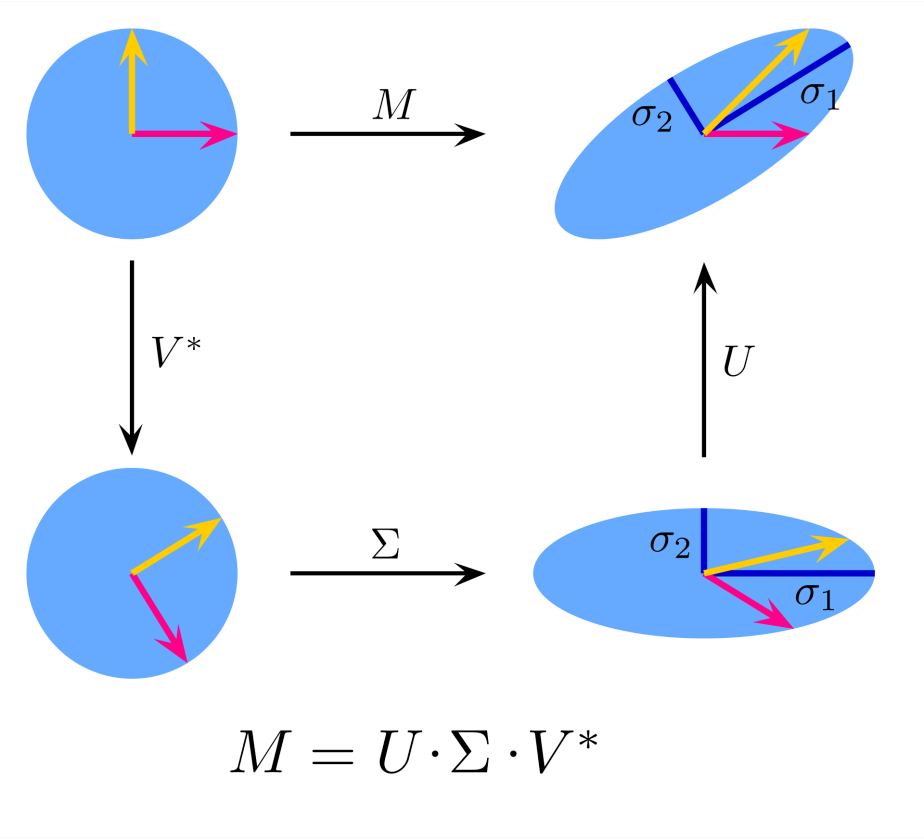
Scientific Computing Project 3: SVD Image Compression

Brian Livian

Introduction

Singular value decomposition, or SVD for short, is a powerful tool in linear algebra with many applications. When utilized with a computer, SVD can be applied to many areas, such as data science as well as image compression. SVD is the process of the factorization of a matrix that generalizes the eigendecomposition of a square normal matrix to any m x n matrix. An illustration of SVD is shown below.



$$M = U \cdot \Sigma \cdot V^*$$

The problem I explore in this project is the application of SVD to black & white/ color image compression. I use a picture of myself taken from my computer for the experiment. I am also tasked with calculating the percent of memory used for various quality images (various values of p singular values). I am then tasked with calculating the error of the compressed image compared to the original image, the results of which are shown in the following pages.

Methodology

To apply SVD through python, I use numpy's linear algebra library. I first explore image compression of a black and white image. First I upload a picture of myself, and convert it to black and white. I then convert the image into a numpy matrix. I also save the storage size of the original image to a variable named original_st. Finally, I run a for loop for singular values between 2 and 50, with a spacing of 5 for each iteration. The algorithm applies SVD to the image and saves it to a variable called 'reconstimg', while saving the memory of each iterated compressed image to the variable named 'reconstimg_st'. The algorithm then displays the compressed black and white image for each iterated value.

