Homework 1: Part 1

DATA 605

Stephanie Chiang

Fall 2025

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.

- Here we define a simple right triangle:

```
triangle <- data.frame(
  x = c(0, 1, 0, 0),
  y = c(0, 0, 1, 0)
)

triangle
```

```
##   x y
## 1 0 0
## 2 1 0
## 3 0 1
## 4 0 0
```

Apply Transformations:

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

- Each of the first 3 functions below returns a matrix to be used as a multiplier on the original shape's points, resulting in transformations. The final function is a generic helper that applies the multiplication and returns the new point coordinates of the altered shape.

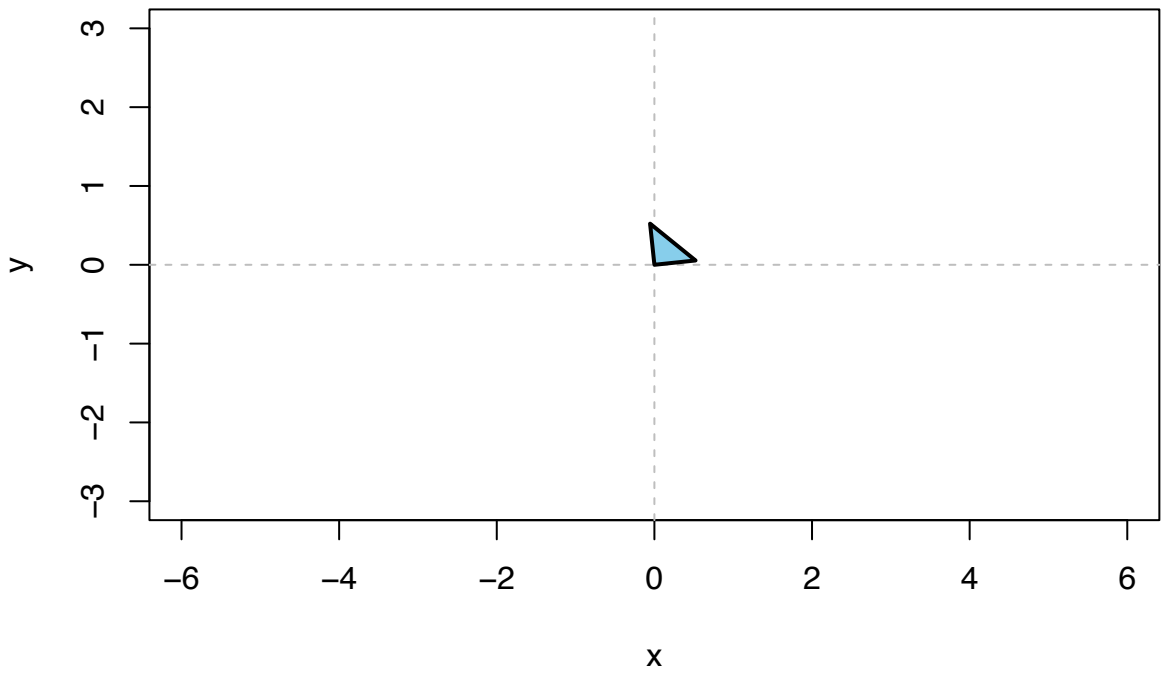
```
# function to return a matrix for resizing
scaling <- function(sx, sy) {
  matrix(c(sx, 0,
           0, sy), nrow = 2, byrow = TRUE)
}

# function to return a matrix for rotation
rotating <- function(theta) {
```

