DATA 605 - Homework #1

Julian Adames-Ng

2024-10-05

1. 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 shapes 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 multiplying a transformation matrix by each of the point vectors. Demonstrate these transformations through animated plots.

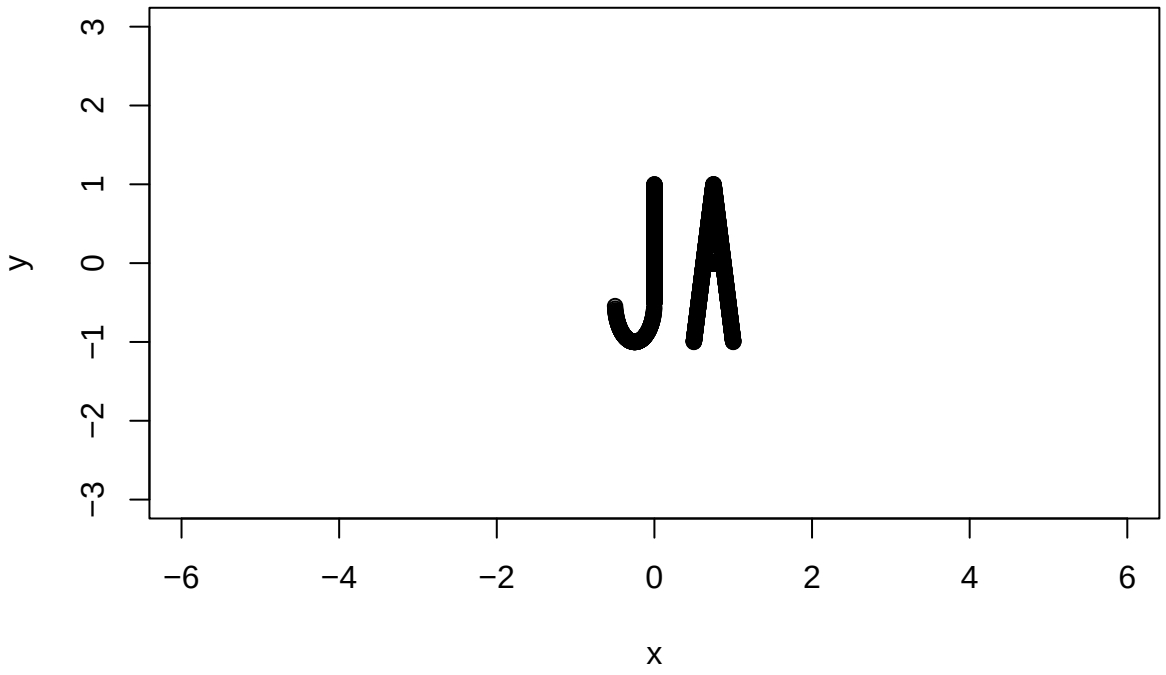
Create a Shape:

Define a simple shape (e.g., a square) using a set of point coordinates. I made sure the aspect ratio was 1:1 using the "asp" parameter for the plot() function.

```
rm(list = ls())

x=c(rep(0,500),seq(0.65,0.85,length.out=1000), seq(0.75,1,length.out=500), seq(0.5,0.75,length.out=500)
y=c(seq(-0.5,1,length.out=500),rep(0,1000), seq(1,-1,length.out=500), seq(-1,1,length.out=500), -0.5-sq
```

```
## Warning in sqrt(0.25 - (seq(-1, 1, length.out = 500) - 0.5)^2): NaNs produced
```

```
z <- rbind(x, y)

# Create a plot
plot(y~x, xlim=c(-3,3), ylim=c(-3,3), asp = 1)
```



Apply Transformations:

- Scaling: Enlarge or shrink the shape.
- Rotation: Rotate the shape by a given angle.
- Reflection: Reflect the shape across an axis.

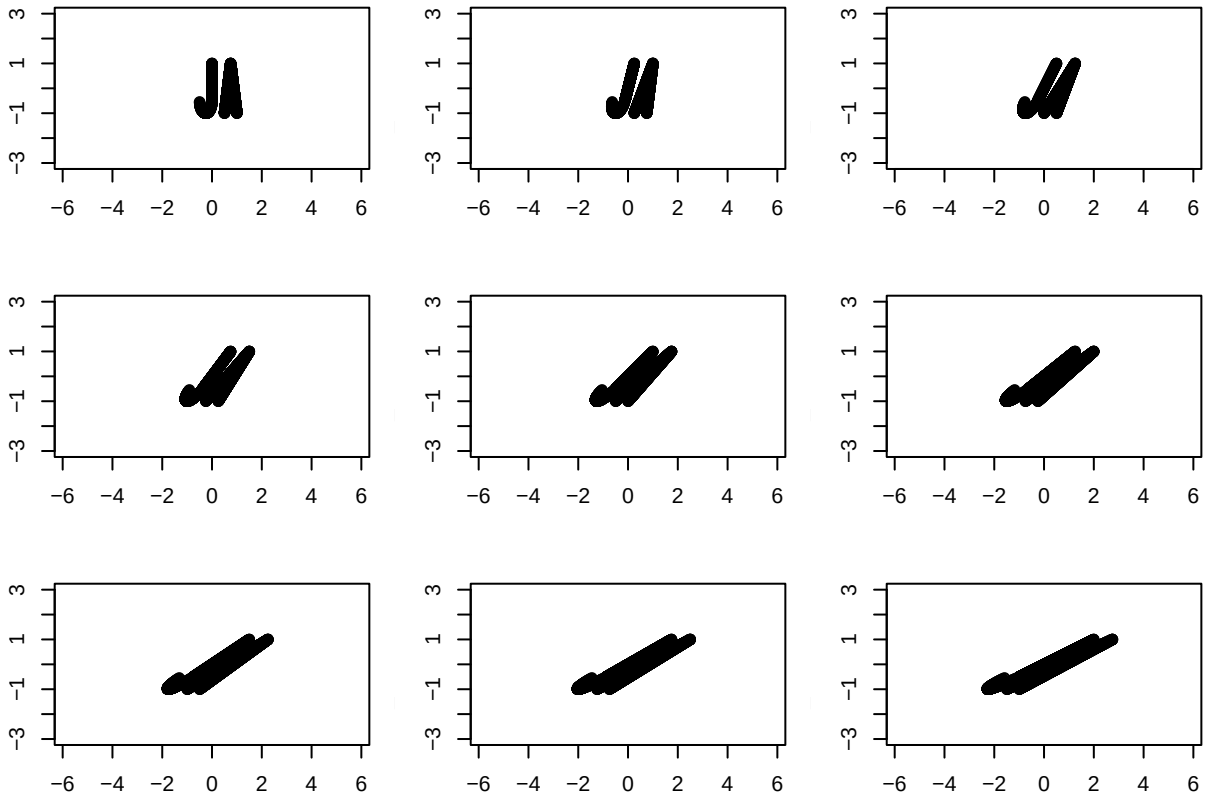
Plot:

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

Shearing:

Here I created a function called "leftMultiply" which performs matrix multiplication.

