Subhasis_Biswas_22571

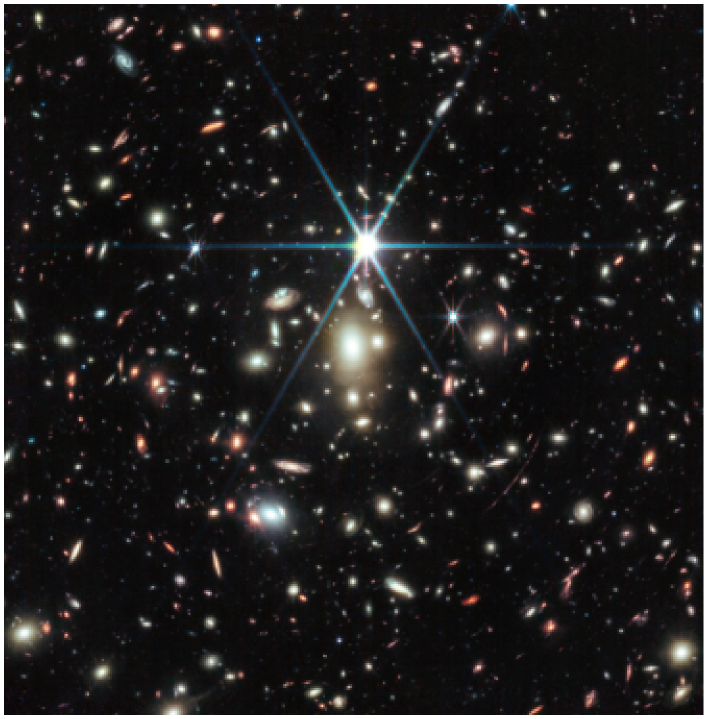
October 5, 2023

1 Setting up the environment

The original image on this local machine is saved as "*img.png*".

[1]:
```
import numpy as np
import matplotlib.pyplot as plt
from scipy.linalg import svd
import pandas as pd


#Loading the image

img_raw=plt.imread('img.png')

#print the first element of the first row of img_raw
print('The first element of the first row of img_raw is: ',img_raw[0,0])
print('The shape of img_raw is: ',img_raw.shape)
plt.imshow(img_raw)
#don't show the axes
plt.axis('off')
plt.show()
```

```
The first element of the first row of img_raw is:  [0.03529412 0.02352941
0.01568628]
The shape of img_raw is:  (2000, 1968, 3)
```



As we can see, the element at $(0,0)$ is not a scalar, rather a vector consisting of three componenets, namely the intensities of the Red, Green and Blue colour channels with values in the interval $[0,1]$. The *plt.imread*() shows the intensities as floats in $[0,1]$, whereas some other module may show the pixel data as unsigned 8bit integers in $[0,255]$ (image depth 8bit). Here we proceed with the floating point representation, since SVD will be performed on the matrices and orthonormality is required for the column vectors of U and V, there might be a significant loss of information while converting the floats of the $k-rank\ approximation$ to uint8.

The image is a 2000x1968x3 sized tensor, and we cannot directly apply SVD on the image itself. Therefore we intend to split this image into three channels and perform singular value decomposition on those respective channels and recombine the image to obtain a compressed version of the original image.

In order to not repeatedly write the code for splitting and recombining the channels, we write a function that takes an image tensor of the form $m \times n \times 3$ and returns the intensities of the red, green and the blue channels in respective order in the form of $m \times n$ matrices. We write another function that takes three matrices of the same size $m \times n$ as inputs in the same order as R,G,B and returns a tensor of the size $m \times n \times 3$.

Additionally, we define another function that will come in handy from time to time. This function takes the intensity of a single colour channel as a matrix M and a keyword $\alpha \in \{r, g, b\}$ as inputs and returns an image tensor that has only M matrix on channel α and the rest of the colour intensities are set to 0. This will help us to see the image in three different colour modes when necessary.

1.1 Defining the RGB Splitting and Re-Combining Functions

[2]:
```
def split_rgb(image_tensor):
    #Split the image tensor into its red, green and blue components.
    red = image_tensor[:,:,0]
    green = image_tensor[:,:,1]
    blue = image_tensor[:,:,2]
    return red, green, blue

def combine_rgb(red, green, blue):
    #Combine the red, green and blue components to form a new image tensor.
    #axis=2 means that the stacking will be along the third dimension
    return np.stack((red, green, blue), axis=2)


def single_channel_img(matrix,keyword:str):
    #create a blank image of size matrix.shape x 3
    #3 is for the three channels
    blank_image = np.zeros((matrix.shape[0],matrix.shape[1],3))
    img_new = blank_image.copy()
    if keyword=='r':
        #assign the red channel of the image to be matrix
        img_new[:,:,0]=matrix
    elif keyword=='g':
        #assign the green channel of the image to be matrix
        img_new[:,:,1]=matrix
    elif keyword=='b':
        #assign the blue channel of the image to be matrix
        img_new[:,:,2]=matrix
    else:
        print('Wrong keyword')
        return None
    return img_new
```

1.2 Defining the Singular Value Decomposition and k-rank Approximation Functions

[3]:
```
def perform_svd(matrix):
        #Perform SVD on the input matrix and return the U, S and V^T matrices␣
↪from the decomposition.
        U, S, VT = svd(matrix)
```

```
        return U, S, VT

def low_rank_approx(U,S,VT,k):
        #Perform low rank approximation of the input matrix using the first k
 ↪singular values.
        #reconstruct the image using the first k singular values
        S = S[:k]
        U = U[:,:k]
        VT = VT[:k,:]
        return U@np.diag(S)@VT  # @ is matrix multiplication in numpy module
```

