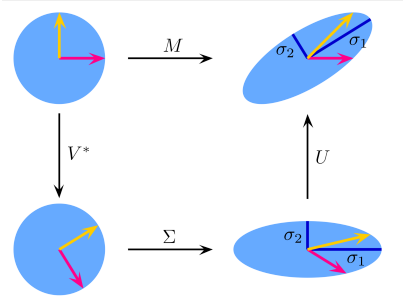
In [1]:
```
import numpy as np
import numpy.linalg
import matplotlib.pyplot as plt
import scipy.integrate
import time
%matplotlib notebook
# For plotting. Don't include this if you submit
                        # a Jupyter Notebook to Gradescope.
```

We will skip the information between these headers (below), you can read it if you'd like.

Okay, so we know that matrices rotate and stretch vectors. To understand this more, we are going to think about "what happens to a sphere (a circle in 2D) when we apply a circle to it. Well, that is the image we had above:



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

We see that a circle becomes an ellipse. This ellipse is defined by its semimajor and semiminor axis. Those are two vectors and form a new *basis* . We see that **the semimajor and semiminor form a new coordinate system (because they are two orthogonal directions) that define the ellipse.** Let's call those two new unit vectors (unit vectors are vectors of length 1) u_1 and u_2. We see that they both don't have the same length, so the semiminor and semimajor axes are actually stretched versions of the unit vectors: $\sigma_1 u_1$ and $\sigma_2 u_2$.

So let's think about what is happening: we started with one set of basis vectors, v_1, v_2. When we apply the matrix to it (matrix multiplication), we arrived at a new set of basis vectors, u_1 and u_2 that were stretched/shrunken by the factors σ_1 and σ_2 respectively.

In other words,

$$Av_1 = \sigma_1 u_1,$$

where v_1 and u_1 both have length 1 ($\|v_1\| = \|u_1\| = 1$). If we had n vectors (n-dimensional space), we would have

$$Av_j = \sigma_j u_j, \qquad j = 1, \ldots, n.$$

In matrix-multiplication form, this looks like

$$A \begin{pmatrix} v_1 & v_2 & \cdots & v_n \end{pmatrix} = \begin{pmatrix} u_1 & u_2 & \cdots & u_n \end{pmatrix} \begin{pmatrix} \sigma_1 & 0 & \cdots & 0 \\ 0 & \sigma_2 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & \cdots & \cdots & \sigma_n \end{pmatrix},$$

where the matrices with v_j and those with u_j are comprised of the columns v_j, u_j side by side. For example, in two dimensions it is

$$\begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix} \begin{pmatrix} v_{11} & v_{12} \\ v_{21} & v_{22} \end{pmatrix} = \begin{pmatrix} u_{11} & u_{12} \\ u_{21} & u_{22} \end{pmatrix} \begin{pmatrix} \sigma_1 & 0 \\ 0 & \sigma_2 \end{pmatrix},$$

assuming that $v_j = \begin{pmatrix} v_{1j} & v_{2j} \end{pmatrix}^T$, and u_j is defined similarly. We write this matrix-matrix product as

$$AV = U\Sigma,$$

which defines the matrices V, U, and Σ. Again what we are doing is going from one *basis* (v_1 and v_2) to another *basis* (u_1 and u_2). The U and V matrices are like rotation matrices and the Σ is the stretching matrix. It happens to be that we can rewrite this as $A = U\Sigma V^T$. Again, what do matrices do? Rotate and stretch. So what does the matrix A do? It first rotates (V^T), then stretches (Σ) and then rotates again (U).

We will skip the information between these headers (above), you can read it if you'd like.

The *Singular-Value Decomposition* is when we *decompose* the matrix A as

$$AV = U\Sigma,$$

which defines the matrices V, U, and Σ. We are going from one *basis* (v_1 and v_2, or v_i) to another *basis* (u_1 and u_2, or u_i). The U and V matrices are like rotation matrices and the Σ is the stretching matrix. It happens to be that we can rewrite this as $A = U\Sigma V^T$. Again, what do matrices do? Rotate and stretch. So what does the matrix A do? It first rotates (V^T), then stretches (Σ) and then rotates again (U).

The main reason why we are interested in doing this is because calculating *the SVD of a matrix* (meaning the three matrices, U, Σ and V), *allows us to find the natural coordinate system for the system* through the basis vectors in U and V.