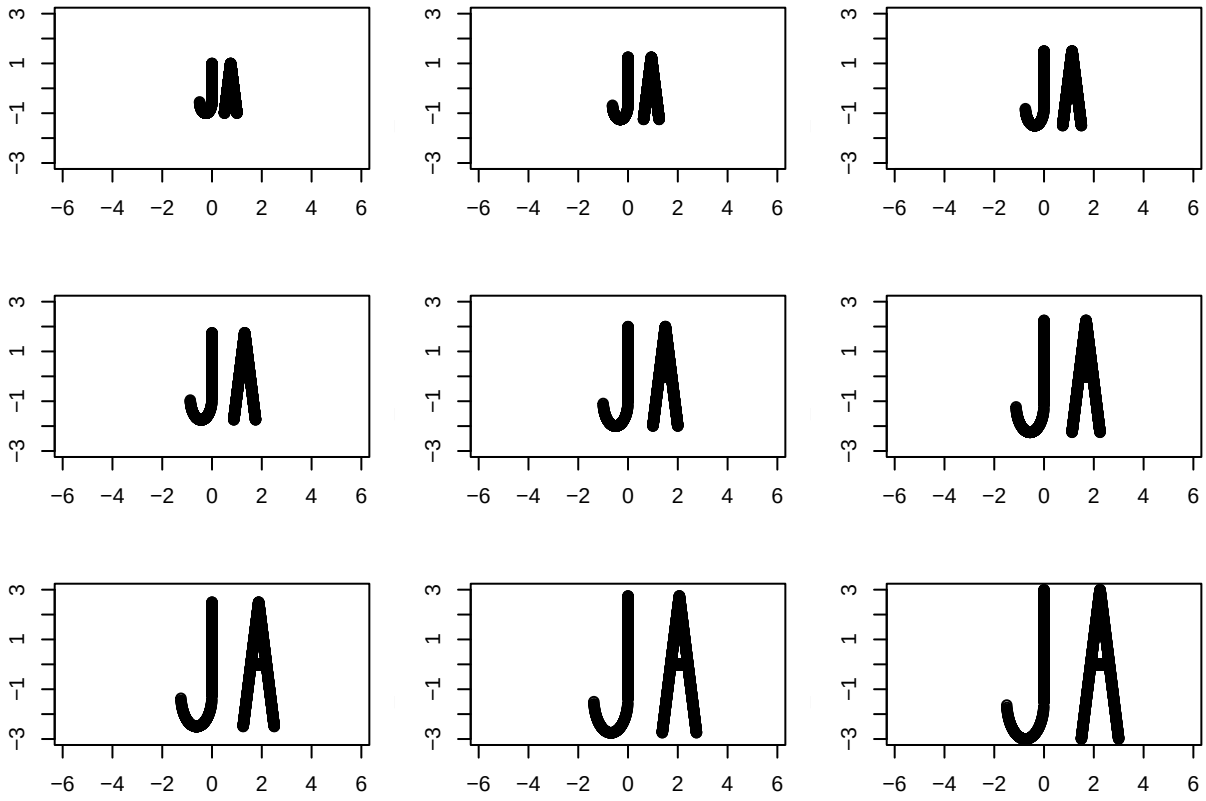
```
leftMultiply  <- function(x,y){
   x %*% y
}

par(mfrow = c(3,3), mar = c(3,3,2,1))
for (i in seq(0,2,length.out=9)) {
  z_shear<-apply(z,2,function(x) leftMultiply(x,matrix(c(1,i,0,1),nrow=2,ncol=2)))
  plot(z_shear[2,]~z_shear[1,], xlim=c(-3,3), ylim=c(-3,3), col='black', asp = 1)
}
```



Scaling:

Enlarge or shrink the shape.

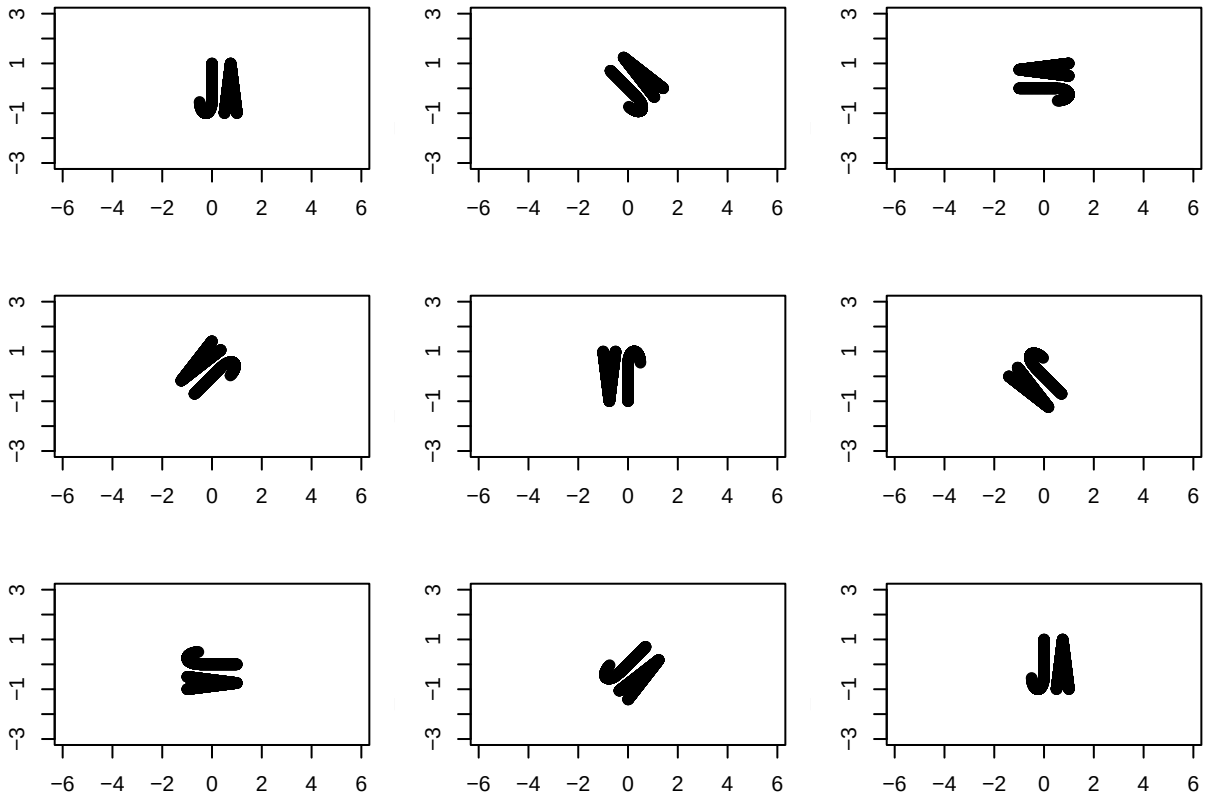
```
par(mfrow = c(3,3), mar = c(3,3,2,1))
for (i in seq(1,3,length.out=9)) {
  z_scale<-apply(z,2,function(x) leftMultiply(x,matrix(c(i,0,0,i),nrow=2,ncol=2)))
   plot(z_scale[2,]~z_scale[1,], xlim=c(-3,3), ylim=c(-3,3), col='black', asp = 1)
}
```



