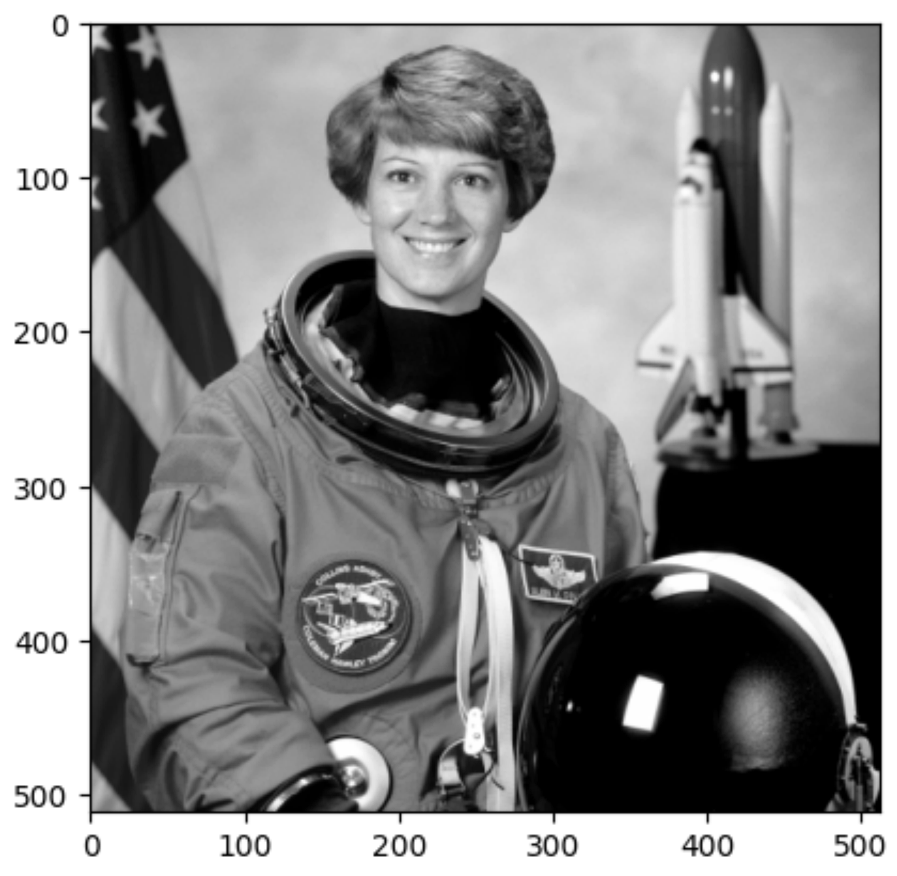
In [18]:
```python
import numpy as np
import matplotlib.pyplot as plt
import skimage
from skimage.data import astronaut
from skimage.color import rgb2gray
```

Compute SVD

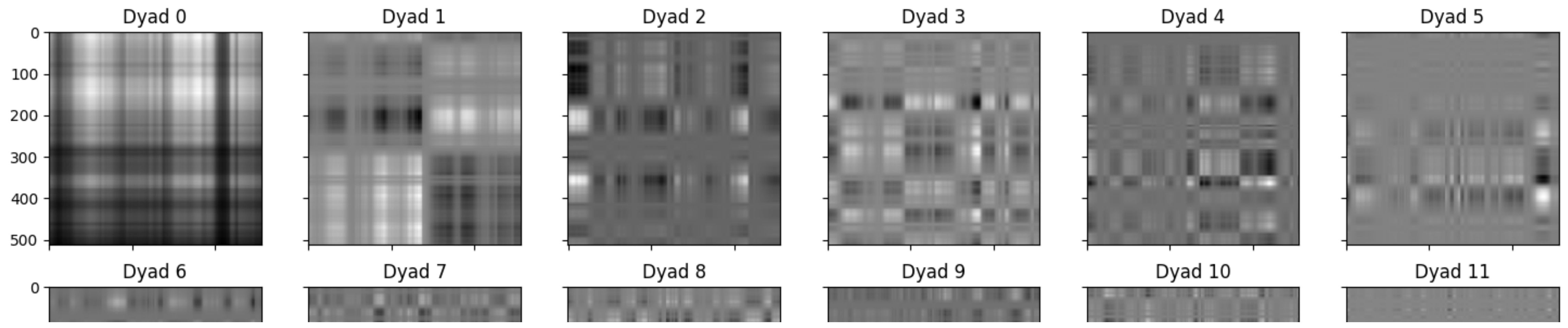
In [20]:
```python
img = rgb2gray(astronaut())
print(f'Image shape: {img.shape}')
U, s, VT = np.linalg.svd(img, full_matrices=True)

plt.imshow(img, cmap='gray')
plt.show()
```

