DATA 605 Homework 1

Kevin Havis

2025-02-16

```
library(matlib)
library(gganimate)
library(pracma)
```

Geometric transformation of shapes using matrix multiplication

> *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.*

We'll first start by identifying some common two dimensional transformations, such as rotation and shears. By representing these transformations as matrices, we'll be able to use any of these in our upcoming animation function.

```
# Rotate 90 degrees counter clockwise
rotate_90 <- matrix(c(1,0,
                      0,-1), nrow=2, byrow=TRUE)

# Apply a right-ward shear
shear <- matrix(c(1,0,
                  1,1), nrow=2, byrow=TRUE)

# Reflect across line of symmetry
mirror <- matrix(c(-1,0,
                   0,-1), nrow=2, byrow=TRUE)

# Reflect across y axis
reflect_y <- matrix(c(-1, 0,
                      0, 1), nrow=2, byrow=TRUE)
```

```
# Reflect across x axis
reflect_x <- matrix(c(1,0,
                      0,-1), nrow=2, byrow=TRUE)
```

We'll now build the transformation and animations. To properly animate, we need to interpolate through the transformation matrix, performing partial transformations until the entire transformation is complete.

We will do this by establishing a frame rate, looping through the transformation, and incrementing the transformation by the current frame for each loop. Mathematically, we are using the value of the current frame as a scalar by which we scale the transformation matrix, incrementing each loop by adding it back to the transformation matrix.

We calculate and store these intermediate values in a dataframe to make it easy to animate later.

```
interpolate_matrix_transformation <- function(vectors,
                                              frames,
                                              transformation_matrix){

  # Initialize an identity matrix to later interpolate
  # This way we can interpolate any transformation matrix we want
  i_matrix <- matrix(c(1, 0,
                0, 1), nrow=2, byrow=TRUE)

  # Init df to store transformed vectors
  transformed_vectors <- data.frame()

  # For each animation frame
  for (f in 1:frames){

    scale <- f / frames

    # progressively interpolate the identify matrix
    # and multiply by the transformation matrix
    interpolation_matrix <- (1 - scale) * i_matrix +
                            scale * transformation_matrix


    # Use the now interpolated transformation matrix to calculate new points
    transformed_shape <- vectors %*% interpolation_matrix
```

```
    # Get the new vectors and the frame they were created on
    step_vectors <- as.data.frame(transformed_shape)
    step_vectors$frame <- f

    # Append dataframe
    transformed_vectors <- rbind(transformed_vectors, step_vectors)

  }
  # names the dataframe columns
  colnames(transformed_vectors) <- c('x', 'y', 'frame')
  return(transformed_vectors)
}
```

Now with our function defined, we'll define a "square" as a matrix of vector points and build the dataframe containing all the incremental vector values we'll need in order to animate.

```
# Create a "square"
square <- data.frame(x=c(-1, 1, 1, -1),
                     y=c(-1, -1, 1, 1))
# Convert to matrix
m <- as.matrix(square)

# Set animation params
frames <- 60
scale <- seq(1, -1, length.out = frames)

# Build dataframe of interpolated vectors
transformed_vectors <- interpolate_matrix_transformation(m, frames, reflect_y)
```

Finally, we'll plot a simple x-y coordinate system to display the transformation upon.

```
# Build coordinate plot
# and run the animation

p_anim <- ggplot(transformed_vectors, aes(x=x, y=y)) +
    geom_point(size = 4, aes(color=frame), shape = "square") +
    geom_hline(yintercept = -10:10, color = "lightgray", linetype = "dotted") +  # Grid lines
    geom_vline(xintercept = -10:10, color = "lightgray", linetype = "dotted") +
    geom_hline(yintercept = 0, color = "black", linetype = "solid", linewidth = 1) +  # X-axi
    geom_vline(xintercept = 0, color = "black", linetype = "solid", linewidth = 1) +  # Y-axi
    scale_x_continuous(breaks = -10:10) +
```

```
    scale_y_continuous(breaks = -10:10) +
    coord_cartesian(xlim = c(-10, 10), ylim = c(-10, 10)) +
    theme_minimal() +
    theme(panel.grid = element_blank()) +
    transition_states(frame, transition_length = 1, state_length = 1)

# The GIF will not render in pdf
# print(p_anim)
```

We can now view the animation in its entirely - a few still frames for PDF format below.

See the full animation on my GitHub here.