The algorithm for compressing color images is similar to the algorithm for black & white images in that it utilizes the SVD function in numpy's linear algebra library. The problem I am faced with is that SVD is useful for 2 dimensional matrices. Black and white images are 2 dimensional, so there's no problem there. However, color images have a 3rd dimension, thus introducing a problem for the SVD algorithm. Color images have a dimension for the base colors of red, green and blue. To solve this issue, I create 3 matrices out of the original uploaded image, one for red, one for green and one for blue. I define a function named 'compress' with inputs for image and i, which implements the algorithm of SVD for color images. Within the function I apply a similar methodology as the black and white image processing, but now I apply it to the 3 newly created 2 dimensional matrices. For each of the 3 color matrices, I apply SVD and compress the image. I also calculate the memory of each of the 3 matrices. After compressing the 3 matrices, I recompile the compressed images into a matrix, 'm'. This new matrix, 'm', is the recompiled, compressed color image. To display the image correctly in matplotlib, I convert the image to a uint8 data type. However, uint8 data types only represent values on the range [0,255]. This would cause errors in the recompiled image, since there are possible negative values or values greater than 255 in the matrix. To mitigate this issue, I run an algorithm to change any negative values m to positive, and any values greater than 255 to 255. I then calculate the memory of the recompiled color image by adding the memory values of each of the 3 compiled matrices. I calculate the percentage of memory for the compiled image compared to the original. I calculate the error of the recompiled image compared to the original as well. Finally, I display the recompiled image, and iterate for p values on the range [2,47], spaced out for every 5 values.

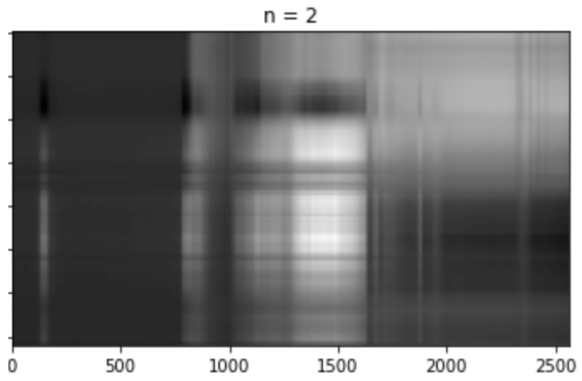
Computer Simulation and Result

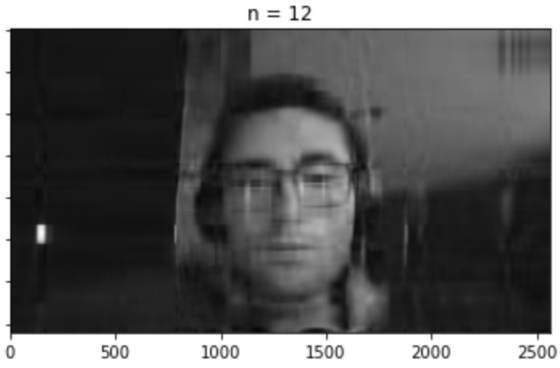
Below is the program for converting the black & white image to a matrix, and storing the original memory value. The program for compressing black & white images, and the resulting compressed images, are shown below as well.

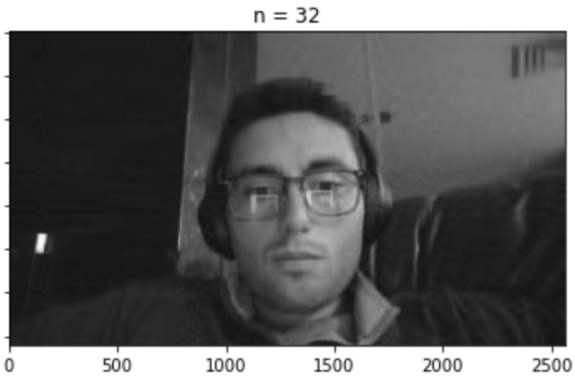
```
imgmat = np.array(list(imgconv.getdata(band=0)), float)
imgmat.shape = (imgconv.size[1], imgconv.size[0])
imgmat = np.matrix(imgmat)
print(imgmat)
print(np.amax(imgmat))
original_st=imgmat.shape[0]*imgmat.shape[1]
print('Storage memory=',original_st)
plt.figure(figsize=(9,6))
plt.imshow(imgmat, cmap= 'gray')
```

