',asp=1,xlim=c(lower,upper), ylim=c(lower,upper),ma
in="Reflected over X axis"))
        Sys.sleep(1)
      }
      else{
        y_axis_reflect <- matrix(c(-1,0,0,1),ncol=2)
        running_matrix <- running_matrix%*%y_axis_reflect
        print(plot(running_matrix,type='l', asp=1, xlim=c(lower,upper), ylim=c(lower,upper),m
ain="Reflected over Y axis"))
        Sys.sleep(1)
        }}
  }
}
```



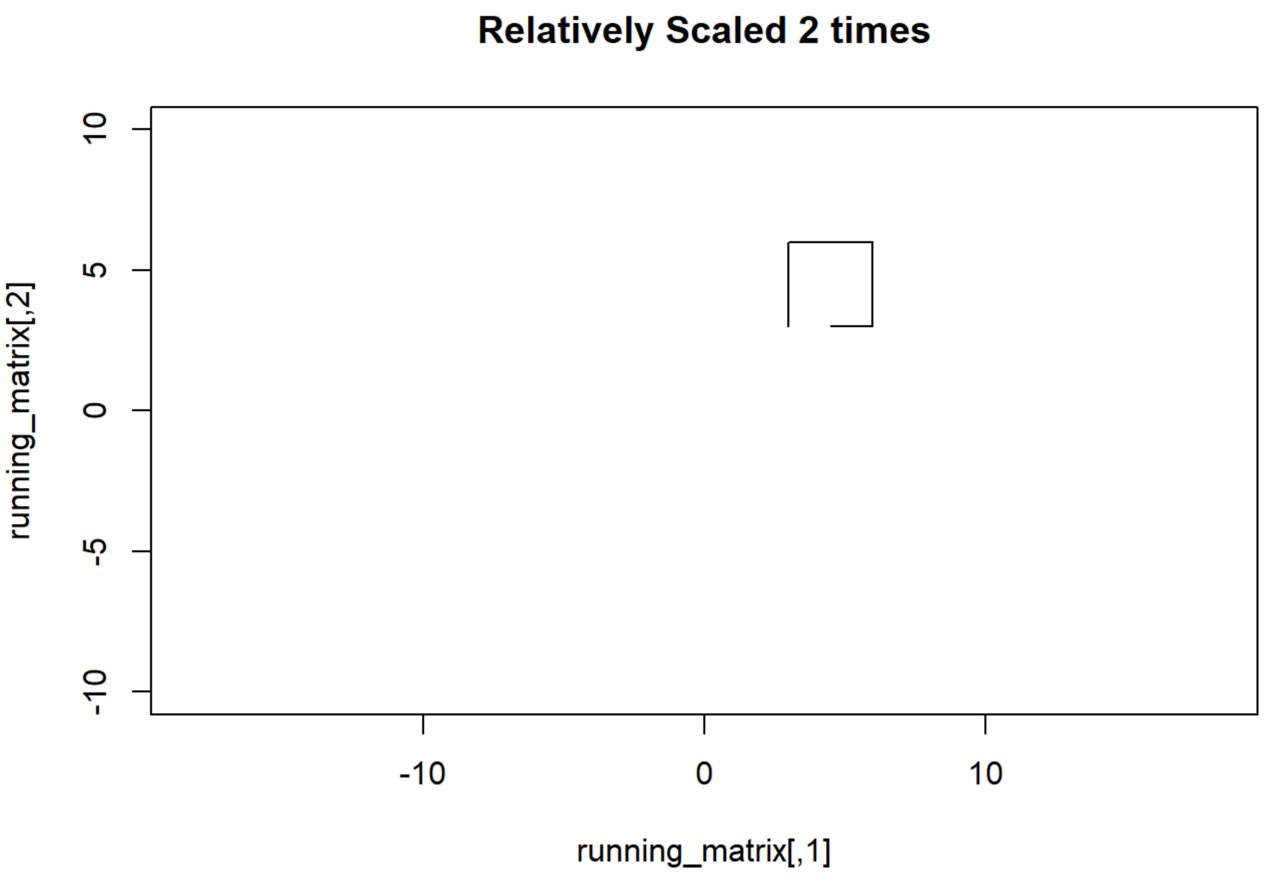
```
## NULL
## [1] "scaled plot"
```


Relatively Scaled 0.5 times
running_matrix[,2]
running_matrix[,1]
10
5
0
-5
-10
-10
0
10

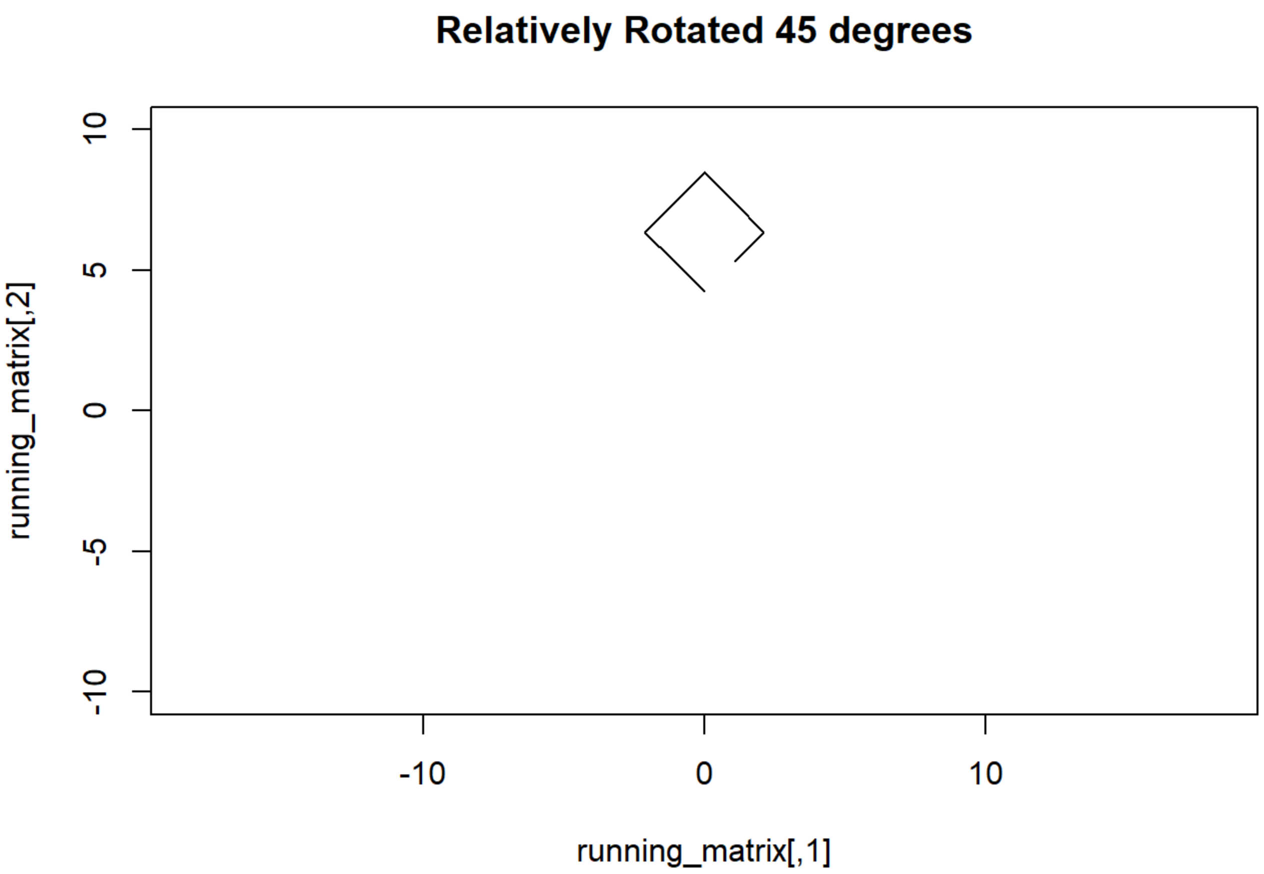
```
## NULL
```


Relatively Scaled 1.5 times
running_matrix[,2]
running_matrix[,1]
10
5
0
-5
-10
-10
0
10