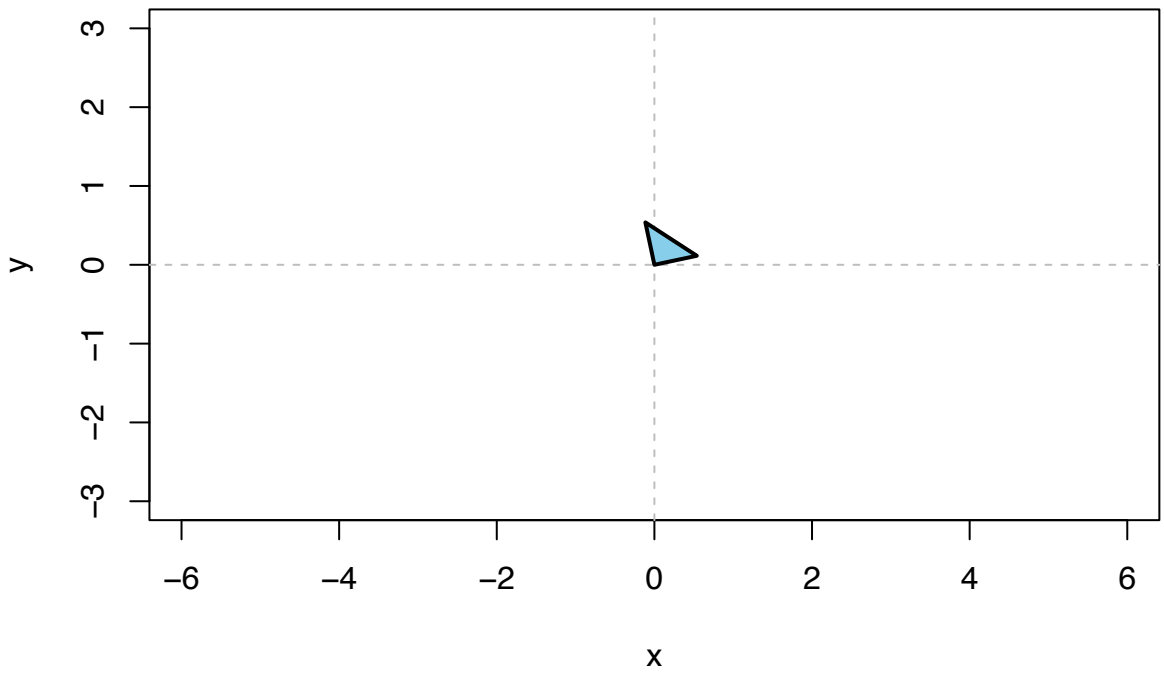
```
  matrix(c(cos(theta), -sin(theta),
           sin(theta),  cos(theta)), nrow = 2, byrow = TRUE)
}

# function to return a matrix for reflection
reflect_x <- matrix(c(1, 0,
                      0, -1), nrow = 2, byrow = TRUE)

# helper function
apply_transform <- function(shape, M) {
  coords <- as.matrix(shape[,c("x","y")])
  result <- t(M %*% t(coords))
  data.frame(x=result[,1], y=result[,2])
}
```

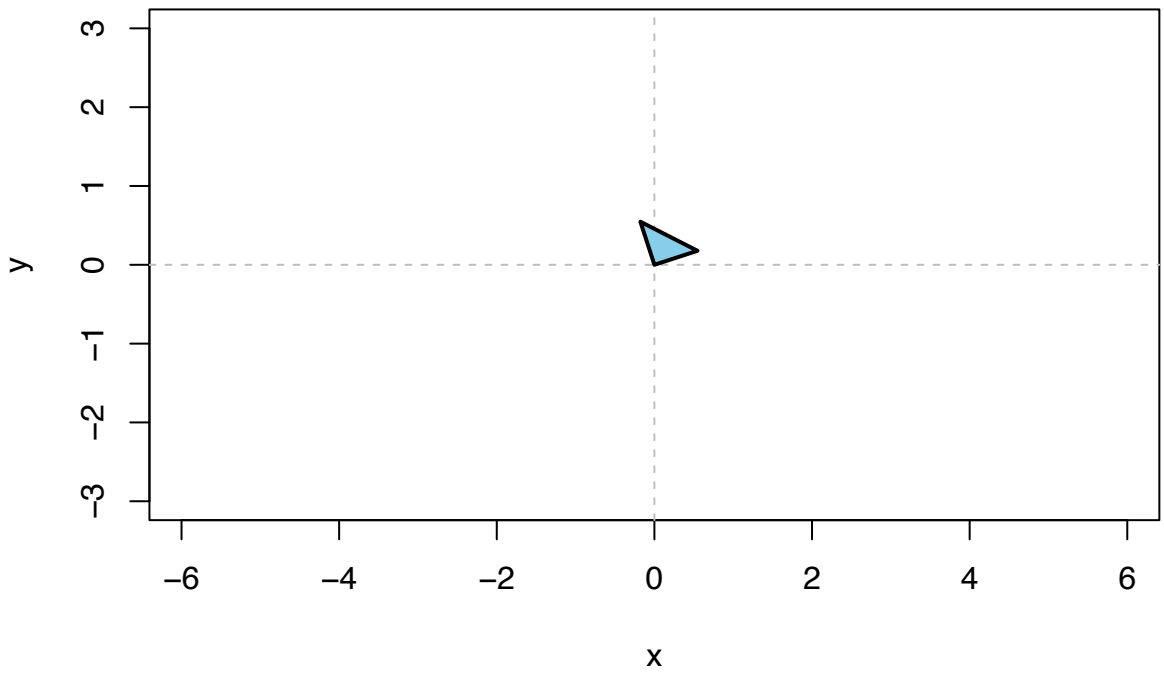
Animate Transformations: Use a loop to incrementally change the transformation matrix and visualize the effect on the shape over time.