Rotation:

Rotate the shape by a given angle.

```
par(mfrow = c(3,3), mar = c(3,3,2,1))
for (i in seq(0,pi*2,length.out=9)) {
  z_rotate<-apply(z,2,function(x) leftMultiply(x,matrix(c(cos(i),-sin(i),sin(i),cos(i)),nrow=2,ncol=2)))
   plot(z_rotate[2,]~z_rotate[1,], xlim=c(-3,3), ylim=c(-3,3), col = 'black', asp = 1)
}
```

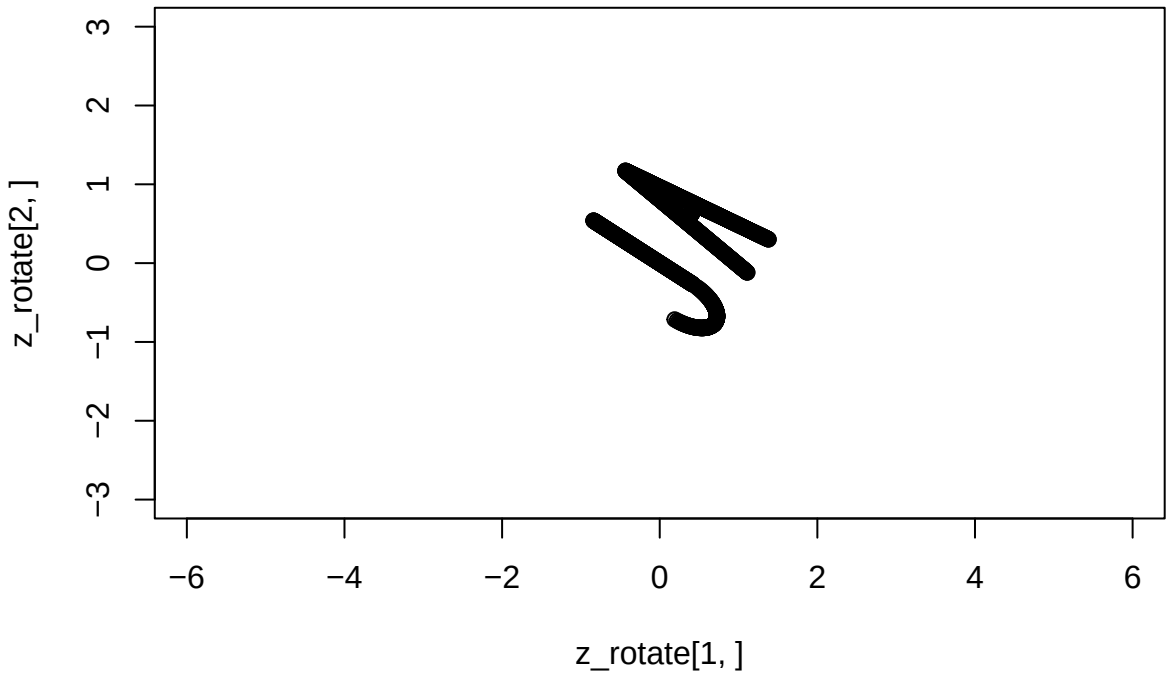


Reflection:

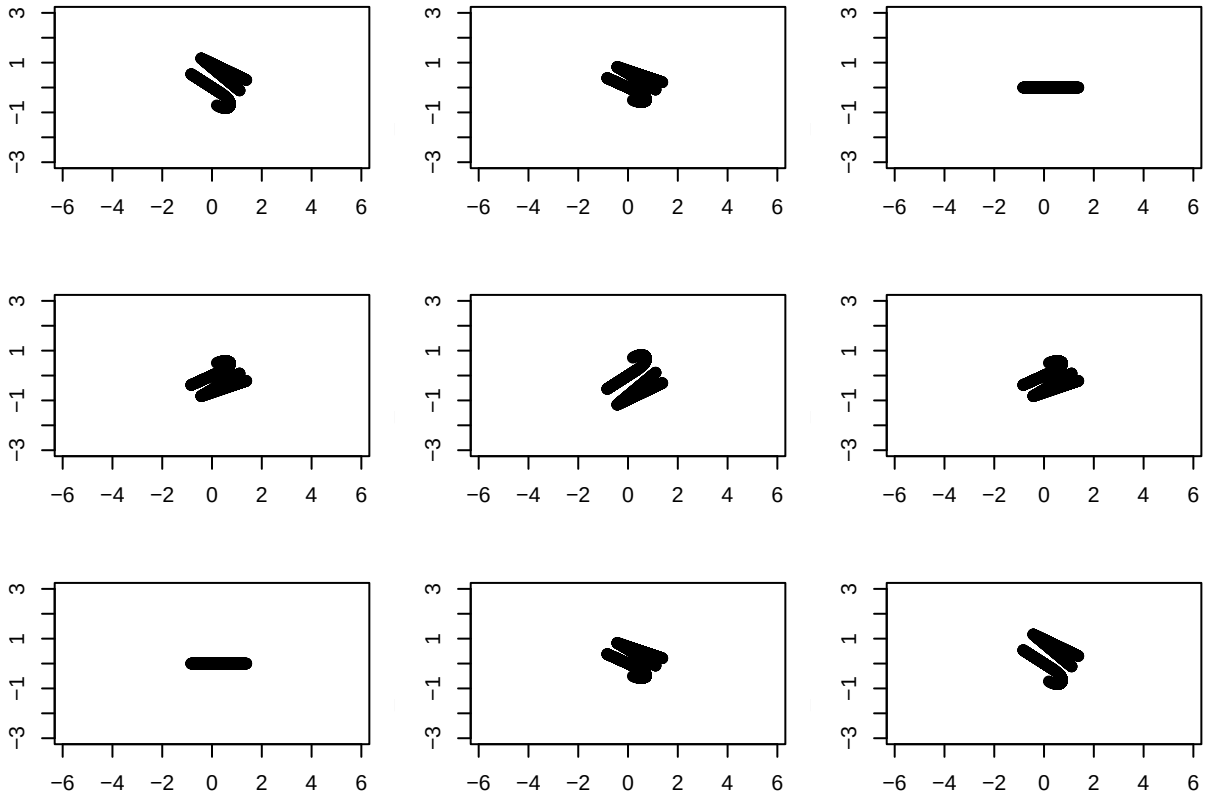
Reflect the shape across an axis.