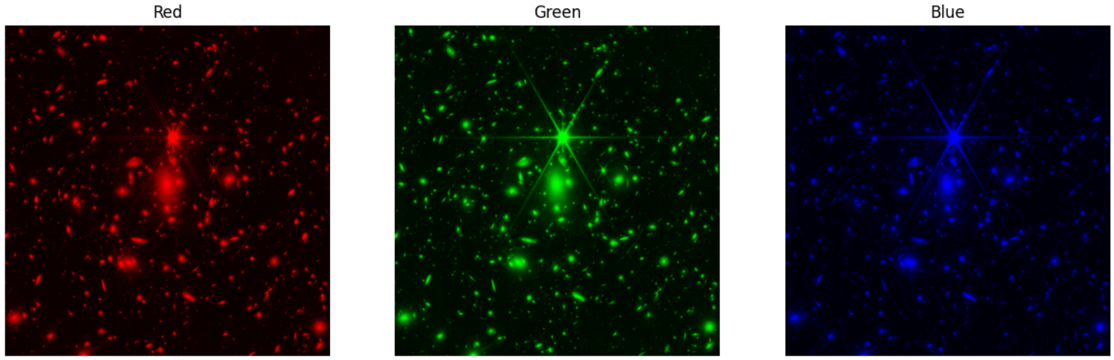
1.3 Error Measurement

The following function takes two matrices A and B as inputs and returns the Frobenius and the 2-Norm of $(A - B)$ as a tuple.

[4]:
```
def error(A,B):
    #Compute the Frobenius and 2-norm error between the matrices A and B.
    return np.linalg.norm(A-B), np.linalg.norm(A-B,ord=2)
```

1.4 The original image on three different channels

[5]:
```
original_red, original_green, original_blue = split_rgb(img_raw)

fig, ax = plt.subplots(1,3,figsize=(15,15))
ax[0].imshow(single_channel_img(original_red,'r'))
ax[0].axis('off')
ax[0].set_title('Red')
ax[1].imshow(single_channel_img(original_green,'g'))
ax[1].axis('off')
ax[1].set_title('Green')
ax[2].imshow(single_channel_img(original_blue,'b'))
ax[2].axis('off')
ax[2].set_title('Blue')
plt.show()
```



2 Problem 2(a) & 2(c)

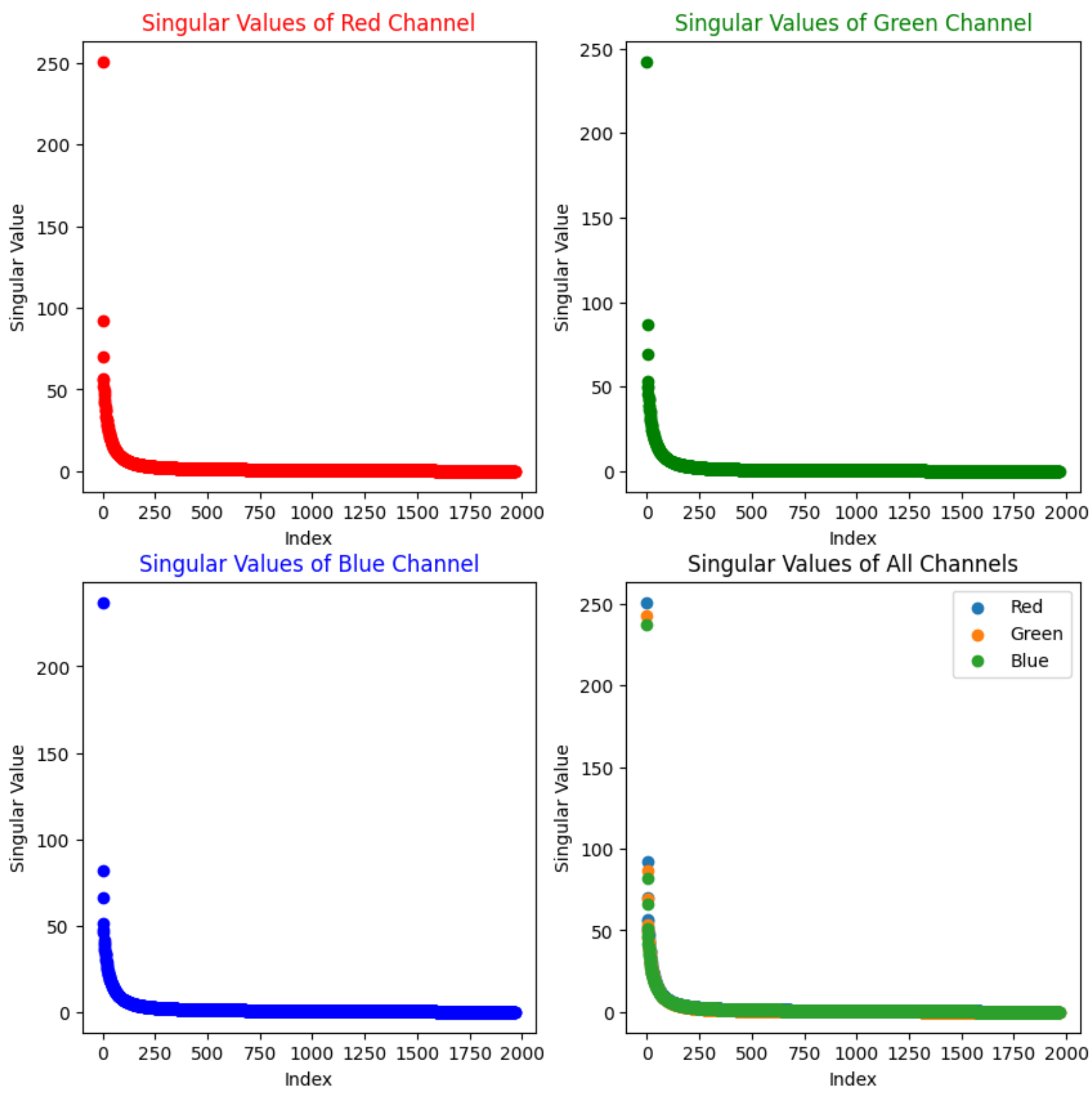
As per the problems, we first plot the singular values corresponding to the three channels to get a better idea regarding the minimum number of singular values required to make the compressed image almost indistinguishable from the original one.

