Caitlyn Jones

Prof. Thomas Fai

MATH 122A: Numerical Methods and Big Data

8 October 2024

Homework 4

1.

a. To prove that $F = I - \frac{2vv^T}{v^Tv}$ is orthogonal ($F^T = F^{-1}$), we must prove that F is symmetric, $F^T = F$, and that F is its own inverse, $F^2 = I$.

First prove $F^T = F$:

$$F = I - \frac{2vv^T}{v^Tv}$$

$$F^T = \left(I - \frac{2vv^T}{v^Tv}\right)^T$$

$$F^T = I^T - \frac{2(vv^T)^T}{(v^Tv)^T}$$

$$F^T = I - \frac{2v^T(v^T)^T}{(v^T)^Tv^T}$$

$$F^T = I - \frac{2v^Tv}{vv^T}$$

Since we know that $F = I - \frac{2vv^T}{v^Tv}$, we can conclude that $F^T = F$.

Now prove $F^2 = I$:

$$F = I - \frac{2vv^T}{v^Tv}$$

$$F^2 = \left(I - \frac{2vv^T}{v^Tv}\right)^2$$

$$F^2 = \left(I - \frac{2vv^T}{v^Tv}\right)\left(I - \frac{2vv^T}{v^Tv}\right)$$

$$F^2 = I^2 - I \cdot \frac{2vv^T}{v^Tv} - I \cdot \frac{2vv^T}{v^Tv} + 4\frac{vv^T}{v^Tv} \cdot \frac{vv^T}{v^Tv}$$

$$F^2 = I - 4\frac{vv^T}{v^Tv} + 4\frac{v(v^Tv)v^T}{v^T(vv^T)v}$$

$$F^2 = I - 4\frac{vv^T}{v^Tv} + 4\frac{vv^T}{v^Tv}$$

$$F^2 = I$$

Since we have proved that F is symmetric and that F is its own inverse, we can conclude that $F^T = F^{-1}$, so F is orthogonal. ■

b. To compute the reduced and complete QR factorizations of matrix $A = \begin{bmatrix} 1 & 2 \\ 0 & 1 \\ 1 & 0 \end{bmatrix}$, I first created matrix A.

```
A = np.array([[1.,2.],[0.,1.],[1.,0.]])
```

Then, I computed the complete QR factorization, $A = QR$.

```
Q_complete, R_complete = np.linalg.qr(A, mode = "complete")
print("Q complete is \n", Q_complete)
print("R complete is \n", R_complete)
```

```
Q complete is
 [[-0.70710678 -0.57735027 -0.40824829]
 [-0.         -0.57735027  0.81649658]
 [-0.70710678  0.57735027  0.40824829]]
R complete is
 [[-1.41421356 -1.41421356]
 [ 0.         -1.73205081]
 [ 0.          0.        ]]
```

And the reduced QR factorization, $A = \widehat{Q}\widehat{R}$.

```
Q_reduced,R_reduced = np.linalg.qr(A, mode = "reduced")
print("Q reduced is \n", Q_reduced)
print("R reduced is \n", R_reduced)
```

```
Q reduced is
 [[-0.70710678 -0.57735027]
 [-0.         -0.57735027]
 [-0.70710678  0.57735027]]
R reduced is
 [[-1.41421356 -1.41421356]
 [ 0.         -1.73205081]]
```

To check that $\widehat{Q}$ and Q have orthogonal columns, I computed $Q^T Q = I$.

```
Q_complete_transpose = np.transpose(Q_complete)
print("The transpose of Q complete is \n", Q_complete_transpose)
print("Checking orthogonality of Q complete: \n", np.matmul(Q_complete_transpose,Q_complete))
```

```
The transpose of Q complete is
 [[-0.70710678 -0.         -0.70710678]
 [-0.57735027 -0.57735027  0.57735027]
 [-0.40824829  0.81649658  0.40824829]]
Checking orthogonality of Q complete:
 [[ 1.00000000e+00 -1.28818495e-16 -1.35284554e-16]
 [-1.28818495e-16  1.00000000e+00 -1.40562122e-16]
 [-1.35284554e-16 -1.40562122e-16  1.00000000e+00]]
```