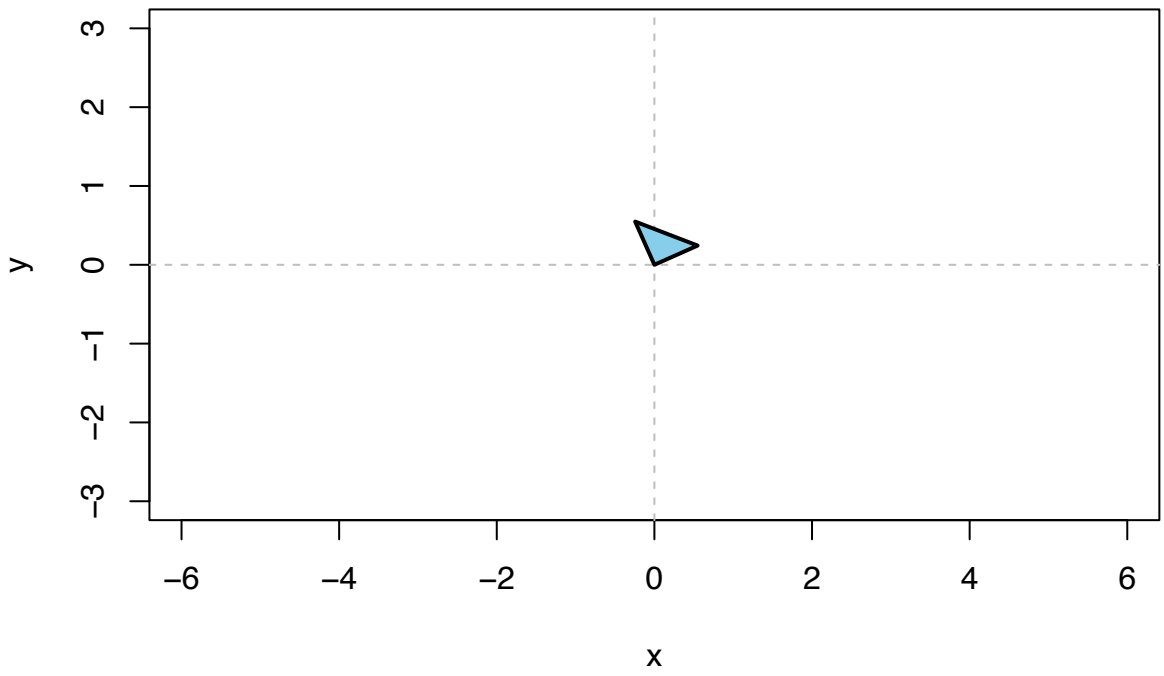
- The below loops 15 times (an arbitrary number), increasingly rotating and resizing the triangle and generating a new plot each loop for use as a frame in an animation:

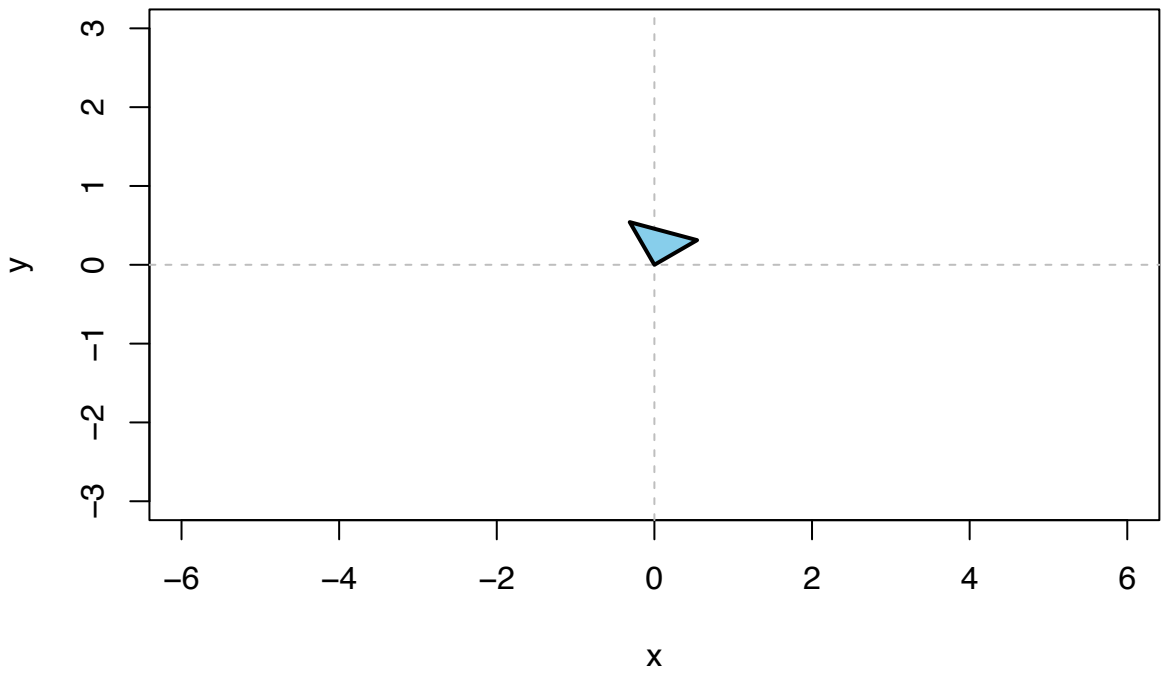
```
for (i in 1:5) {
  angle <- i * (2*pi/60)
  size <- 0.5 + (i/60)*1.5
  M <- rotating(angle) %*% scaling(size, size)
  new_tri <- apply_transform(triangle, M)

  plot(NULL, xlim=c(-3,3), ylim=c(-3,3), asp=1, xlab="x", ylab="y")
  abline(h=0, v=0, col="gray", lty=2)

  polygon(new_tri$x, new_tri$y, col="skyblue", border="black", lwd=2)
}
```