Let's see an example of how we do this on the computer. First define

$$A = \begin{pmatrix} 1 & 2 \\ 0.3 & 0.4 \end{pmatrix}.$$

In [2]:
```
# Define A
A = np.array([[1, 2], [0.3, 0.4]])

# Calculate SVD with np.linalg.svd(A, full_matrices=False). Print the resul
print(np.linalg.svd(A, full_matrices=False))
```

```
(array([[-0.97657551, -0.21517498],
       [-0.21517498,  0.97657551]]), array([2.28962221, 0.08735066]), arr
ay([[-0.45471607, -0.89063646],
       [ 0.89063646, -0.45471607]]))
```

We can see that it is outputting 3 things. Those are U, Σ and V^T (note that it is V^T not V!). Let's define those.

In [3]:
```
# Save the three outputs: U, S, and Vt from np.linalg.svd(A)
U, S, Vt = np.linalg.svd(A, full_matrices = False)

#Print each of them
print(U)
print(S)
print(Vt)
```

```
[[-0.97657551 -0.21517498]
 [-0.21517498  0.97657551]]
[2.28962221 0.08735066]
[[-0.45471607 -0.89063646]
 [ 0.89063646 -0.45471607]]
```

We see that U and Vt are matrices as expected, but not S! That's because it only gives us the diagonal. To turn it into a matrix we can use `np.diag` .

In [4]:
```
# Set S = np.diag(S)
S = np.diag(S)
print(S) # now it is a matrix
```

```
[[2.28962221 0.        ]
 [0.         0.08735066]]
```

By the way let's check: Is $A = U\Sigma V^T$?

In [5]:
```
# Print A
print(A)
# Print the product of the three matrices.
print(U@S@Vt)
```

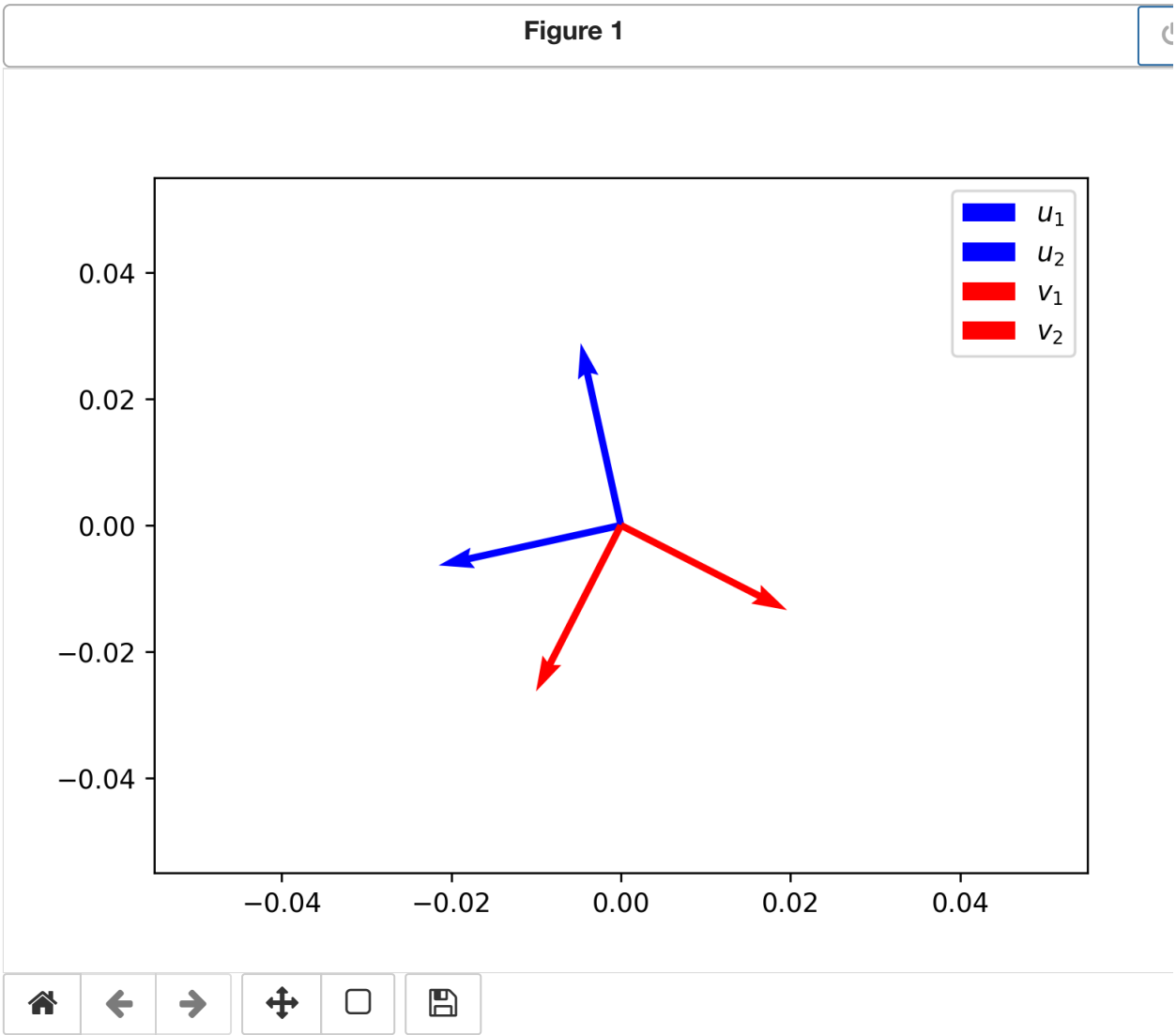
```
[[1.  2. ]
 [0.3 0.4]]
[[1.  2. ]
 [0.3 0.4]]
```