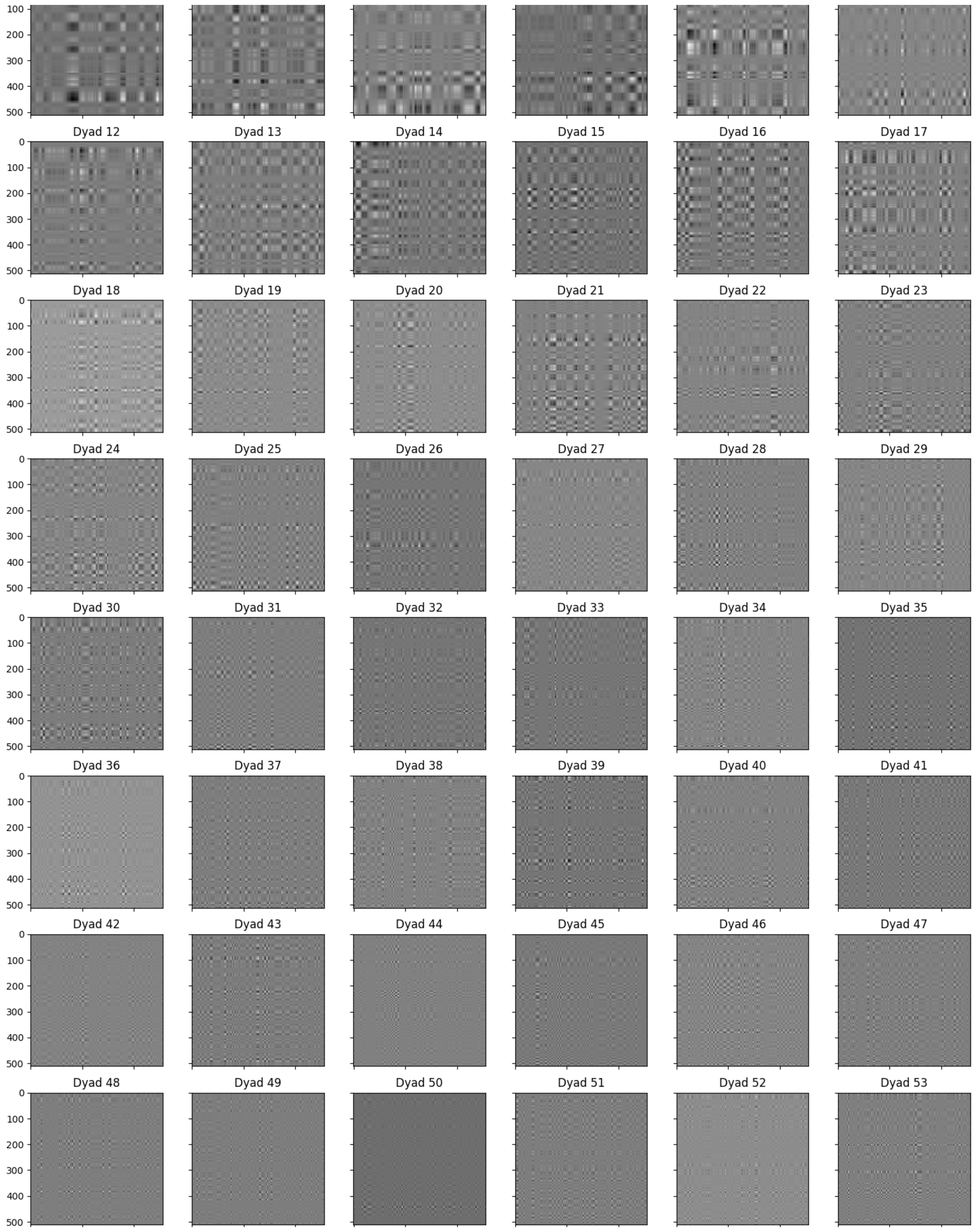
```
Image shape: (512, 512)
```

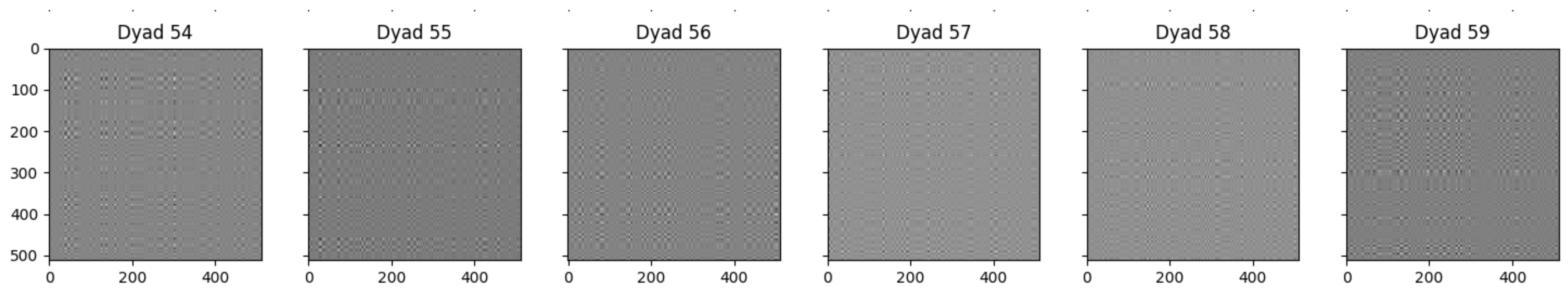


Visualizing dyads

In [38]:
```python
nrows = 10
ncols = 6
n_dyads = ncols*nrows
dyads = []

fig, axs = plt.subplots(figsize=(3*ncols, 3*nrows), nrows=nrows, ncols=ncols, sharex=True, sharey=True)

iter = 0
for i in range(nrows):
    for j in range(ncols):
        # given two 1-dimensional vectors, np.outer returns a MATRIX with the PAIRWISE product of the two vectors
        dyad = np.outer(U[:, iter] * s[iter], VT[iter, :])
        ax = axs[i][j]
        ax.imshow(dyad, cmap='gray')
        ax.set_title(f"Dyad {iter}")

        iter+=1
```