I rotated my initials just a bit so that the visualization shows a clear reflection process.

```
z_rotate<-apply(z,2,function(x) leftMultiply(x,matrix(c(cos(1),-sin(1),sin(1),cos(1)),nrow=2,ncol=2)))
   plot(z_rotate[2,]~z_rotate[1,], xlim=c(-3,3), ylim=c(-3,3), col = 'black', asp = 1)
```




```
par(mfrow = c(3,3), mar = c(3,3,2,1))
for (i in seq(0,2*pi,length.out=9)) {
  z_new <-rbind(z_rotate,rep(0,ncol(z)))
  z_reflect<-apply(z_new,2,function(x) leftMultiply(x,matrix(c(1,0,0,0,cos(i),-sin(i),0,sin(i),cos(i)),
   plot(z_reflect[2,]~z_reflect[1,], xlim=c(-3,3), ylim=c(-3,3), col='black', asp = 1)
}
```



Animating Transformations:

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

Code:

The following code chunks generate GIF files of the aforementioned animations.

Shearing Animation:

```
saveGIF({
  for (i in seq(0,2,length.out=10)) {
    z_shear<-apply(z,2,function(x) leftMultiply(x,matrix(c(1,i,0,1),nrow=2,ncol=2)))
    plot(z_shear[2,]~z_shear[1,], xlim=c(-3,3), ylim=c(-3,3), col='black', asp = 1)
  }
}, movie.name = "shear_animation.gif", interval = 0.1)
```

```
## Output at: shear_animation.gif
```

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

```
#shear_gif <- image_read("/Users/jadamesng/Brooklyn College Dropbox/Julian Adames-Ng/MS Data Science/DA
shear_gif <- image_read("shear_animation.gif")

#shear_gif
```

Scaling Animation:

```
saveGIF({
  for (i in seq(1,3,length.out=10)) {
    z_scale<-apply(z,2,function(x) leftMultiply(x,matrix(c(i,0,0,i),nrow=2,ncol=2)))
    plot(z_scale[2,]~z_scale[1,], xlim=c(-3,3), ylim=c(-3,3), col='black', asp = 1)
  }
}, movie.name = "scale_animation.gif", interval = 0.1)
```

```
## Output at: scale_animation.gif
```

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

```
#scale_gif <- image_read("/Users/jadamesng/Brooklyn College Dropbox/Julian Adames-Ng/MS Data Science/DA
scale_gif <- image_read("scale_animation.gif")

#scale_gif
```

Rotating Animation:

```
saveGIF({
  for (i in seq(0,pi*2,length.out=20)) {
    z_rotate<-apply(z,2,function(x) leftMultiply(x,matrix(c(cos(i),-sin(i),sin(i),cos(i)),nrow=2,ncol=2
    plot(z_rotate[2,]~z_rotate[1,], xlim=c(-3,3), ylim=c(-3,3), col = 'black', asp = 1)
  }
}, movie.name = "rotation_animation.gif", interval = 0.1)
```

```
## Output at: rotation_animation.gif
```

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

```
#rotation_gif <- image_read("/Users/jadamesng/Brooklyn College Dropbox/Julian Adames-Ng/MS Data Science/
rotation_gif <- image_read("rotation_animation.gif")

#rotation_gif
```

Reflecting Animation:

```
saveGIF({
  for (i in seq(0,2*pi,length.out=20)) {
    z_new <-rbind(z_rotate,rep(0,ncol(z)))
    z_reflect<-apply(z_new,2,function(x) leftMultiply(x,matrix(c(1,0,0,0,cos(i),-sin(i),0,sin(i),cos(i)
    plot(z_reflect[2,]~z_reflect[1,], xlim=c(-3,3), ylim=c(-3,3), col='grey', asp = 1)
  }
}, movie.name = "ref_animation.gif", interval = 0.1)
```

```
## Output at: ref_animation.gif
```

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

```
#reflection_gif <- image_read("/Users/jadamesng/Brooklyn College Dropbox/Julian Adames-Ng/MS Data Scien
reflection_gif <- image_read("ref_animation.gif")

#reflection_gif
```

2. Matrix Properties and Decomposition

Proofs

- **Prove** that $AB \neq BA$

  Proof (by counterexample): Let A and B be two square, non-zero matrices such that

  $$A = \begin{pmatrix} 1 & 5 \\ 3 & 2 \end{pmatrix}, B = \begin{pmatrix} 3 & 11 \\ 7 & 1 \end{pmatrix}$$

  The following *left-hand-side* matrix multiplication for AB yields:

  $$AB = \begin{bmatrix} 1 & 5 \\ 3 & 2 \end{bmatrix} \begin{bmatrix} 3 & 11 \\ 7 & 1 \end{bmatrix} = \begin{bmatrix} 38 & 16 \\ 23 & 35 \end{bmatrix}$$

  However, the *right-hand-side* matrix multiplication for BA yields:

  $$BA = \begin{bmatrix} 3 & 11 \\ 7 & 1 \end{bmatrix} \begin{bmatrix} 1 & 5 \\ 3 & 2 \end{bmatrix} = \begin{bmatrix} 36 & 37 \\ 10 & 37 \end{bmatrix}$$

  Thus, $AB \neq BA$. ■

  Note that the example above shows multiplication of two square matrices. If A and B were not square, matrix multiplication may not even be possible in a given direction. In particular, in order to be able to matrix multiply in *both* directions, we must have an $m \times n$ matrix and an $n \times m$ matrix. This way, regardless of the multiplication order, the rows for the first matrix matches with the number of columns in the second matrix.

- **Prove** that A^TA is always symmetric

  Proof: Let A be an $m \times n$ matrix. Then A^T is an $n \times m$ matrix. So the product A^TA will be an $n \times n$ matrix. This satisfies the requirement that symmetric matrices are square. It should be known that for two matrices X and Y where their matrix product XY is defined, the following equality holds true:

  $$(XY)^T = Y^TX^T \text{(Property: Transpose of a Matrix Product)}$$

  Applying this to the matrix product A^TA yields:

  $$(A^TA)^T = A^T(A^T)^T$$

  Continuing the **right-hand-side** simplification,

  $$A^T(A^T)^T = A^TA$$

  since $(A^T)^T = A$. ■

- **Prove** that the determinant of A^TA is non-negative

  Proof: Let A be an $m \times n$ matrix. Then A^TA is a square, $n \times n$ matrix. We say that a matrix, B, is **positive semi-definite** if for any vector $x \in \mathbb{R}$, the following holds true:

  $$x^TBx \geq 0$$

  Let's prove this is true for matrix (A^TA) by substituting it in for B. We then have:

  $$x^T(A^TA)x$$

  which can be rewritten as

  $$x^TA^T(Ax)$$

  by *the associative property of matrix multiplication*. Using the previously used *Property: Transpose of a Matrix Product*, we can rewrite the left most matrix product, x^TA^T, from the previous expression as follows:

  $$x^TA^T = (Ax)^T,$$

  so the full expression

  $$\mathbf{x^TA^T}(Ax) = (\mathbf{Ax})^{\mathbf{T}}(Ax).$$

  In the resulting expression above, the *right-hand-side* of the equation is the product of a $1 \times m$ row vector, $(Ax)^T$, and an $m \times 1$ column vector, Ax. Although this is matrix product notation, it can also be thought of as a *dot product* of the *vector* Ax with itself which results in the *squared norm* of vector $A\boldsymbol{x}$:

  $$(A\boldsymbol{x}) \cdot (A\boldsymbol{x}) = ||A\boldsymbol{x}||^2$$

  where $||A\boldsymbol{x}||^2 \in \mathbb{R}$. In particular, since this is a *squared* norm, it follows that

  $$||A\boldsymbol{x}||^2 \geq 0$$

  and therefore

  $$x^T(A^TA)x \geq 0$$

  proving that A^TA is a **positive semi-definite** matrix.

  We will use the property of **positive semi-definite** matrices that states that their **eigenvalues** are non-negative. It should also be known that the **determinant** of a matrix can be expressed as the product of its eigenvalues. This is represented by the equation below:

  $$\det(A^TA) = \prod_{i=1}^{n} \lambda_i \geq 0, \text{ where } \lambda_i \text{ are the eigenvalues of } A^TA.$$

From a previous result, we know that A^TA is **positive semi-definite** and we can conclude that its determinant, $\det(A^TA)$, is also non-negative since it is just a product of its non-negative eigenvalues. Therefore,

$$\det(A^TA) \geq 0. \blacksquare$$

Context:

Every time you upload a photo to social media, algorithms often compress your image to reduce file size without significantly degrading quality. One of the key techniques used in this process is Singular Value Decomposition (SVD), which is another form of matrix factorization.

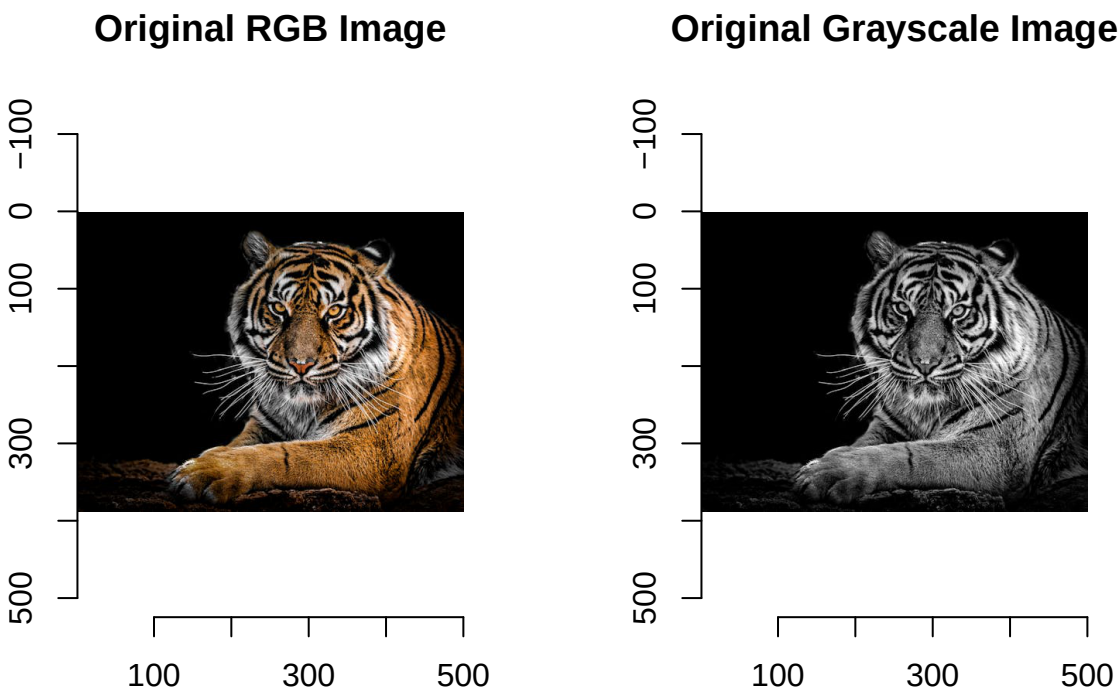
Task:

Write an R function that performs Singular Value Decomposition (SVD) on a grayscale image (which can be represented as a matrix). Use this decomposition to compress the image by keeping only the top k singular values and their corresponding vectors. Demonstrate the effect of different values of k on the compressed image's quality. You can choose any open-access grayscale image that is appropriate for a professional program.

Instructions:

Read an Image: Convert a grayscale image into a matrix.

```r
rm(list = ls())

#t_image <- readJPEG("Tiger_Grayscale.jpg")
t_image <- load.image("tiger.jpeg")

t_gray <- grayscale(t_image)

par(mfrow = c(1, 2))

plot(t_image, main = "Original RGB Image")

plot(t_gray, main = "Original Grayscale Image")
```




```
t_matrix <- as.matrix(t_gray)
```

Perform Singular Value Decomposition: Factorize the image matrix A into $U \sum V^T$ using R's built-in svd() function.

```
t_svd <- svd(t_matrix)
```

Compress the Image: Reconstruct the image using only the top k singular values and vectors.

