1 Problem 1

1.1 a

For a function $q(s) = a_m s^m + a_{m-1}s^{m-1}... + a_0$, we can say that $Q(A) = a_m A^m + a_{m-1}A^{m-1}... + a_0 I$. Since we know that the matrix A^k is equivalent to $V\Lambda^k V^{-1}$, this is equivalent to $Q(A) = a_n(V\Lambda^n V^{-1}) + a_{n-1}V\Lambda^{n-1}V^{-1}... + a_0 I$.

We know that for the same polynomial $q(s)$, we can say that when $q(\Lambda) = diag(q(\lambda_1)...q(\lambda_n)$, $Vq(\Lambda)V^{-1} = Vdiag(q(\lambda_1)...q(\lambda_n)V^{-1}$. This means that $Vq(\Lambda)V^{-1} = V(a_n\lambda_n + a_n\lambda_{n-1}... + a_0)V^{-1} + V(a_{n-1}\lambda_n + a_n\lambda_{n-2}... + a_0)V^{-1}... + V(a_0)V^{-1}$. This is directly equivalent to $q(A) = a_nV\Lambda^nV^{-1} + a_{n-1}V\Lambda^{n-1}V^{-1}... + a_0 I$, meaning the two are the same.

1.2 b

Since we know from a. that the two are the same, this means that $p(A) = Vp(\Lambda)V^{-1}$, where $p(\Lambda) = \lambda^n + a_{n-1}\lambda^{n-1}... + a_0$. This applies for every eigenvalue $\lambda_1...\lambda_n$, where this is equivalent to $V\Lambda^nV^{-1} + a_{n-1}V\Lambda^{n-1}V^{-1}... + a_0$, which we know is equal to 0.

2 Problem 2

2.1 a

If $A = A^T$, then if $A = VBV^{-1}$, $A^T = (VBV^{-1})^T = (V^{-1})^T B^T V^T$. Therefore, $A = VBV^{-1} = A^T = (V^{-1})^T B^T V^T$. If $V^TV = I$, then $(VBV^{-1})(V^{-1})^T = VB$, and $((V^{-1})^T B^T V^T)(V^{-1})^T = (VBV^{-1})^T(V^{-1})^T = (V^{-1}VBV^{-1})^T$. This does not lead us to the same equation as for A, meaning we cannot prove that $B = B^T$ from this, making it false.

2.2 b

Two matrices A and B are similar when there exists some matrix P such that $A = PBP^{-1}$ and $B = P^{-1}AP$. Then if $A = A^T$, $PBP^{-1} = (PBP^{-1})^T = (P^{-1})^T B^T P^T$. This must mean that $P = P^{-1}$ and $B = B^T$, meaning that B is therefore symmetric, so this is True.

2.3 c

This is true, because if a matrix is not diagonalizable, then that implies there are remaining eigenvalues that are equal 0 (the number of 0 eigenvalues is determined by rank-nullity). Summing products involving these eigenvalues that are equal to 0 will give us the same solution as before+0, or still 0. Therefore, this is true.

3 Problem 3

Let A be the following mxn matrix:

a_{11} a_{12} ... a_{1n}

a_{21} a_{22} ... a_{2n}

...a_{m1} a_{m2} ... a_{mn}

We know that for this matrix A, the max 1-norm, $||A||_1$, is equal to $||Ax||_1$ where $||x||_1 = 1$. Since x has dimension nx1, this means that we can write $||Ax||_1$ as $\sum_{i=1}^{m} ||(A_i)x|| <= \sum_{i=1}^{m} ||(A_i)|||x|$. We know that this is going to be maximized when the value of $|x| = 1$ because any value less than 1 will decrease the total sum, since they are decimal values less than 1.

We also know that this is maximized when we only include the maximum column i of A because that is the only column that will be multiplied by the vector with norm 1, x. Therefore, $||A||_1$ is equal to $||(A_i)||$ where A_i is the maximum column of A. This is identical to max $\sum_{i=1}^{m} |a_{ij}|$ for the maximum column of A j, or the max column sum.

4 Problem 4

For a stochastic matrix, the columns and rows (depending on column-stochastic and row-stochastic) will each add to 1 with no negative values. If a matrix is column-stochastic, then its columns, specifically, will add up to 1.

For column-stochastic matrix A, let us say we know that it has a set of eigenvalues λ_1... λ_n. We can say that there is some eigenvector, x, corresponding to the maximum value eigenvalue, λ so that $Ax = \lambda x$. For the product Ax, we know that since no element is less than 0, then the highest possible value of Ax will be equal to 1 times the maximal value of x. For the eigenvalue λ, the maximal value will be equal to λ times the maximal value of x. Therefore, λ has a maximum value of 1, otherwise it would be impossible for A to be a column=stochastic matrix.