2.1 Plotting the singular values of R, G, B intensity matrices.

[6]:
```
#Perform SVD on the red channel
U_r, S_r, VT_r = perform_svd(original_red)
#Perform SVD on the green channel
U_g, S_g, VT_g = perform_svd(original_green)
#Perform SVD on the blue channel
U_b, S_b, VT_b = perform_svd(original_blue)
```

[7]:
```
fig, ax = plt.subplots(2,2,figsize=(10,10))
ax[0,0].scatter(np.arange(len(S_r)),S_r,color='r')
ax[0,0].set_title('Singular Values of Red Channel',c='r')
ax[0,0].set_xlabel('Index')
ax[0,0].set_ylabel('Singular Value')
ax[0,0].set_xticks(np.arange(0,2250,250))
ax[0,1].scatter(np.arange(len(S_g)),S_g,color='g')
ax[0,1].set_title('Singular Values of Green Channel',c='g')
ax[0,1].set_xlabel('Index')
ax[0,1].set_ylabel('Singular Value')
ax[0,1].set_xticks(np.arange(0,2250,250))
ax[1,0].scatter(np.arange(len(S_b)),S_b,color='b')
ax[1,0].set_title('Singular Values of Blue Channel',c='b')
ax[1,0].set_xlabel('Index')
ax[1,0].set_ylabel('Singular Value')
```

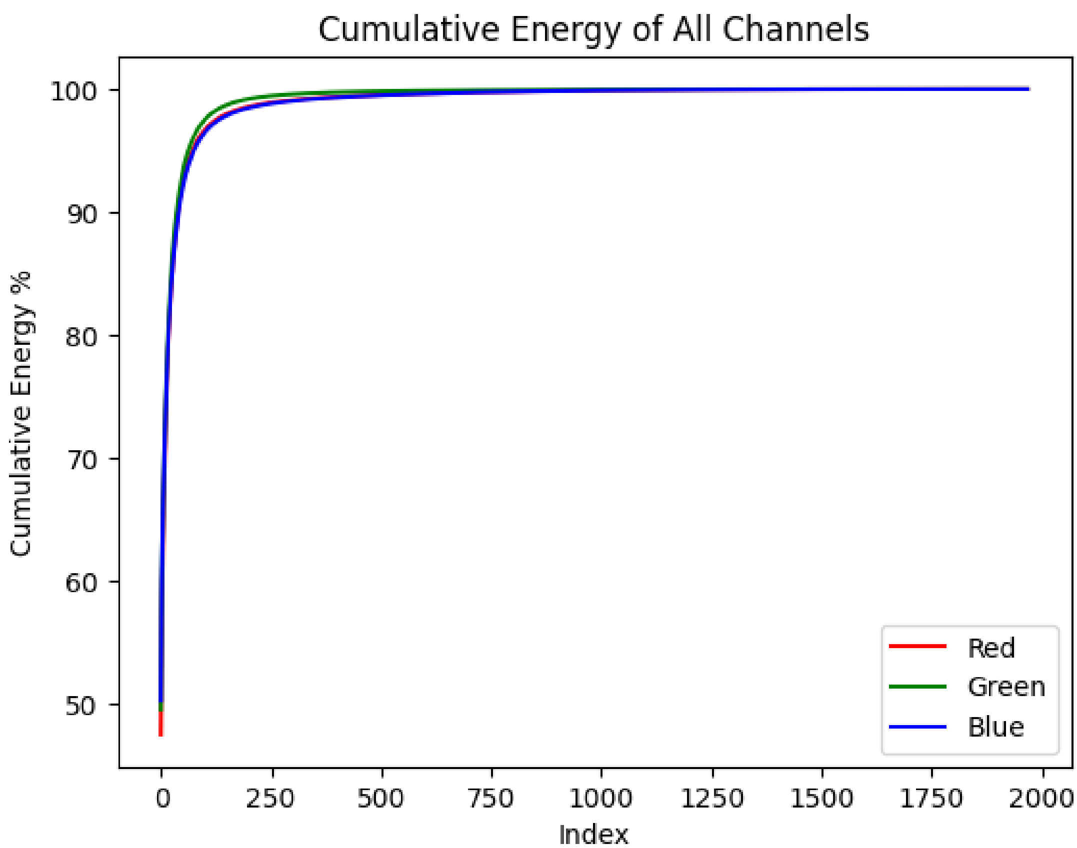
```python
ax[1,0].set_xticks(np.arange(0,2250,250))
ax[1,1].scatter(np.arange(len(S_r)),S_r,label='Red')
ax[1,1].scatter(np.arange(len(S_g)),S_g,label='Green')
ax[1,1].scatter(np.arange(len(S_b)),S_b,label='Blue')
ax[1,1].set_title('Singular Values of All Channels')
ax[1,1].set_xlabel('Index')
ax[1,1].set_ylabel('Singular Value')
ax[1,1].set_xticks(np.arange(0,2250,250))
ax[1,1].legend()
plt.show()
```



As we can observe from the plots, most of the "energy" of all the three matrices are being captured within the first 250 of the singular values. Before we proceed to approximate the image corresponding to different approximations, we plot the cumulative % plot for the information captured. $\sum_i \sigma_i^2$ is the total energy captured by the matrices.

[8]:
```
#cumulative percentage of the total energy contained
#squaring the singular values for the energy captured

cumulative_energy_r = np.cumsum(S_r**2)/np.sum(S_r**2)*100
cumulative_energy_g = np.cumsum(S_g**2)/np.sum(S_g**2)*100
cumulative_energy_b = np.cumsum(S_b**2)/np.sum(S_b**2)*100

print('The first 250 cumulative energy of the red channel is:␣
 ↪',cumulative_energy_r[250])
print('The first 500 cumulative energy of the red channel is:␣
 ↪',cumulative_energy_r[500])
plt.figsize=(10,10)
plt.plot(np.
 ↪arange(len(cumulative_energy_r)),cumulative_energy_r,color='r',label='Red')
plt.plot(np.
 ↪arange(len(cumulative_energy_g)),cumulative_energy_g,color='g',label='Green')
plt.plot(np.
 ↪arange(len(cumulative_energy_b)),cumulative_energy_b,color='b',label='Blue')
plt.title('Cumulative Energy of All Channels')
plt.xlabel('Index')
plt.ylabel('Cumulative Energy %')
plt.xticks(np.arange(0,2250,250))
plt.legend()
plt.show()
```

```
The first 250 cumulative energy of the red channel is:  98.89116
The first 500 cumulative energy of the red channel is:  99.51962
```



2.2 Images for different k-rank approximations

2.2.1 Defining a helper function *image_approx*

This function uses the decomposition of the matrices r, g, b and takes k as an input to do the following:

1. Obtain the k-rank approximation of the matrices
2. If an entry γ in the approximated channel is $\gamma < 0$, set $\gamma = 0$, if $\gamma > 1$, set $\gamma = 1$.
3. Combine the three approximations into tensor of the same size as the original image.
4. Frobenius error, 2-Norm error, $(k+1)$ th singular value (ideal 2-norm), $\sqrt{\sum_{k+1}^{1968} \sigma_i^2}$ (ideal Frobenius norm) alongside the total entries sent back to earth is recorded as a row in a dataframe. We record the comparisons twice, before and after chopping off the illegal values.