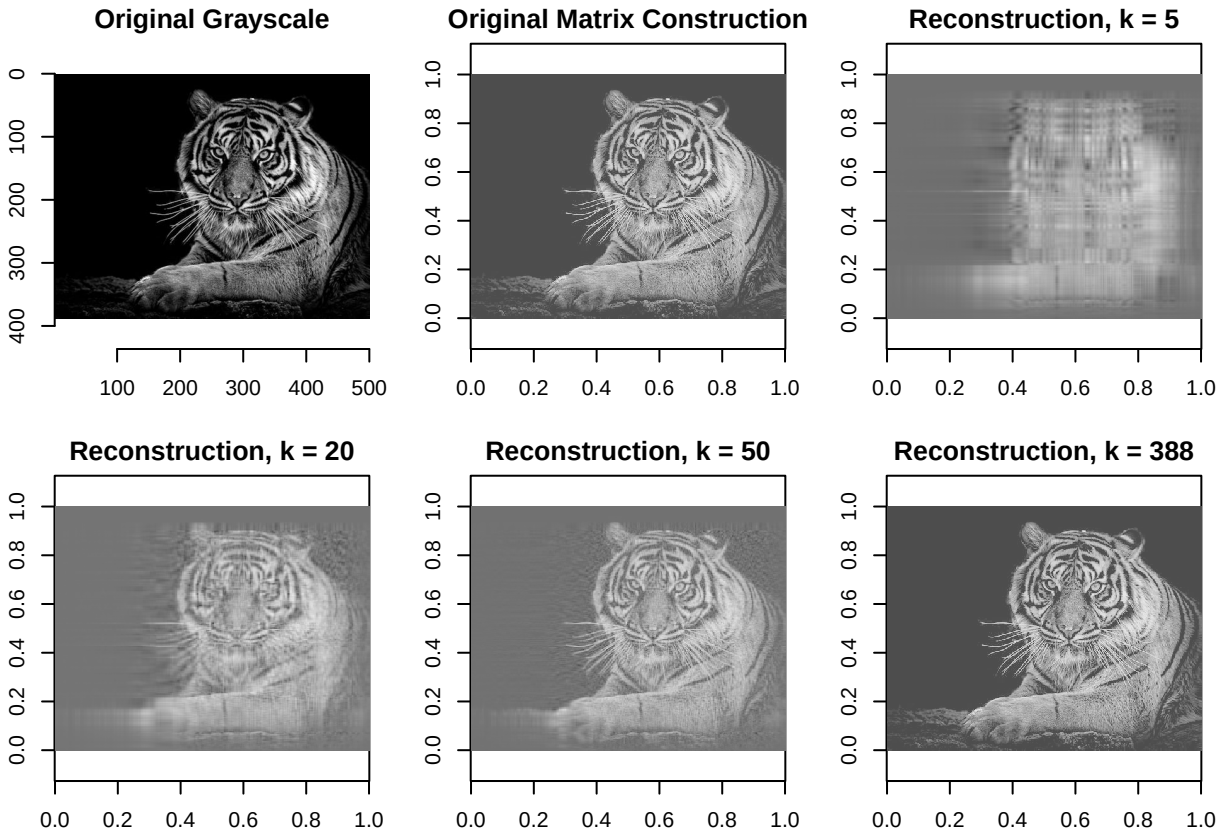
```
t_reconstructed <- function(k){
  U <- t_svd$u[, 1:k]
  D <- diag(t_svd$d[1:k])
  V <- t_svd$v[,1:k]
  return(U %*% D %*% t(V))
}

dim(t_matrix)
```

```
## [1] 500 388
```

Visualize the Result: Plot the original image alongside the compressed versions for various values of k (e.g., k = 5, 20, 50).

```
t_plot <- function(ks){
  par(mfrow = c(2, 3), mar = c(3,3,2,1))
  plot(t_gray, main = "Original Grayscale")
  image(t_matrix[1:nrow(t_matrix), ncol(t_matrix):1], col = gray.colors(256), main = "Original Matrix C
  for (k in ks){
    t_compressed <- t_reconstructed(k)
    image(t_compressed[1:nrow(t_compressed), ncol(t_compressed):1], col = gray.colors(256), main = paste
  }
}

k_vals <- c(5, 20, 50, 388)
```

```
t_plot(k_vals)
```



3. Matrix Rank, Properties, and Eigenspace

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.

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

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

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(A)$rank

rank_A
```

```
## [1] 4
```

Rank of Matrix A: Here we see that the rank of matrix A is 4. This means that there are 4 linearly independent rows, which also means that there are 4 linearly independent columns. Thus, rank tells us the dimension of the row and column space. Since the rank is equal to the number of rows and the number of columns, then there are no linear dependencies among rows or columns. This means that no row can be expressed as a linear combination of the other rows and, likewise, no column can be expressed as a linear combination of the other columns.

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.

Max and Min Rank The maximum possible rank for such a matrix is n since the rank cannot be larger than either the number of rows or columns and therefore must have a max value of the smaller value. The minimum possible rank would be 1 since this is a non-zero matrix.

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

Proof: From the **Rank-Nullity Theorem**, $\forall A \in \mathbb{R}^{m \times n}$, the row rank is equal to the column rank. We also know that the dimension of a row space is given by the number of linearly independent rows in A and likewise for the column space. Since we know that the **rank** of such a matrix, A, is defined as the number of linearly independent rows or columns also, we can now say that the **rank** of A is the same as the **dimension** of the row space or the column space:

$$\text{rank}(A) = \dim\left(\text{Row}(A)\right) = \dim\left(\text{Col}(A)\right) \blacksquare$$

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?)

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

Rank and Row Reduction: The code below computes the rank of matrix B,

$$\text{rank}(B) = 1$$

and performs a row reduction on matrix B. We will see that the row reduction simplifies the matrix B down to

$$\text{RREF}(B) = \begin{pmatrix} 1 & 2.5 & 3.5 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix}$$

which shows one non-zero row. This non-zero row tells us that there is only one linearly independent row in the original matrix B. This verifies the rank of 1 that was computed.

Rank-Deficiency: We consider an $m \times n$ matrix, A, to be **rank deficient** if and only if its rank is less than the minimum of the number of its rows and columns

$$\text{matrix } A \text{ is } \textbf{rank deficient} \iff \text{rank}(A) < \min(m, n)$$

We must determine if this is true for matrix B. We know that

$$\text{rank}(B) = 1$$

and that it is a 3×3 matrix. So,

$$\min(3, 3) = 3$$

Since

$$1 < 3,$$

we have

$$\text{rank}(B) < \min(3, 3)$$

Therefore, matrix B is **rank deficient**. ■

```
library(reticulate)
library(pracma)
```

```
##
## Attaching package: 'pracma'
```

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

```
## The following objects are masked from 'package:magrittr':
##
##     and, mod, or
```

```
#create the matrix
B <- matrix(c(2,5,7,4,10,14,1,2.5,3.5), nrow = 3, ncol = 3, byrow = TRUE)
B
```

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

```
#find the rank of matrix B
rank_B <- qr(B)$rank
cat(paste("\nThe rank of matrix B is:", rank_B, "\n\n"))
```

```
##
## The rank of matrix B is: 1
```

```
#perform row reduction on matrix B
row_reduce_B <- rref(B)
row_reduce_B
```

```
##      [,1] [,2] [,3]
## [1,]    1  2.5  3.5
## [2,]    0  0.0  0.0
## [3,]    0  0.0  0.0
```

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.

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

Eigenvalues and Eigenvectors: The code below computes each for matrix A.