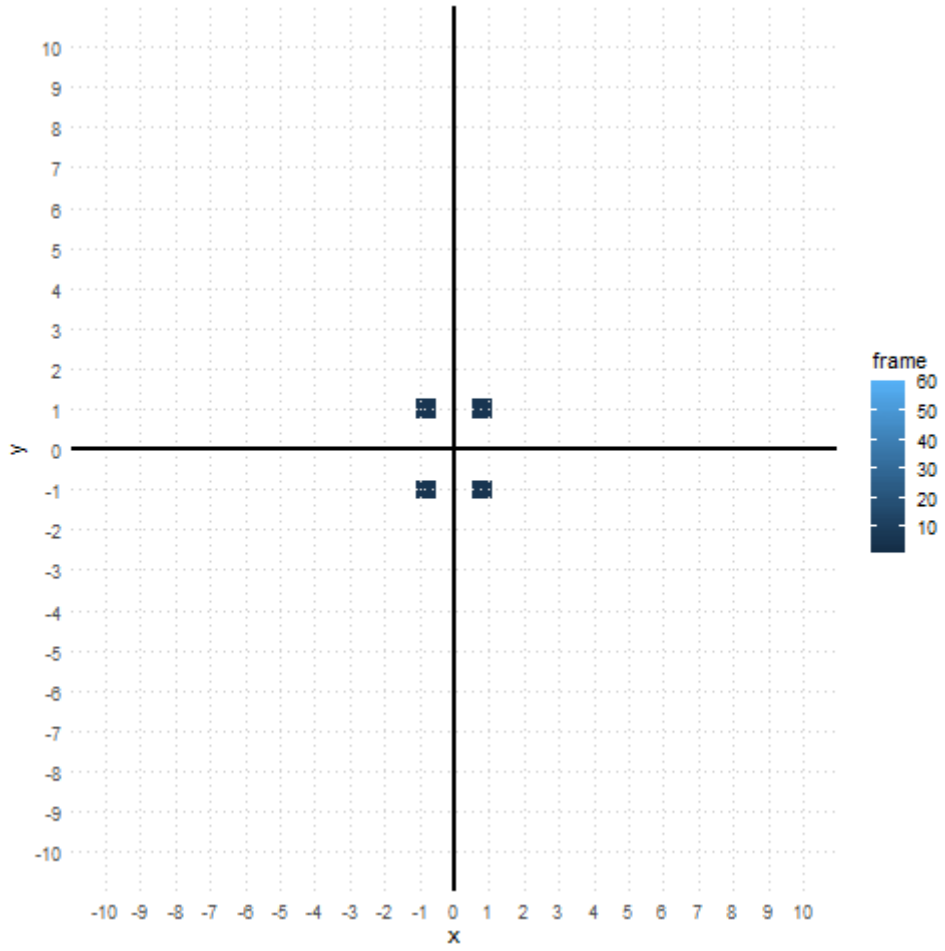


Figure 1: Frame 1

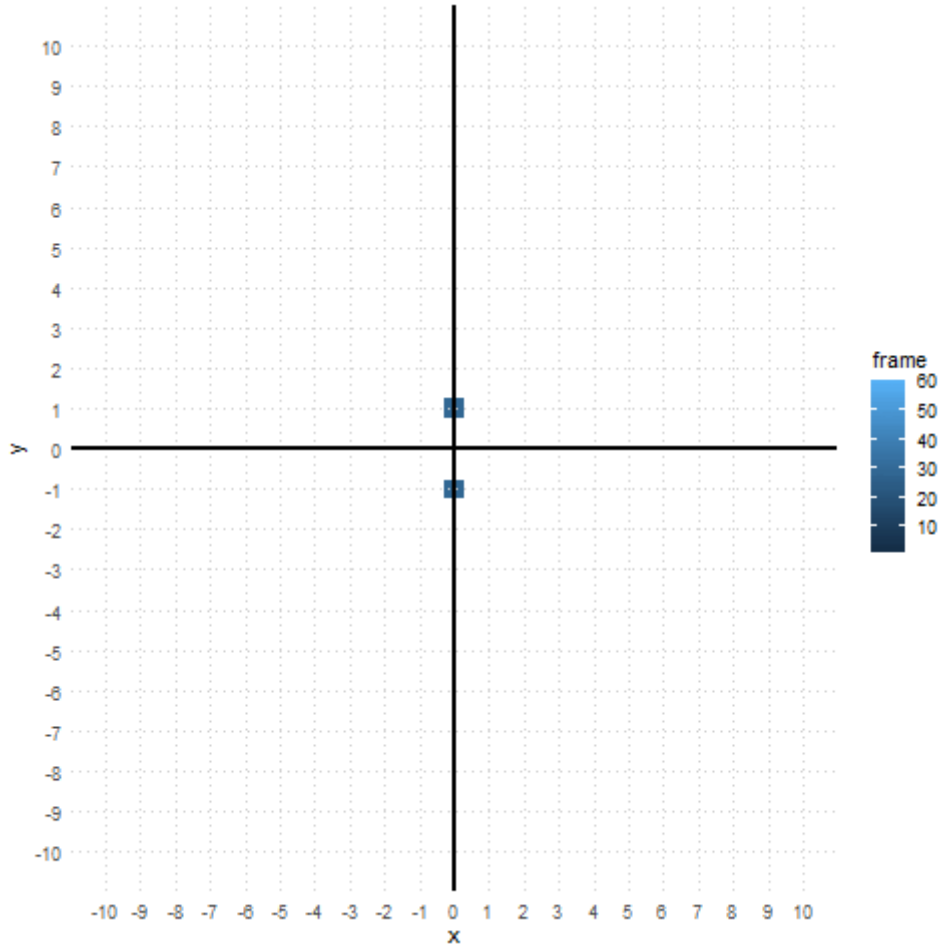


Figure 2: Frame 30

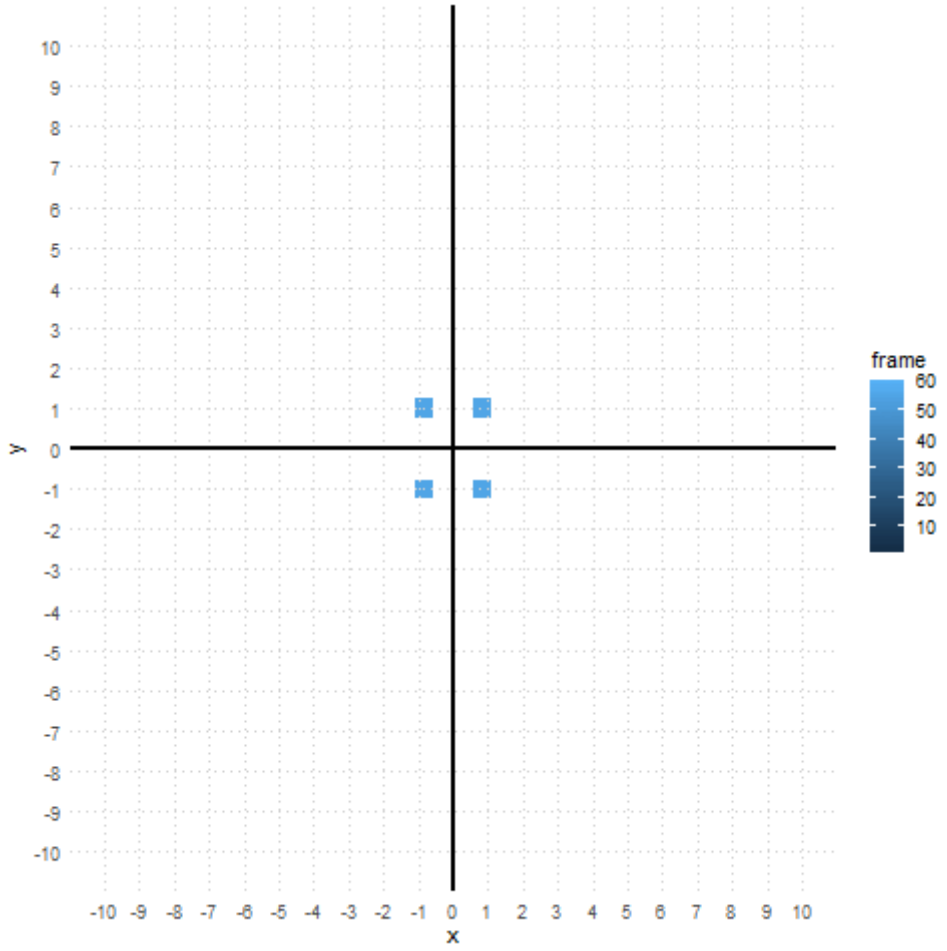


Figure 3: Frame 60

Matrix properties and decomposition

Prove that $AB \neq BA$

Matrices do not enjoy the commutative multiplicative property, unlike arithmetic algebra. We can easily prove that $AB \neq BA$ empirically with a few lines of R code.

```
A <- matrix(c(1,2,3,
              4,5,6,
              7,8,9), nrow=3, byrow=TRUE)
```

```r
B <- matrix(c(1, 1, 1,
              2, 2, 2,
              3, 3, 3), nrow=3, byrow=TRUE)

AB <- A%*%B

BA <- B%*%A

print(AB)
```

```
     [,1] [,2] [,3]
[1,]   14   14   14
[2,]   32   32   32
[3,]   50   50   50
```

```r
print(BA)
```

```
     [,1] [,2] [,3]
[1,]   12   15   18
[2,]   24   30   36
[3,]   36   45   54
```

```r
print(all.equal(AB, BA))
```

```
[1] "Mean relative difference: 0.1527778"
```

Prove that A^TA is always symmetric

A transpose of a matrix of rows m and columns n is simply re-ordering a matrix such that A^T is composed of n rows and m columns. As such, the matrix maintains symmetry *with respect to the main diagonal*.