y
3
2
1
0
−1
−2
−3
−6
−4
−2
0
2
4
6
x


y
3
2
1
0
−1
−2
−3
−6
−4
−2
0
2
4
6
x

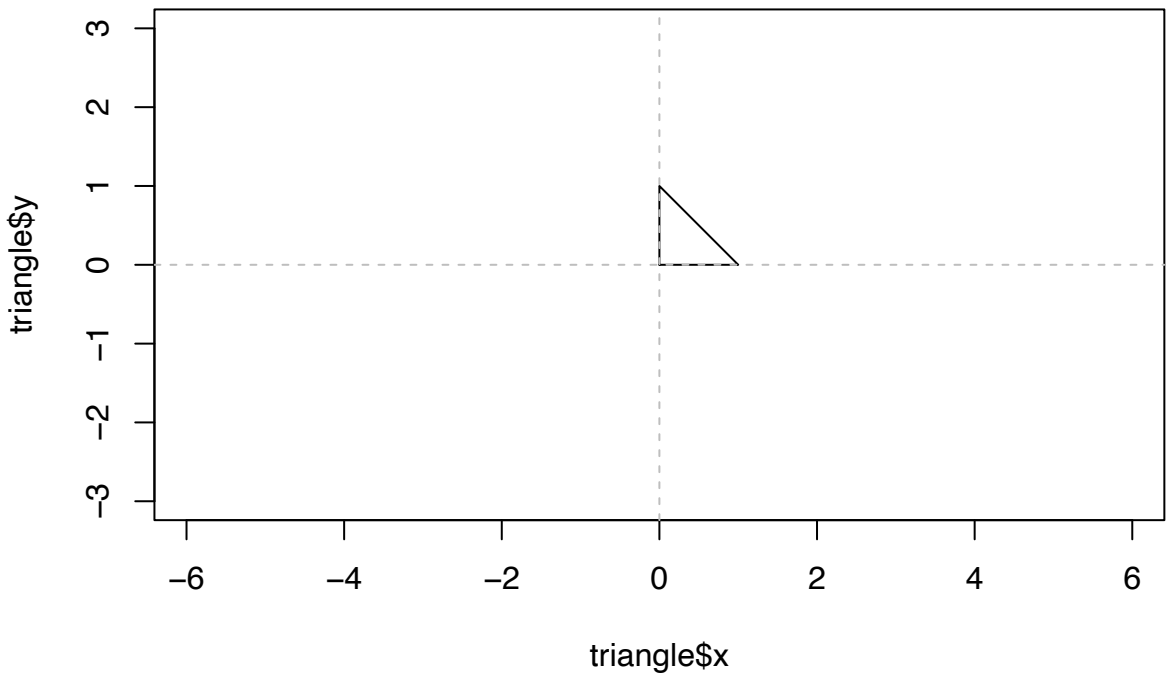

y
3
2
1
0
−1
−2
−3
−6
−4
−2
0
2
4
6
x


y
3
2
1
0
−1
−2
−3
−6
−4
−2
0
2
4
6
x

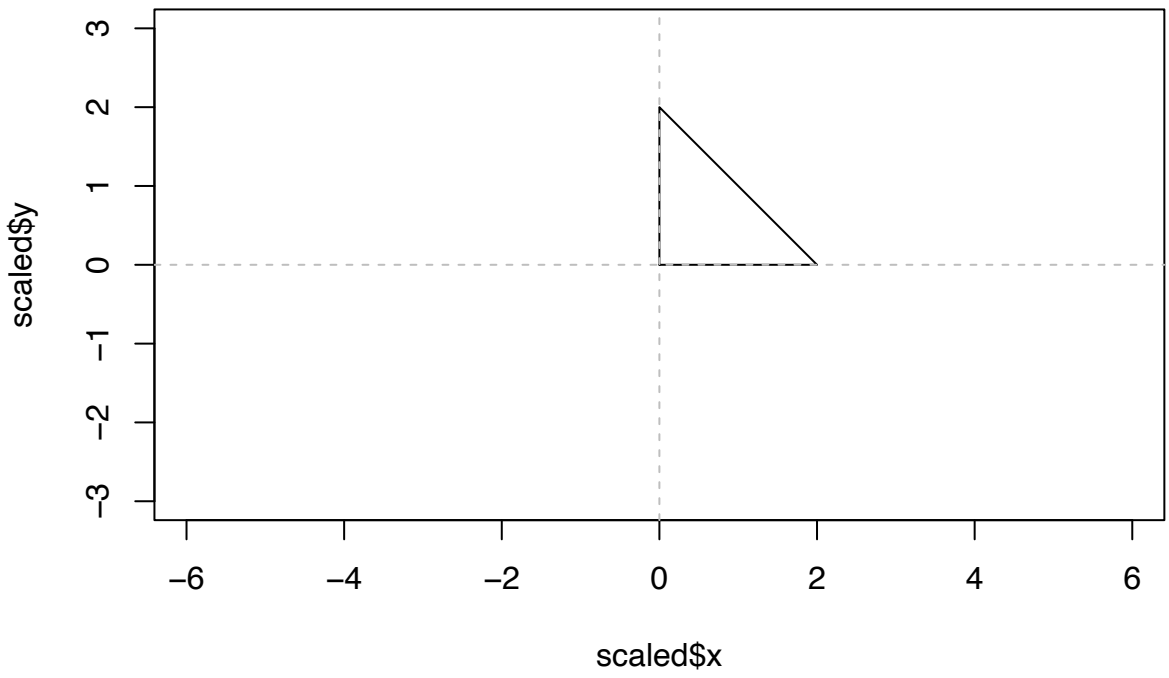