And I computed $\widehat{Q}^T \widehat{Q} = I$.

```
Q_reduced_transpose = np.transpose(Q_reduced)
print("The transpose of Q reduced is \n", Q_reduced_transpose)
print("Checking orthogonality of Q complete: \n", np.matmul(Q_reduced_transpose,Q_reduced))
```

```
The transpose of Q reduced is
 [[-0.70710678 -0.         -0.70710678]
 [-0.57735027 -0.57735027  0.57735027]]
Checking orthogonality of Q complete:
 [[ 1.00000000e+00 -1.28818495e-16]
 [-1.28818495e-16  1.00000000e+00]]
```

Because of machine precision, the numerical values above and below the diagonal are not exactly zero, but instead are approaching zero. Since they are so close to zero, we are able to conclude that they are zero.

c. To complete the Householder QR factorization, I first initialized identity matrix Q.

```
Q = np.eye(m,m)
```

Then, I define the variable `i` that will hold the indices for the rows.

```
i = range(k,m)
```

Then, I modified the first index of the reflection matrix.

```
v[0] = sigma+v[0]
```

Then, I wrote the code that will update the values in the kth column of matrix A.

```
A[k,k] = sigma
A[k+1:m,k] = 0
```

Then, I created the variable `j` that will hold the indices for the columns.

```
j = range(k+1,n)
```

I then applied the reflections to the given index of A.

```
A[np.ix_(i,j)] = A[np.ix_(i,j)]-2*v@(v.T@A[np.ix_(i,j)])
```

Then I updated matrix Q.

```
Q[:,i] = Q[:,i]-2*Q[:,i]@(v@v.T)
```

Lastly, I created upper triangular matrix R to have the same contents as our updated A.

```
R = A
```

Given the following code:

```
A = np.array([[1.,2.],[0.,1.],[1.,0.]])

Q_true, R_true = np.linalg.qr(A,mode = 'complete')
print("Q_true is\n", Q_true)
print("R_true is\n", R_true)

Q, R = myqrsteps(A)
print("Q is\n", Q)
print("R is\n", R)
```

I was able to confirm that my results for A in (b) are equivalent to the result above.

```
Q_true is
 [[-0.70710678 -0.57735027 -0.40824829]
 [-0.         -0.57735027  0.81649658]
 [-0.70710678  0.57735027  0.40824829]]
R_true is
 [[-1.41421356 -1.41421356]
 [ 0.         -1.73205081]
 [ 0.          0.        ]]
Q is
 [[ 0.70710678  0.57735027 -0.40824829]
 [ 0.          0.57735027  0.81649658]
 [ 0.70710678 -0.57735027  0.40824829]]
R is
 [[1.41421356 1.41421356]
 [0.         1.73205081]
 [0.         0.        ]]
```

However, as you can see above, the signs on some of the numerical values are different. This is because matrix A has many different solutions and both solutions are correct. We can see this by multiplying a column of Q by -1 and multiplying its corresponding row in R by -1; we will still get the same answer for the full factorization.

2. In order to use function `myqrsteps` from problem (1), I first had to read in the text file `sphinxmoth.txt`.

```
data = pd.read_csv("sphinxmoth.txt", sep=" ", header = None)
```

In order to solve a least squares equation, I first need to compute the QR factorization $A = QR$. Then, with the results of Q and R, I can then plug those into the equation

$R\underline{x} = Q^T\underline{b}$, where we will be solving for $\underline{x}$. In this case, we can define A = logW and b = logR. After reading in the text file, I now can define variable b to hold all the values of logR from the text file.

```
R = np.array(data[1])
b = np.array(np.log10(R))
```

Defining A is slightly different than defining b, as the first column of A will ones and the second column of A will have the values of logW. To do this I must first define a variable `W_log` to hold the values of logW. Then, I must create a new array that is the same size as `W_log`, except every value is a one. Then I can reshape `W_log` to make space for the array of ones we previously define. Lastly, I can define A as the design array with ones in the first column and logW in the second column.

```
W = np.array(data[0])
W_log = np.array(np.log10(W))
ones = np.ones((W_log.shape[0], 1))
data_column = W_log.reshape(-1, 1)
A = np.hstack((ones, data_column))
```