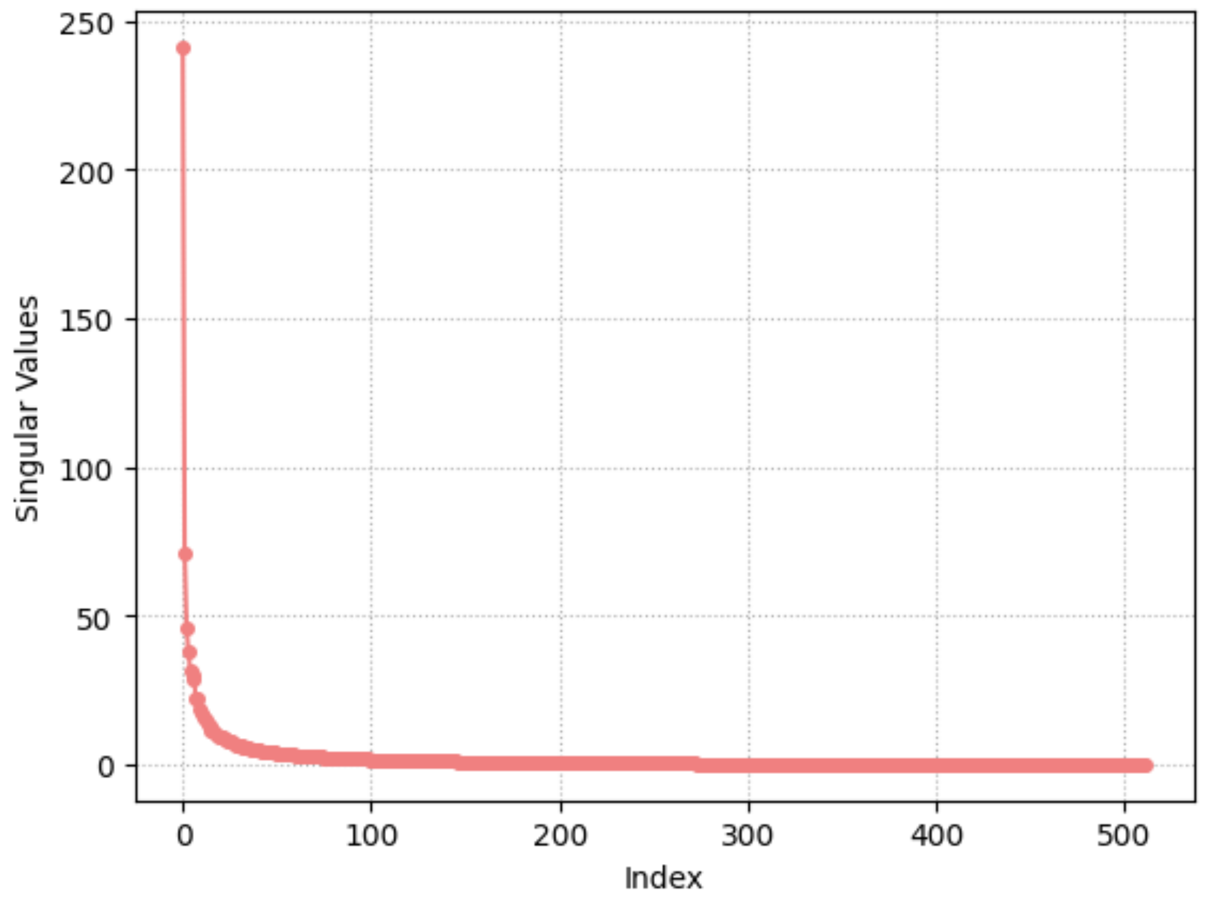

Dyad 12
Dyad 13
Dyad 14
Dyad 15
Dyad 16
Dyad 17
Dyad 18
Dyad 19
Dyad 20
Dyad 21
Dyad 22
Dyad 23
Dyad 24
Dyad 25
Dyad 26
Dyad 27
Dyad 28
Dyad 29
Dyad 30
Dyad 31
Dyad 32
Dyad 33
Dyad 34
Dyad 35
Dyad 36
Dyad 37
Dyad 38
Dyad 39
Dyad 40
Dyad 41
Dyad 42
Dyad 43
Dyad 44
Dyad 45
Dyad 46
Dyad 47
Dyad 48
Dyad 49
Dyad 50
Dyad 51
Dyad 52
Dyad 53
0
100
200
300
400
500