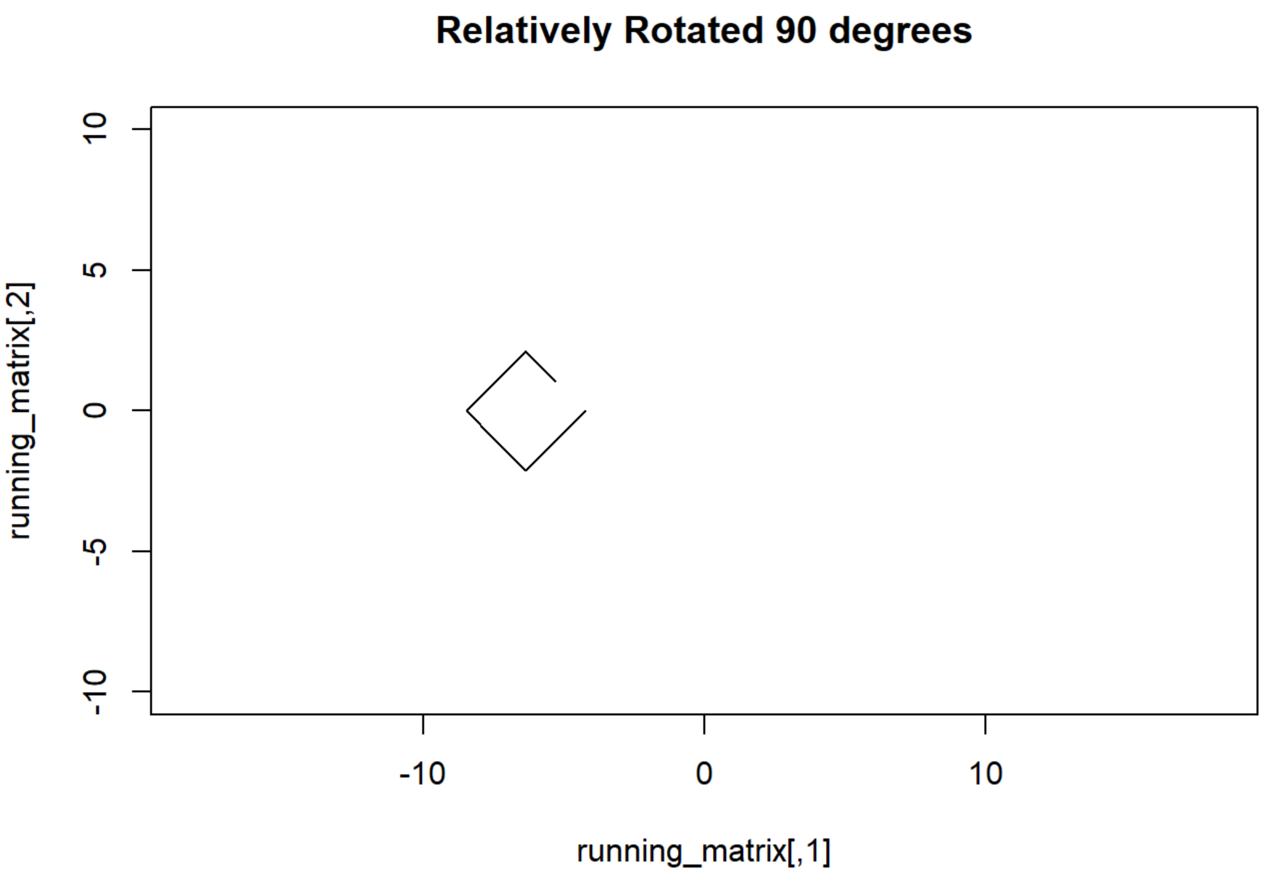
```
## NULL
```


Relatively Scaled 2 times
running_matrix[,2]
running_matrix[,1]
10
5
0
-5
-10
-10
0
10

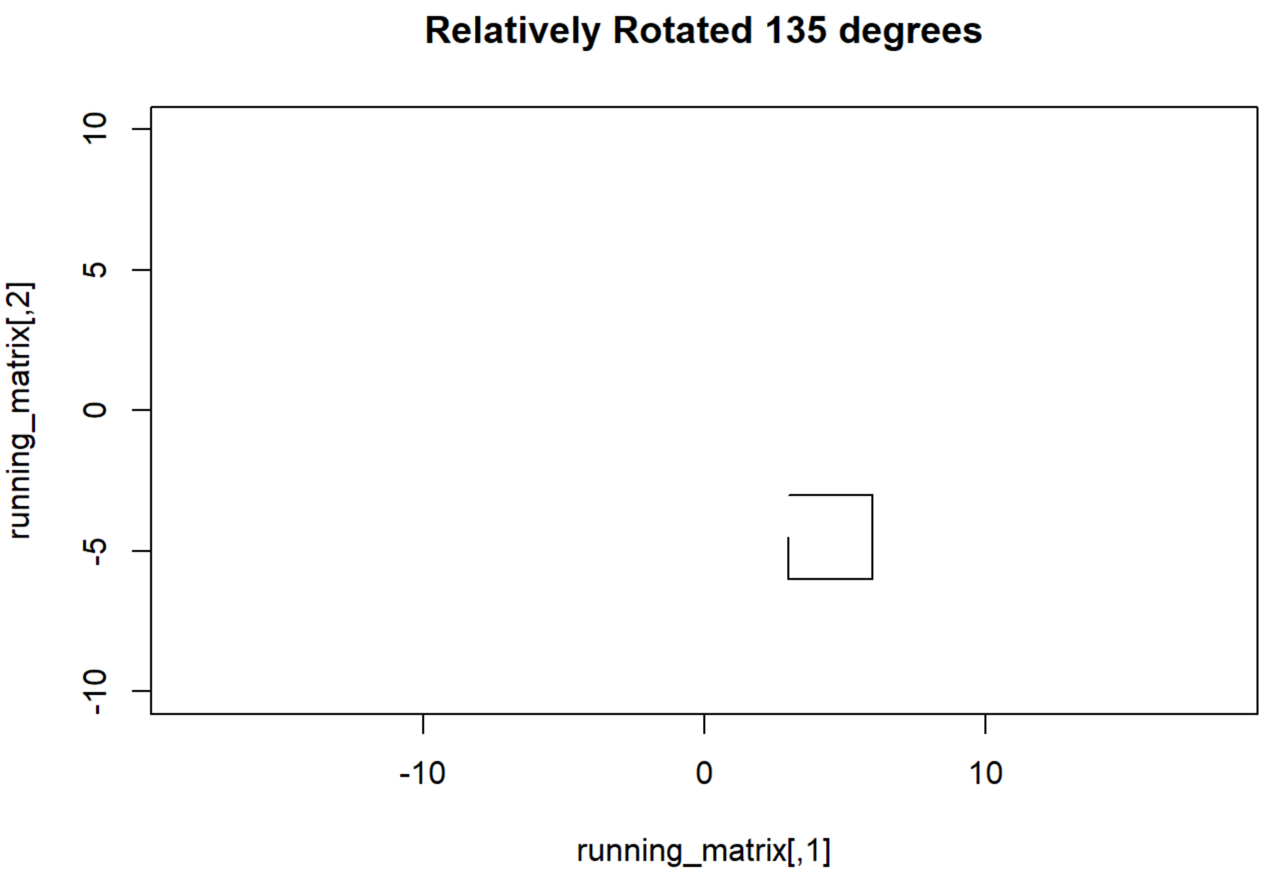
```
## NULL
```


Relatively Rotated 45 degrees
running_matrix[,2]
running_matrix[,1]
-10
0
10
-10
-5
0
5
10

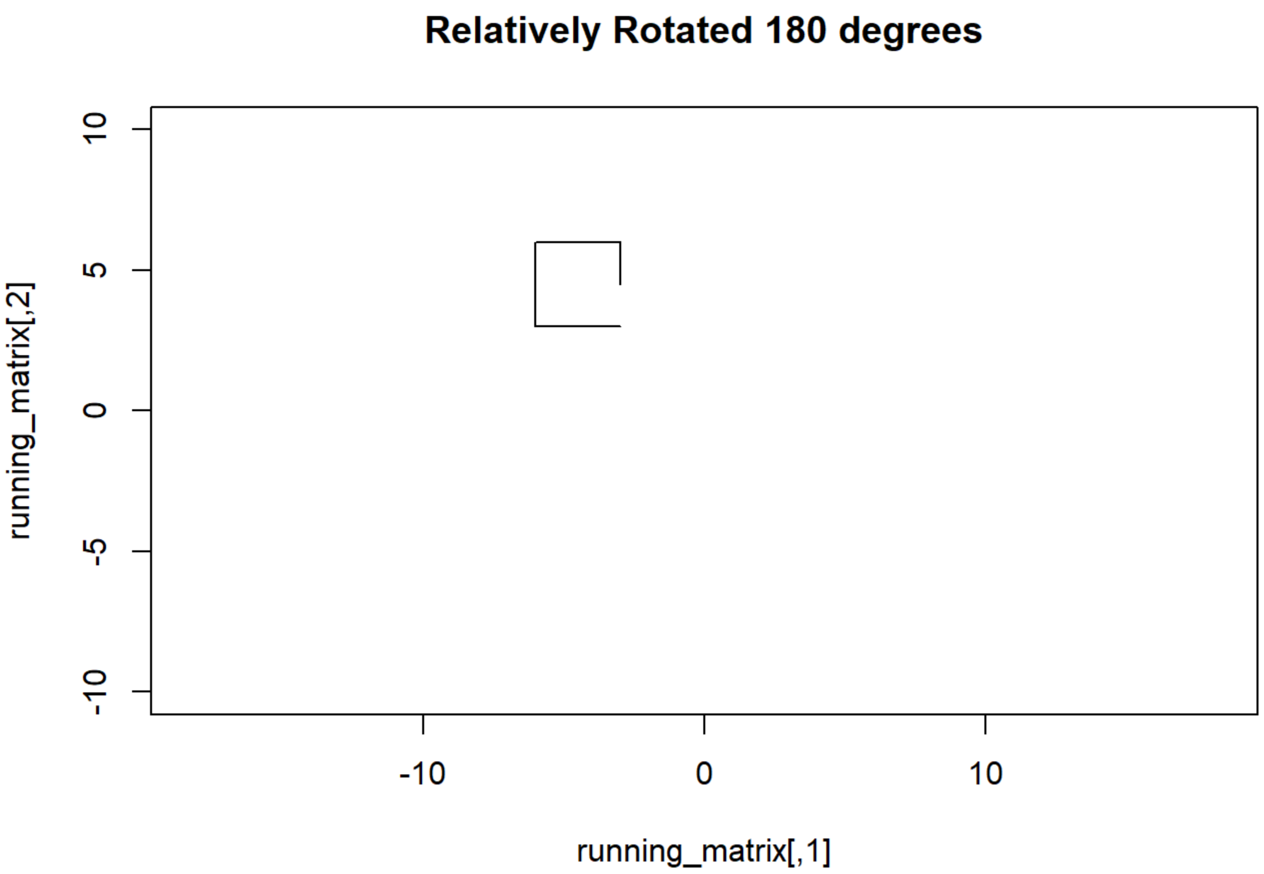
```
## NULL
```


Relatively Rotated 90 degrees
running_matrix[,2]
running_matrix[,1]
10
5
0
-5
-10
-10
0
10

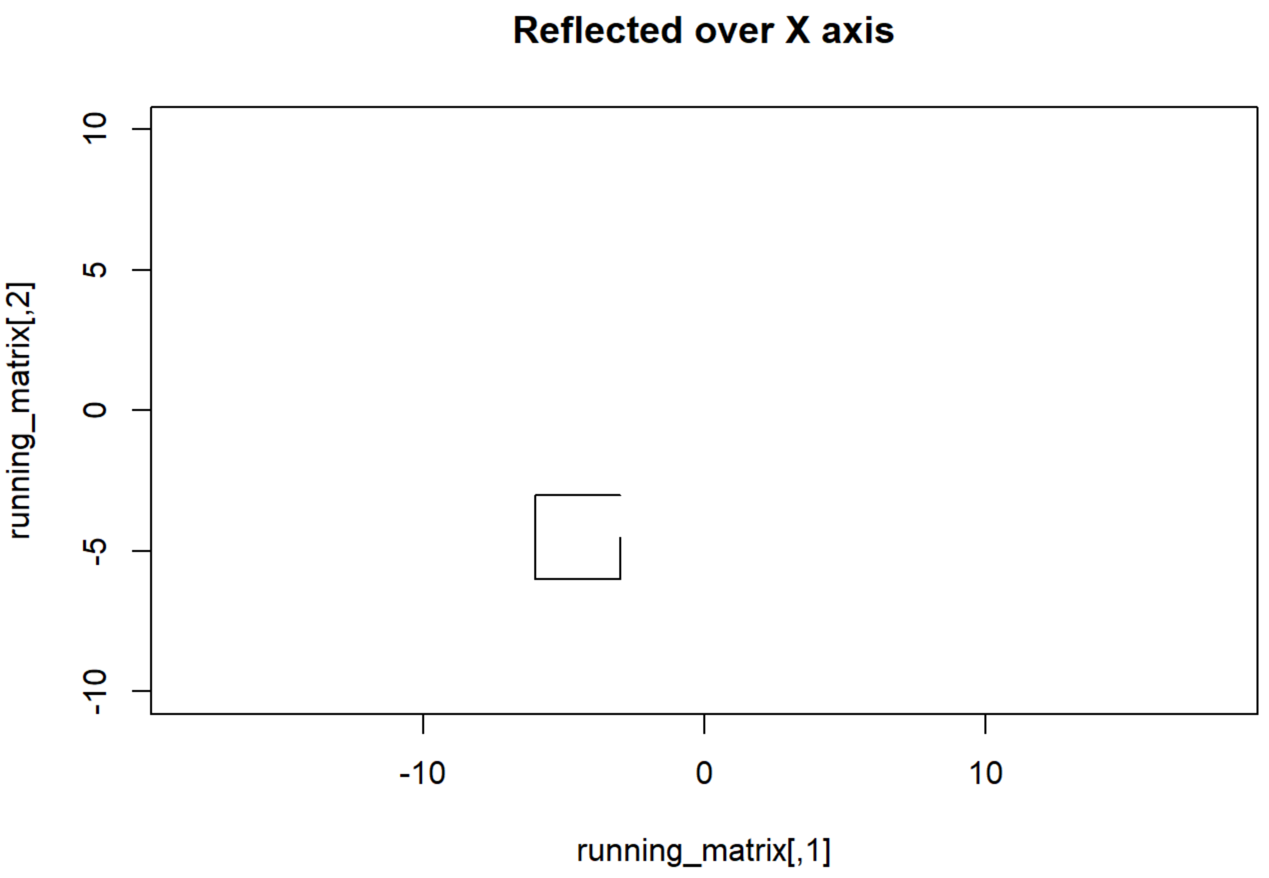
```
## NULL
```


Relatively Rotated 135 degrees
running_matrix[,2]
running_matrix[,1]
10
5
0
-5
-10
-10
0
10

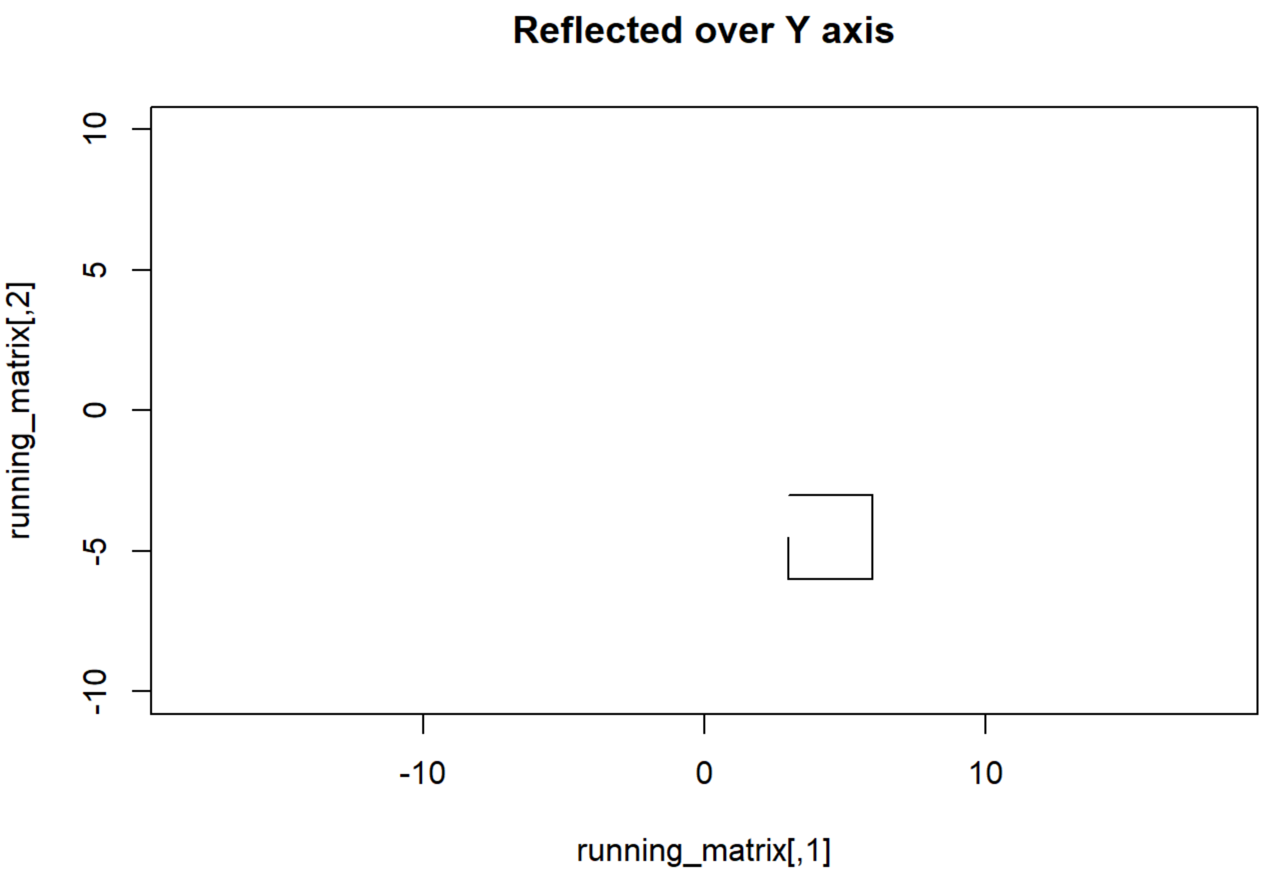
```
## NULL
```


Relatively Rotated 180 degrees
running_matrix[,2]
running_matrix[,1]
10
5
0
-5
-10
-10
0
10