   Note: In the code, we need to shift the indices by -1, as the indexing starts from 0, i.e the $(k+1)$ th singular value has the index k in the array.
5. Return the compressed image and the dataframes (before and after).

For approximating the image with the first k singular values, we get back $k(1928+2000+1)$ entries each for R, G, B. The reasoning is that we send back the first k columns of the matrix V^T, so

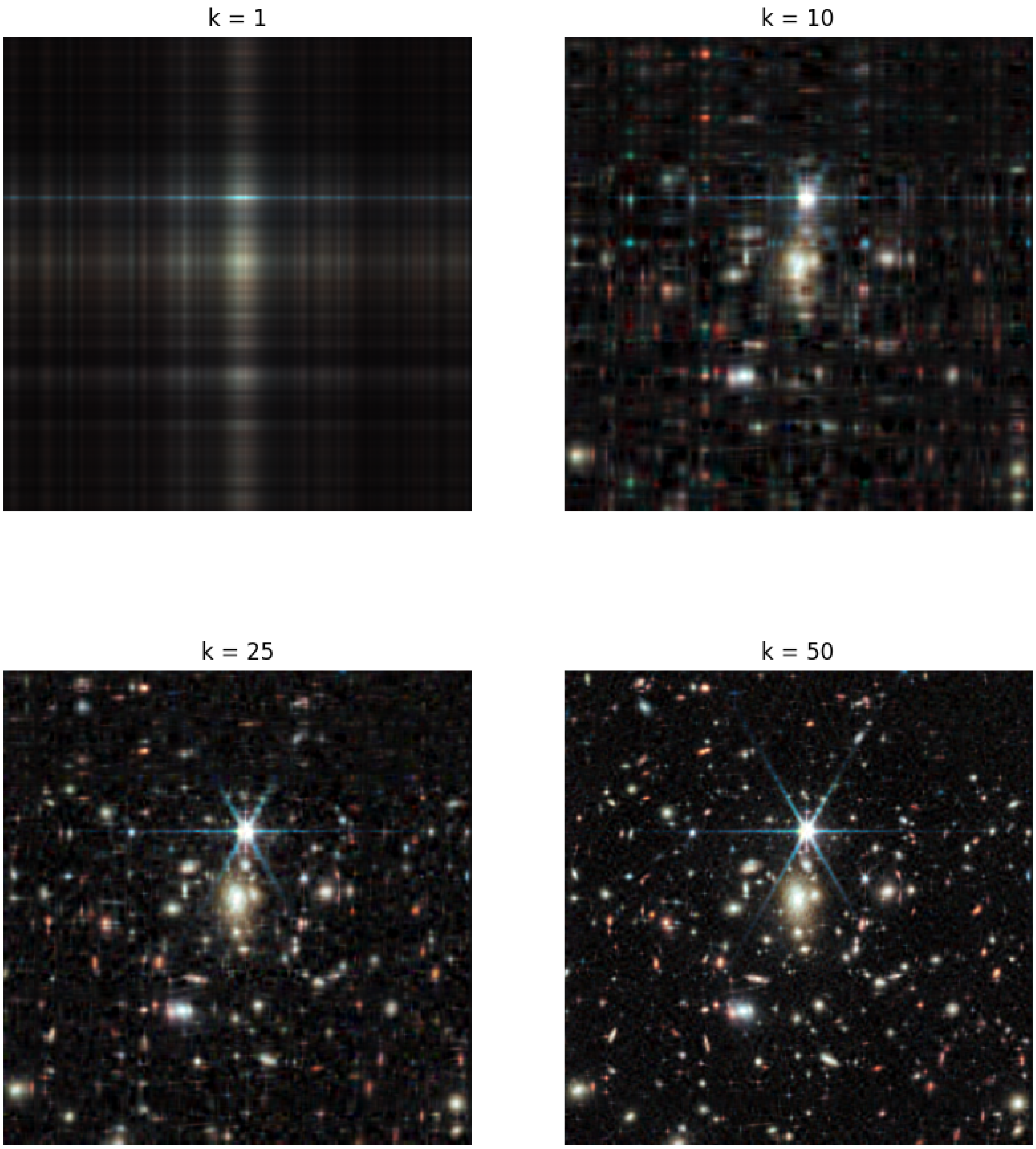
1968k entries, the first k singular values and the first k columns of the matrix U, i.e 2000k entries. Therefore one needs to send a total of $3k(1968 + 2000 + 1)$ entries to Earth, combining all the channels.