We can notice that the information contained in each dyad decreases when increasing the index i. This is a consequence of the fact that the i-th singular value σ_i represent the importance of the i-th dyad.

Plot the singular values of the matrix representing the image

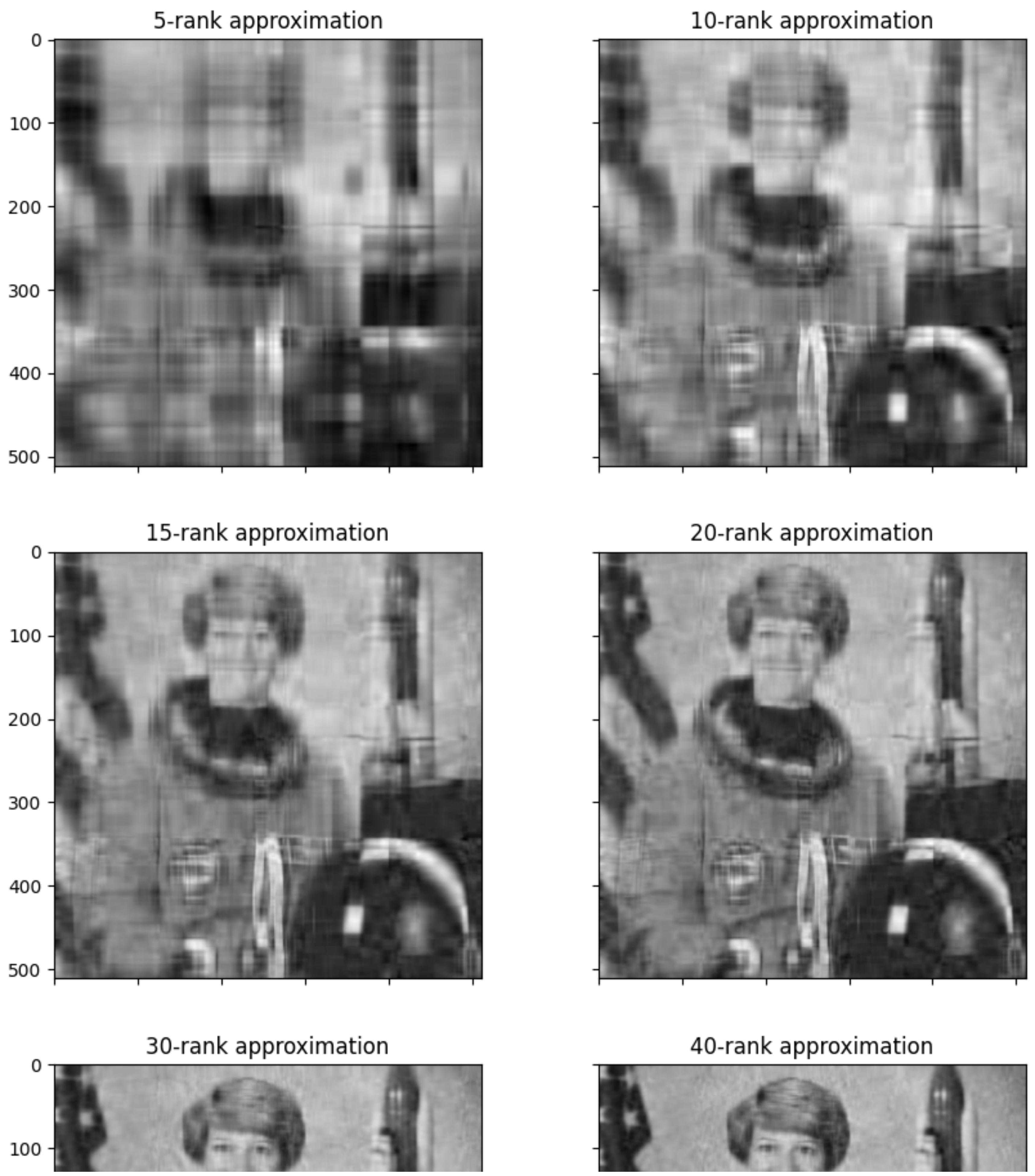
In [64]:
```
plt.plot(s, marker='o', color='lightcoral', ms=4)
plt.grid(linestyle=':')
plt.ylabel('Singular Values')
plt.xlabel('Index')
plt.show()
```