Good, we see they are the same. We were saying that the vectors in U and V^T were *basis* vectors. In order for that to be true, they need to have length 1 and be orthogonal. We should check the *columns* of U and the *rows* of V^T (or the columns of V). Let's plot them.

In [6]:
```
# Create a new figure
fig, ax = plt.subplots()
# Plot the two columns of U in blue, using quiver
# We are plotting the 0th column ([:, 0]),
# but the x component is the 0th entry in the 0th column
# and the y component is the 1st entry in the 0th column
ax.quiver(0, 0, U[0, 0], U[1, 0], color='blue', scale=5,
                  label = '$u_1$')

# Same now for the 1st column.
ax.quiver(0, 0, U[0, 1], U[1, 1], color='blue', scale=5,
          label = '$u_2$')

# Show legends
ax.legend()
```



Out[6]:
```
<matplotlib.legend.Legend at 0x7fe64dc31490>
```

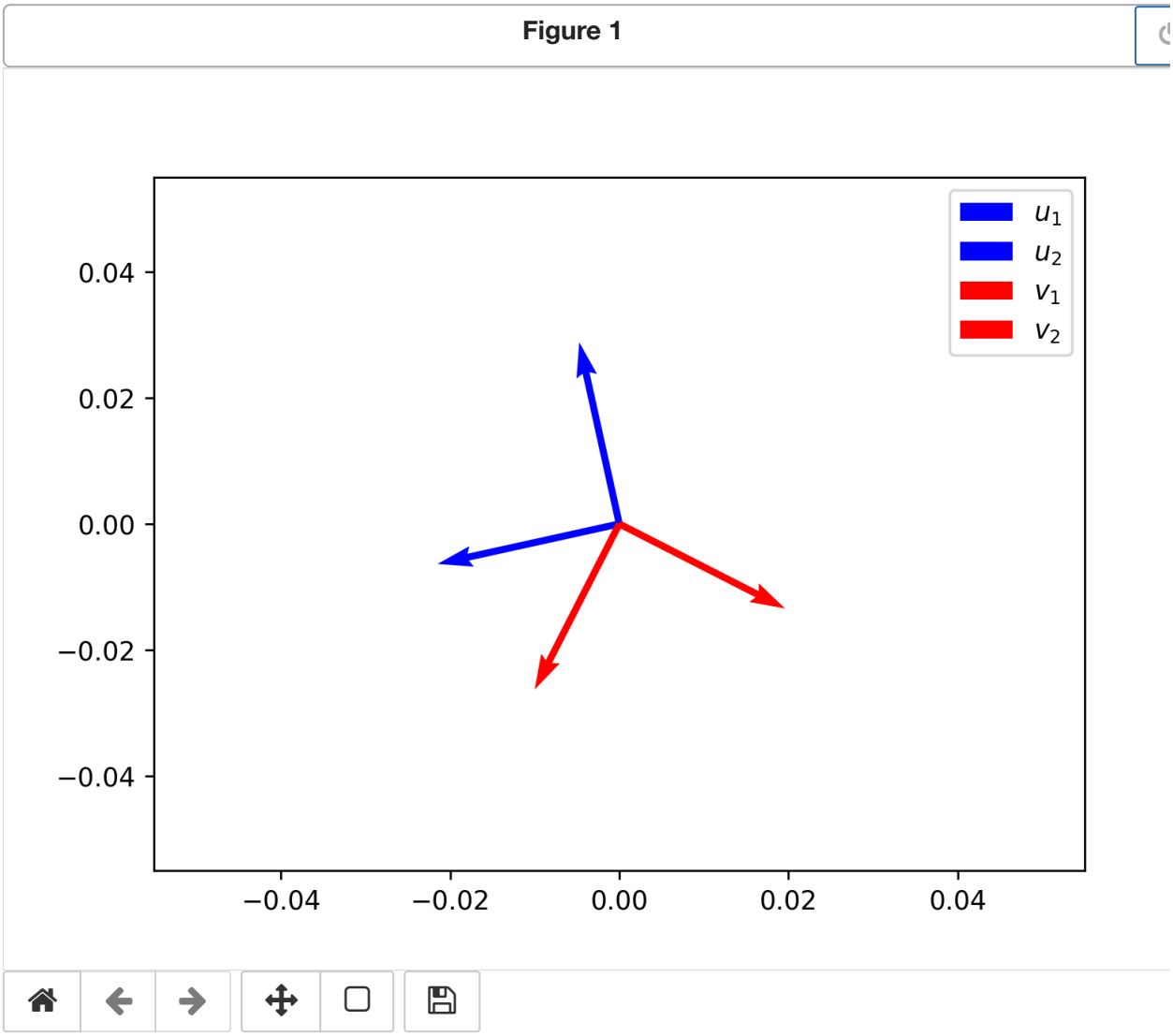
We can see that those are orthogonal and have the same length. Now let's do the same for the *rows* of V^T. Plot in red.