```
A <- matrix(c(3,1,2,0,5,4,0,0,2), nrow = 3, byrow = TRUE)
A
```

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

```
eigens <- eigen(A)

e_vals_A <- eigens$values
e_vecs_A <- eigens$vectors

e_vals_A
```

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

```
e_vecs_A
```

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

Results:

The **eigenvalues** of A are **5, 3, and 2**.

The **eigenvectors** are

$$\begin{bmatrix} 0.4472136 \\ 0.8944272 \\ 0 \end{bmatrix}, \quad \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix}, \quad \begin{bmatrix} -0.3713907 \\ -0.7427814 \\ 0.5570860 \end{bmatrix}$$

The Characteristic Polynomial of a square matrix is given by the following formula:

$$p(\lambda) = \det(A - \lambda I)$$

where $I =$ the identity matrix of same size as A.

However, since we already have the eigenvalues of A, we can use these as the roots of the characteristic polynomial and simplify to find it. The factored form will look like:

$$p(\lambda) = (5 - \lambda)(3 - \lambda)(2 - \lambda)$$

and the R code below uses the "polynom" package to expand $p(\lambda)$.

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

```
##
## Attaching package: 'polynom'
```

```
## The following object is masked from 'package:pracma':
##
##     integral
```

```
char_poly <- polynomial(c(5, -1))*polynomial(c(3, -1))*polynomial(c(2, -1))
char_poly
```

```
## 30 - 31*x + 10*x^2 - x^3
```

Results:

The **characteristic polynomial** for matrix A in factored form is

$$p(\lambda) = (5 - \lambda)(3 - \lambda)(2 - \lambda)$$

and from the output above, we see that the expanded form (in order of descending powers) is

$$p(\lambda) = -\lambda^3 + 10\lambda^2 - 31\lambda + 30$$

Linear Independence: To determine if the eigenvectors are linearly independent, we can use row reduction on the matrix formed by these vectors.

```
e_matrix_A <- cbind(e_vecs_A)

rref(e_matrix_A)
```

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

```
det(e_matrix_A)
```

```
## [1] -0.4982729
```

Results:

Since the **reduced row echelon form** of the 3×3 matrix A returns the identity matrix with the same dimensions, then A has full rank of 3, implying that all columns and thus **eigenvectors** of A are linearly independent. We can confirm this by finding the determinant of the matrix formed by the eigenvectors and seeing that it is non-zero. ■

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.

  Results: Since the eigenvectors are linearly independent, we *can* diagonalize matrix A. The diagonal matrix is formed using the eigenvalues of matrix A as the diagonal values and 0 for all other matrix entries. It is shown below:

  $$\begin{bmatrix} 5 & 0 & 0 \\ 0 & 3 & 0 \\ 0 & 0 & 2 \end{bmatrix}$$

  The matrix of eigenvectors that *diagonalizes* A is

  $$\begin{bmatrix} 0.4472136 & 1 & -0.3713907 \\ 0.8944272 & 0 & -0.7427814 \\ 0 & 0 & 0.5570860 \end{bmatrix}$$

- 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 $\mathbb{R}^3$?

Results: In general, eigen**vectors** indicate the linear paths that remain unchanged during a matrix transformation, whereas the eigen**values** represent the scaling factors corresponding to those paths or directions. In our example, we have the following:

$$\text{The } \textbf{first eigenvector} \begin{bmatrix} 0.4472136 \\ 0.8944272 \\ 0 \end{bmatrix} \text{ is stretched by a factor of } \textbf{5}, \text{ which is the } \textbf{first eigenvalue}.$$

$$\text{The } \textbf{second eigenvector} \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix} \text{ is stretched by a factor of } \textbf{3}, \text{ which is the } \textbf{second eigenvalue}.$$

$$\text{The } \textbf{third eigenvector} \begin{bmatrix} -0.3713907 \\ -0.7427814 \\ 0.5570860 \end{bmatrix} \text{ is stretched by a factor of } \textbf{2}, \text{ which is the } \textbf{third eigenvalue}.$$

4. Project: Eigenfaces from the LFW (Labeled Faces in the Wild) Dataset

Context:

Eigenfaces are a popular application of Principal Component Analysis (PCA) in computer vision. They are used for face recognition by finding the principal components (eigenvectors) of the covariance matrix of a set of facial images. These principal components represent the "eigenfaces" that can be combined to approximate any face in the dataset.

Task:

Using the LFW (Labeled Faces in the Wild) dataset, build and visualize eigenfaces that account for 80% of the variability in the dataset. The LFW dataset is a well-known dataset containing thousands of labeled facial images, available for academic research.

Instructions:

1. Download the LFW Dataset:

```
\begin{itemize}
  \item The dataset can be accessed and downloaded using the lfw module from the sklearn library in Pyt
  \item In this case, we'll use the lfw module from Python's sklearn library.
\end{itemize}
```

Setting up the Python Environment

```
options(repos = c(CRAN = "https://cran.rstudio.com/"))
install.packages('reticulate')
```

```
##
## The downloaded binary packages are in
##  /var/folders/q8/00jxpyg135x_d3mhbg8pylr40000gn/T//RtmpU0fqdM/downloaded_packages
```

```
py_install("scikit-learn")
```

```
## Using virtual environment '/Users/jadamesng/.virtualenvs/r-reticulate' ...
```

```
## + /Users/jadamesng/.virtualenvs/r-reticulate/bin/python -m pip install --upgrade --no-user scikit-lea
```

```
py_install("Pillow")
```

```
## Using virtual environment '/Users/jadamesng/.virtualenvs/r-reticulate' ...
```

```
## + /Users/jadamesng/.virtualenvs/r-reticulate/bin/python -m pip install --upgrade --no-user Pillow
```

```
py_install("matplotlib")
```

```
## Using virtual environment '/Users/jadamesng/.virtualenvs/r-reticulate' ...
```

```
## + /Users/jadamesng/.virtualenvs/r-reticulate/bin/python -m pip install --upgrade --no-user matplotlil
```

```
from sklearn.datasets import fetch_lfw_people
```

2. Preprocess the Images:

- 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.

```
import sklearn
from sklearn.datasets import fetch_lfw_people
import numpy as np
from PIL import Image
import matplotlib.pyplot as plt
from sklearn.decomposition import PCA

#Download the LFW dataset (resized to smaller size for computational efficiency)
lfw_people = fetch_lfw_people(min_faces_per_person=70, resize=1)

#Get the image data and target labels
#2D array: each face image
images = lfw_people.images
#Height and width of images
h, w = lfw_people.images.shape[1], lfw_people.images.shape[2]

#Convert to grayscale and resize to 64x64
grayscale_images = []
for img in images:
    #Normalize the pixel values to range [0, 255] and convert to uint8
    img = (img - img.min()) / (img.max() - img.min()) * 255
    img = img.astype(np.uint8)

    #Convert NumPy array to PIL Image
    pil_img = Image.fromarray(img)
    #Convert to grayscale ('L' mode)
    pil_img = pil_img.convert('L')
    #Resize to 64x64
    pil_img = pil_img.resize((64, 64), Image.Resampling.LANCZOS)
    #Convert back to NumPy array
    grayscale_images.append(np.array(pil_img))

#Convert the list of images into a NumPy array
grayscale_images = np.array(grayscale_images)
```

```
#Flatten each image into a vector
n_samples, h_new, w_new = grayscale_images.shape
flat_images = grayscale_images.reshape((n_samples, h_new * w_new))