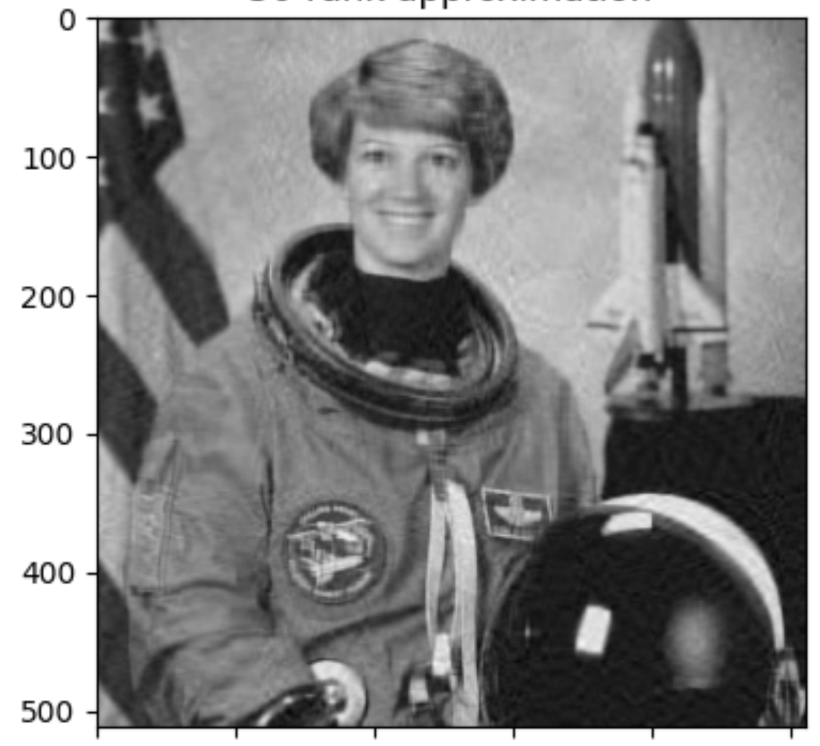


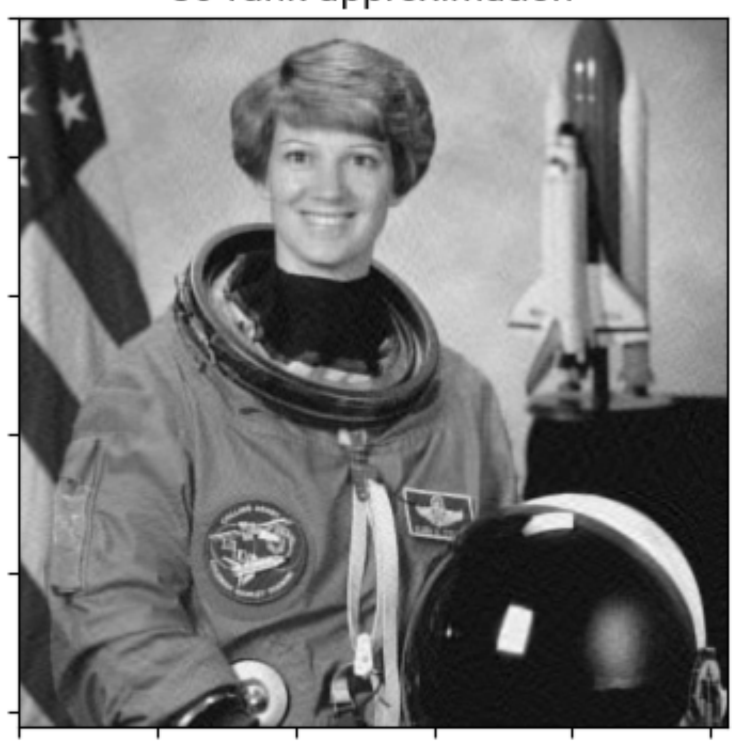
Only few of the singular values are far enough from zero