In [7]:
```
# Plot the two rows of V^T in red, using quiver
# We are plotting the 0th row ([0, :]),
# but the x component is the 0th entry in the 0th row
# and the y component is the 1st entry in the 0th row
ax.quiver(0, 0, Vt[0, 0], Vt[0, 1], color='red', scale=5,
          label = '$v_1$')

# Same now for the 1st row.
ax.quiver(0, 0, Vt[1, 0], Vt[1, 1], color='red', scale=5,
          label = '$v_2$')

# Show the legend
ax.legend()

# Show the figure
fig
```



We have verified that there are two different bases, in U and V. The matrix A rotates between the two bases.

The columns of U are called the "*left-singular vectors*" and the columns of V (or rows of V^T) are called the "*right-singular vectors.*" Σ contains the "*singular values.*"

So how does this all relate to the movie recommendation algorithm?

When we can draw arrows in 2D, as we have done above, it's easy to say what the directions are because we can visualize them. When we have other data sets, like the scores for movies for a bunch of students in class, we can't do that so easily. First off, our data set is going to be huge: hundreds of dimensions (hundreds of students) or tens of dimensions (tens of movies). We can't visualize that. Luckily, we can still interpret what the result is.

For now we are going to not worry about the Σ matrix. So we are going to work with

$$A = U\Sigma V^T = \tilde{U}V^T,$$

where A is our data matrix, and $\tilde{U} = U\Sigma$ is a scaled version (scaled by the singular values) of our left-singular vectors. Remember this doesn't change the coordinate system, it just stretches the vectors.

So we have

$$A = \begin{bmatrix} | & | & \cdots & | \\ \tilde{u}_1 & \tilde{u}_2 & \cdots & \tilde{u}_n \\ | & | & \cdots & | \end{bmatrix} \begin{bmatrix} & v_1^T & \\ & v_2^T & \\ & \vdots & \\ & v_n^T & \end{bmatrix}.$$

Notice that we have $\tilde{U}$ having the columns and V^T having the rows. Compare this to our data matrix, A,

$$A = \begin{pmatrix} 3 & 4 & 2 & 1 \\ 5 & 1 & 3 & 1 \\ 1 & 1 & 2 & 4 \\ 3 & 3 & 3 & 3 \\ 2 & 1 & 4 & 4 \end{pmatrix},$$

where the rows are people's scores and the columns are movie ratings. So **the left-singular vectors**, which are columns, **form a basis for the movies** and **the right-singular vectors**, which are rows, **form a basis for the people.**