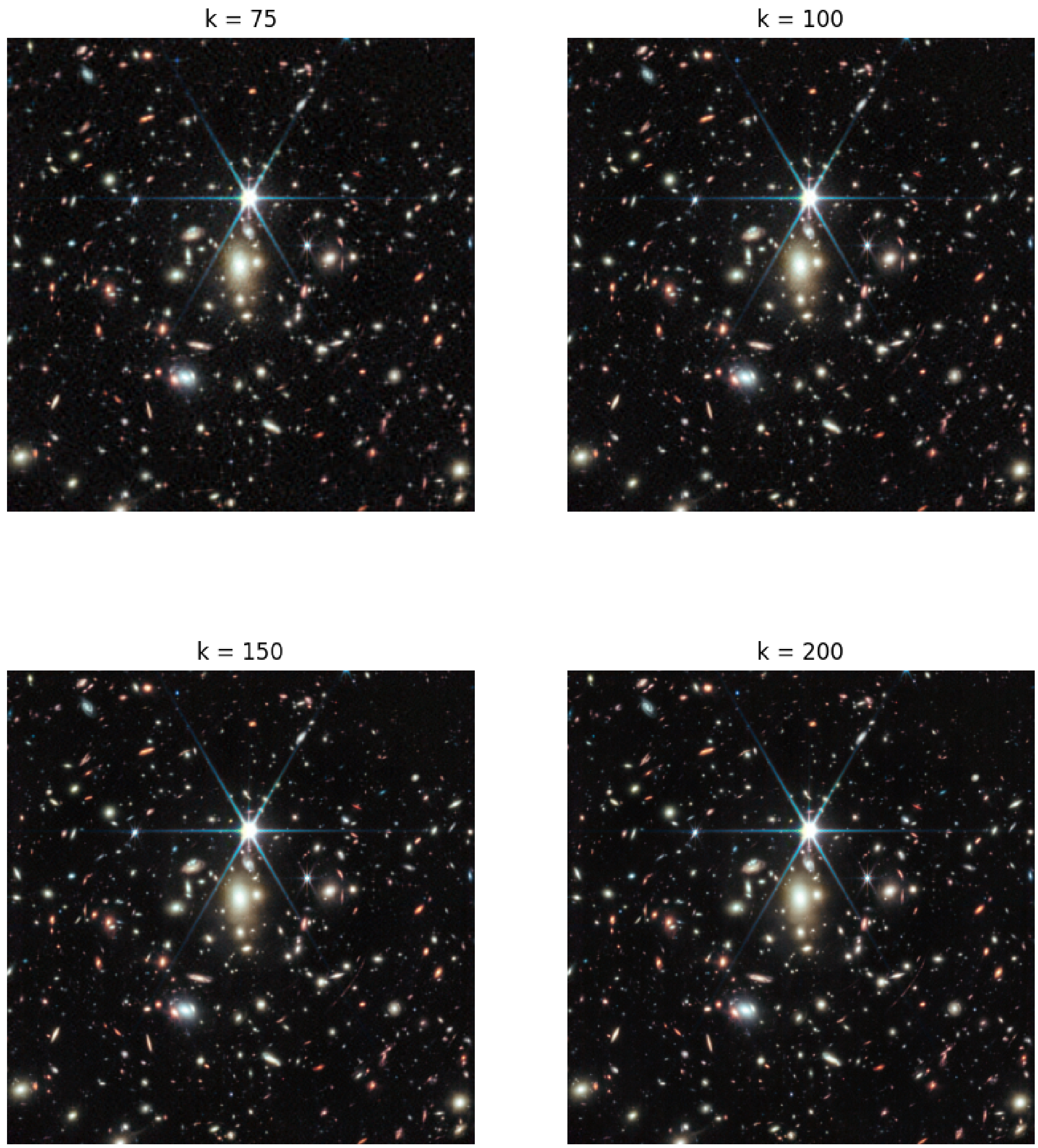
[9]:
```
r=split_rgb(img_raw)[0]
g=split_rgb(img_raw)[1]
b=split_rgb(img_raw)[2]

U_r, S_r, VT_r = perform_svd(r)
U_g, S_g, VT_g = perform_svd(g)
U_b, S_b, VT_b = perform_svd(b)


def image_approx(k):
    r_new = low_rank_approx(U_r,S_r,VT_r,k)
    g_new = low_rank_approx(U_g,S_g,VT_g,k)
    b_new = low_rank_approx(U_b,S_b,VT_b,k)

    ### k+1 th singular value has the index k ###
    # Measuring the error before clipping the values of the matrices to be␣
 ↪between 0 and 1.

    r_errors_before_clipping = error(r,r_new), np.sqrt(np.sum(S_r[k:]**2)),␣
 ↪S_r[k]
    g_errors_before_clipping = error(g,g_new), np.sqrt(np.sum(S_g[k:]**2)),␣
 ↪S_g[k]
    b_errors_before_clipping = error(b,b_new), np.sqrt(np.sum(S_b[k:]**2)),␣
 ↪S_b[k]

    r_new=np.clip(r_new,0,1)
    g_new=np.clip(g_new,0,1)
    b_new=np.clip(b_new,0,1)

    #ideal Frobenius norm error is np.sqrt(np.sum(S_r[k:]**2))
    #ideal 2-norm error is S_r[k]


    ### k+1 th singular value has the index k ###
    # Measuring the error after clipping the values of the matrices to be␣
 ↪between 0 and 1.

    r_errors_after_clipping = error(r,r_new), np.sqrt(np.sum(S_r[k:]**2)),␣
 ↪S_r[k]
    g_errors_after_clipping = error(g,g_new), np.sqrt(np.sum(S_g[k:]**2)),␣
 ↪S_g[k]
```