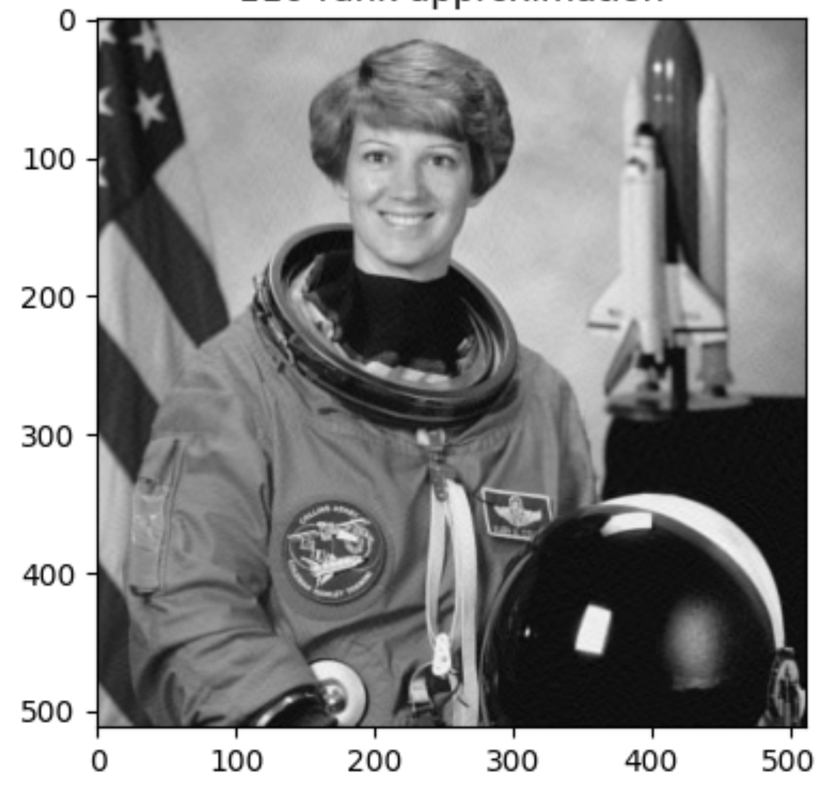
K-Rank approximation

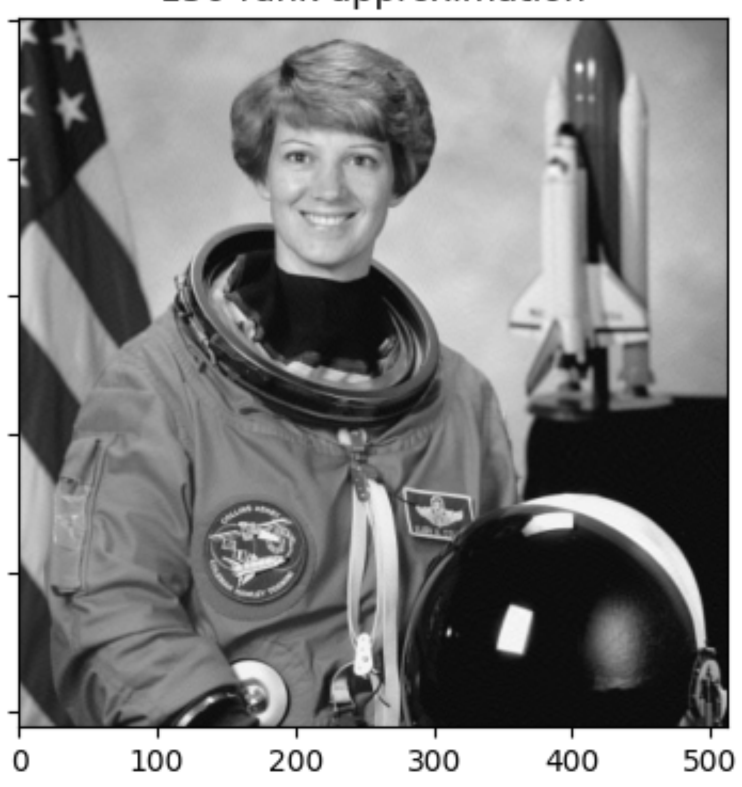
In [68]:
```python
k_vals = [5, 10, 15, 20, 30, 40, 50, 80, 110, 150]

def k_rank_approx(matrix, k):
    approx = np.zeros_like(matrix)
    for i in range(k):
        approx += s[i] * np.outer(U[:, i], VT[i, :])

    return approx


nrows=5
ncols=2
fig, axs = plt.subplots(figsize=(5*ncols, 5*nrows), nrows=nrows, ncols=ncols, sharex=True, sharey=True)

iter = 0
for i in range(nrows):
    for j in range(ncols):
        k = k_vals[iter]
        k_approx = k_rank_approx(img, k)
        ax = axs[i][j]
        ax.imshow(k_approx, cmap='gray')
        ax.set_title(f"{k}-rank approximation")

        iter+=1
```