Plot: Display the original shape and its transformations in your compiled pdf. Demonstrate the effect of the transformations with fixed images.

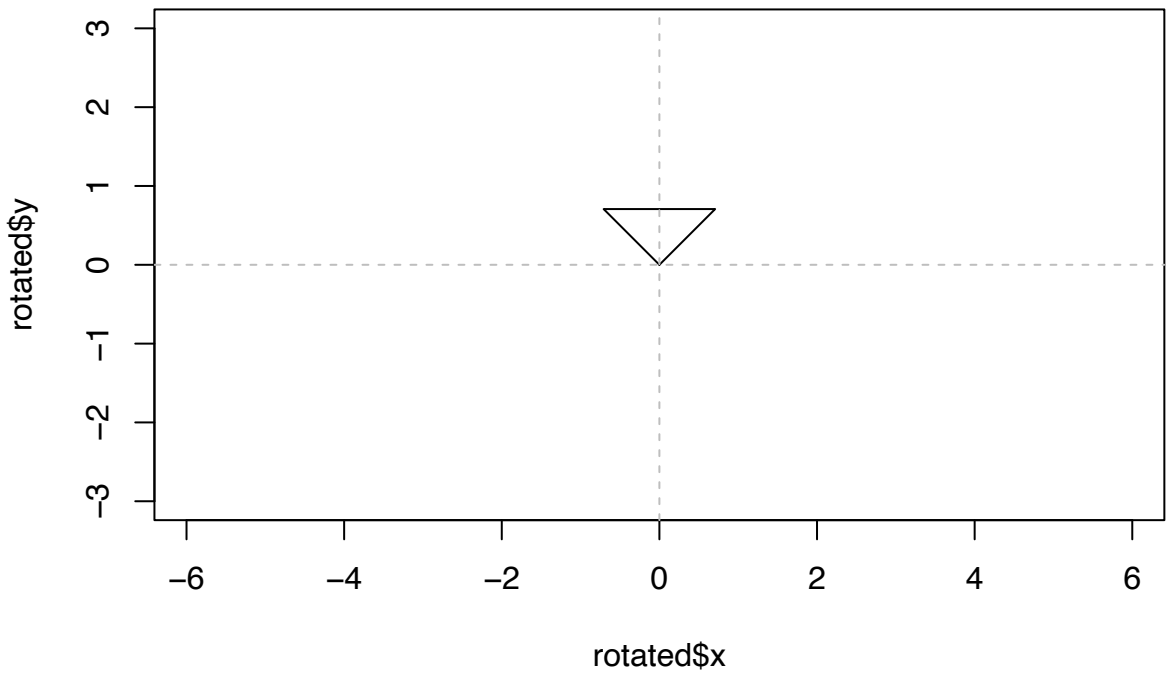
```
# original shape
plot(triangle$x, triangle$y, type = "l", asp = 1, xlim=c(-3,3), ylim=c(-3,3))
abline(h=0, v=0, col="gray", lty=2)
```