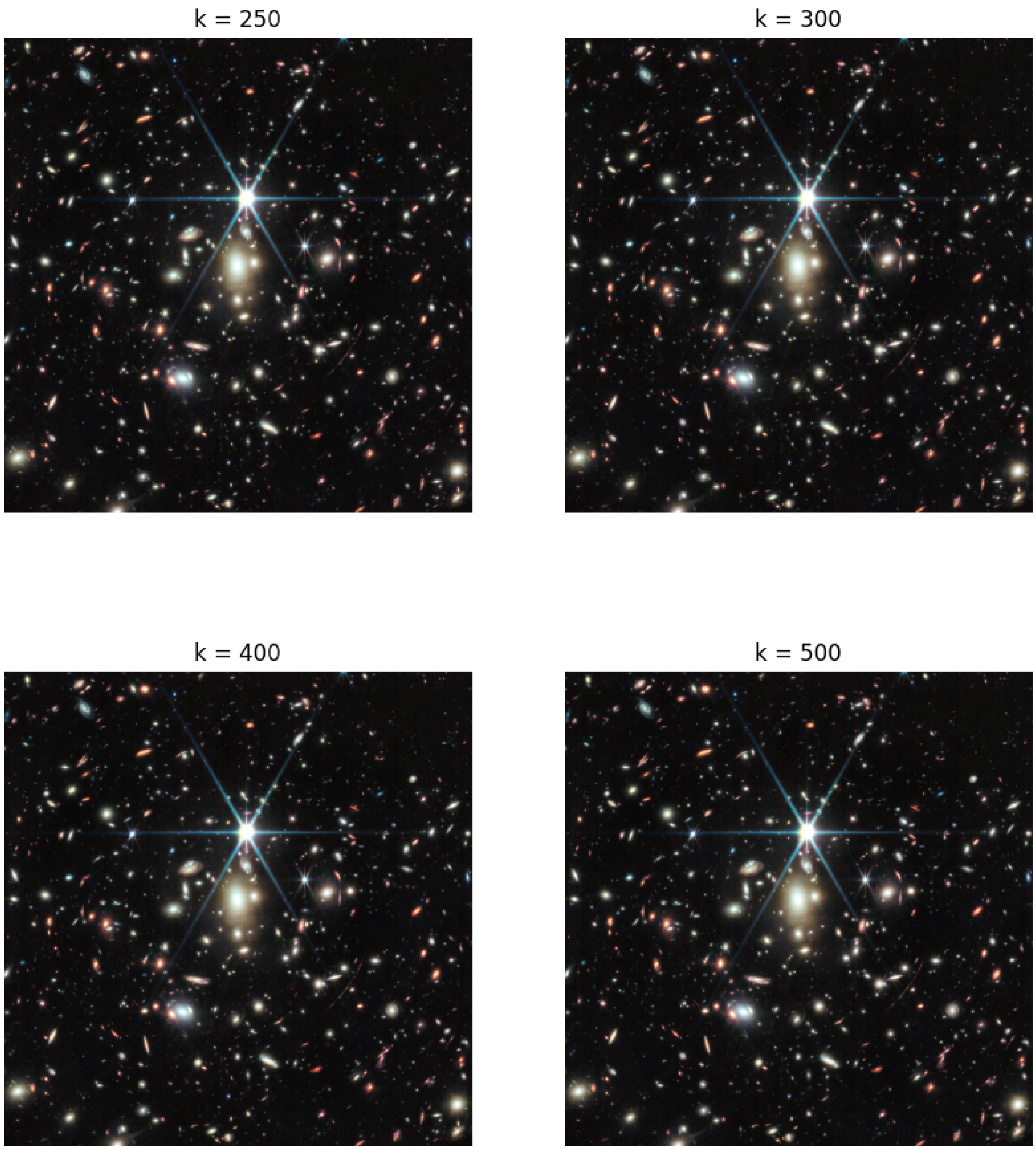
```
    b_errors_after_clipping = error(b,b_new), np.sqrt(np.sum(S_b[k:]**2)),␣
↪S_b[k]




    compressed_img = combine_rgb(r_new,g_new,b_new)

    #read the size of the compressed image

    entries=k*(r.shape[0]+r.shape[1]+1)+k*(g.shape[0]+g.shape[1]+1)+k*(b.
↪shape[0]+b.shape[1]+1)

    error_data_before_clipping=pd.DataFrame({
        'k':[k], 'Frobenius Error (R)': [r_errors_before_clipping[0][0]],␣
↪'Ideal Frobenius Error (R)': [r_errors_before_clipping[1]], '2-norm Error␣
↪(R)': [r_errors_before_clipping[0][1]], 'Ideal 2-norm Error (R)':␣
↪[r_errors_before_clipping[2]],
        'Frobenius Error (G)': [g_errors_before_clipping[0][0]], 'Ideal␣
↪Frobenius Error (G)': [g_errors_before_clipping[1]], '2-norm Error (G)':␣
↪[g_errors_before_clipping[0][1]], 'Ideal 2-norm Error (G)':␣
↪[g_errors_before_clipping[2]],
        'Frobenius Error (B)': [b_errors_before_clipping[0][0]], 'Ideal␣
↪Frobenius Error (B)': [b_errors_before_clipping[1]], '2-norm Error (B)':␣
↪[b_errors_before_clipping[0][1]], 'Ideal 2-norm Error (B)':␣
↪[b_errors_before_clipping[2]],
    })

    error_data_after_clipping=pd.DataFrame({
        'k':[k], 'Frobenius Error (R)': [r_errors_after_clipping[0][0]], 'Ideal␣
↪Frobenius Error (R)': [r_errors_after_clipping[1]], '2-norm Error (R)':␣
↪[r_errors_after_clipping[0][1]], 'Ideal 2-norm Error (R)':␣
↪[r_errors_after_clipping[2]],
        'Frobenius Error (G)': [g_errors_after_clipping[0][0]], 'Ideal␣
↪Frobenius Error (G)': [g_errors_after_clipping[1]], '2-norm Error (G)':␣
↪[g_errors_after_clipping[0][1]], 'Ideal 2-norm Error (G)':␣
↪[g_errors_after_clipping[2]],
        'Frobenius Error (B)': [b_errors_after_clipping[0][0]], 'Ideal␣
↪Frobenius Error (B)': [b_errors_after_clipping[1]], '2-norm Error (B)':␣
↪[b_errors_after_clipping[0][1]], 'Ideal 2-norm Error (B)':␣
↪[b_errors_after_clipping[2]],
        'Entries Sent': [entries]
    })

    return compressed_img, error_data_before_clipping, error_data_after_clipping
```

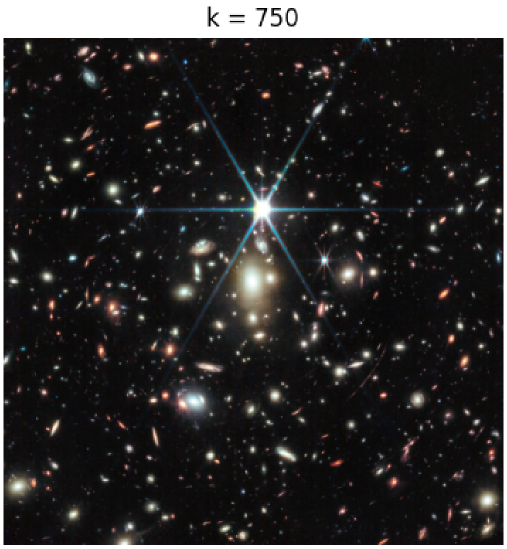
2.2.2 Images obtained for different k.