```
## NULL
```


Reflected over X axis
running_matrix[,1]
running_matrix[,2]
-10
0
10
-10
-5
0
5
10

```
## NULL
```



```
## NULL
```

Plot

Display the original shape and its transformations in your compiled PDF. Demonstrate the effect of the transformations with fixed images.

Matrix Properties and Decomposition

Proofs

Non-Commutativity of Matrix Multiplication

Prove that $AB \neq BA$ in general.

```
## Need to prove order matters in matrix multiplication

a <- matrix(c(1,2,3,4),ncol=2)
print(a)
```

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

```
b <- matrix(c(2,2,2,2),ncol=2)
print(b)
```

```
##      [,1] [,2]
## [1,]    2    2
## [2,]    2    2
```

```
## A*B
a_b <- a %*% b
print(a_b)
```

```
##      [,1] [,2]
## [1,]    8    8
## [2,]   12   12
```

```
## B*A
b_a <- b %*% a
print(b_a)
```

```
##      [,1] [,2]
## [1,]    6   14
## [2,]    6   14
```

```
## You can see by the output from these two matrices being multiplied by each other that the
products are different depending on the order.
```

Symmetric Property

Prove that $A^T A$ is always symmetric.

```
## Running a 1000 different simulations of randomly generating matrices of different sizes.

iterate <- seq(1, 1000, by=1)
running_symmetry = c()
for (i in iterate) {
## For the sake of scaling computations limiting to 4 through 50 for vec limits
  mat_dim <- sample(4:50, 1,replace=TRUE)
  random_vec <- sample(1:50, mat_dim, replace=TRUE)
## Det. the number of cols based on random vec length. (limit 2 through 5 for matrix size b/c
computation time). Using that to determine rows.
  n_col_input <- sample(2:5, 1, replace=TRUE)
## Ensures an integer that is a multiple of the column number
  n_row_input <- ceiling(mat_dim / n_col_input) * n_col_input
  temp_matrix <- matrix(random_vec,nrow=n_row_input, ncol=n_col_input)
  trans_temp_matrix <- t(temp_matrix)
  product_matrix <- trans_temp_matrix %*% temp_matrix
  result <- isSymmetric(product_matrix)
  running_symmetry <- c(running_symmetry,result)
}

print(unique(running_symmetry))
```

```
## [1] TRUE
```

```
### This loop generates 1000 different random matrices (within certain size boundries) and no
tes if the matrix and its transpose product is symmetrical. If there was an instance where th
is was false, it would be found in the print statement showing the distinct values found for
the scenarios. However, only True is in the vector.
```

Determinant Property

Prove that the determinant of $A^T A$ is non-negative.

```
### I will be doing the same type of thing as i did for the symmetric "proof"

## Running a 1000 different simulations of randomly generating matrices of different sizes.
iterate <- seq(1, 1000, by=1)
running_det = c()
for (i in iterate) {
## For the sake of scaling computations limiting to 4 through 50 for vec limits
  mat_dim <- sample(4:50, 1,replace=TRUE)
  random_vec <- sample(1:50, mat_dim, replace=TRUE)
## Det, the number of cols based on random vec length. (limit 2 through 5 for matrix size b/c
computation time). Using that to determine rows.
  n_col_input <- sample(2:5, 1, replace=TRUE)
  n_row_input <- ceiling(mat_dim / n_col_input) * n_col_input
  temp_matrix <- matrix(random_vec,nrow=n_row_input, ncol=n_col_input)
  trans_temp_matrix <- t(temp_matrix)
  product_matrix <- trans_temp_matrix %*% temp_matrix
  result <- det(product_matrix)
  running_det<- c(running_det,result>=0)
}