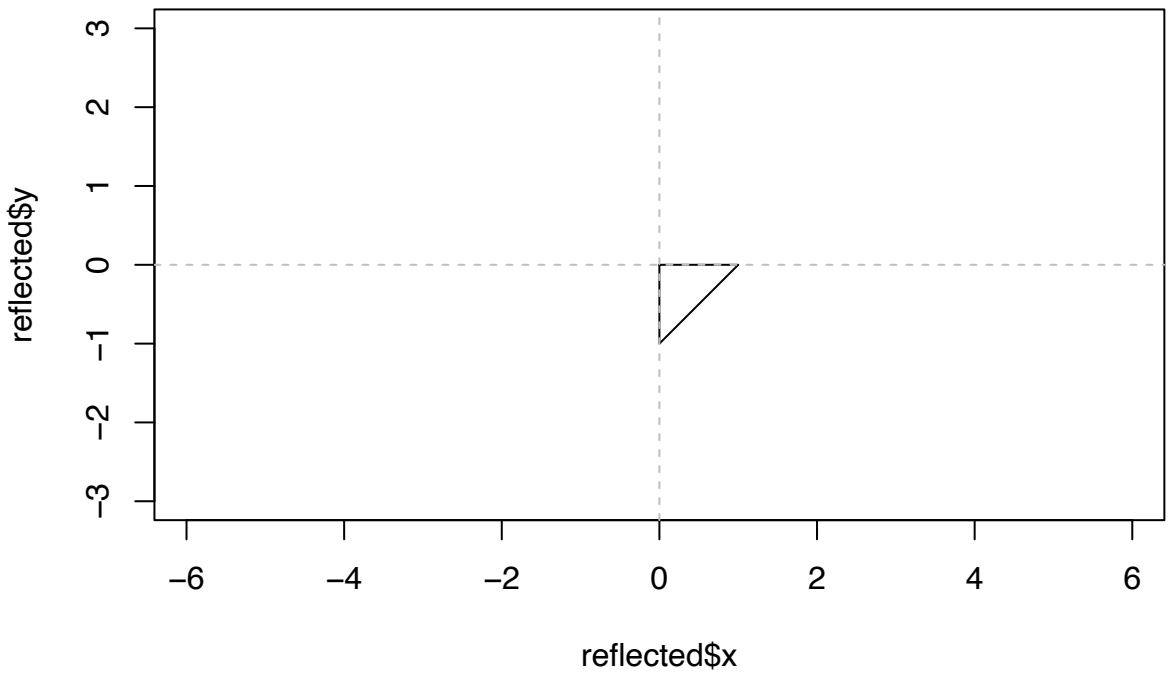

```
# scaled x 2 all around
scaled <- apply_transform(triangle, scaling(2, 2))

plot(scaled$x, scaled$y, type = "l", asp = 1, xlim=c(-3,3), ylim=c(-3,3))
abline(h=0, v=0, col="gray", lty=2)
```




```r
# rotated by 45°
rotated <- apply_transform(triangle, rotating(pi/4))

plot(rotated$x, rotated$y, type = "l", asp = 1, xlim=c(-3,3), ylim=c(-3,3))
abline(h=0, v=0, col="gray", lty=2)
```




```r
# reflected
reflected <- apply_transform(triangle, reflect_x)

plot(reflected$x, reflected$y, type = "l", asp = 1, xlim=c(-3,3), ylim=c(-3,3))
abline(h=0, v=0, col="gray", lty=2)
```



2. Matrix Properties and Decomposition

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.

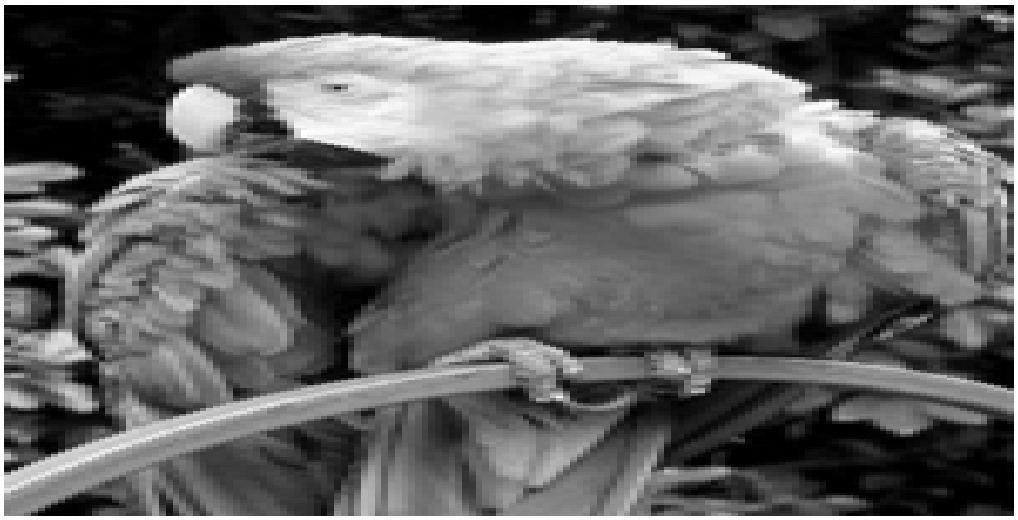
Instructions:

- Read an Image: Convert a grayscale image into a matrix.
- Perform SVD: Factorize the image matrix A using R's 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).