From here, I can use `myqrsteps` from problem (1) to get the values for Q and R.

```
Q,R = myqrsteps(A)
```

For the sake of space, I cannot fit the entirety of Q in one screen clipping as Q is a 37x37 matrix, but here is the excerpt that my console printed out.

```
[[ 0.16439899 -0.31678855  0.09402475 ...  0.18505762
0.20363308
   0.20809654]
 [ 0.16439899 -0.17799579 -0.19989726 ...  0.01684137
0.06106738
   0.07169437]
 [ 0.16439899 -0.11907246  0.95497151 ... -0.02862096
-0.02527296
  -0.02446848]
 ...
 [ 0.16439899  0.1155763  -0.01071084 ...  0.96130496
-0.04440527
  -0.04577738]
 [ 0.16439899  0.1634569  -0.00370825 ... -0.04075068
0.95169073
  -0.05012551]
 [ 0.16439899  0.17496205 -0.0020256  ... -0.04124462
-0.04924735
   0.94882969]]
```

And here are the results for R, which is a 37x2 matrix.

```
[[ 6.08276253 -0.91927795]
 [ 0.          5.10884218]
 [ 0.          0.        ]
 [ 0.          0.        ]
 [ 0.          0.        ]
 [ 0.          0.        ]
 [ 0.          0.        ]
 [ 0.          0.        ]
 [ 0.          0.        ]
 [ 0.          0.        ]
 [ 0.          0.        ]
 [ 0.          0.        ]
 [ 0.          0.        ]
 [ 0.          0.        ]
 [ 0.          0.        ]
 [ 0.          0.        ]
 [ 0.          0.        ]
 [ 0.          0.        ]
 [ 0.          0.        ]
 [ 0.          0.        ]
 [ 0.          0.        ]
 [ 0.          0.        ]
 [ 0.          0.        ]
 [ 0.          0.        ]
 [ 0.          0.        ]
 [ 0.          0.        ]
 [ 0.          0.        ]
 [ 0.          0.        ]
 [ 0.          0.        ]
 [ 0.          0.        ]
 [ 0.          0.        ]
 [ 0.          0.        ]
 [ 0.          0.        ]
 [ 0.          0.        ]
 [ 0.          0.        ]
 [ 0.          0.        ]
 [ 0.          0.        ]]
```

Then I computed Q^T, which is also a 37x37 matrix.

```
Q_T = np.transpose(Q)
```

Now, I can use b = logR and Q^T to plug into the equation $R\underline{x} = Q^T\underline{b}$ to get the least squares solution $\underline{x}$. To do this, I used `np.linalg.lstsq` to find the solution.

```
x,_,_,_  = np.linalg.lstsq(R,Q_T_b, rcond=None)
```

```
[0.11551651 0.57622652]
```

We use `x[0]` since `np.linalg.lstsq` returns four values, so the first index returned is the least squares solutions logb and a. From here, I was able to find b.

```
log_b, a = x
b = 10**(log_b)
```

```
1.304717562851326
```