### Printing unique values
print(unique(running_det))
```

```
## [1] TRUE
```

```
### This code runs 1000 different simulations, simlar to the code previous, but this time cal
culated the determinant of the product of transpose a * a. The values for the determinant are
determined to be greater than or equal to zero. The boolean result is appened to a list. Once
complete the unique values are printed, and should only show True to demonstrate the non-nega
tive results.
```

Singular Value Decomposition (SVD) and Image Compression

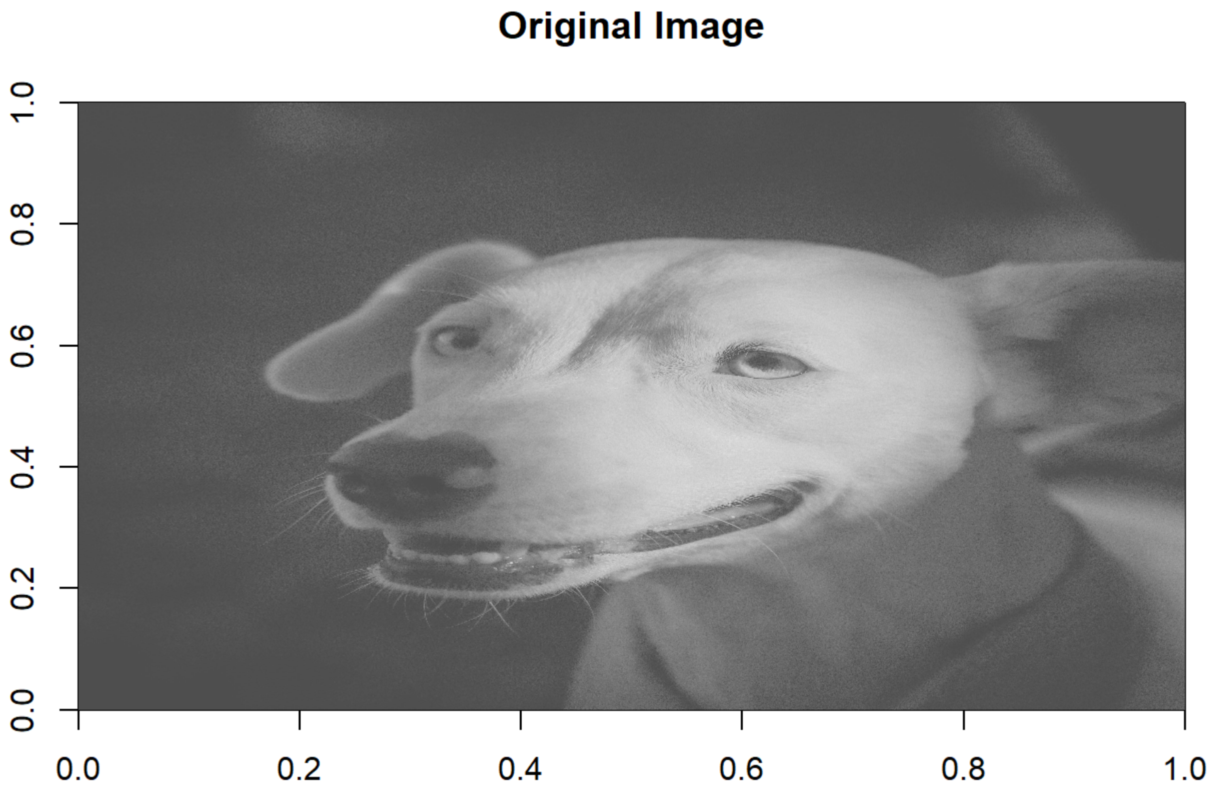
Context

SVD is a key technique used in image compression, allowing images to be stored efficiently while maintaining quality.

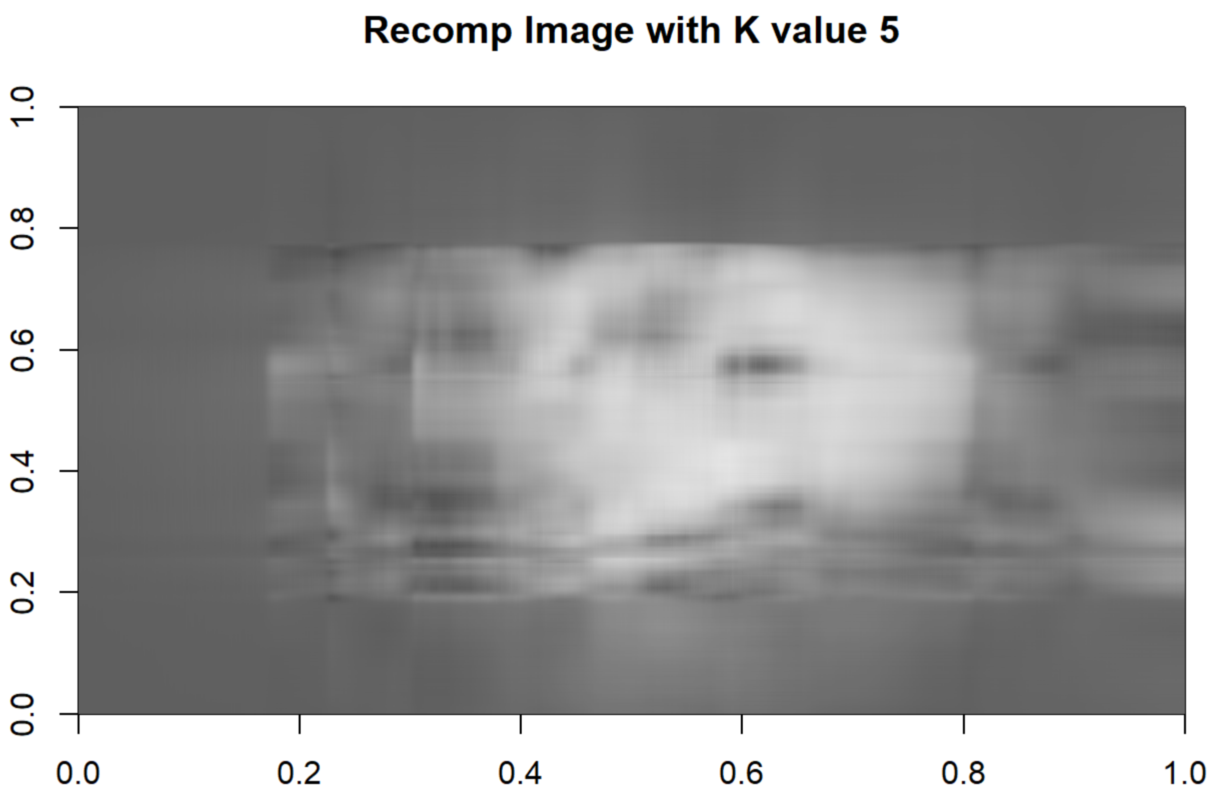
Instructions

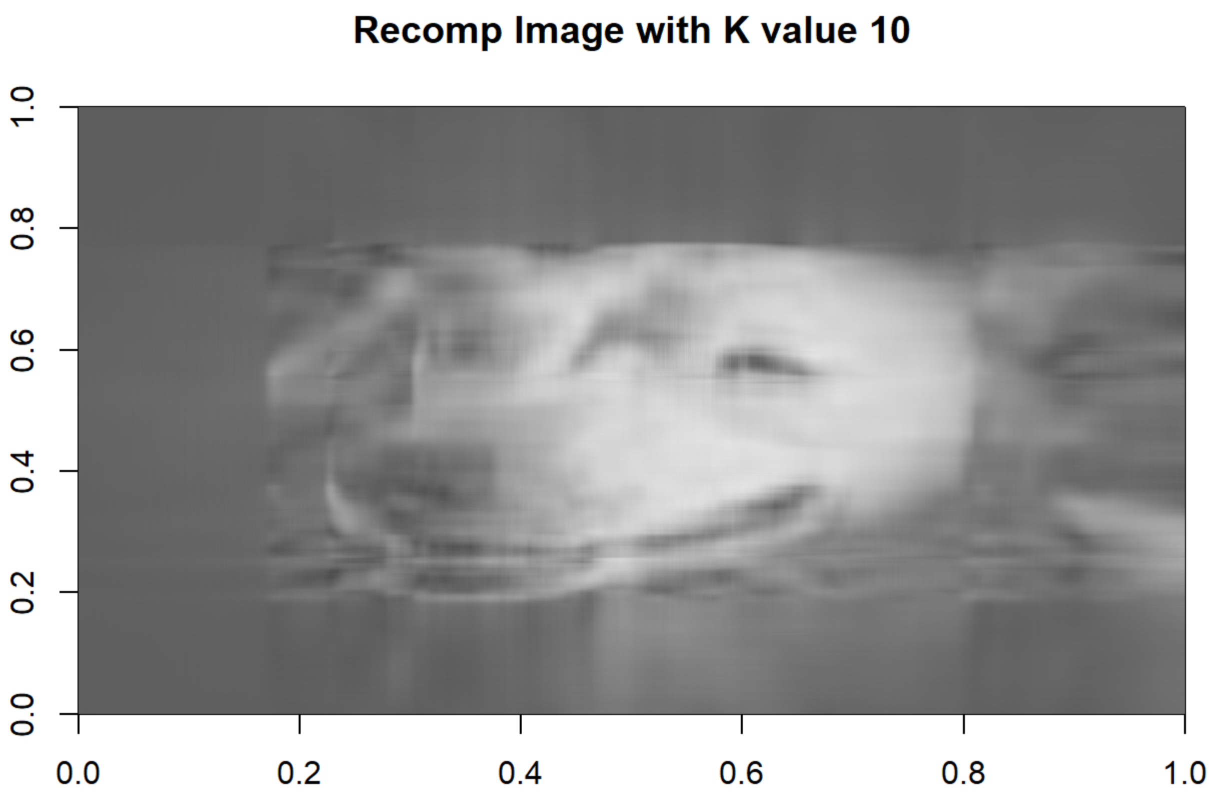
- **Read an Image:** Convert a grayscale image into a matrix.
- **Perform SVD:** Factorize the image matrix A into $U\Sigma V^T$ using Rs built-in `svd()` function.
- **Compress the Image:** Reconstruct the image using only the top k singular values and vectors.
- **Visualize the Result:** Plot the original image alongside the compressed versions for various values of k (e.g., $k = 5, 20, 50$).

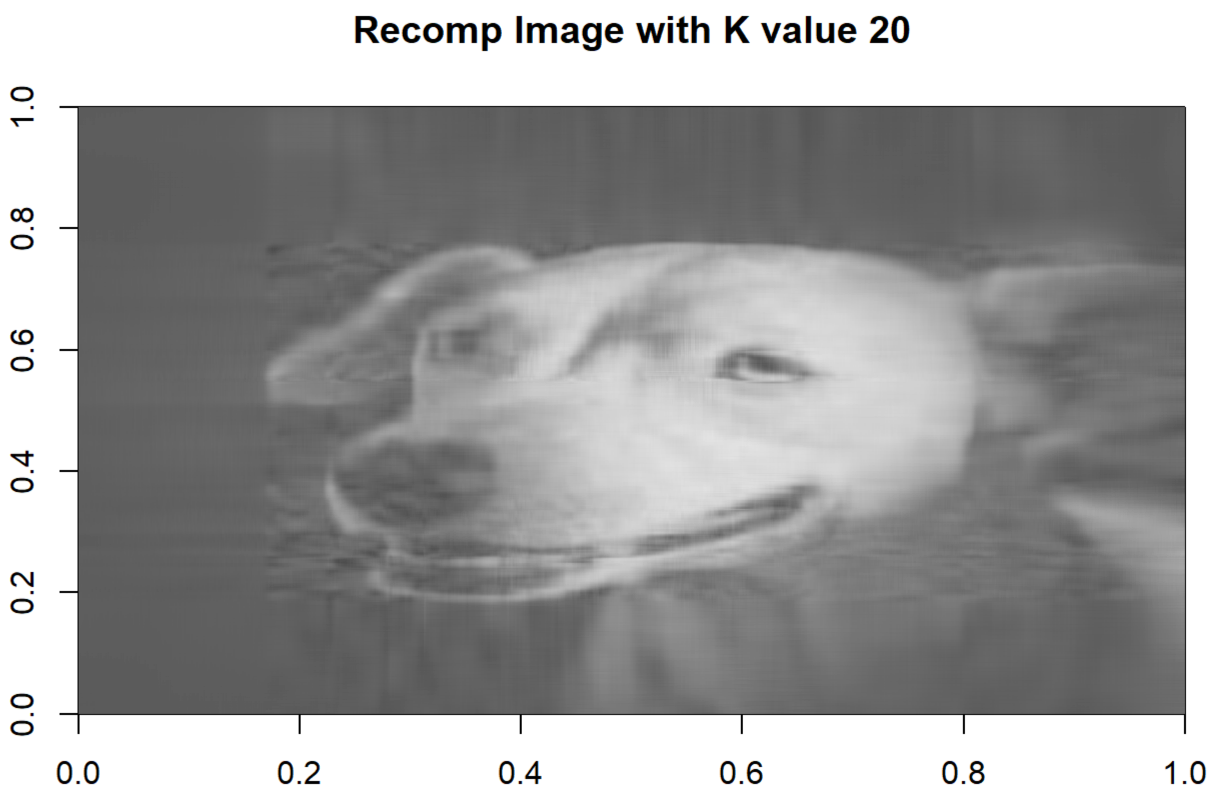
```r
## install.packages("jpeg")