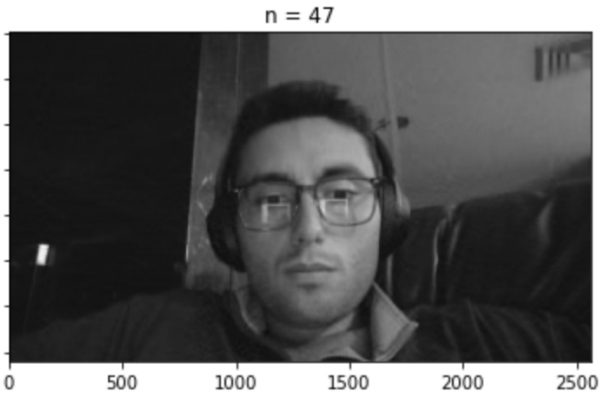
```
U, sigma, V = np.linalg.svd(imgmat)
#print(sigma)
plt.plot(sigma)
```

```
for i in range(2,50,5):
    reconstimg = i+np.matrix(U[:, :i]) * np.diag(sigma[:i]) * np.matrix(V[:i, :])
    reconstimg_st=np.shape(U[:, :i])[0]*np.shape(U[:, :i])[1]+np.shape(V[:i, :])[0]*np.shape(V[:i, :])[1]
    print('Percent of storage memory=',reconstimg_st/original_st)
    plt.imshow(reconstimg, cmap='gray')
    title = "n = %s" % i
    print(np.shape)
    plt.title(title)
    plt.show()
```







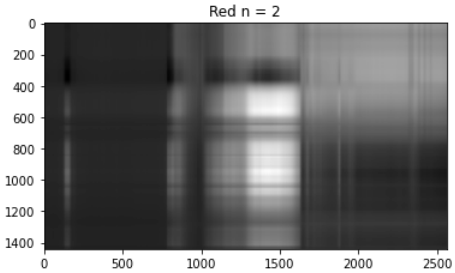


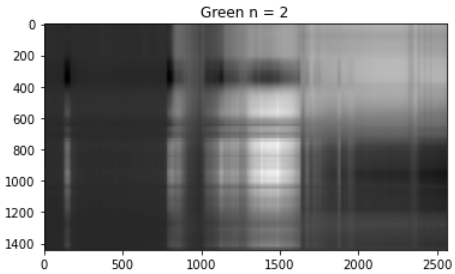
Below is the program for compressing the color image. The original image is already uploaded, and the original memory storage is calculated to be 11059200. As mentioned in the methodology, SVD image compression is applied for each of the 3 color matrices, then recompiled to produce a compressed color image.