#Print the shape of the flattened images (each row is a flattened 64x64 image)
print(f"Shape of the flattened image data: {flat_images.shape}")
```

```
## Shape of the flattened image data: (1288, 4096)
```

```
#Visualize some of the grayscale, resized images
fig, axes = plt.subplots(2, 5, figsize=(15, 5))
for i, ax in enumerate(axes.flat):
    ax.imshow(grayscale_images[i], cmap='gray')
    ax.set(xticks=[], yticks=[])
```

```
## <matplotlib.image.AxesImage object at 0x12ca65fa0>
## [[], []]
## <matplotlib.image.AxesImage object at 0x12c7b74f0>
## [[], []]
## <matplotlib.image.AxesImage object at 0x12c86a820>
## [[], []]
## <matplotlib.image.AxesImage object at 0x12c8a54f0>
## [[], []]
## <matplotlib.image.AxesImage object at 0x12c8e22b0>
## [[], []]
## <matplotlib.image.AxesImage object at 0x12c91d250>
## [[], []]
## <matplotlib.image.AxesImage object at 0x12ca74880>
## [[], []]
## <matplotlib.image.AxesImage object at 0x12c98d7c0>
## [[], []]
## <matplotlib.image.AxesImage object at 0x12c9ca5b0>
## [[], []]
## <matplotlib.image.AxesImage object at 0x12ca08370>
## [[], []]
```

```
plt.show()
```

Apply PCA:

- Compute the PCA on the flattened images.

- Determine the number of principal components required to account for 80

```
#Apply PCA on the flattened images
#Set the number of components (e.g., enough components to explain 80% of the variance)
pca = PCA(n_components=0.80, whiten=True)  # 80% of variance
pca.fit(flat_images)
```

```
## PCA(n_components=0.8, whiten=True)
```

```
#Transform the images into the PCA space
pca_images = pca.transform(flat_images)

#Retrieve the eigenfaces (principal components)
eigenfaces = pca.components_.reshape((-1, h_new, w_new))

#Print information about the PCA result
print(f"Original shape: {flat_images.shape}")
```

```
## Original shape: (1288, 4096)
```

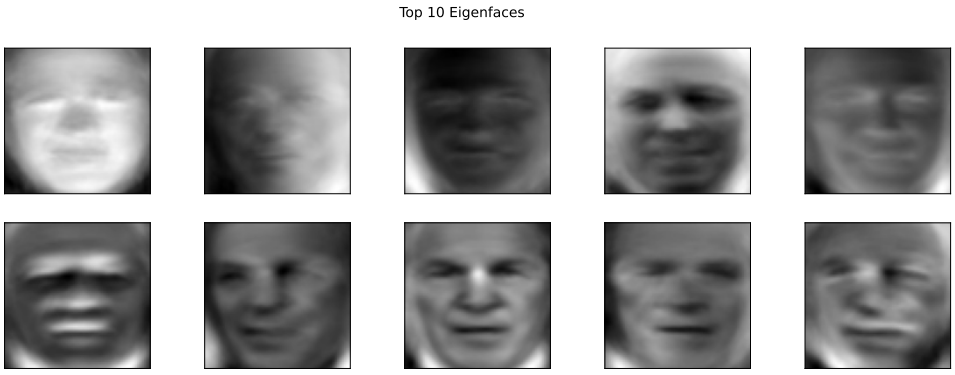
```
print(f"Reduced shape after PCA: {pca_images.shape}")
```

```
## Reduced shape after PCA: (1288, 42)
```

```
print(f"Number of eigenfaces: {eigenfaces.shape[0]}")
```

```
## Number of eigenfaces: 42
```

```
#Visualize some eigenfaces
fig, axes = plt.subplots(2, 5, figsize=(15, 5), subplot_kw={'xticks':[], 'yticks':[]})
for i, ax in enumerate(axes.flat):
    ax.imshow(eigenfaces[i], cmap='gray')
plt.suptitle('Top 10 Eigenfaces')
plt.show()
```

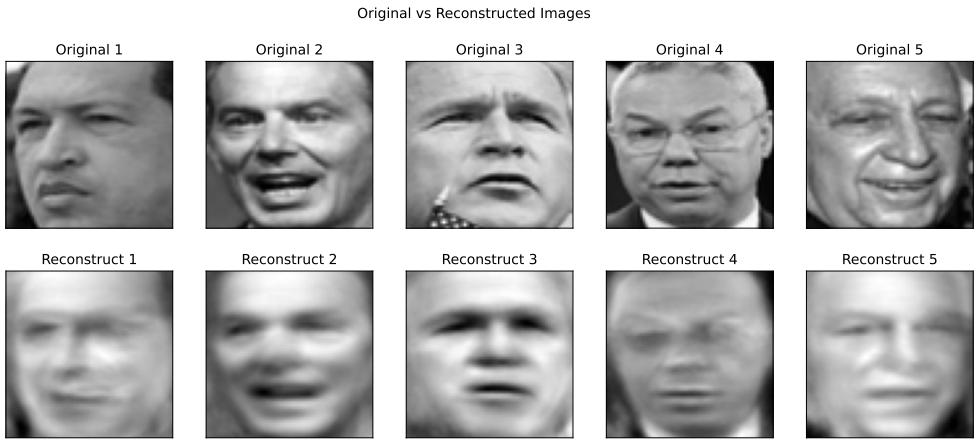


4. Visualize Eigenfaces:

- 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.

```
#Reconstruct some original images using the top eigenfaces
#Reconstruct the images from their PCA projection (approximation)
#Project back into original space
reconstructed_images = pca.inverse_transform(pca_images)
```

```python
reconstructed_images = reconstructed_images.reshape((n_samples, h_new, w_new))

#Compare original images with reconstructed images
#Number of images to show for comparison
n_images_to_show = 5
fig, axes = plt.subplots(2, n_images_to_show, figsize=(15, 6), subplot_kw={'xticks':[], 'yticks':[]})

for i in range(n_images_to_show):
    #Show original image
    axes[0, i].imshow(grayscale_images[i], cmap='gray')
    axes[0, i].set_title(f"Original {i+1}")

    #Show reconstructed image
    axes[1, i].imshow(reconstructed_images[i], cmap='gray')
    axes[1, i].set_title(f"Reconstruct {i+1}")

plt.suptitle('Original vs Reconstructed Images')
plt.show()
```



Results:

The original images are crisp and clear, showcasing a high level of detail in each face, allowing for easy recognition of the individuals. In contrast,the reconstructed images obtained through Principal Component Analysis (PCA) appear somewhat blurry, making it difficult to identify the individuals with certainty. However, they still retain some vague yet identifiable features. This is the essence of PCA. By reducing the dimensionality, it retains significant features from high-dimensional datasets in a lower-dimensional context. This not only decreases computational complexity, potentially reducing rendering times, but also makes it easier to interpret the data by minimizing 'noise' that may obscure recognizable patterns.