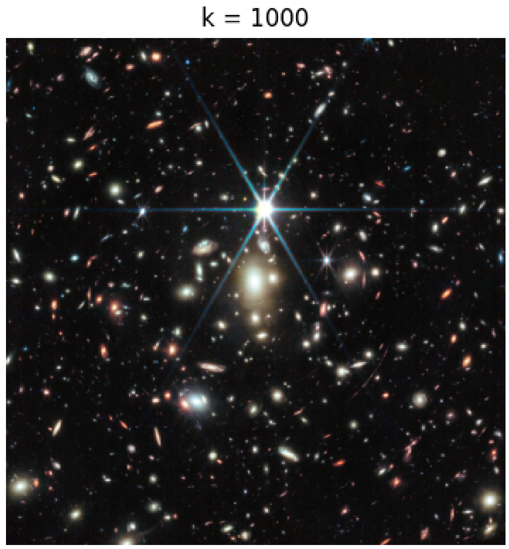
[10]:
```
error_df_after_clip=pd.DataFrame(columns=['k','Frobenius Error (R)','Ideal␣
↪Frobenius Error (R)','2-norm Error (R)','Ideal 2-norm Error (R)',
                                  'Frobenius Error (G)','Ideal Frobenius Error␣
↪(G)','2-norm Error (G)','Ideal 2-norm Error (G)',
                                  'Frobenius Error (B)','Ideal Frobenius Error␣
↪(B)','2-norm Error (B)','Ideal 2-norm Error (B)','Entries Sent'])

error_df_before_clip=pd.DataFrame(columns=['k','Frobenius Error (R)','Ideal␣
↪Frobenius Error (R)','2-norm Error (R)','Ideal 2-norm Error (R)',
                                     'Frobenius Error (G)','Ideal Frobenius␣
↪Error (G)','2-norm Error (G)','Ideal 2-norm Error (G)',
                                     'Frobenius Error (B)','Ideal Frobenius␣
↪Error (B)','2-norm Error (B)','Ideal 2-norm Error (B)'])



list_k=[1,10,25,50,75,100,150,200,250,300,400,500,750,1000,1250,1500]

for i in range(int(len(list_k)/2)):
    compressed_img1,error_data1_before ,error_data1_after =␣
↪image_approx(list_k[2*i])
    compressed_img2,error_data2_before ,error_data2_after =␣
↪image_approx(list_k[2*i+1])
    error_df_after_clip=pd.
↪concat([error_df_after_clip,error_data1_after,error_data2_after],ignore_index=True)
    error_df_before_clip=pd.
↪concat([error_df_before_clip,error_data1_before,error_data2_before],ignore_index=True)

    fig, ax = plt.subplots(1,2,figsize=(10,10))
    ax[0].imshow(compressed_img1)
    ax[0].axis('off')
    ax[0].set_title('k = '+str(list_k[2*i]))
    ax[1].imshow(compressed_img2)
    ax[1].axis('off')
    ax[1].set_title('k = '+str(list_k[2*i+1]))
    plt.show()


error_df_before_clip.to_csv('error_data_before_clip.csv',index=False) #saving␣
↪the prior error data to a csv file
error_df_after_clip.to_csv('error_data_after_clip.csv',index=False) #saving the␣
↪posterior error data to a csv file
```

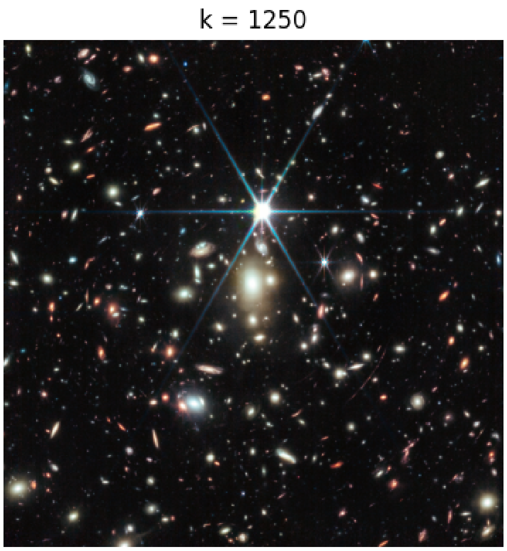

k = 1
k = 10
k = 25
k = 50


k = 75
k = 100
k = 150
k = 200


k = 250
k = 300
k = 400
k = 500







The following table contains the errors and the entries sent for different values of k. Following are the abbreviations used in the header:

- $FE\ (c) :=$ Frobenius Error in channel c
- $IFE\ (c) :=$ Ideal Frobenius Error in channel c
- $2E\ (c) :=$ 2-norm Error in channel c
- $I2E\ (c) :=$ Ideal 2-norm Error in channel c

Before Clipping

k	FE (R)	IFE (R)	2E (R)	I2E (R)	FE (G)	IFE (G)	2E (G)	I2E (G)	FE (B)	IFE (B)	2E (B)	I2E (B)
1	263.82364	263.82983	91.78814	91.78815	244.88927	244.8957	86.52972	86.52969	235.96927	235.97462	81.67768	81.67768
10	196.10655	196.11082	42.79276	42.79277	180.37108	180.37465	38.84556	38.84557	176.78348	176.78662	36.64282	36.64283
25	144.27325	144.276	26.10782	26.10782	131.39792	131.40039	24.49553	24.49553	132.72484	132.72716	23.28854	23.28854
50	101.51534	101.51711	15.3626	15.3626	90.20885	90.21011	14.61239	14.61239	95.48482	95.48635	14.2475	14.24751
75	79.56965	79.57091	10.25232	10.25232	68.001	68.00219	9.38636	9.38636	75.3294	75.33059	9.43965	9.43965
100	66.48619	66.48712	7.42038	7.42038	54.6604	54.66123	6.79449	6.7945	63.11312	63.11388	7.02153	7.02154
150	51.80519	51.80595	4.65542	4.65542	39.37001	39.3706	4.16215	4.16215	49.22598	49.2266	4.39074	4.39074
200	43.68872	43.68929	3.33135	3.33135	30.97953	30.97993	2.80337	2.80337	41.51716	41.51767	3.14446	3.14446
250	38.42093	38.42147	2.57446	2.57446	25.82411	25.82444	2.07056	2.07056	36.53117	36.53164	2.4573	2.4573
300	34.66683	34.66726	2.12311	2.12311	22.3583	22.35856	1.61301	1.613	32.93604	32.9364	2.03509	2.03509
400	29.30243	29.30279	1.62943	1.62943	17.81466	17.81488	1.11785	1.11785	27.71697	27.71727	1.55349	1.55349
500	25.267	25.26732	1.35195	1.35195	14.87342	14.8736	0.86325	0.86325	23.84906	23.84932	1.28498	1.28498
750	17.69354	17.69376	0.95519	0.95519	10.06994	10.07006	0.55566	0.55566	16.69879	16.69897	0.89937	0.89937
1000	12.00128	12.00142	0.69513	0.69513	6.76112	6.76119	0.39434	0.39434	11.35152	11.35163	0.65656	0.65656
1250	7.48504	7.48512	0.49042	0.49046	4.20501	4.20506	0.27603	0.27603	7.09979	7.09987	0.4633	0.4633
1500	3.92492	3.92496	0.31339	0.31339	2.20183	2.20186	0.17555	0.17555	3.72476	3.7248	0.29653	0.29653