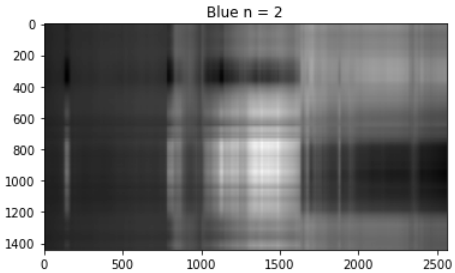
```
def compress(img, i):
    # Create 3 arrays, one for red, one for green and for blue
    red = img[:, :, 0]
    green = img[:, :, 1]
    blue = img[:, :, 2]

    # Apply SVD to 3 new matrices
    URed, sigmaRed, VRed = np.linalg.svd(red)
    UGreen, sigmaGreen, VGreen = np.linalg.svd(green)
    UBlue, sigmaBlue, VBlue = np.linalg.svd(blue)


    # Apply same process as black and white image compression
    # Except this time apply the process for each of the 3 colors
    Red_reconstimg = np.matrix(URed[:, :i]) * np.diag(sigmaRed[:i]) * np.matrix(VRed[:i, :])
    Red_reconstimg_st=np.shape(URed[:, :i])[0]*np.shape(URed[:, :i])[1]+np.shape(VRed[:i, :])[0]*np.shape(VRed[:i, :])[1]
    plt.imshow(Red_reconstimg, cmap= 'gray')
    title = "Red n = %s" % i
    plt.title(title)
    plt.show()

    Green_reconstimg = np.matrix(UGreen[:, :i]) * np.diag(sigmaGreen[:i]) * np.matrix(VGreen[:i, :])
    Green_reconstimg_st=np.shape(UGreen[:, :i])[0]*np.shape(UGreen[:, :i])[1]+np.shape(VGreen[:i, :])[0]*np.shape(VGreen[:i,
    plt.imshow(Green_reconstimg, 'gray')
    title = "Green n = %s" % i
    plt.title(title)
    plt.show()

    Blue_reconstimg = np.matrix(UBlue[:, :i]) * np.diag(sigmaBlue[:i]) * np.matrix(VBlue[:i, :])
    Blue_reconstimg_st=np.shape(UBlue[:, :i])[0]*np.shape(UBlue[:, :i])[1]+np.shape(VBlue[:i, :])[0]*np.shape(VBlue[:i, :])[1
    plt.imshow(Blue_reconstimg, cmap= 'gray')
    title = "Blue n = %s" % i
    plt.title(title)
    plt.show()

    # Create new array of zeroes in the size of the original image
    m = np.zeros(img.shape)

    # Recompile the compressed images of 3 colors into matrix m defined above
    m[:, :, 0] = Red_reconstimg
    m[:, :, 1] = Green_reconstimg
    m[:, :, 2] = Blue_reconstimg


    # np.unit8 is a datatype with range [0,255]
    # Algorithm to change any negative values into positive, and any values greater than 255 into 255
    for i1, row in enumerate(m):
        for i2, col in enumerate(row):
            for i3, x in enumerate(col):
                if x > 255:
                    m[i1,i2,i3] = 255
                if x < 0:
                    m[i1,i2,i3] = abs(x)


    compressed = m.astype(np.uint8)
    error = (np.linalg.norm(img - m))/(np.linalg.norm(img))
    percentmemory = ((Red_reconstimg_st + Green_reconstimg_st + Blue_reconstimg_st)/ (color_original_st)) * 100

    plt.imshow(compressed)
    title = "Recompiled Image n = %s" % i
    plt.title(title)
    plt.show()
    print('Recompiled image percent of storage memory= ' + str(percentmemory) + ' %')
    print('The error is ' + str(error))

    return percentmemory, error
```

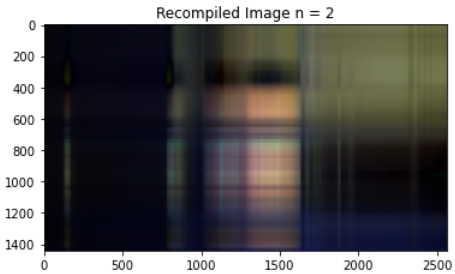
The image compression function is iterated for p singular values on the range [2,47], spaced by 5 in between every iteration. I also create lists for percent memory and error which are appended for each iteration. I explore the result of the image compression for various quality images (aka various p values). The results of which are shown below:

```
percentlist = []
errorlist = []
plist = []

for i in range (2, 50, 5):
    plist.append(i)
    a = compress(image, i)
    percentlist.append(a[0])
    errorlist.append(a[1])
```

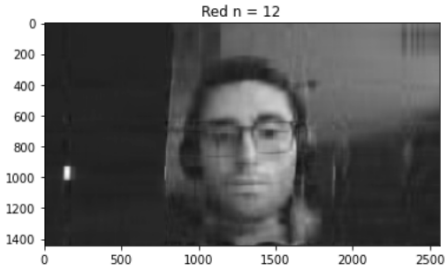


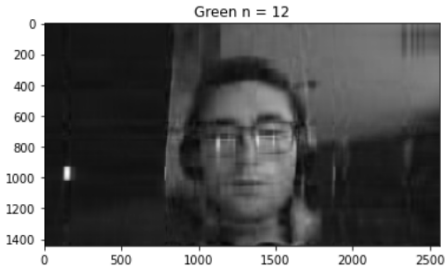


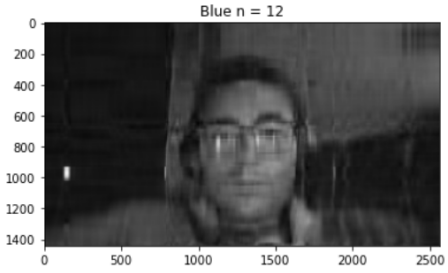


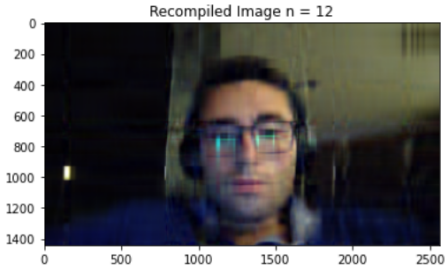



```
Recompiled image percent of storage memory= 0.2170138888888889 %
The error is 0.30179400417945434
```