50-rank approximation


80-rank approximation

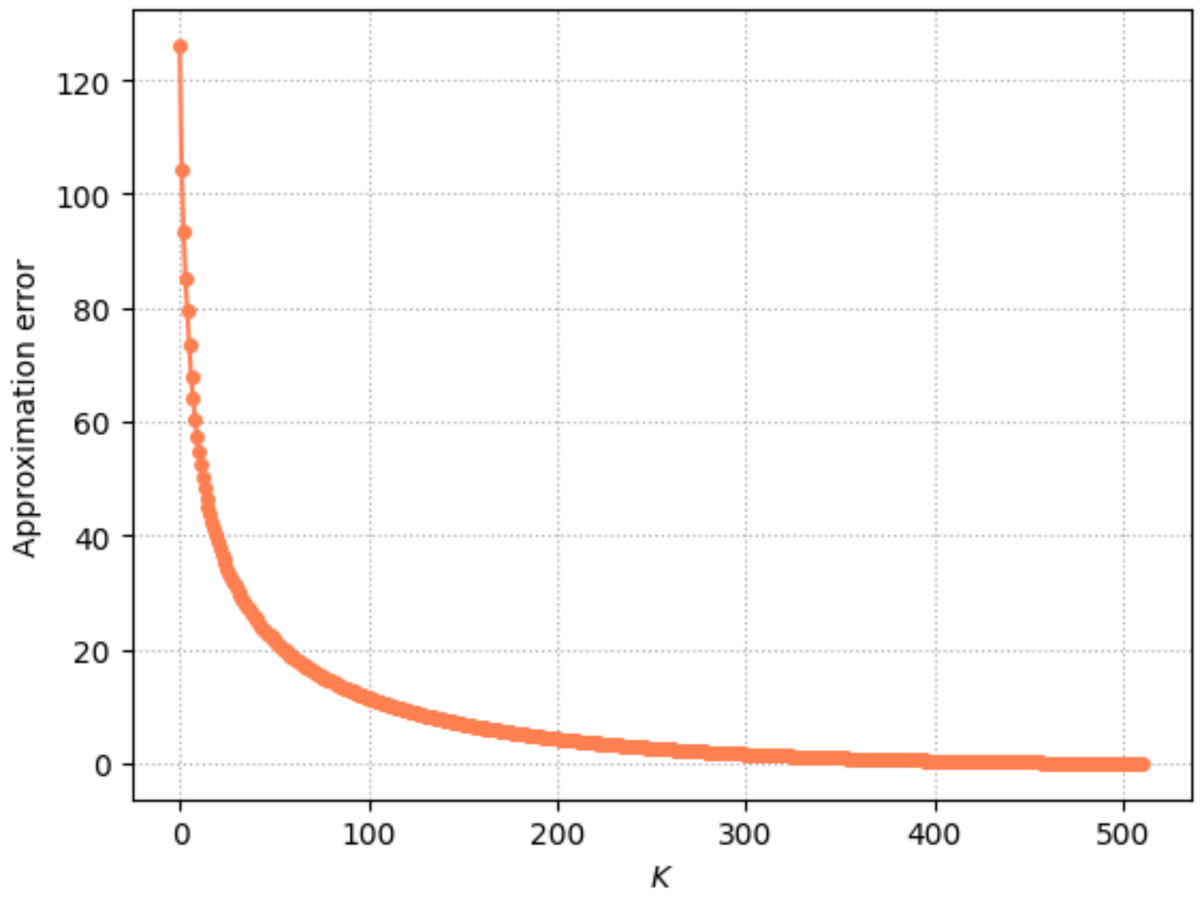

110-rank approximation


150-rank approximation

As we can see from the singular values plot and from the k-rank approximations, less then 100 singular values are sufficient to have a good reconstruction of the image.