Again, we can show this using a concrete example.

```r
# Initialize a matrix
A <- matrix(c('a', 'b', 'c',
              'd', 'e', 'f',
              'g', 'h', 'i'), nrow=3, byrow=TRUE)

# transpose
print(t(A))
```

```
     [,1] [,2] [,3]
[1,] "a"  "d"  "g"
[2,] "b"  "e"  "h"
[3,] "c"  "f"  "i"
```

As we can see, the main diagonal of a, e, i is retained, demonstrating symmetry throughout the transpose.

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

The determinant is the scaling factor by which vector space is expanded after a matrix transformation. A negative determinant would indicate a reflection, meaning vector space is "stretched" towards either or both the opposite x and y axis (given a standard two dimensional coordinate system).

Considering a two dimensional square matrix, $A = \begin{bmatrix} a & b \\ c & d \end{bmatrix}$, we can calculate the product of its transpose $A = \begin{bmatrix} a & c \\ b & d \end{bmatrix}$ as $A^T A = \begin{bmatrix} a^2 + c^2 & ab + cd \\ ab + cd & b^2 + d^2 \end{bmatrix}$

When we express this as as the determinant, we end up simply with $det_A = (ad - bc)^2$.

Since we are taking the square, the value of the determinant must be non-negative.

This is demonstrated in code below.

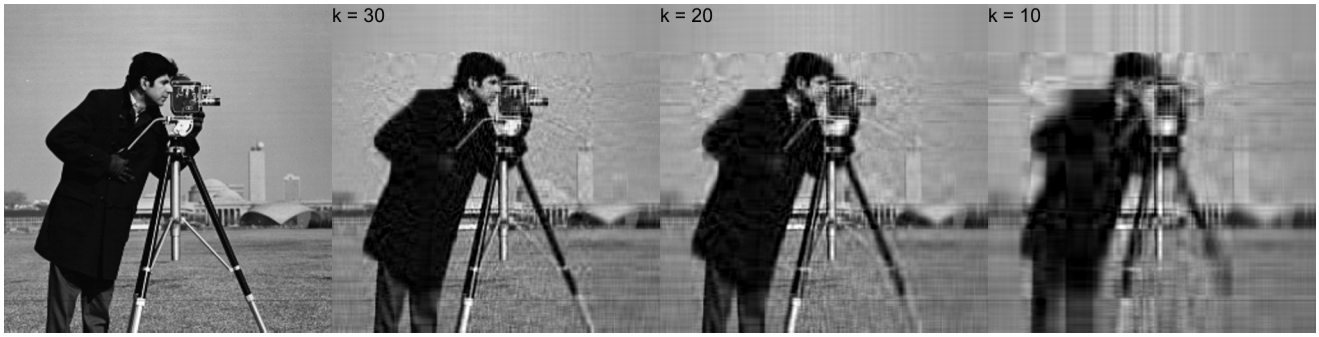
```r
# Initiate matrix
A <- matrix(c(1, 2,
              3, 4), nrow=2, byrow=TRUE)

# Transpose
A_t <- t(A)

# Show the transpose determinant
print(det(A_t %*% A))
```

```
[1] 4
```

Image Singular Value Decomposition

We can use singular value decomposition, or SVD, to compress images. This works by analyzing the image as a matrix of pixels, each with a value of saturation. In the case of a color image, you would also have red, green, and blue matrices, but we will work with a simpler greyscale image in this example.

```
library(magick)
options(magick.info = FALSE)

# Read a grayscale image
image <- image_read('image.png')
grey_image <- image_convert(image, colorspace = 'gray')

# Convert to vector
image_matrix <- as.numeric(image_data(grey_image))

# Set matrix dimension to those of the image
dim(image_matrix) <- c(image_info(grey_image)$width, image_info(grey_image)$height)

# Factorize the image matrix A using svd()
svd_matrix <- svd(image_matrix)

# Set up k values to test
k_values <- c(30, 20, 10)
compressed_images <- list()

# Loop through k values to create compressed images
for(k in k_values) {
    compress_matrix <- svd_matrix$u[, 1:k] %*% diag(svd_matrix$d[1:k]) %*% t(svd_matrix$v[,
    compress_matrix_3d <- array(compress_matrix, dim = c(dim(compress_matrix)[1], dim(compres
    img <- image_read(compress_matrix_3d)
    img <- image_annotate(img, paste("k =", k), size = 30, color = "red")
    compressed_images[[length(compressed_images) + 1]] <- img
}

# Combine all images including original
all_images <- c(grey_image, compressed_images)

# Turn off metadata display and show combined images
image_append(all_images)
```



With increasing levels of k, we see that we retain more and more information in the compressed image.