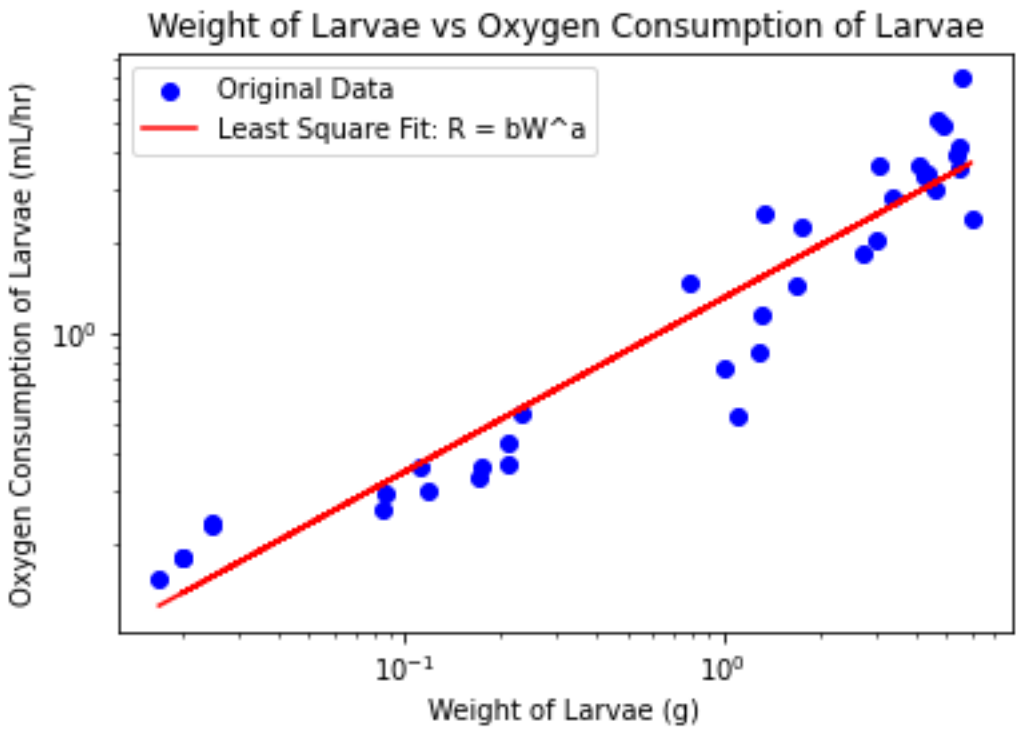
Now that we have values for b and a, we can plot the least squares fit versus the data, or the equation $R = bW^a$ versus the R and W data in `sphinxmoth.txt`. To do this, I first calculated the values for the equation $R = bW^a$.

```
R_lsf = b * (W**a)
```

```
[0.1246965  0.31947684 0.47632222 1.38558384 1.79526004 2.93770175
 3.46615279 3.64950501 0.15572812 0.36762606 0.53229224 1.30396559
 2.4666597  3.01558209 3.13562937 3.17489922 0.1369383  0.31522402
 0.47157256 1.51091917 2.47605948 3.01964003 3.41085669 0.1369383
 0.382668   0.53083713 1.53106774 2.61404532 3.47713425 0.15572812
 0.563599   1.13318083 1.55102316 1.7653502  2.33706966 3.23314353
 3.49538024]
```

Now that we have the values needed for the plots of the least squares fit and the data text file, we can plot these against each other on the same graph.

```
plt.scatter(W,R, label="Original Data", color = "blue")
plt.plot(W, R_lsf, label="Least Square Fit: R = bW^a", color = "red")
plt.xlabel("Weight of Larvae (g)")
plt.ylabel("Oxygen Consumption of Larvae (mL/hr)")
plt.title("Weight of Larvae vs Oxygen Consumption of Larvae")
plt.xscale("log")
plt.yscale("log")
plt.legend()
plt.show()
```



The exponent a represents a sub-linear relationship between the weight of the larvae and the oxygen consumption of the larvae. Because $a < 1$, we can predict that the weight of the larvae increases faster than the oxygen consumption. Thought their oxygen is increasing, it is at a much lower rate when compared to its increase it weight.

3.

a. To compute the singular values from my name matrix from Homework 1, I first read the file into my code.

```
name_matrix = np.loadtxt("cjones_name.txt")
```

Since `np.linalg.svd` returns three values, I only created variable `svd` to hold the singular value decomposition of `name_matrix`.

```
_, svd, _ = np.linalg.svd(name_matrix, full_matrices=False)
```

Then I computed the rank of my matrix.

```
name_rank = np.linalg.matrix_rank(name_matrix)
```

```
Rank of name_matrix:  9
```

In terms of singular value decomposition, the rank of a matrix is equal to the number of non-zero singular values, which we can see given the results.

```
[1.04370512e+01 3.35084497e+00 3.21754072e+00 2.41026135e+00
 1.29532716e+00 1.00000000e+00 1.00000000e+00 1.00000000e+00
 1.00000000e+00 1.21303708e-16 3.11496619e-17 5.97658638e-33
 0.00000000e+00 0.00000000e+00 0.00000000e+00 0.00000000e+00]
```