library(jpeg)

## For reproducibility Downloading and saving image locally
image_url <- "https://github.com/jhnboyy/CUNY_SPS_WORK/raw/main/Spring2025/DATA605/Homework/H
omework_1/grayscale_dog.jpeg"
file <- "grayscale_dog.jpeg"
download.file(image_url, destfile = file, mode = "wb")
image_array <- readJPEG("grayscale_dog.jpeg")


gray_matrix <- image_array[,,1]


## Convert to grayscale
svd_result <- svd(gray_matrix)

## Firstly, i want to plot the original image.
image(t(gray_matrix)[, nrow(gray_matrix):1], col=gray.colors(256), main="Original Image")
```

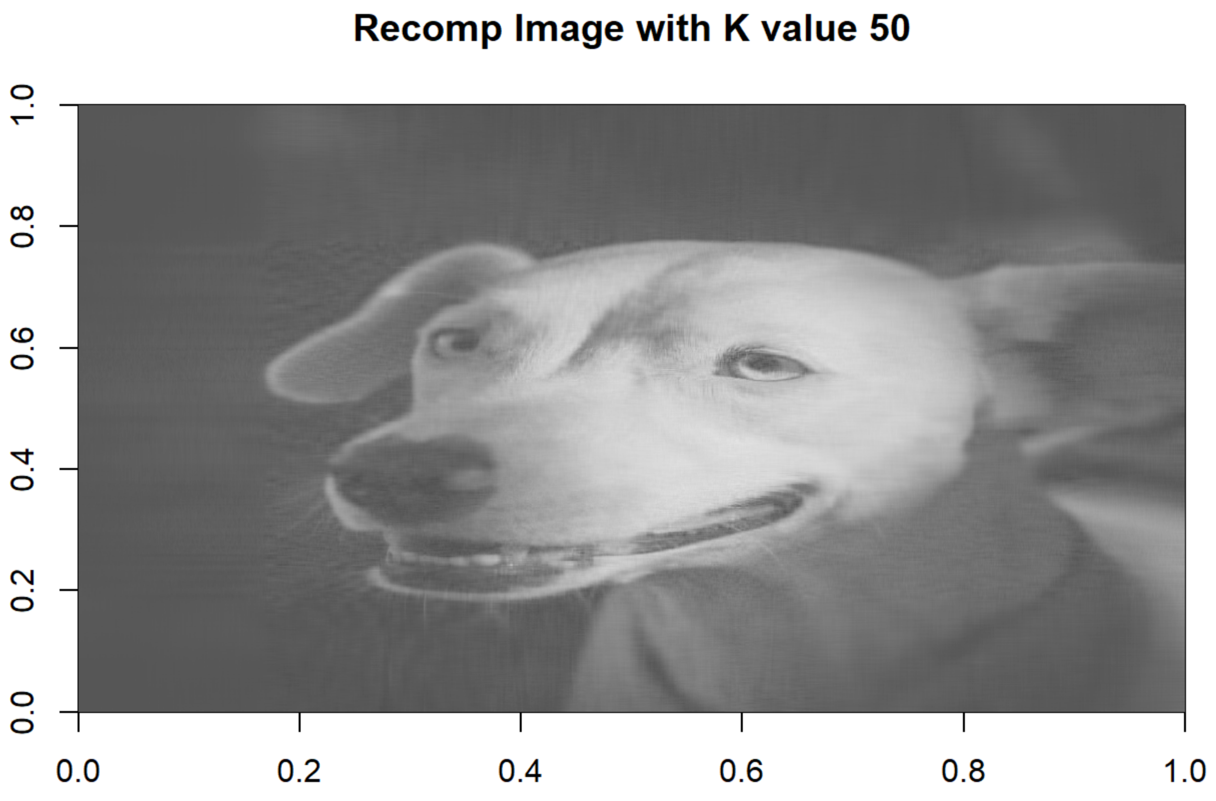


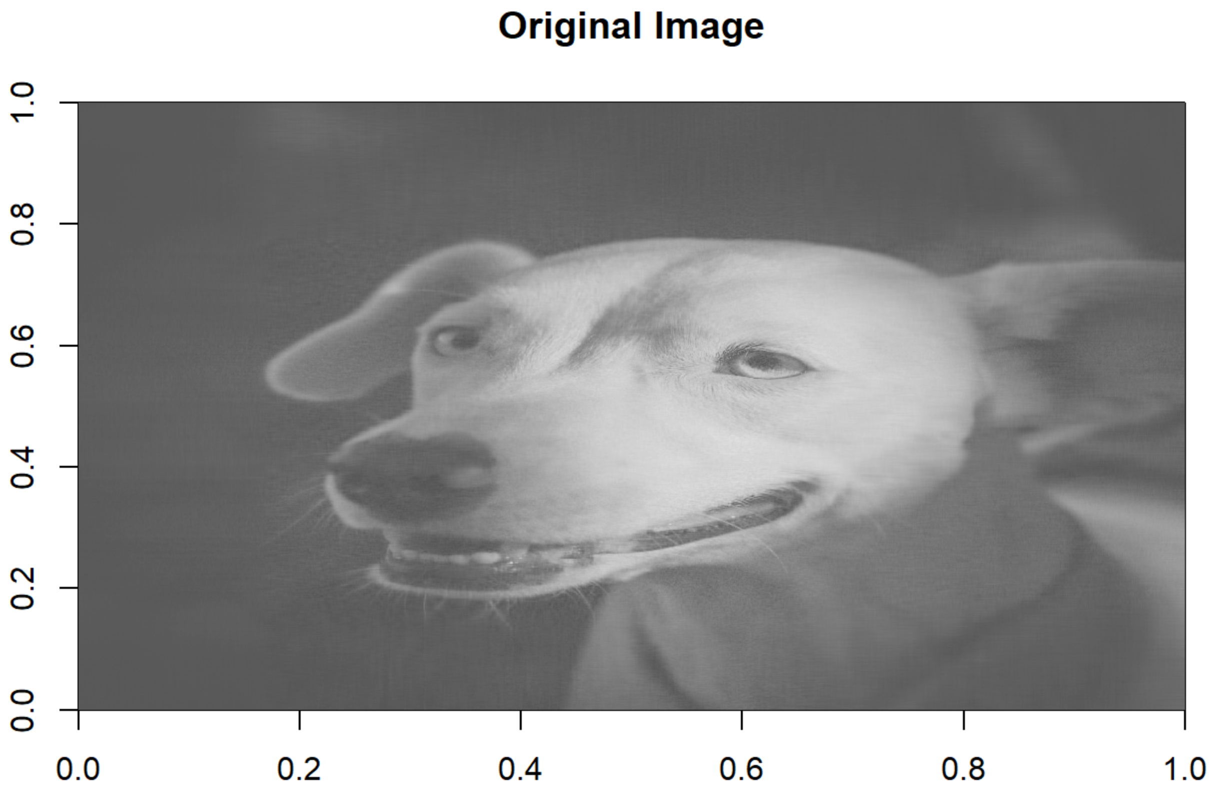
```r
## Beginning of work to reconstruct image and visualize results via a function;
image_processing <- function(k_value, svd_result) {
### Splitting the SVD component results into their different components
  U <- svd_result$u
  d <- svd_result$d
  V <- svd_result$v
## Once the components are broken out we can work with compressing diag matrix in components;
  comp_d <- diag(d)
## Limiting by k for the components
  lim_u <-U[, 1:k_value]
  lim_d <- comp_d[1:k_value, 1:k_value]
  lim_v <- V[, 1:k_value]
  recomp_image <- lim_u %*% lim_d %*% t(lim_v)
  return(recomp_image)
}

# Looping throuhg each k value and printing result;
k_vals<-c(5,10,20,50,100)
for (k in k_vals){
  recomp_image <- image_processing(k,svd_result)
  if (k == 100){
    title <- "Original Image"
    }
  else {
    title <- paste("Recomp Image with K value",k)
    }
  image(t(recomp_image)[, nrow(recomp_image):1], col=gray.colors(256), main=title)
  Sys.sleep(2)
  }
```


Recomp Image with K value 5
0.0
0.2
0.4
0.6
0.8
1.0
0.0
0.2
0.4
0.6
0.8
1.0


Recomp Image with K value 10
0.0
0.2
0.4
0.6
0.8
1.0
0.0
0.2
0.4
0.6
0.8
1.0


Recomp Image with K value 20
1.0
0.8
0.6
0.4
0.2
0.0
0.0
0.2
0.4
0.6
0.8
1.0


Recomp Image with K value 50
1.0
0.8
0.6
0.4
0.2
0.0
0.0
0.2
0.4
0.6
0.8
1.0



```
## Note for this task, I asked AI to help walk me through this step by step to ensure i fully
understood the process. I also didnt know how to display an image in r. The transformation wi
thin the image function is controlling for how R reads and stores the info.
```

Matrix Rank, Properties, and Eigenspace

Rank of a Matrix

Determine the Rank of the Given Matrix

Find the rank of the matrix A. Explain what the rank tells us about the linear independence of the rows and columns of matrix A. Identify if there are any linear dependencies among the rows or columns.

$$A = \begin{bmatrix} 2 & 4 & 1 & 3 \\ -2 & -3 & 4 & 1 \\ 5 & 6 & 2 & 8 \\ -1 & -2 & 3 & 7 \end{bmatrix}$$

Matrix Rank Boundaries

- Given an $m \times n$ matrix where $m > n$, determine the maximum and minimum possible rank, assuming that the matrix is non-zero.

- Prove that the rank of a matrix equals the dimension of its row space (or column space). Provide an example to illustrate the concept.