So for example, we might find that

$$\begin{aligned} \tilde{u}_1 &= \text{a typical action movie} \\ \tilde{u}_2 &= \text{a typical comedy movie} \\ \tilde{u}_3 &= \text{a typical horror movie} \\ \tilde{u}_4 &= \text{a typical anime movie} \\ &\vdots \end{aligned},$$

etc. So just like before when I wrote,

$$\begin{pmatrix} 6 \\ 4 \end{pmatrix} = 6 \begin{pmatrix} 1 \\ 0 \end{pmatrix} + 4 \begin{pmatrix} 0 \\ 1 \end{pmatrix},$$

writing the vector as a sum of its two basis elements, now we can have something like

$$\text{The Batman} = \alpha_1 \tilde{u}_1 + \alpha_2 \tilde{u}_2 + \dots,$$

where α_1 tells me "*how much of an action movie The Batman is*, α_2 tells me "*how much of a comedy movie The Batman is*", etc. Then *every single movie will be some combination of those different types.* Now we don't actually get to choose those genres/basis. Instead, if we plug in 4 movies, the SVD will find a 4-dimensional basis for us: it will find 4 characteristics. If we plug in 1000 movies it will find a 1000-dimensional basis: 1000 characteristics. We won't be able to

actually say in words what those characteristics are, that's what SVD does on its own and it's hard to interpret what the basis elements are, but what it's doing is *finding some interrelatedness between movies*, based on how they were rated by people. For example, it could actually be that one of the basis elements is *year the movie came out* because certain groups of people may rank movies similarly based on the year, or *whether or not the movie had Leonardo Dicaprio in it*, etc. **We don't know what that basis is.**

What about the right-singular vectors? Well, they provide a basis for our other space: people. For example, we may have something like

$$\begin{aligned} v_1 &= \text{typical action fan} \\ v_2 &= \text{typical comedy fan} \\ v_3 &= \text{typical horror fan} \\ v_4 &= \text{typical anime fan} \\ &\vdots \end{aligned}$$

etc. So now we have something like

$$A = \tilde{U}V^T$$

$$= \begin{bmatrix} | & | & \cdots \\ \text{action movie} & \text{comedy movie} & \cdots \\ | & | & \cdots \end{bmatrix} \begin{bmatrix} & \text{action fan} & \\ & \text{comedy fan} & \\ \vdots & \vdots & \vdots \end{bmatrix}$$

So if we want to know Adnan's score for "The Batman", which is $A(1,1)$ we do

$$\begin{aligned} A(1,1) &= \text{Adnan's score for "The Batman"} \\ &= (\text{How much of an action movie is The Batman}) \\ &\quad \cdot (\text{How much does Adnan like action movies}) \\ &\quad + (\text{How much of a comedy movie is The Batman}) \\ &\quad \cdot (\text{How much does Adnan like comedy movies}) + \\ &\quad \cdots \end{aligned}$$

etc.

In other words, to figure out someone's score in the movie, we just have to know how much of each component in the basis a particular movie is and then weight it by how much that person likes movies in that basis.

The SVD allows us to do this **when the coordinate basis is not obvious** and in fact finds **ideal coordinate bases for us.**

This allows us to now think about "what are the most important characteristics of a movie, in order for it to be ranked a particular way?" That's where the singular values come in. **The singular-values are ranked in terms of largest to smallest, which corresponds to their importance in characterizing the data.** For instance, we may find that a basis vector for the movies is "movie runtime", but we may find that that doesn't have much impact on the movie ratings. In that case, **the singular value for that basis vector would be small.**

This leads us to the concept of

Low rank approximation using SVD

Using the SVD for low-rank approximations of our data is the key to what makes SVD so powerful. When we use all of our data, we have a *full-rank* system. When we use *only the one largest singular-value*, we have a rank-1 approximation. The *rank* corresponds to how many of the singular values we use to describe/approximate the system. To demonstrate this, we are going to discuss image compression using low-rank approximations.