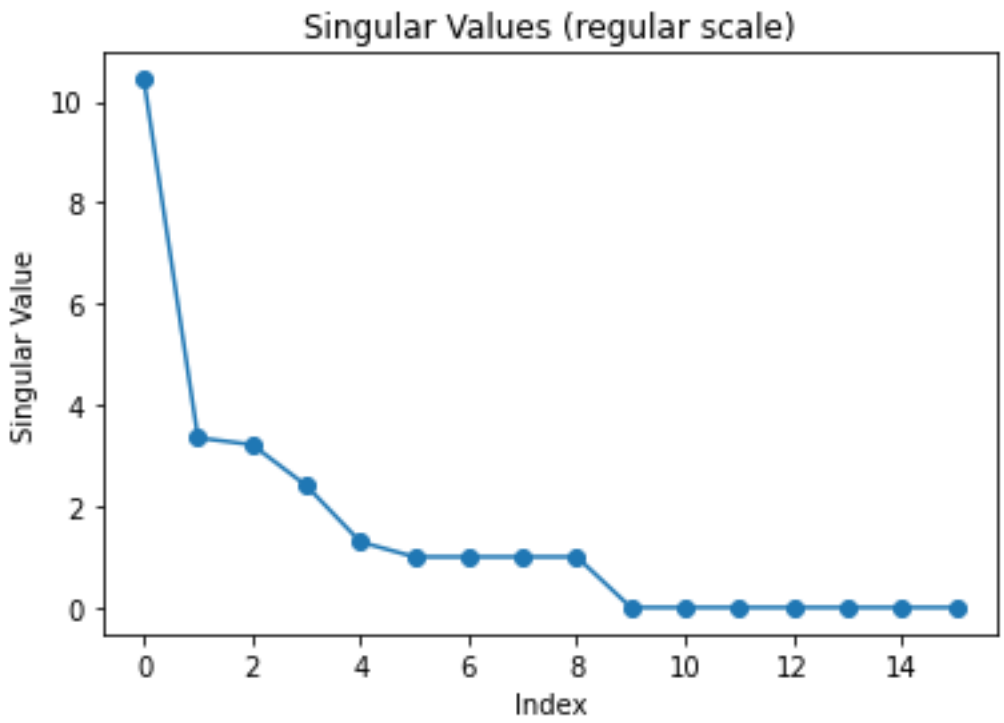
With machine precision, we can see that the last 3 values in row 3 are incredibly small numbers that are dangerously close to zero, so we can assume that these numbers are zero. Given that information, we can confirm that the rank of `name_matrix` is equivalent to the number of non-zero singular values.

Lastly, I graphed the singular values on a regular scale and on a log scale.

On a regular scale:

```
plt.plot(svd, "o-", label="Singular Values")
plt.title("Singular Values (regular scale)")
plt.xlabel("Index")
plt.ylabel("Singular Value")
plt.show
```

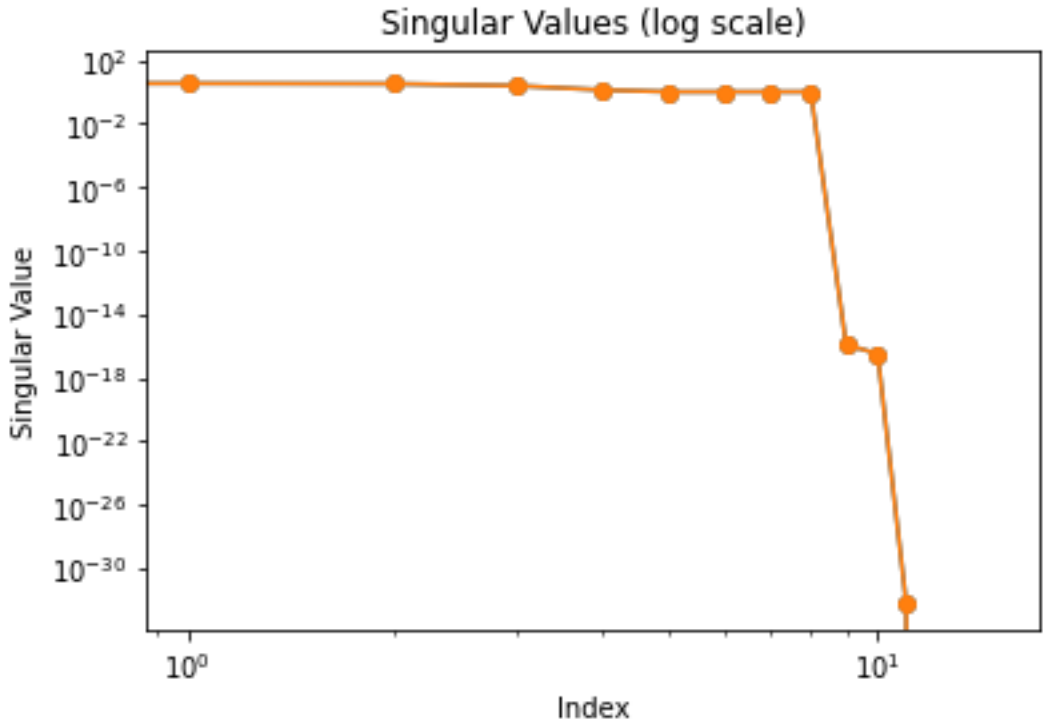
With results:



On a log scale:

```
plt.plot(svd, "o-", label="Singular Values")
plt.title("Singular Values (log scale)")
plt.xlabel("Index")
plt.ylabel("Singular Value")
plt.xscale("log")
plt.yscale("log")
plt.show()
```

With results:



One thing I observe about these graphs is that they seem flipped or a reflection of each other. For the graph on the regular scale, you can clearly see every positive value, but every value close to zero is plotted as zero. However, for the graph on the log scale, you can clearly see every value less than zero, as machine precision

allows for the values on the graph to be accurate regarding the magnitude of their decimal places; at the same time, every value greater than zero is plotted very close to 10, so the value seem to be closer in value than they actually are.

b. To construct low rank approximations of `name_matrix`, I first initialized values `U`, `S`, and `V` from running `np.linalg,svd(name_matrix)`.

```
U,S,V = np.linalg.svd(name_matrix, full_matrices=False)
```

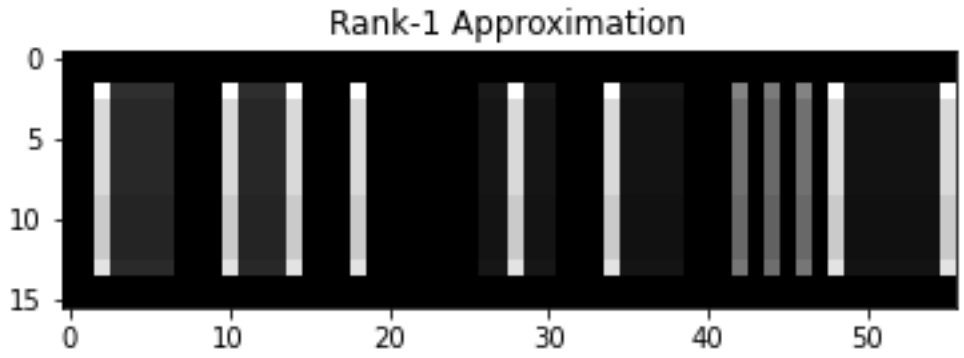
From there, I was able to compute low-rank approximations for rank-1, rank-2, rank-3, and rank-4.

```
#rank1
S_one = np.diag(S[:1])
U_one = U[:, :1]
V_one = V[:1, :]
rank_one = U_one @ S_one @ V_one

#rank2
S_two = np.diag(S[:2])
U_two = U[:, :2]
V_two = V[:2, :]
rank_two = U_two @ S_two @ V_two

#rank3
S_three = np.diag(S[:3])
U_three = U[:, :3]
V_three = V[:3, :]
rank_three = U_three @ S_three @ V_three

#rank4
S_four = np.diag(S[:4])
U_four = U[:, :4]
V_four = V[:4, :]
rank_four = U_four @ S_four @ V_four
```