```
matrix_A <- matrix(c(2, 4, 1, 3, -2, -3, 4, 1, 5, 6, 2, 8, -1, -2, 3, 7), nrow=4, byrow=TRUE)
print(matrix_A)
```

```
##      [,1] [,2] [,3] [,4]
## [1,]    2    4    1    3
## [2,]   -2   -3    4    1
## [3,]    5    6    2    8
## [4,]   -1   -2    3    7
```

```
rank_A <- qr(matrix_A)$rank
print(rank_A)
```

```
## [1] 4
## Prove that the rank is the row or column space.
## Matrix a
##      [,1] [,2] [,3] [,4]
## [1,]    2    4    1    3
## [2,]   -2   -3    4    1
## [3,]    5    6    2    8
## [4,]   -1   -2    3    7

## Manually row reducing in order to prove.
## Getting the 1 in first 1,1 placement (divide first row by 2)
##| 1, 2, 0.5, 1.5 |
##|-2,-3, 4,  1    |
##| 5, 6, 2, 8     |
##|-1,-2, 3, 7     |

## Using first row to deal with the other rows.Additional steps as well.
##| 1, 2, 0.5, 1.5 |                     | 1, 2, 0.5, 1.5 |                    | 1, 2, 0.5, 
1.5 |             | 1, 2, 0.5, 1.5 |
##|-2,-3, 4,  1    | + 2(1,2,0.5,1.5) => | 0, 1,  5,  4   |                    | 0, 1,  5, 
4   |             | 0, 1,  5,  4   |
##| 5, 6, 2, 8     | - 5(1,2,0.5,1.5) => | 0,-4,-0.5, 0.5 | + 4(0, 1, 5, 4) =>  | 0, 0, 19.5, 
16.5| / (19.5) =>| 0, 0 , 1, 0.85 |
##|-1,-2, 3, 7     | + 1(1,2,0.5,1.5) => | 0, 0, 3.5, 8.5 |                    | 0, 0, 3.5, 
8.5 |             | 0, 0, 3.5, 8.5 |

## Continuing Row reduction
##| 1, 2, 0.5, 1.5 |                      | 1, 2, 0.5, 1.5 |            | 1, 2, 0.5, 1.5 |
##| 0, 1,  5,  4   |                      | 0, 1,  5,  4   |            | 0, 1,  5,  4   |
##| 0, 0 , 1, 0.85 |                      | 0, 0 , 1, 0.85 |            | 0, 0 , 1, 0.85 |
##| 0, 0, 3.5, 8.5 | - 3.5(0,0,1,0.85) => | 0, 0 , 0 ,5.52 | = / 5.52 => | 0, 0 , 0 , 1   |

## Finally, this matrix has been row reduced, there are no zero columns, so the number of lin
early independent columns is 4. This means that the rank is correct, in being the same number
of columns. Additionally, assuming the matrizx is square (n), according to the text book: r
(A)=n.
```

Rank and Row Reduction

- Determine the rank of matrix B. Perform a row reduction on matrix B and describe how it helps in finding the rank. Discuss any special properties of matrix B (e.g., is it a rank-deficient matrix?).

$$B = \begin{bmatrix} 2 & 5 & 7 \\ 4 & 10 & 14 \\ 1 & 2.5 & 3.5 \end{bmatrix}$$

```
## Firstly, finding the rank with r
matrix_b <- matrix(c(2, 5, 7, 4, 10, 14,1, 2.5, 3.5), ncol = 3, byrow=TRUE)
print(matrix_b)
```

```
##      [,1] [,2] [,3]
## [1,]    2  5.0  7.0
## [2,]    4 10.0 14.0
## [3,]    1  2.5  3.5
```

```
print(qr(matrix_b)$rank)
```

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

```
## The rank function returns: 1



##     [,1] [,2] [,3]
##[1,]    2  5.0  7.0
##[2,]    4 10.0 14.0
##[3,]    1  2.5  3.5

## Manually row reducing in order to prove.
## Getting the 1 in first 1,1 placement.
##| 2 , 5.0 , 7.0  | => | 1, 2.5 , 3.5|                        | 1, 2.5 ,3.5|
##| 4 , 10.0 , 14.0| => | 4 , 10.0 , 14.0| - 4(1, 2.5, 3.5) =>| 0,  0  ,0  |
##| 1 , 2.5  , 3.5 | => | 1 , 2.5  , 3.5 | - (1, 2.5, 3.5) => | 0,  0  ,0  |

## Final row reduced form:
##| 1, 2.5 ,3.5|
##| 0,  0  ,0  |
##| 0,  0  ,0  |

## The rank returns one because there is only one non-zero row in the matrix. The rest of the
rows are reduced to zero. The matrix is infact "rank deficient" because based on the size of
the matrix the maximum rank would be 3. However, the rank is 1 because of the two zero rows,
so its deficient.
```

Compute the Eigenvalues and Eigenvectors

- Find the eigenvalues and eigenvectors of the matrix A. Write out the characteristic polynomial and show your solution step by step. After finding the eigenvalues and eigenvectors, verify that the eigenvectors are linearly independent. If they are not, explain why.

$$A = \begin{bmatrix} 3 & 1 & 2 \\ 0 & 5 & 4 \\ 0 & 0 & 2 \end{bmatrix}$$

```
matrix_a <- matrix(c(3, 1, 2, 0, 5, 4, 0, 0, 2), ncol = 3, byrow=TRUE)
print(matrix_a)
```

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

```
## Firstly finding with the r function
egn <- eigen(matrix_a)
print(egn$values)
```

```
## [1] 5 3 2
```

```
##[1] 5 3 2
print(egn$vectors)
```

```
##           [,1] [,2]       [,3]
## [1,] 0.4472136    1 -0.3713907
## [2,] 0.8944272    0 -0.7427814
## [3,] 0.0000000    0  0.5570860
##          [,1] [,2]       [,3]
##[1,] 0.4472136    1 -0.3713907
##[2,] 0.8944272    0 -0.7427814
##[3,] 0.0000000    0  0.5570860

### Second, beginning the manual proof for eigenvalues.
##det(matrix_a - Lambda * I) = 0

##     |3 ,  1 ,  2|        |1, 0, 0 |                  |3 ,  1 ,  2|   |L, 0, 0 |
|3-L ,  1 ,  2|
##det( |0 ,  5 ,  4|  - L * |0, 1, 0 | ) = 0  ==> det(  |0 ,  5 ,  4| - |0, L, 0 | ) = 0  ==>
|0 ,  5-L ,  4|
##     |0 ,  0 ,  2|        |0 , 0, 1|                  |0 ,  0 ,  2|   |0, 0, L |
|0 ,  0 ,  2-L|

## And because the diagonal product is the determinant we can then do:
## (3-L)(5-L)(2-L)=0
## which allows us to conclude the eigenvalues are: 3, 5, 2 which was also shown to us with t
he r function.

## Third the eigenvector calculations. With the equation: (A - LI)v = 0

## For First eigenvalue 3
## |3-3 ,  1 ,  2|    |0 , 1, 2|   |x1|        0x1 + 1x2 + 2x3 = 0    1x2 + 2x3 = 0    x2 = -
(2*0) => 0
## |0 ,  5-3 ,  4| => |0, 2, 4 | * |x2| = 0 => 0x1 + 2x2 + 4x3 = 0 => 2x2 + 4x3 = 0 => 2(-(2*
0)) + 4*0 = 0
## |0 ,  0 ,  2-3|    |0, 0, -1|   |x3|        0x1 + 0x2 - 1x3 = 0    -1x3 = 0          x3 =
0 (so replacing above)

## x2 and x3 are both zero, but because there is nothing in column 1, x1 is free var. So we a
ssign 1
## Eigenvector is [1,0,0]

## For eigenvalue 5
## |3-5 ,  1 ,  2|    |-2 , 1, 2|   |x1|        -2x1 + x2 + 2x3 = 0    -2x1 +x2 + 2(0) => -2x
1 + x2 =0 => 2x1 = x2=>  -2x1 +2x1 => 0=0   free var
## |0 ,  5-5 ,  4| => |0 , 0, 4 | * |x2| = 0 => 0x1 + 0x2 + 4x3 = 0 =>  0x1 + 0x2 + 4(0) = 0
=> x2 is free var
## |0 ,  0 ,  2-5|    |0 , 0, -3|   |x3|        0x1 + 0x2 + -3x3 = 0   x3 = 0 (replacing abov
e)
## eigenvector is [1,2,0]

## For eigenvalue 2
## |3-2 ,  1 ,  2|    |1 , 1, 2|   |x1|        x1 + x2 + 2x3 = 0 => x1 - 4x3/3 + 2x3 = 0 => -
x1 = - 4/3 x3 + 2/1 x3 => -2/3 x3 = x1
## |0 ,  5-2 ,  4| => |0 , 3, 4| * |x2| = 0 => 0x1 + 3x2 + 4x3 = 0 => 4x3 = -3x2 = 4/-3 x3 =
x2 =>
## |0 ,  0 ,  2-2|    |0 , 0, 0|   |x3|        0x1 + 0x2 + 0x3 = 0   x3 is free var x3, assig
ning 1
```

```
## Eigenvector is [-2/3, -4/3, 1] => [-2,-4.3]

### Each of these eigenvectors has a free variable, which is indicative of linear independenc
e. Confirming with eigenspace matrix.