Matrix rank, properties, and eigenspace

Determine the rank of the given matrix

$$\textit{Find the rank of matrix } A \; A = \begin{bmatrix} 2 & 4 & 1 & 4 \\ -2 & -3 & 4 & 1 \\ 5 & 6 & 2 & 8 \\ -1 & -2 & 3 & 7 \end{bmatrix}$$

Note: we assume A is a coefficient matrix, not an augmented matrix in this example.

In linear algebra, **rank** refers to the number of dimensions affected by a matrix transformation. If a matrix transformation retains all dimensions of a vector space, it is said to be **full rank**.

A lesser rank occurs when a matrix transformation forces vector space into a smaller dimension; for example, a cube to a plane, a plane to a line, or a line to a point.

We calculate rank by first performing Gaussian elimination to get the matrix into reduced row echelon form. From there, the number of pivot columns is equivalent to the rank, as each pivot column represents linear independence for that given dimension.

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

A_rref <- echelon(A)

print(A_rref)
```

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

In this case, we can say that the rank of A is four given each column is a pivot column.

> *Explain what the rank tells us about the linear independence of the rows and columns of A*

A is full rank. As mentioned above, we have appropriate pivot values in each of our dimensions, which tells us the system is full rank and therefore linearly independent.

> *Identify any linear dependencies among the rows or columns*

If we saw any rows or columns that were strictly all zeroes, we could consider them to be free variables, and as such, linearly dependent.There are none in this case.

Matrix rank boundaries

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

We cannot have a higher rank than we have dimensions, so the maximum rank is $r_{max} \leq n$

If the matrix is non-zero, the smallest it could possibly be is 1x1. Therefore, minimum rank must also be $r_{min} = 1$.

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

Given a linear system of equations LS represented by matrix A such that $LS(A)$;

1. A homogeneous system $LS(A, 0)$ is always consistent
2. The trivial (0) solution to $LS(A, 0)$ is unique
3. The reduced row echelon form B of an augmented matrix A is row equivalent
4. B has pivot columns r which must be less than or equal to the number of columns n
5. $r = n$ when the system has a unique solution
6. $LS(A, 0)$ is consistent and has a unique solution, so for matrix A, $r = n$
7. Rank of A $r(A)$ is equal to r, so $r(A) = r_A = A_n$

8. The transpose A^T of matrix A has rows m and columns n such that $A_m^T = A_n$
9. The rank r of a matrix is equal to the rank of the transpose, so $r_A = r_{A^T}$
10. $r_A = A_m$, or $r_A = A_n$

Example given $4x_1 + 2x_2 = 0$ and $3x_1 + x2 = 0$

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

# Compute RREF form B
B <- echelon(A)

# The number of pivot columns is X
print(B)
```

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

```
# Rank is equal to the number of rows
rank <- nrow(A)

# Transpose of A
A_t <- t(A)

# Assert r is equal to rows
print(identical(rank,nrow(A_t)))
```

```
[1] TRUE
```

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

We can deduce the rank of B by analyzing the number of dependent D and free F variables.

By reducing matrix B, we can examine its pivot columns.

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

# Perform row operations to achieve reduced row echelon form
B_rref <- echelon(B)

print(B_rref)
```

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

In this case, we can see that the first and third columns have pivot values, while the second does not. This means the rank of B is limited to two. Given B is a 3x3 matrix, we can say that with a rank of two, it is rank-deficient (as opposed to full rank).

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.*

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

The roots of the characteristic polynomials of matrix A will be our eigenvalues λ. We can then solve the system of equations for each value of λ to calculate the eigenvectors, the span of which make up our eigenspace.

Detailed steps below. We will use the `pracma` and `matlib` packages to help with the computations.

Steps: 1. Determine characteristic polynomial as $p_A(\lambda) = det(A - \lambda I_3)$

2. Solve the roots of the polynomial for all values of λ
3. Note the algebraic multiplicity of the characteristic polynomial
4. Per value of λ, calculate $(A - \lambda I)\vec{v} = 0$
5. Row reduce each $(A - \lambda I)$ to help find the free variables
6. Solve the row reduced matrix as a system of equations to determine an eigenvector by setting up $(A - \lambda I)\vec{v} = \vec{0}$
7. Repeat for each value of λ; the span of the resulting eigenvectors describe eigenspace
8. Test if eigenspace is linearly independent; if all our eigenvalues are distinct, we can say they are linearly independent. If we have repeated eigenvalues, we should test by confirming the determinant of the eigenvectors is non-zero
9. Check the dimensions of eigenspace todetermine geometric multiplicity

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

# Step 1 characteristic polynomial
A_cp <- charpoly(A)
print(A_cp)
```

```
[1]   1 -10  31 -30
```

```
# Step 2 Eigenvalues as roots of the CP
lambdas <- roots(A_cp)
print(lambdas)
```