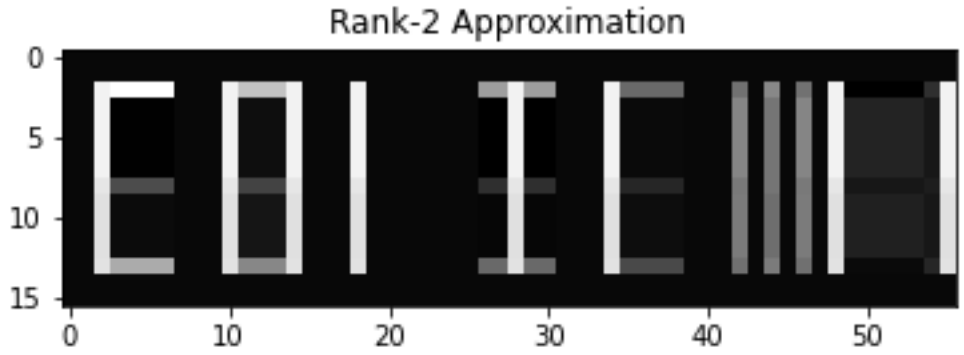
Finally, I was the able plot the images from these low-rank approximations.

```
plt.imshow(rank_one, cmap = "grey")
plt.title("Rank-1 Approximation")

plt.imshow(rank_two, cmap = "grey")
plt.title("Rank-2 Approximation")

plt.imshow(rank_three, cmap = "grey")
plt.title("Rank-3 Approximation")

plt.imshow(rank_four, cmap = "grey")
plt.title("Rank-4 Approximation")
```

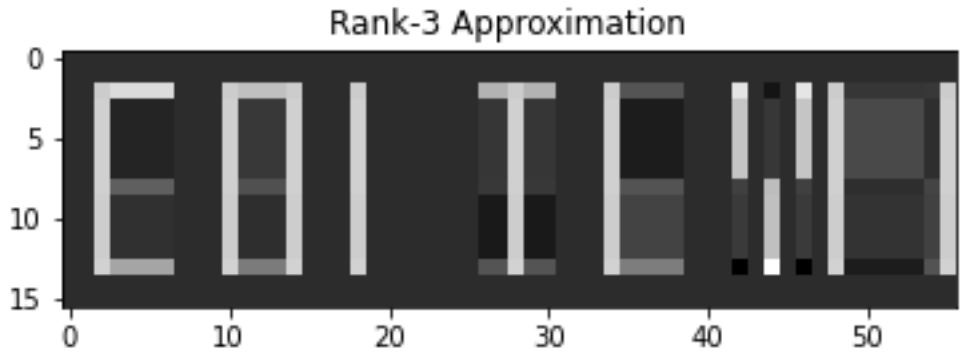
Rank 1:



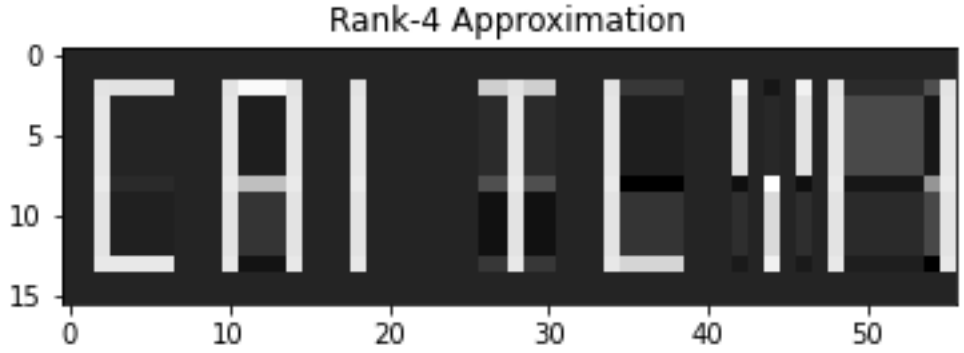
Rank 2:



Rank 3:



Rank 4:



I thought it was really cool that by graphing the low-rank approximations in increasing order, you can see the pixels of the picture become more defined. In rank-1, you can see the outline of the letters in my name, but in rank-4 (and any low-rank approximation from this point up until the given rank of my matrix), you can start to see the intricacy of the details increase as the rank increases; you can also start to see the outlines of each letter actually become the letters.