## Making Eigenspace with the eigenvectors
print(det(matrix(c(1,1,-2,0,2,-4,0,0,3), ncol = 3 , byrow=TRUE))==0)
```

```
## [1] FALSE
```

```
##  Determinant does not equal 0 so they are linearly independent.
```

Diagonalization of Matrix

- Determine if matrix A can be diagonalized. If it can, find the diagonal matrix and the matrix of eigenvectors that diagonalizes A.
- Discuss the geometric interpretation of the eigenvectors and eigenvalues in the context of transformations. For instance, how does matrix A stretch, shrink, or rotate vectors in $\mathbb{R}^3$?

```
## Firstly, matrix A can be diagonalized because every eigenvector is linearly independent. D
eterminant of eigenspace is not 0, so diagonalization is possible.

## Second,semi-manual diagonalization
eigenspace <- matrix(c(1,1,-2,0,2,-4,0,0,3), ncol = 3 , byrow=TRUE)

### eigenspace^-1 * matrix_a * eigenspace is how we would diagonalize.

diag_matrx <- solve(eigenspace) %*% matrix_a %*% eigenspace
print(diag_matrx)
```

```
##      [,1] [,2] [,3]
## [1,]    3    0    0
## [2,]    0    5    0
## [3,]    0    0    2
## For matrix a the eigenvalues are 3,5, and 2. This means that when matrix a is applied to a
vector it will stretch the vector in different directions by 3x, 5x, and 2x. All of the eigen
values are positive and over 1 so they stretch the vectors. Now, for the eigenvectors in rela
tion to the eigenvalues:

# val: 3 | vector: [1,0,0] -> This vector is only in the x-axis direction. So 3x in the x axi
s direction.
# val: 5 | vector: [1,2,0] -> This vector is in the x, y axes directions. it is diagonal in x
& y because they are not equal values. So 5x in the x & y axes directions.
# val: 2 | vector: [-2,-4.3] -> This vector is in all x,y, and z axes. So 2x in all three
(x,y,z) directions, however the directions are diagonal becuase of different values in each a
xis.

### For R3, assuming 1,1,1, the application of matrix a would distort R3 in every axis over a
ll. This would be a net total of a 4x stretch in the x axis (3x+5x-4x), a net total of 2x str
etch in the y axis (0x + 10x - 8x), and then a 5x in the z- axis direction (0x + 0x + 6x).
```

Project: Eigenfaces from the LFW Dataset

Context

Eigenfaces are a popular application of PCA in computer vision for face recognition.

Task

Using the LFW dataset, build and visualize eigenfaces that account for 80% of the variability in the dataset.

Steps

1. Download the LFW Dataset.
    - The dataset can be accessed and downloaded using the lfw module from the sklearn library in Python or by manually downloading it from the LFW website.
    - In this case, well use the lfw module from Pythons sklearn library.
2. Preprocess the images (convert to grayscale and resize).
    - Convert the images to grayscale and resize them to a smaller size (e.g., 64x64) to reduce computational complexity.
    - Flatten each image into a vector.
3. Apply PCA and compute eigenfaces.
    - Compute the PCA on the flattened images.
    - Determine the number of principal components required to account for 80% of the variability.
4. Visualize the top eigenfaces and reconstruct images.
    - Visualize the first few eigenfaces (principal components) and discuss their significance.
    - Reconstruct some images using the computed eigenfaces and compare them with the original images.

```
### Used r so manually downloading the data from this location found in the source code (http
s://github.com/scikit-learn/scikit-learn/blob/main/sklearn/datasets/_lfw.py) of sklearn.py

## Referenced soem of this: https://stackoverflow.com/questions/3053833/using-r-to-download-z
ipped-data-file-extract-and-import-data

## install.packages("imager")
library(imager)
```

```
## Loading required package: magrittr
```

```
##
## Attaching package: 'magrittr'
```

```
## The following object is masked from 'package:purrr':
##
##     set_names
```

```
## The following object is masked from 'package:tidyr':
##
##     extract
```

```
##
## Attaching package: 'imager'
```

```
## The following object is masked from 'package:magrittr':
##
##     add
```

```
## The following object is masked from 'package:stringr':
##
##     boundary
```

```
## The following object is masked from 'package:tidyr':
##
##     fill
```

```
## The following object is masked from 'package:dplyr':
##
##     where
```

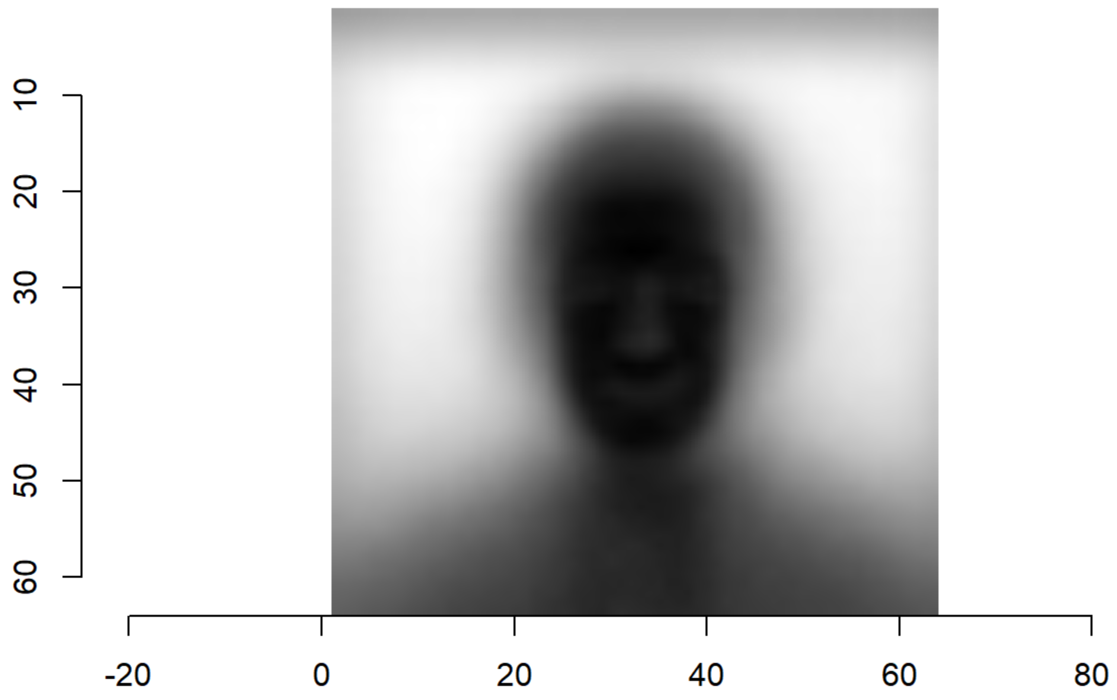
```
## The following objects are masked from 'package:stats':
##
##     convolve, spectrum
## The following object is masked from 'package:graphics':
##
##     frame
```

```
## The following object is masked from 'package:base':
##
##     save.image
## First download the data (NOTE: DURING KNITTING THIS DID NOT WORK SO COMMENTING OUT AND JUS
T READING LOCAL FILE FOR KNIT)
##download.file("https://ndownloader.figshare.com/files/5976018","lfw.tgz",mode = "wb")
##untar("lfw.tgz")

## Loop through the downloaded data and convert the images to grayscale, Resize the images to
64x64
dir.create("processed_lfw")
img_files <- list.files('lfw',pattern="\\.jpg",recursive = TRUE, full.names = TRUE)
for (i_f  in img_files) {
  temp_image <-load.image(i_f)
  img_gray <- grayscale(temp_image)
  processed <- resize(img_gray, size_x = 64, size_y = 64)
  new_filename <- file.path("processed_lfw", basename(i_f))
  save.image(processed, new_filename)
  }

## Flatten each processed image into a vector, place into list and then matrix.

image_vec_list <-list()
proc_img <- list.files('processed_lfw',pattern="\\.jpg",recursive = TRUE, full.names = TRUE)
for (i in seq_along(proc_img)){
  image_matrix <- load.image(proc_img[i])
  image_vec <- as.vector(image_matrix)
  image_vec_list[[i]] <- image_vec
  }


## Putting list of flattened vectors into a matrix for PCA
flattened_matrix <- do.call(rbind, image_vec_list)

## PCA on matrix of flattened images
mypca=princomp(x = flattened_matrix, cor = T)

## mypca has all of the computed Principle Components, getting the std deviation for the PC c
alculations and squaring in order to get total var.

variance <- mypca$sdev^2

total_variance <- sum(variance)

## We want 80 percent of variance explained, so normalizing proporitions
pct_tot_var <-  variance/total_variance

## Now using cumsum to see how many to get to 80%
## Visually looking and i think 68 PC are needed.
cum_var <- cumsum(pct_tot_var)
## print(cum_var)


## doublechecking with code
```

```
print(which(cum_var>= 0.8)[1]) ##Also 68
```

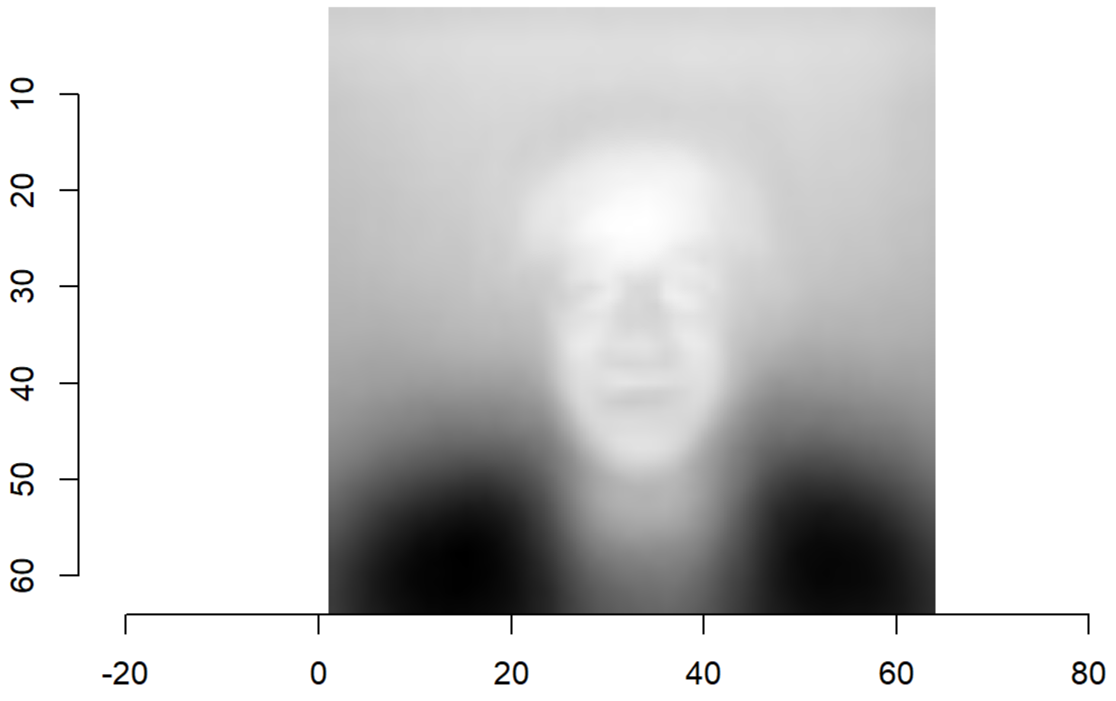
```
## Comp.68
##      68
```

```
## In order to account 80% of the variance 68 Principle Components are needed.