5 Problem 5

If we say that $\hat{X}_k = U\Sigma V^T$ and $X_k = YZ^T$, then we can say V^T has columns that are equal to the eigenvectors of X^TX, by the definition of singular value decomposition. Similarly, we can say that the columns of U are equal to the eigenvectors of XX^T.

Since the matrix Σ is a diagonal matrix filled with values $\sigma_1 ... \sigma_k$ for a decomposition of rank k, we can say that $U\Sigma$ is equal to the eigenvectors of XX^T each scaled by a corresponding singular value (e.g. eigenvector one is scaled by σ_1... eigenvector k is scaled by σ_k).

We can say that this scaling is equivalent to projecting each column of U onto the space generated by eigenvalue k for a space of rank k, thereby creating a matrix with rank k. This matrix is equivalent to Y, and since V^T and Z^T have the same definition, we can say that the two are equivalent, so $\hat{X}_k = X_k$.

6 Problem 6

Attached to bottom of page in separate PDF.

Implementing an image compressor

You will need to install the following commands and enable ipywidgets

```
!pip install scikit-image
!pip install ipywidgets
!jupyter nbextension enable --py widgetsnbextension
```

```
# start by importing some necessary packages
%matplotlib inline
import numpy as np
import matplotlib.pyplot as plt
from skimage.io import imread
import numpy.linalg as la
```

Problem 6 [Hw4] Overview

We will apply low rank approximation to bender in a different way than we did in the class example. And you will apply this code and interactive widget to another image of your choosing.

Part a

For bender.png, what is a reasonable approximation of the rank to get nearly lossless image reconstruction? Submit in your pdf an image of bender next to the image you reconstructed with the chosen rank.

Part b.

For you selected image, what is a reasonable approximation of the rank to get nearly lossless image reconstruction? Submit in your pdf an the original image you chose next to the image you reconstructed with the chosen rank.

What to turn in

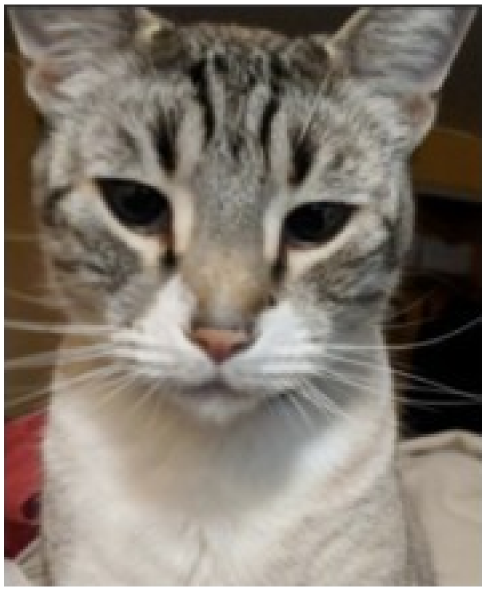
Print the pdf of the notebook and attach it to your pdf that you upload for your homework. The images can be in the printed pdf or you can save them and attach them as part of a written solution for this problem. You must also upload your ipynb.

Part a

We read in a test image and convert to a $[0,1]$ scale before performing any arithmetic.

```
im = imread("./bender.png")
im = im.astype(float)/255.
print("Image dimensions:",im.shape)

plt.imshow(im)
plt.axis('off')
plt.show()
```

```
Image dimensions: (672, 552, 4)
```



Next, we vertically stack the color channels to form a $3m \times n$ image, where the n columns of the image represent our samples, and center the image about its mean to obtain the matrix A.

Part a.1

Below, compute the mean of each row to obtain the column mean. Store this in variable `Sm`. Make sure to pass to the `np.mean` function the argument `keepdims=True`. Then use `Sm` to create the centered matrix $A = S - S_m$.

```
m,n,p = im.shape
S = np.vstack([im[:,:,0],im[:,:,1],im[:,:,2]])

Sm = np.mean(S, axis=0)# compute the mean of each row to obtain a mean
column.

A = S - Sm # compute the centered matrix A.
```

Part a.2

With your group, please write a function `rank_approx(A, k)` which accepts a centered matrix A and a maximum rank k, and assembles a list L of the first k rank-one matrices in the low-rank approximation of the matrix. The list L should be of the form

$$L = \left[\sigma_1 u_1 v_1^T, \sigma_2 u_2 v_2^T, \ldots, \sigma_k u_k v_k^T\right]$$