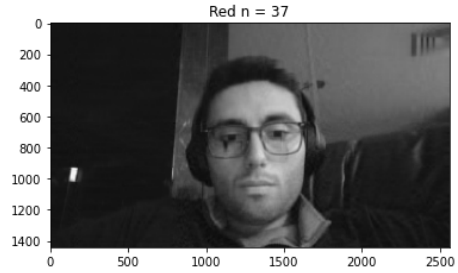


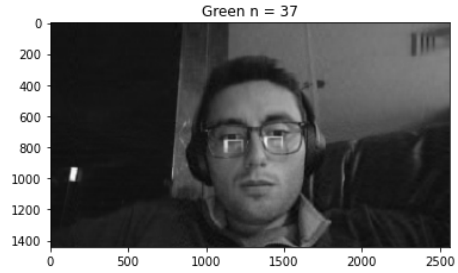


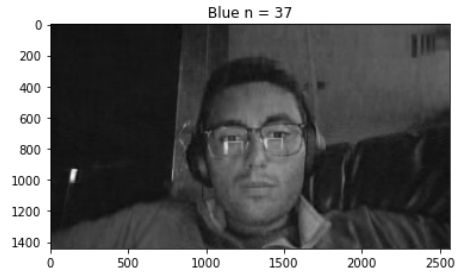




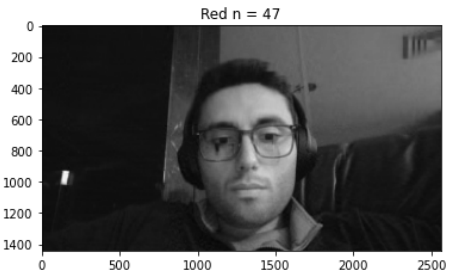

```
Recompiled image percent of storage memory= 1.3020833333333335 %
The error is 0.12919700802255085
```

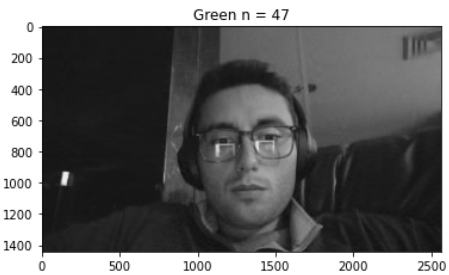


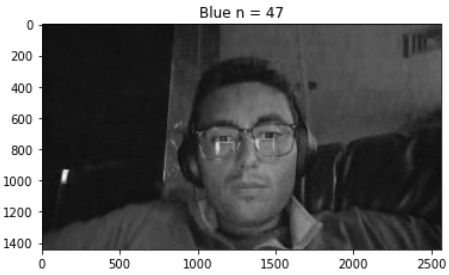


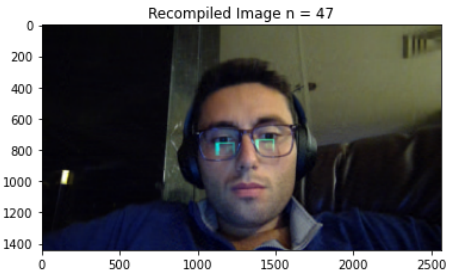



```
Recompiled image percent of storage memory= 4.014756944444445 %
The error is 0.06939482540346444
```