### Visualizing the first 3 Eigenfaces / PC.

## First PC
eigenface_image<- matrix(mypca$loadings[,1], nrow = 64, ncol = 64)
eigenface_norm<- (eigenface_image - min(eigenface_image)) / (max(eigenface_image) - min(eigen
face_image))
plot(as.cimg(eigenface_norm))
```



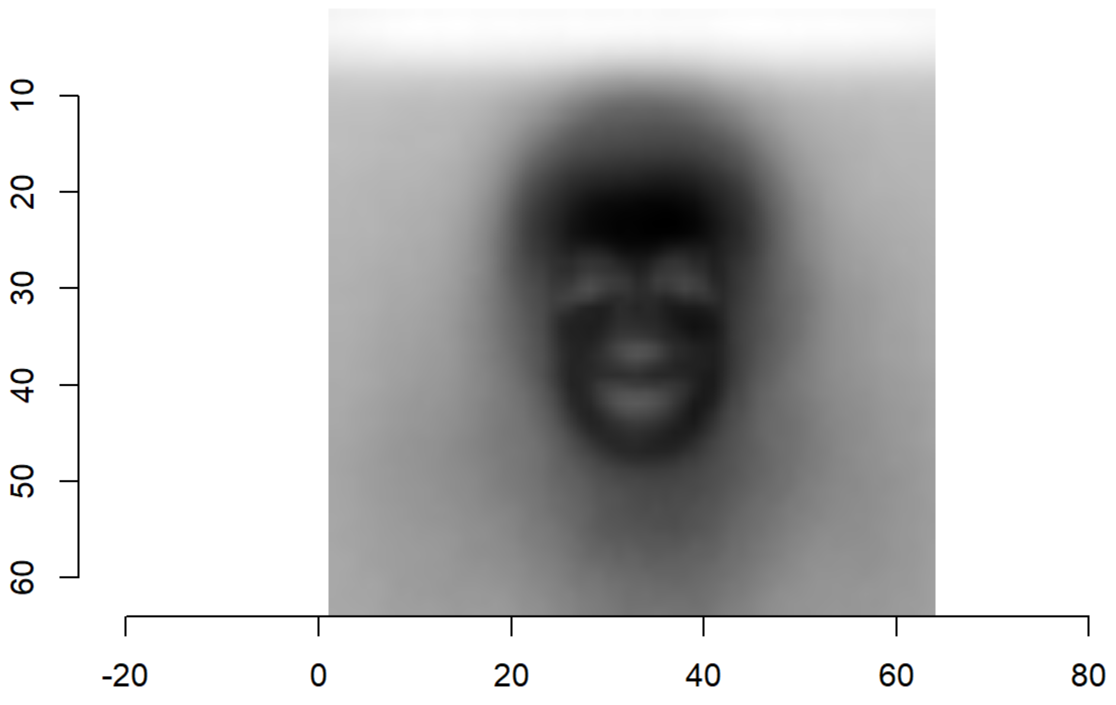
```
print(pct_tot_var[1])
```

```
##    Comp.1
## 0.2300384
## 23 Percent of the total variance is explained by the first PC. THis can be seen in the plo
t, generally speaking you can tell its a portrait photo but the details are not present.

## Second PC
eigenface_image<- matrix(mypca$loadings[,2],nrow = 64, ncol = 64)
eigenface_norm<- (eigenface_image - min(eigenface_image)) / (max(eigenface_image) - min(eigen
face_image))
plot(as.cimg(eigenface_norm))
```



```
print(pct_tot_var[2])
```

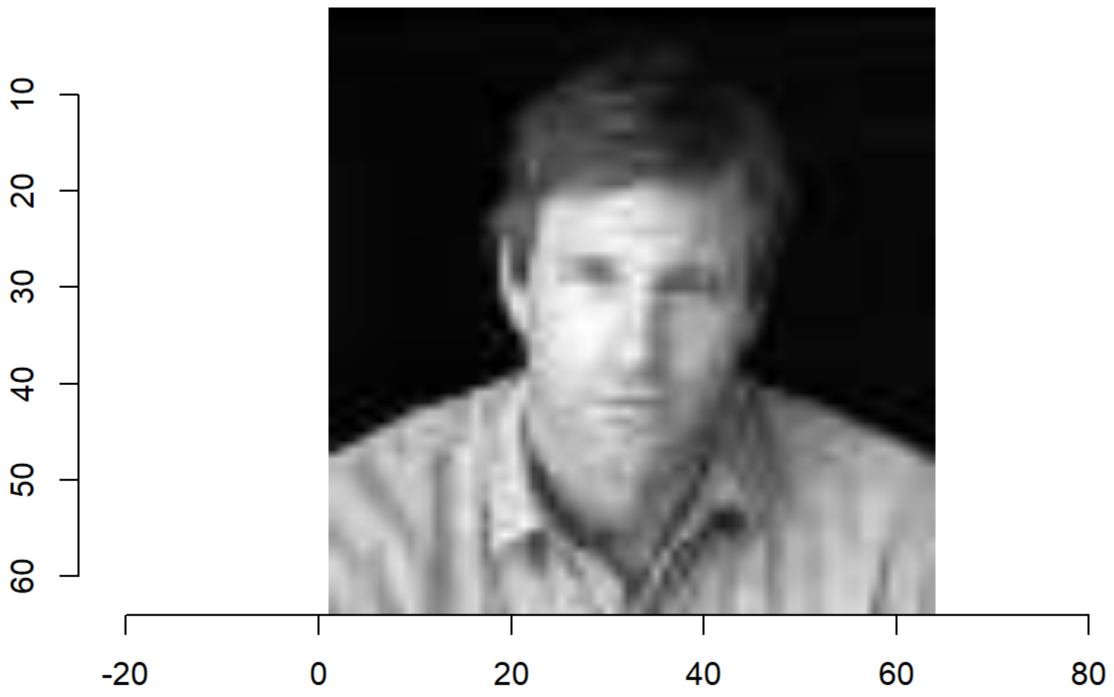
```
##     Comp.2
## 0.07311962
## This eigenface has acout 7% of the total variance. You can see that there is additonal det
ail here that was not captured in the first eigenface, but still a very crude level of compos
ition.

## Third PC
eigenface_image<- matrix(mypca$loadings[,3],nrow = 64, ncol = 64)
eigenface_norm<- (eigenface_image - min(eigenface_image)) / (max(eigenface_image) - min(eigen
face_image))
plot(as.cimg(eigenface_norm))
```



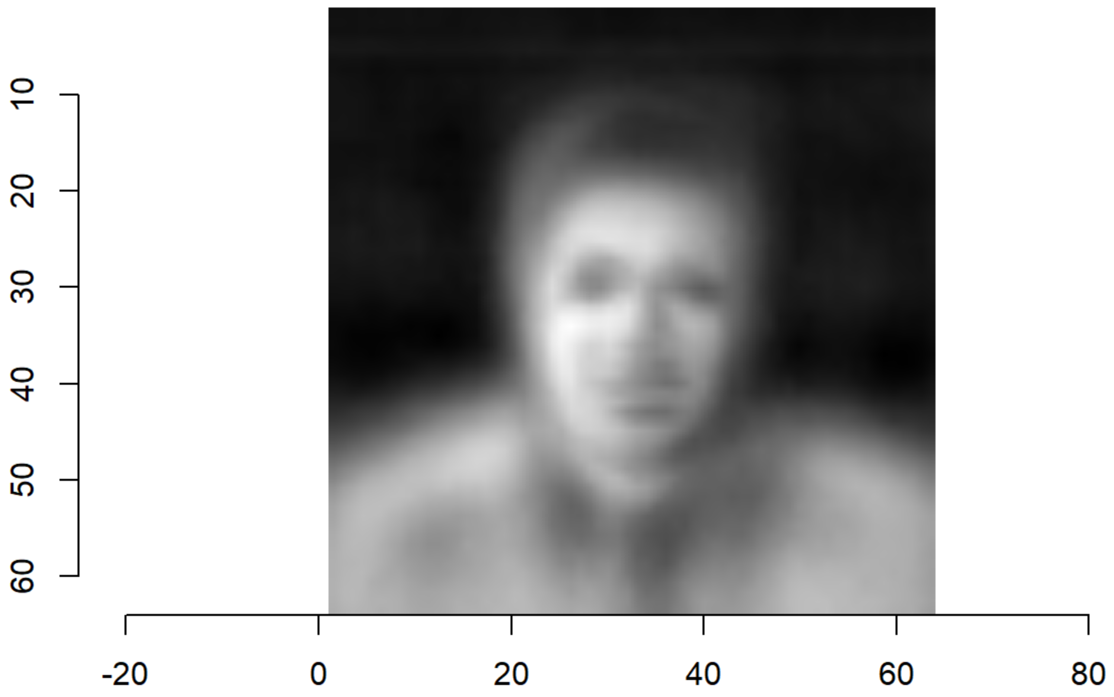
```
## This eigenfaces captures about 5.5 percent of all of the variance.
print(pct_tot_var[3])
```

```
##     Comp.3
## 0.05504029
```

```
## Comparing these eigenfaces to the original photos

## Taking an original processed (64x64 grayscale) photo from the original vector matrix.
first_image_vec <- flattened_matrix[1,]
first_image_mat <- matrix(first_image_vec, nrow = 64, ncol = 64)
plot(as.cimg(first_image_mat))
```



```
## Attempting to reconstruct with eigenface. Lets do 68 PC becuase that was the 80% threshold
first_image_pc_product<- first_image_vec %*% mypca$loadings[, 1:68]
first_image_pc_trans_product <- first_image_pc_product %*% t(mypca$loadings[, 1:68])
recomp <- matrix(first_image_pc_trans_product, nrow = 64, ncol = 64)
plot(as.cimg(recomp))
```



```
## The recomposed photo you can see is similar to about 80% of the original photo, which trac
ks with the PC variance.
```