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

```
# Step 3 Algebraic multiplicity
alpha <- length(unique(lambdas))
print(alpha)
```

```
[1] 3
```

```r
# Step 4 Calculate eigenvectors per eigen value
# Note we are skipping to directly use the eigen() function for easier computation
# but we can easily confirm we have the same eigenvalues
eigen_A <- eigen(A)
print(eigen_A$values) # 5, 3, and 2, same as our polynomial roots
```

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

```r
# Step 5, 6, and 7 are done for us
eigen_vec <- eigen_A$vectors
print(eigen_vec)
```

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

```r
# Step 8 Test for linear independence by checking if any duplicates
print(duplicated(eigen_A$values)) # No duplicates so linearly independent
```

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

```r
# Can validate by asserting the determinant of eigenspace is not zero
eigen_det <- det(eigen_vec)
print(eigen_det)
```

```
[1] -0.4982729
```

```r
# Step 9 The number of eigenvectors is our geometric multiplicity
gamma <- ncol(eigen_vec)
print(gamma)
```

```
[1] 3
```

- Characteristic polynomial: $x^3 - 10x^2 + 31x - 30$
- $\lambda = (5, 3, 2)$
- $\lambda\vec{x} = \begin{bmatrix} 0.4472136 & 1 & -0.3713907 \\ 0.8944272 & 0 & -0.7427814 \\ 0 & 0 & 0.5570860 \end{bmatrix}$

- Algebraic multiplicity = 3
- Determinant of eigen matrix = -0.49827 (i.e. non-zero)
- Geometric mutiplicity = 3

With our results, we can confirm our eigenvectors are linearly independent, as evidenced by the non-zero determinant of eigenspace as well as the distinct eigenvectors themselves.

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.*

As a square matrix, A is considered diagonalizable if it is similar to a diagonal matrix. We would like to find some non-singular matrix S that allows us to diagonlize A such that $A_{diag} = S^{-1}AS$.

In this case, we can use our already-calculated eigenvectors to compose matrix S.As we've concluded that our eigenvectors are independent, we can also state they are non-singular, and thus are appropriate to use for S.

```
# Initiate S as matrix of eigenvectors
S <- eigen_vec

# Calculate diagonal matrix of A
A_diag <- solve(S) %*% A %*% S

print(A_diag)
```

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

As we can see, A is indeed diagonalizable, giving us a lovely diagonal matrix with our eigenvalues as the main diagonal. Beautiful!

> *Discuss the geometric interpretation of the eigenvectors and eigenvalues in the context of transformations. For instance, how does matrix stretch, shrink, or rotate vectors in $\mathbb{R}^3$?*

Eigenvectors represent the span that is unaffected by a geometric transformation. Consider a three dimensional rotation in any x, y, or z direction; the axis of that rotation is itself the span of the eigenvectors. This is why we calculate eigenvalues using a zero determinant.

Some transformations result in all of vector space becoming the eigenspace, such as a streching or shrinking. In these cases, vectors are simply being multiplied by a scalar, which leaves their span unaffected. As such, the span of vector space is the same both before and after the scalar transformation.

In this way, eigenvalues can be thought of as "summaries" of a matrix transformation.

Project: Eigenfaces from the "Labeled Faces in the Wild" dataset

> *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.*
>
> *Apply PCA: Compute the PCA on the flattened images. Determine the number of principal components required to account for 80% of the variability.*
>
> *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.*

I sampled four images from the dataset due to drive space, but the below code can be scaled to fit any number of images.

We are using the `magick` package again to help with the image processing. We read each image in the directory, preprocess it into greyscale and downscale to 64x64, then store the numerical representations in a list. Then it is simple to use the `prcomp` package to calculate the PCA summary for each image.

During my first attempts, I realized that some images had zero pixels (solid backgrounds being the suspected culprit). This meant we needed to perform some preprocessing to remove null elements so we could perform the PCA reliably.

This is necessary, as during the covariance matrix computation, you'll get zero variance which causes issues with the standardization for the rest of the PCA computation. I used Claude AI to help refactor my code so that we could dynamically and elegantly handle images that had this issue.

```
library(magick)

# Read all iamges
image_vector_list <- list.files(path = "lfw/",
```

```
                                pattern = "\\.(jpg)$",
                                full.names = TRUE) |>
  lapply(function(file) {
    # Read each image
    image_read(file) |>
    # Convert to greyscale
    image_convert(colorspace = 'gray') |>
    # Rescale image
    image_scale('64') |>
    # Get numeric image data
    image_data() |>
    as.numeric()
  })
```

```
# Perform PCA on each vector
pca_list <- lapply(image_vector_list, function(image_vec) {
    # Reshape vector to matrix
    mat <- matrix(image_vec, nrow = 64, ncol = 64)

    # Check for columns with non-zero variance, otherwise we get error
    var_cols <- apply(mat, 2, var) > 0

    # If there are any columns with variance
    if(any(var_cols)) {
        # Only perform PCA on columns with variance
        mat_filtered <- mat[, var_cols, drop = FALSE]
        pca_result <- prcomp(mat_filtered, center = TRUE, scale. = TRUE)
        return(list(
            pca = pca_result,
            used_cols = which(var_cols)  # Keep track of which columns we used
        ))
    } else {
        # Return null if no variance in image as a check
        return(NULL)
    }
})

# Remove any null results (images where PCA couldn't be performed)
pca_list <- pca_list[!sapply(pca_list, is.null)]
```