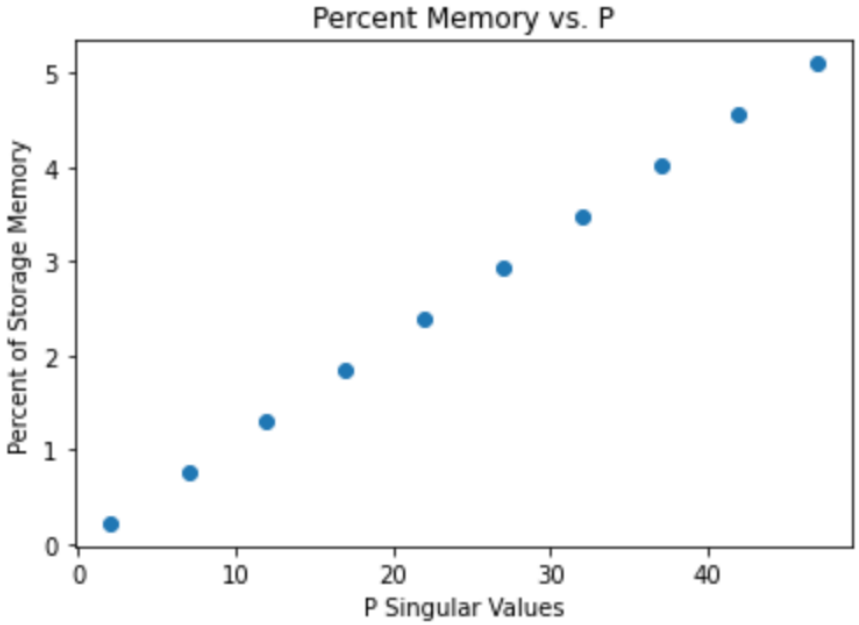


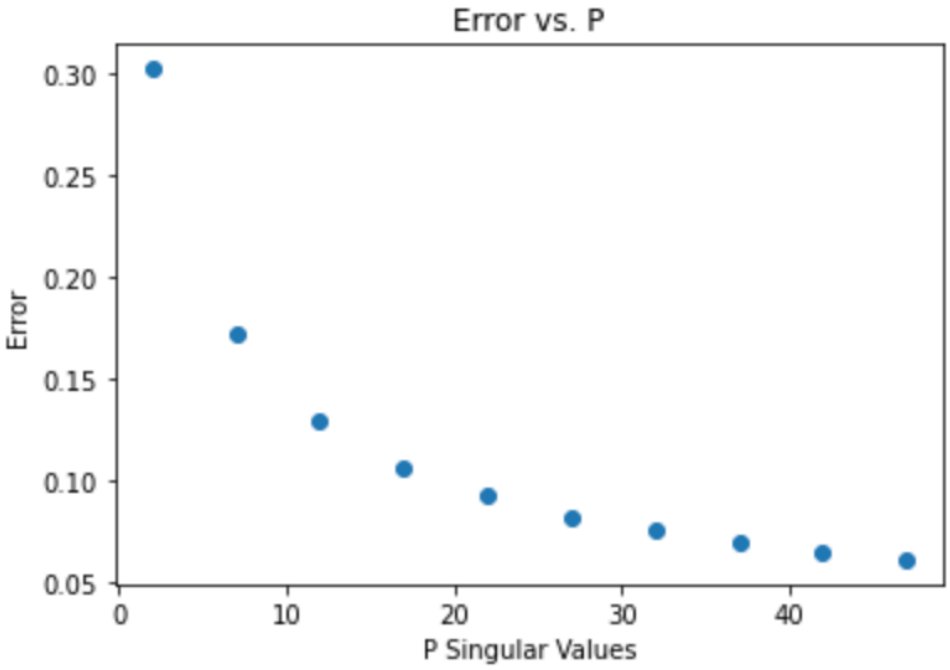



```
Recompiled image percent of storage memory= 5.099826388888888 %
The error is 0.060802337222840314
```

Note that as the n increases, the quality of the image increases.

Below are graphs showing the Percent Memory vs P and Error vs. P. Note that as the P singular values increase (quality of the compression increases), the percent of the memory increases, and the error compared to the original image decreases.





Conclusion

SVD is a powerful scientific computing tool which is applied in this project to image compression. As shown in the previous pages, first I apply SVD to compress a black & white image. Then, I apply SVD to color images by applying the algorithm to the 3 individual color matrices. Next, I explore the results of the image compression with various values of P. Finally, I explore the trends of percent memory and error for increasing values of P. As P singular values increase, the quality of the compressed image increases. Meanwhile, the percent memory of the compressed image increases, and the error decreases.

References Cited

[1] Marian Gidea. Eigenvalues and Singular Values Lecture

[2] Timothy Sauer. Numerical Analysis. 2nd Edition