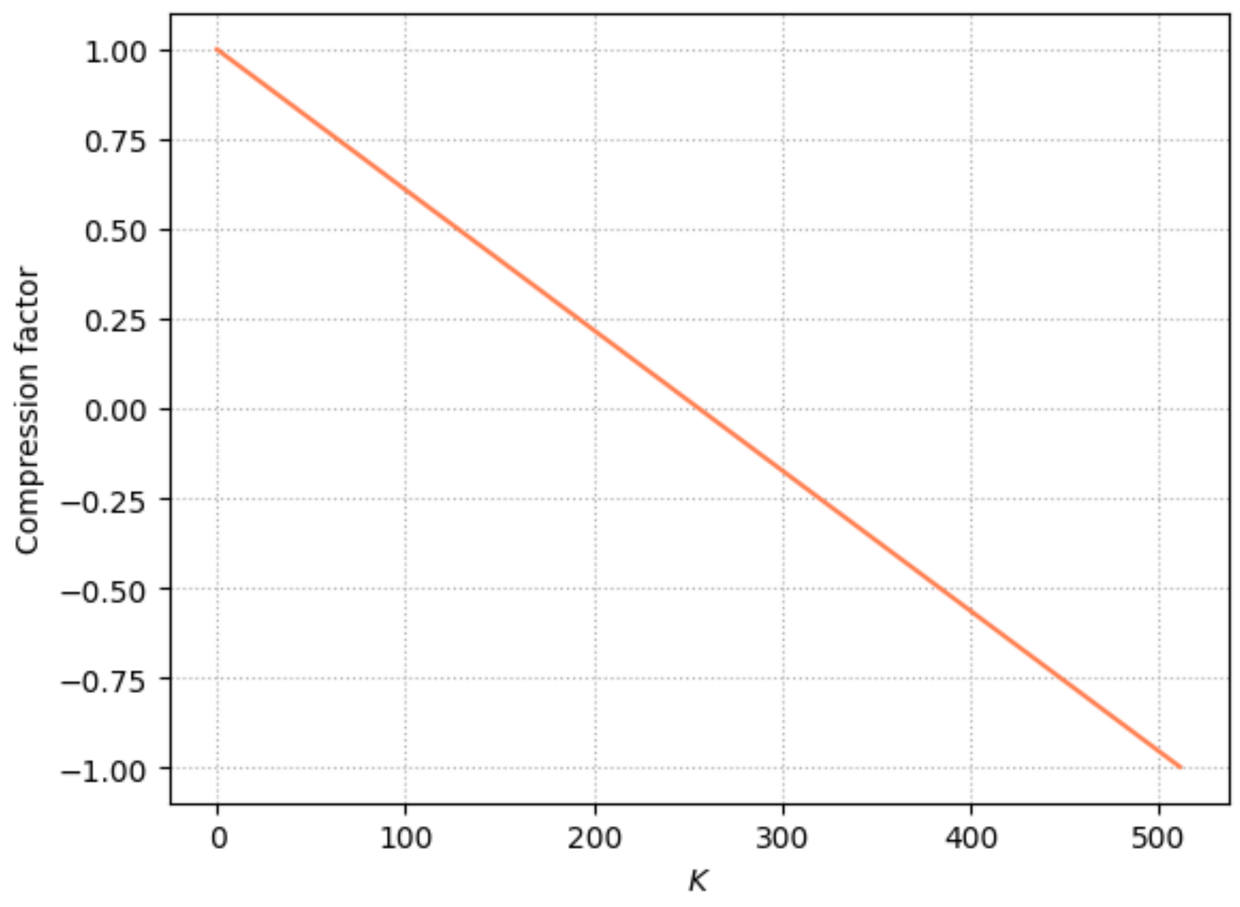
Approximation error

```
In [70]: errs = []

         for k in range(1, s.shape[0]):
             errs.append(np.linalg.norm(img-k_rank_approx(img, k), ord='fro'))

         plt.plot(errs, marker='o', ms=4, color='coral')
         plt.grid(linestyle=':')
         plt.ylabel('Approximation error')
         plt.xlabel('$K$')
         plt.show()
```



Compression factor

```
In [74]: c_factors = []
         m = U.shape[0]
         n = VT.shape[0]
         for k in range(s.shape[0]):
             # k*(m+n+1) is the number of values needed to save the k-rank approximation
             # m*n is the number of values needed to save the entire image
             # the fraction is the percentage of pixel to save in memory
             # 1 - percentage is the percentage of the pixel saved
             c_factors.append(1-(k*(m+n+1) / (m*n)))

         plt.plot(c_factors, color='coral')
         plt.grid(linestyle=':')
         plt.ylabel('Compression factor')
         plt.xlabel('$K$')
         plt.show()
```



We can find the value k for which the the compressed image requires the same amount of informations of those of the uncompressed image by imposing $1 - \frac{k*(m+n+1)}{m*n} = 0$. Making the math, we obtain $k = \frac{m*n}{m+n+1}$. The value for which $c_k \leq 0$ is 256. As we can see from the plot, the compression factor is inverse proportional to k, meaning that higher is k, less is the amount of memory saved. Starting from $k = 256$ it is better to save the original image rather then saving the k-rank approximation.

In [80]:
```
k = (m*n)/(m+n+1)
print(k)

app_err = np.linalg.norm(img-k_rank_approx(img, int(k+1)), ord='fro')
print(app_err)
```

```
255.75024390243902
2.5760597106598815
```