With the PCA computed for each image, we can view the results and see for each how many principle components are need to explain more than 80% of the variability, by reviewing the

`Cumulative Proportion` of each. Considering the first image as an example, we can see that we need five principal components in order to get to our benchmark.

```
# The first image needs PC5
print(summary(pca_list[[1]]$pca))
```

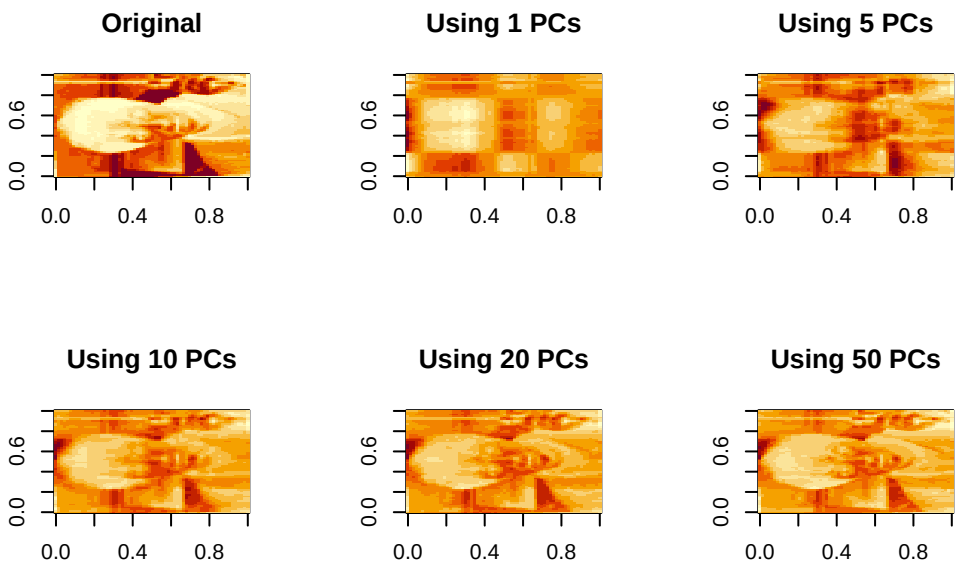
```
Importance of components:
                          PC1    PC2    PC3     PC4     PC5     PC6    PC7
Standard deviation     4.7700 3.7944 2.8403 2.16678 1.69458 1.66032 1.3718
Proportion of Variance 0.3555 0.2250 0.1261 0.07336 0.04487 0.04307 0.0294
Cumulative Proportion  0.3555 0.5805 0.7065 0.77989 0.82476 0.86783 0.8972
                           PC8     PC9    PC10    PC11    PC12    PC13    PC14
Standard deviation     1.09038 1.06778 0.94741 0.72527 0.65320 0.61815 0.54554
Proportion of Variance 0.01858 0.01781 0.01402 0.00822 0.00667 0.00597 0.00465
Cumulative Proportion  0.91581 0.93362 0.94765 0.95587 0.96253 0.96850 0.97316
                          PC15    PC16    PC17    PC18    PC19    PC20    PC21
Standard deviation     0.51554 0.48051 0.40999 0.37618 0.34293 0.32536 0.31293
Proportion of Variance 0.00415 0.00361 0.00263 0.00221 0.00184 0.00165 0.00153
Cumulative Proportion  0.97731 0.98092 0.98354 0.98575 0.98759 0.98924 0.99077
                          PC22   PC23    PC24    PC25    PC26    PC27    PC28
Standard deviation     0.26686 0.2652 0.24440 0.21932 0.20809 0.19989 0.18806
Proportion of Variance 0.00111 0.0011 0.00093 0.00075 0.00068 0.00062 0.00055
Cumulative Proportion  0.99189 0.9930 0.99392 0.99467 0.99535 0.99597 0.99652
                          PC29    PC30    PC31    PC32    PC33    PC34   PC35
Standard deviation     0.16910 0.16172 0.14539 0.13580 0.12474 0.12182 0.1134
Proportion of Variance 0.00045 0.00041 0.00033 0.00029 0.00024 0.00023 0.0002
Cumulative Proportion  0.99697 0.99738 0.99771 0.99800 0.99824 0.99847 0.9987
                          PC36    PC37    PC38    PC39    PC40    PC41    PC42
Standard deviation     0.10650 0.09845 0.09195 0.09062 0.08308 0.07624 0.07286
Proportion of Variance 0.00018 0.00015 0.00013 0.00013 0.00011 0.00009 0.00008
Cumulative Proportion  0.99885 0.99900 0.99913 0.99926 0.99937 0.99946 0.99954
                          PC43    PC44    PC45    PC46    PC47    PC48    PC49
Standard deviation     0.07055 0.06396 0.06152 0.05841 0.05134 0.04682 0.04246
Proportion of Variance 0.00008 0.00006 0.00006 0.00005 0.00004 0.00003 0.00003
Cumulative Proportion  0.99962 0.99969 0.99975 0.99980 0.99984 0.99987 0.99990
                          PC50    PC51    PC52    PC53    PC54    PC55    PC56
Standard deviation     0.03722 0.03353 0.03148 0.02692 0.02277 0.02065 0.01915
Proportion of Variance 0.00002 0.00002 0.00002 0.00001 0.00001 0.00001 0.00001
Cumulative Proportion  0.99992 0.99994 0.99996 0.99997 0.99998 0.99998 0.99999
                          PC57    PC58   PC59     PC60     PC61     PC62
Standard deviation     0.01529 0.01404 0.0104 0.009078 0.007451 0.003464
Proportion of Variance 0.00000 0.00000 0.0000 0.000000 0.000000 0.000000
```

```
Cumulative Proportion  0.99999 1.00000 1.0000 1.000000 1.000000 1.000000
                           PC63      PC64
Standard deviation     0.001674 2.091e-15
Proportion of Variance 0.000000 0.000e+00
Cumulative Proportion  1.000000 1.000e+00
```