Image compression using SVD

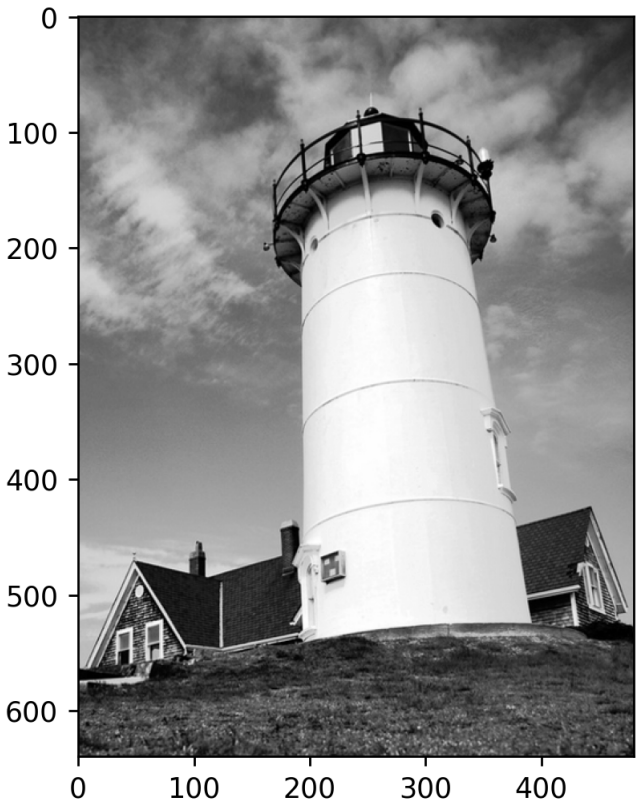
We first need to load in our image. We will do this using cv2.imread().

In [8]:
```
# Import cv2
import cv2

A = cv2.imread('lighthouse.png')

# View the image
fig, ax = plt.subplots()
ax.imshow(A)
```

Figure 2



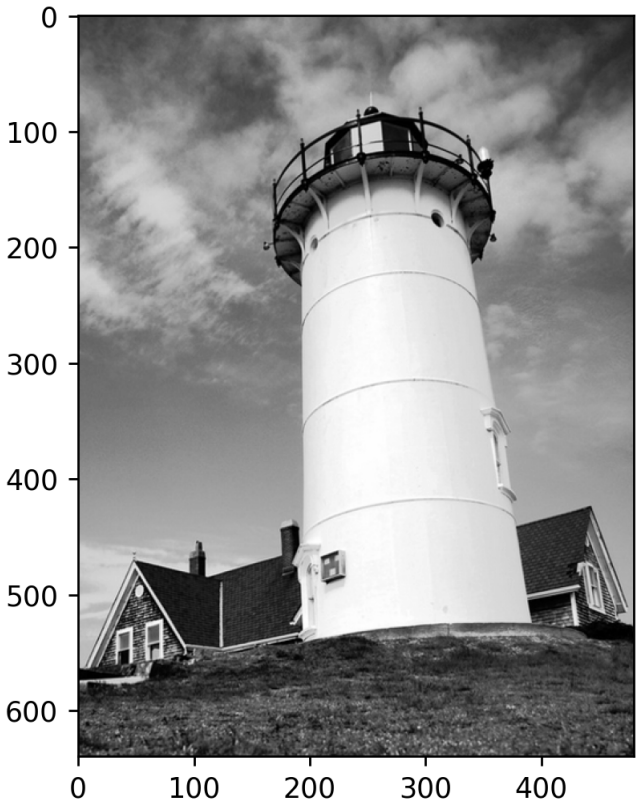
Out[8]: <matplotlib.image.AxesImage at 0x7fe64dd66580>

The image compression we will describe only works on grayscale images, so we will load and plot our images using grayscale.

In [9]:
```
# Reload the data, using grayscale
A = cv2.imread('lighthouse.png', 0) # The 0 means grayscale

# View the image
ax.imshow(A, cmap='gray')
fig
```

Figure 2



We can see how the image, `A`, is stored.

In [10]:
```
print(A)
```

```
[[ 63  64  66 ...  38  39  38]
 [ 65  65  67 ...  39  40  38]
 [ 69  69  69 ...  40  40  38]
 ...
 [ 72  89  82 ...  16   9  18]
 [ 74 118 132 ...  23  28  41]
 [ 68  93  98 ...  47  46  44]]
```

We see it's just a matrix! We can check its shape.

In [11]:
```
print(A.shape)
```

```
(640, 480)
```

This means there are 640x480 pixels.

We know how to take the SVD of matrices, so let's do that.