Return the list as your output.

```
def rank_approx(A, k):
    L = []
    U, sigs, VH = la.svd(A, compute_uv = True)
    V = np.transpose(VH)
    for i in range(k):
        ui = U[:, i]
        vi = V[:, i]
        ui = np.reshape(ui, (len(ui), 1))
        vi = np.reshape(vi, (len(vi), 1))
        vi = np.transpose(vi)
        L.append(np.dot(sigs[i]*ui, vi))
    return L

print(np.shape(A))
L=rank_approx(A,3)
```

```
(2016, 552)
```

Part a.3

Create the function `reconstruct(L, r, Sbar)` below such that it accepts as inputs:

- the list of low-rank approximations L,
- a target rank $r<k$, and
- the column mean $\acute{S}$,

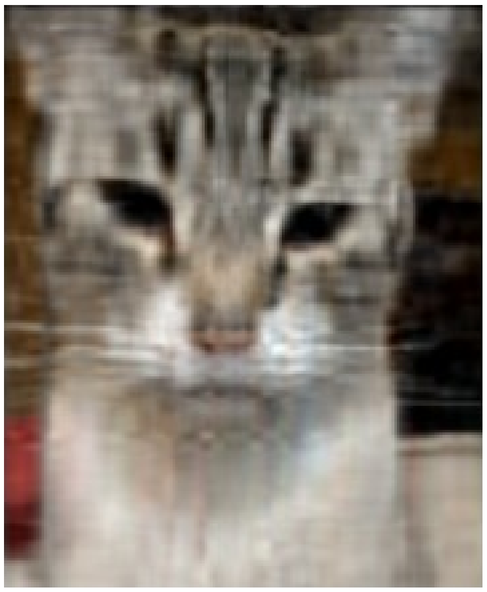
And returns

- the rank-r reconstructed image.

If your `rank_approx` function is set up correctly, you should be able to run the following cell and plot a test reconstruction:

```
def reconstruct(L, r, Sbar):
    if r >= len(L):
        print("Error: r must be less than the length of L.")
        return

    m = len(Sbar)//3 # the leading dimension of a single channel


    Ar = np.zeros(L[0].shape)
    # Reconstruct the original image up to rank j.
    for i in range(r):
        Ar = Ar + L[i]

    Ar += Sbar # Add back the mean Sbar and reshape into m by n by p.

    m = Ar.shape[0]//3

    imr = np.stack([Ar[:m,:],Ar[m:2*m,:],Ar[2*m:,:]], axis=2)
    # truncate values and return the reconstructed image.
    imr[imr<0] = 0; imr[imr>1] = 1
    imr = imr.real
    return imr

# Run a test:
k = 16; r = 12
print("Ran rank")
L = rank_approx(A, k)
print("Ran reconstruct")
imr = reconstruct(L, r, Sm)

plt.imshow(imr)
plt.axis("off")
plt.show()
```

```
Ran rank
Ran reconstruct
```



part a.4

Given the above code works, you can now run this cool little widget to visualize bender at different target values of k. Decide how many singular vectors you feel are necessary to reconstruct the image before differences between the original become imperceptible.

Provide an plot of this reconstructed image next to the original.

```
import ipywidgets as widgets
from ipywidgets import interactive

rmax = min(80,n)
```

```
L = rank_approx(A, rmax+1)
rank_slider = widgets.IntSlider(
    value=1, min=1, max=rmax, step=1,
    description='rank:', continuous_update=False)

def rank_slider_plot(r = 1):
    plt.close()
    imr = reconstruct(L, r, Sm)

    # Plot the original and compressed images.
    fig, (ax1, ax2) = plt.subplots(1,2, figsize=(12,6))
    ax1.imshow(im)
    ax1.set_title("original", size=16)
    ax1.axis("off")

    ax2.imshow(imr)
    ax2.set_title("reconstruction", size=16)
    ax2.axis("off")
    plt.show()

interactive_plot = interactive(rank_slider_plot, r=rank_slider)
output = interactive_plot.children[-1]
output.layout.height = '400px'
interactive_plot
```

{"version_major":2,"version_minor":0,"model_id":"9f4f7574dc04443c91161
b73aeaed9c3"}

Part b

Now find an image online or select one of your own images, and apply the above code to that image. Determine an appropriate target rank to reconstruct the image and provide a plot of the original next to the reconstructed image as you did in part a.