Now to visualize how much information we get at each principal component, we will reconstruct the images at a series of n components.

```r
# Function to reconstruct image from n principal components
reconstruct_image <- function(pca_result, n_components) {
    # Get the eigenvectors for first n components
    eigenvec <- pca_result$pca$rotation[, 1:n_components, drop = FALSE]

    # Get the scores for first n components
    scores <- pca_result$pca$x[, 1:n_components, drop = FALSE]

    # Reconstruct using matrix multiplication
    reconstructed <- scores %*% t(eigenvec)

    # Un-scale and un-center (reverse the standardization)
    # Check if scale is TRUE/FALSE rather than a vector
    if(!is.null(pca_result$pca$scale.) && pca_result$pca$scale.) {
        reconstructed <- scale(reconstructed,
                               center = FALSE,
                               scale = 1/pca_result$pca$sdev[1:n_components])
    }

    # Add back the center
    if(!is.null(pca_result$pca$center)) {
        reconstructed <- t(t(reconstructed) + pca_result$pca$center)
    }

    # Create full-size image matrix
    full_image <- matrix(0, nrow = 64, ncol = 64)
    full_image[, pca_result$used_cols] <- reconstructed

    return(full_image)
}

# Get first image's PCA result
first_pca <- pca_list[[1]]
```

```
# Original image for comparison
original_mat <- matrix(image_vector_list[[1]], nrow = 64, ncol = 64)

# Set up plotting parameters
par(mfrow = c(2,3))

# Original
image(original_mat, main = "Original")

# Reconstructions with different numbers of PCs
for(n_pc in c(1, 5, 10, 20, 50)) {
    reconstructed <- reconstruct_image(first_pca, n_pc)
    image(reconstructed,
          main = paste("Using", n_pc, "PCs"))
}
```



Comparing this to the original image, we can see how the PCA helps us identify the most important pixel values to retain the most visual information possible. Thanks Zorica!

```
# A tibble: 1 x 7
  format width height colorspace matte filesize density
  <chr>  <int>  <int> <chr>      <lgl>    <int> <chr>
```

```
1 JPEG     250    250 sRGB       FALSE    10095 72x72
```

Conclusion

In this paper we've demonstrated our ability to leverage linear algebra and the constructs of matrices to do manty important data related tasks, from image processing to eigenspace calculations. In reflection, I at first found much of these calculations to be tedious and opaque, and not very interesting at all. However, pursuing these exercises, developing an understanding of the relationships between these concepts, and applying them in a context that demosntrates real practical value, I started to find them as puzzles that require solving - a far more satisfying prospect.

I now find a great deal of value in thinking of problems in the way linear algebra encourages; ignoring the "noise", focusing on the values that matter, and most important, the patterns we can recognize and reapply as we see them over and over again, and what that allows us to infer from them.