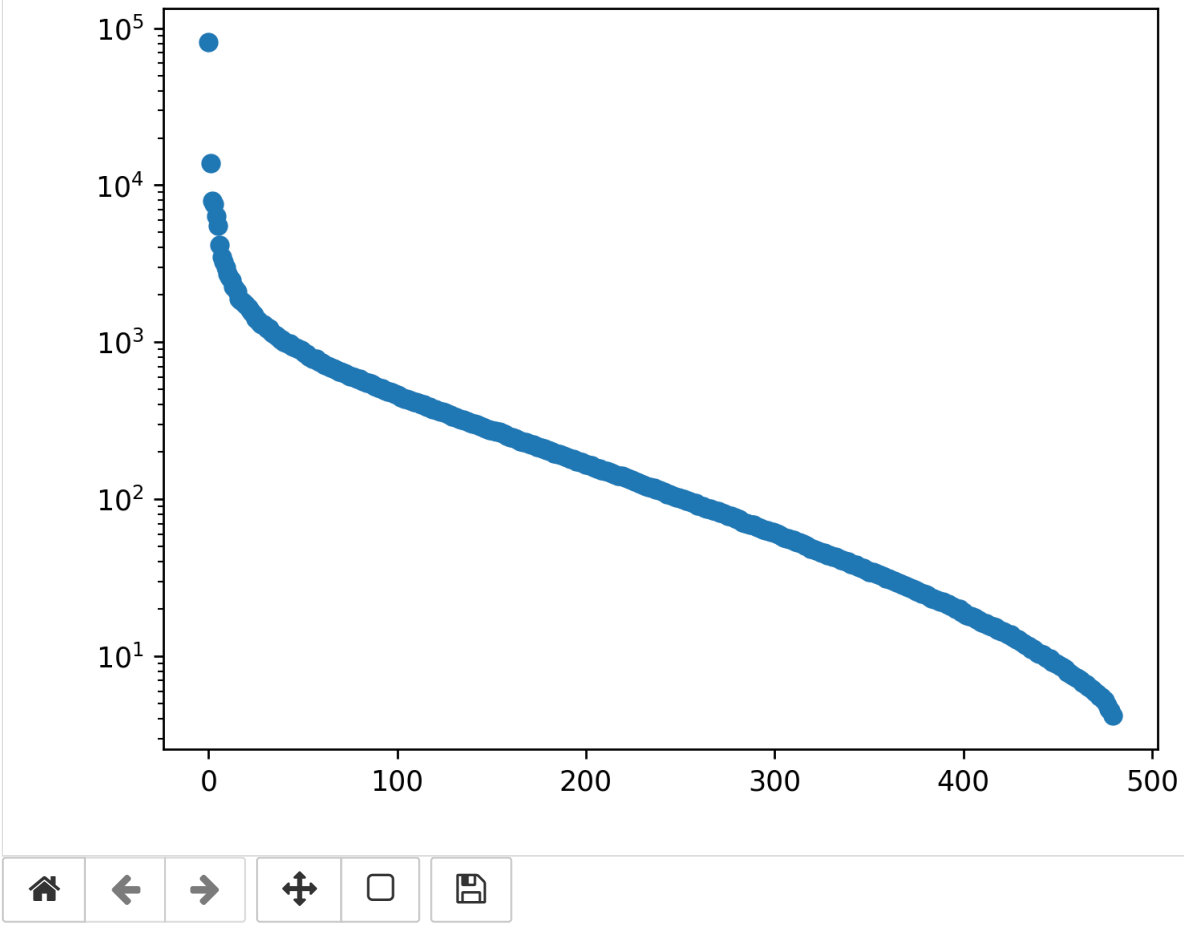
In [12]:
```
# Compute the SVD
U, S, Vt = np.linalg.svd(A, full_matrices=False)
# Print the shape of each part
print(U.shape)
print(S.shape)
print(Vt.shape)
```

```
(640, 480)
(480,)
(480, 480)
```

Let's plot the singular values in order of how they are returned. We will use a "semilogy" plot, which means logarithmic in the "y" variables but not in the "x" variables.

In [13]:
```
fig2, ax2 = plt.subplots()
ax2.semilogy(S, 'o')
```

Figure 3



Out[13]:
```
[<matplotlib.lines.Line2D at 0x7fe64e044f40>]
```

We note that:

- the singular values are ordered from largest to smallest; and
- some of them are much larger than others.