Compare the target ranks to reconstruct the images in part a and b. Are they similar?

```
## enter your code here
im = imread("./russ_bronco.jpg")
im = im.astype(float)/255.
print("Image dimensions:",im.shape)

plt.imshow(im)
plt.axis('off')
plt.show()

m,n,p = im.shape
S = np.vstack([im[:,:,0],im[:,:,1],im[:,:,2]])
Sm = np.mean(S, axis=0)# compute the mean of each row to obtain a mean
column.
A = S - Sm # compute the centered matrix A.
```

```
# Run a test:
k = 60; r = 59
print("Ran rank")
L = rank_approx(A, k)
print("Ran reconstruct")
imr = reconstruct(L, r, Sm)

plt.imshow(imr)
plt.axis("off")
plt.show()
```

```
Image dimensions: (550, 900, 3)
```

```
Ran rank
Ran reconstruct
```

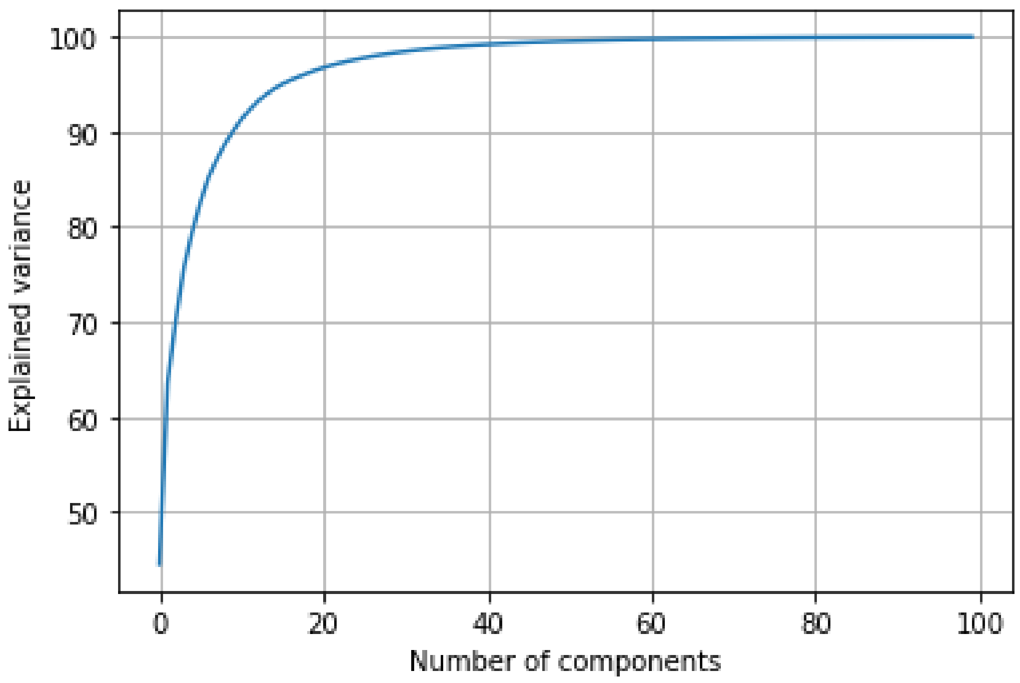
I used a target rank of 60, and I see that it is less than the target rank of 80 used in the previous part. This may be due to the fact that there is a greater variety of pixels in the picture I used.

Supplementary Material: Image Compression with PCA

You can also do image compression and reconstruction with PCA. Below you can see how using `scikit-learn`. The cool thing is that you can plot the explained variance and use that to pick a reasonable target value for k.

```
import numpy as np
from sklearn.decomposition import PCA
pca_100=PCA(n_components=100)
pca_100.fit(S)
plt.grid()
plt.plot(np.cumsum(pca_100.explained_variance_ratio_ * 100))
plt.xlabel('Number of components')
plt.ylabel('Explained variance')

n_components=20
pca_ = PCA(n_components=n_components)
bender_reduced = pca_.fit_transform(S)
bender_recovered = pca_.inverse_transform(bender_reduced)
print(np.shape(bender_recovered))

plt.figure()
m = len(Sm)//3
Ar_ = bender_recovered
imr = np.stack([Ar_[:m,:],Ar_[m:2*m,:],Ar_[2*m:,:]], axis=2)
plt.imshow(imr, cmap='gray_r')
plt.title('Compressed image with {} components'.format(n_components),
fontsize=15, pad=15)
plt.savefig("image_pca.png")
```

```
(2016, 552)
```



```
<Figure size 432x288 with 0 Axes>
```