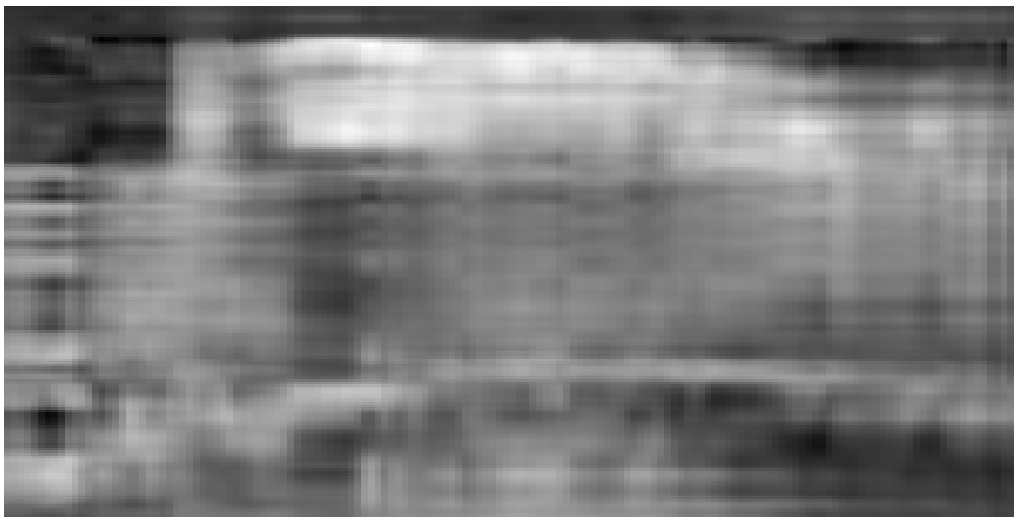
```
library(jpeg)
library(png)

# import a grayscale image as a matrix
url <- "https://upload.wikimedia.org/wikipedia/commons/f/fa/Grayscale_8bits_palette_sample_image.png"
download.file(url, destfile = "parrot.png", mode="wb", )
img <- readPNG("parrot.png")
```

```r
# factorize with svd()
result <- svd(img)

# compress using only the top k singular values and vectors
U <- result$u # left singular vectors
D <- diag(result$d) # singular values matrix
V <- result$v # right singular vectors

# helper function to reconstruct
reconstruct_image <- function(k, U, D, V) {
  U_k <- U[, 1:k]
  D_k <- D[1:k, 1:k]
  V_k <- V[, 1:k]
  U_k %*% D_k %*% t(V_k)
}

# test with k = 5, 20, or 100
img_k5 <- reconstruct_image(5, U, D, V)
img_k20 <- reconstruct_image(20, U, D, V)
img_k100 <- reconstruct_image(100, U, D, V)

# visualize
image(t(apply(img, 2, rev)), col=gray((0:255)/255), axes=FALSE, main="Original")
```

Original

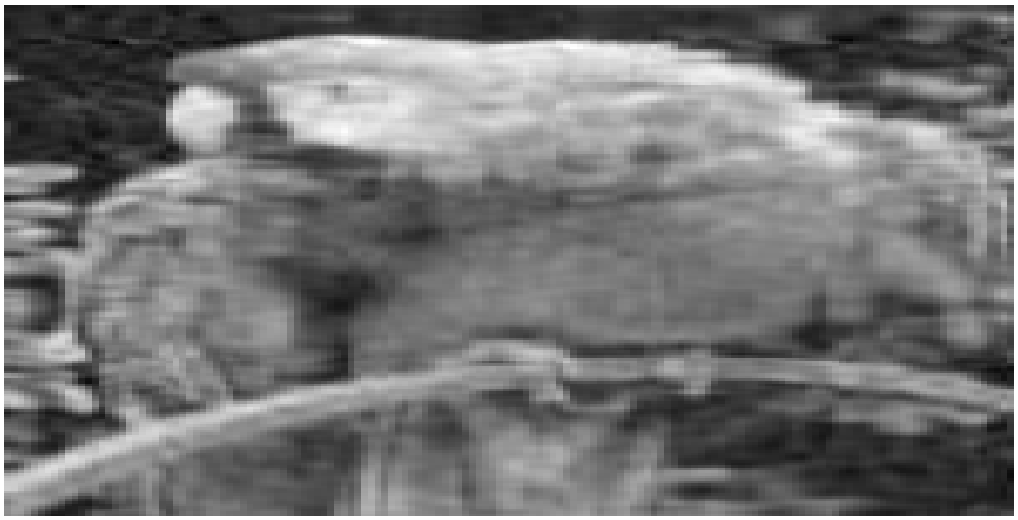


```r
image(t(apply(img_k5,   2, rev)), col=gray((0:255)/255), axes=FALSE, main="k = 5")
```



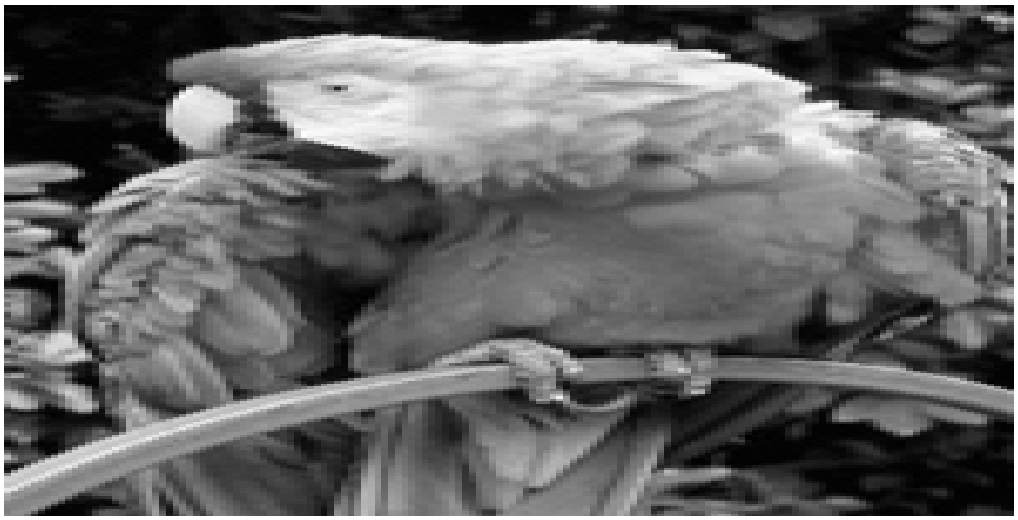

```r
image(t(apply(img_k20,  2, rev)), col=gray((0:255)/255), axes=FALSE, main="k = 20")
```