After Clipping

k	FE (R)	IFE (R)	2E (R)	I2E (R)	FE (G)	IFE (G)	2E (G)	I2E (G)	FE (B)	IFE (B)	2E (B)	I2E (B)	Entries Sent
1	263.824	263.83	91.788	91.788	244.888	244.896	86.538	86.53	235.956	235.975	81.69	81.678	11907
10	194.966	196.111	42.065	42.793	179.315	180.375	38.38	38.846	175.87	176.787	35.96	36.643	119070
25	143.064	144.276	25.543	26.108	130.233	131.4	23.854	24.496	131.897	132.727	22.761	23.289	297675
50	100.998	101.517	15.136	15.363	89.764	90.21	14.36	14.612	95.076	95.486	14.053	14.248	595350
75	79.372	79.571	10.178	10.252	67.845	68.002	9.336	9.386	75.155	75.331	9.381	9.44	893025
100	66.392	66.487	7.396	7.42	54.604	54.661	6.778	6.794	63.034	63.114	6.995	7.022	1190700
150	51.771	51.806	4.649	4.655	39.354	39.371	4.158	4.162	49.2	49.227	4.383	4.391	1786050
200	43.667	43.689	3.33	3.331	30.972	30.98	2.802	2.803	41.501	41.518	3.14	3.144	2381400
250	38.4	38.421	2.575	2.574	25.819	25.824	2.069	2.071	36.52	36.532	2.455	2.457	2976750
300	34.647	34.667	2.121	2.123	22.354	22.359	1.612	1.613	32.926	32.936	2.034	2.035	3572100
400	29.283	29.303	1.628	1.629	17.812	17.815	1.118	1.118	27.706	27.717	1.552	1.553	4762800
500	25.249	25.267	1.35	1.352	14.872	14.874	0.863	0.863	23.838	23.849	1.284	1.285	5953500
750	17.682	17.694	0.954	0.955	10.068	10.07	0.556	0.556	16.687	16.699	0.899	0.899	8930250
1000	11.994	12.001	0.694	0.695	6.76	6.761	0.394	0.394	11.343	11.352	0.656	0.657	11907000
1250	7.481	7.485	0.49	0.49	4.204	4.205	0.276	0.276	7.095	7.1	0.463	0.463	14883750
1500	3.923	3.925	0.313	0.313	2.201	2.202	0.175	0.176	3.723	3.725	0.296	0.297	17860500

From the first table, it's evident that the theorem holds empirically as well. The slight mismatches are due to round-off errors.

In the second table, the differences are larger due to the matrices being modified to only contain values in $[0, 1]$.

3 Problem 2(b)

Visually, there seems to be almost no difference between $k = 100$ and the original image. For $k = 75$ the image seems a bit blurry. We now check out the energy captured by our different choices of k and the % of amount of entries that we need to send as compared to the original $2000 \times 1968 \times 3 = 11808000$ entries.

[11]:
```
No_entries=2000*1968*3
k_list=[75,100,150,200,250,300,400,500]
for i in k_list:
    print('Cumulative energy for RGB with k=',i,':␣
 ↪',cumulative_energy_r[i],cumulative_energy_g[i],cumulative_energy_b[i])
    print('% of entries required for k=',i,': ',␣
 ↪round(error_df_after_clip[error_df_after_clip['k']==i]['Entries Sent'].
 ↪values[0]/No_entries*100,2))
```

```
Cumulative energy for RGB with k= 75 :  95.30186 96.1799 95.0098
% of entries required for k= 75 :  7.56
Cumulative energy for RGB with k= 100 :  96.706024 97.5227 96.48528
```

```
% of entries required for k= 100 :  10.08
Cumulative energy for RGB with k= 150 :  97.99125 98.709236 97.85228
% of entries required for k= 150 :  15.13
Cumulative energy for RGB with k= 200 :  98.56814 99.19838 98.468834
% of entries required for k= 200 :  20.17
Cumulative energy for RGB with k= 250 :  98.89116 99.44199 98.81307
% of entries required for k= 250 :  25.21
Cumulative energy for RGB with k= 300 :  99.0966 99.58121 99.034515
% of entries required for k= 300 :  30.25
Cumulative energy for RGB with k= 400 :  99.3541 99.733765 99.315765
% of entries required for k= 400 :  40.34
Cumulative energy for RGB with k= 500 :  99.51962 99.81431 99.493286
% of entries required for k= 500 :  50.42
```

Depending on the constraints on board, if high compression is required while preserving more than 96% of the details, one can select $k = 100$. This achieves $\approx 90\%$ compression.

If 99% of the details is a bare minimum, $k = 300$ can be selected, which still achieves around 70% compression. Going beyond this seems like an overkill, which worsens compression and it adds almost nothing to the details.

Taking a value in-between the two, $k = 200$ would be my personal choice, as it maintains 98% of the original data and around 80% reduction is achieved.