k = 20



```
image(t(apply(img_k100, 2, rev)), col=gray((0:255)/255), axes=FALSE, main="k = 100")
```

k = 100



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.

- The `qr()` function in R computes the A = QR decomposition of a matrix. Here, A is a 4x4 matrix so the maximum possible rank is 4. The below code returns a rank of 4, meaning all of the rows and columns are linearly independent and cannot be written as linear combinations of the others. The rank being equal to the dimensions also means there are no linear dependencies.

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

qr(A)$rank
```

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

Matrix Rank Boundaries:

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

- The minimum possible rank of a non-zero matrix is 1, since it has at least 1 row or column. The maximum possible rank of this matrix would be n, or the smaller dimension.

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

- The rank of a matrix is the maximum number of linearly independent rows or columns. For example, using the same 4x4 matrix A as above, we can apply Gaussian elimination row operations (here the `rref()` function from the pracma package) to convert A to reduced row echelon form. The RREF shows that each column has a pivot, or a leading/unique 1 value, signaling linear independence. Since there are 4 pivots, the rank of the matrix is 4.

```
library(pracma)
```

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

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

```
rref(A)
```

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

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?).

- Below, the rank and RREF of matrix B are calculated using R. The RREF includes only 1 pivot, equal to its rank. This is a rank-deficient matrix, since the rank is only 1 but the maximum possible rank would be 3.

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

qr(B)$rank
```

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

```
rref(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 is upper-triangular with diagonal entries 3,5,2. Those are three distinct eigenvalues and eigenvectors corresponding to distinct eigenvalues are always linearly independent.

- Using R's built-in `eigen()` and `charpoly()` functions, the characteristic polynomials, eigenvalues and eigenvectors can be computed easily:

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

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

```
eigA <- eigen(A_eigen)
eigA$values
```

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

```
eigA$vectors
```

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

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.

- Again, since we have 3 distinct eigenvalues, the matrix is also diagonalizable because its eigenvectors will be linearly independent. The `diag()` function below constructs the diagonal matrix. Using this, we can also calculate VDV^(-1) = A to verify this.

```
V <- eigA$vectors
D <- diag(eigA$values)
D
```

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

```
A_reconstructed <- V %*% D %*% solve(V)
A_reconstructed
```

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

```
all.equal(A_eigen, A_reconstructed)
```

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

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 R^3?

- Matrix A transforms space by stretching vectors along three directions, the eigenvectors. The eigenvalue tells how much stretching occurs. Since A is an upper triangular matrix, its action is simple; it's mostly a scaling transformation, stretching along three independent axes by factors of 3, 5, and 2. And because A is diagonalizable, there exists another coordinate system (the eigenvector basis) where the

transformation just stretches space differently along three independent axes. In the standard basis, this looks like a mixture of stretching + shearing.

(Cont. . .)

Stephanie Chiang

Homework 1: Part 2

DATA 605

Fall 2025

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:

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, we'll use the lfw module from Python's sklearn library.

Preprocess the Images:

Convert the images to grayscale and resize them to a smaller size (e.g., 64x64) to reduce computational complexity.

In [5]:
```
import sklearn
from sklearn.datasets import fetch_lfw_people

lfw_people = fetch_lfw_people(color=False, resize=0.4)
```

Flatten each image into a vector.

In [ ]:
```
n_samples = len(lfw_people.images)
data = lfw_people.images.reshape((n_samples, -1))
```

Apply PCA:

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

In [11]:
```
# apply PCA
from sklearn.decomposition import PCA
pca = PCA(n_components=0.8, svd_solver='full')
pca.fit(data)
eigenfaces = pca.components_.reshape((pca.n_components_, lfw_people.images.s
print(f'Number of principal components to account for 80% variability: {pca.
```

```
Number of principal components to account for 80% variability: 35
```

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 image

In [12]:
```
# visualize the first few eigenfaces
import matplotlib.pyplot as plt
n_eigenfaces_to_show = 5
fig, axes = plt.subplots(1, n_eigenfaces_to_show, figsize=(15, 8))
for i in range(n_eigenfaces_to_show):
    axes[i].imshow(eigenfaces[i], cmap='gray')
    axes[i].set_title(f'Eigenface {i+1}')
    axes[i].axis('off')
plt.show()
```