The idea behind a *low-rank approximation* is that **we will only use the basis vectors corresponding to the largest singular values to *approximate* the data.**

To form the low-rank approximation, we need to multiply $U\Sigma V^T$, but only some of it. For instance, if we want a rank-1 approximation, it is

In [14]:
```
# (We first need to change S to a matrix)
S_mat = np.diag(S) # Converts an array to a diagonal matrix.
rank_1 = (U[:, 0:1]@S_mat[0:1, 0:1])@Vt[0:1, :]
print(rank_1.shape)
```

```
(640, 480)
```

We can see that `rank_1` has the same dimensions as `A`, our original data. But, **we are actually using a lot less information.** Instead of storing all 640×480 pixels, we need only the first column of U, the first singular value of Σ, and the first row of V^T. That means we have $640 + 1 + 480 = 1121$ numbers stored, instead of $640 \times 480 = 307,200$ numbers stored. **This is a major savings!** But that savings doesn't mean much if the rank-1 approximation doesn't look like our image. Before showing a plot of it, we will define *the energy* of a rank$-r$ approximation. The *total energy* of a set of data is *the sum of all of the eigenvalues*. So we can define `total_energy`

In [15]:
```
# First turn the matrix S back
total_energy = sum(S)
print(total_energy)
```

```
269894.9223718478
```

Meanwhile, the energy for the rank-1 approximation is the energy when we use only one singular value. That is:

In [16]:
```
rank_1_energy = sum(S[0:1])
print(rank_1_energy)
```

```
81516.45754347282
```

So the rank-1 approximation contains $81516/269894$ of the "energy" or information of the whole image.

In [17]:
```
# Calculate percentage
# of energy in the rank-1 approximation
rank_1_percentage = rank_1_energy / total_energy
print(rank_1_percentage)
```

```
0.3020303488005732
```

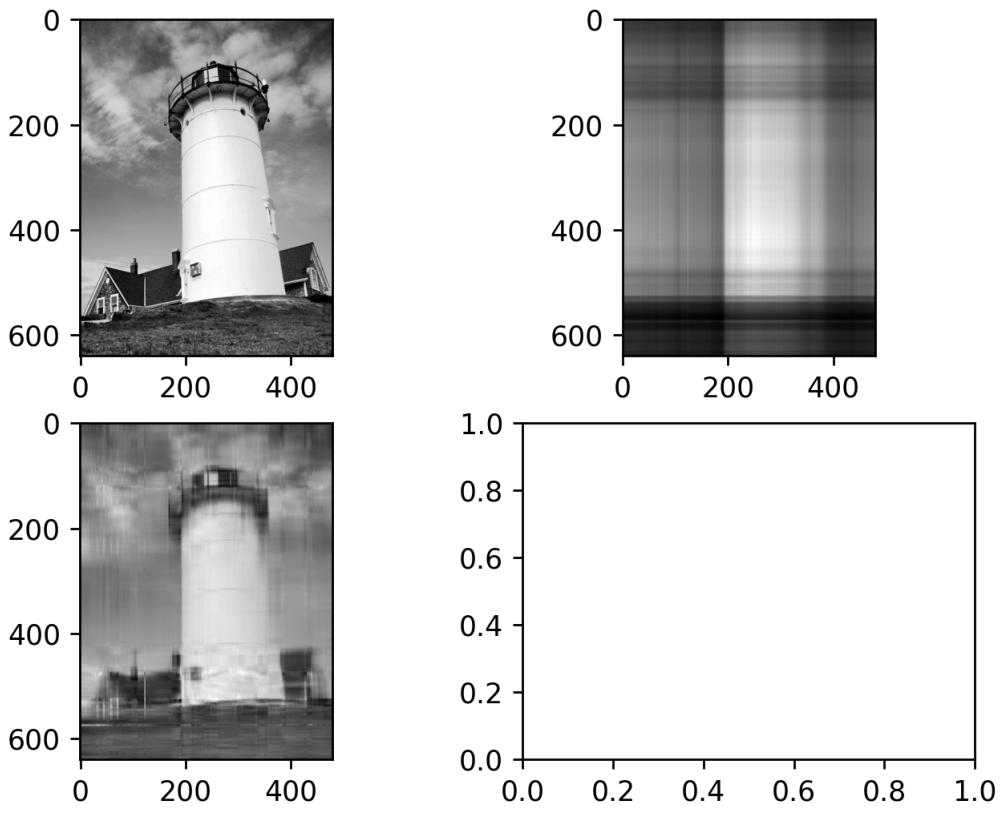
We see that the rank-1 approximation contains 30% of the total energy!

Let's see what the image looks like.

In [18]:
```
# Setup a 2x2 grid of images
fig, ax = plt.subplots(2,2)
# Plot the full image. We need to index "ax"
# like it's a matrix.
ax[0,0].imshow(A, cmap='gray')

# Then plot the rank-1 approximation in the
#  0th row, 1st column
ax[0, 1].imshow(rank_1, cmap='gray')
```

Figure 4



Out[18]: <matplotlib.image.AxesImage at 0x7fe64ddee550>

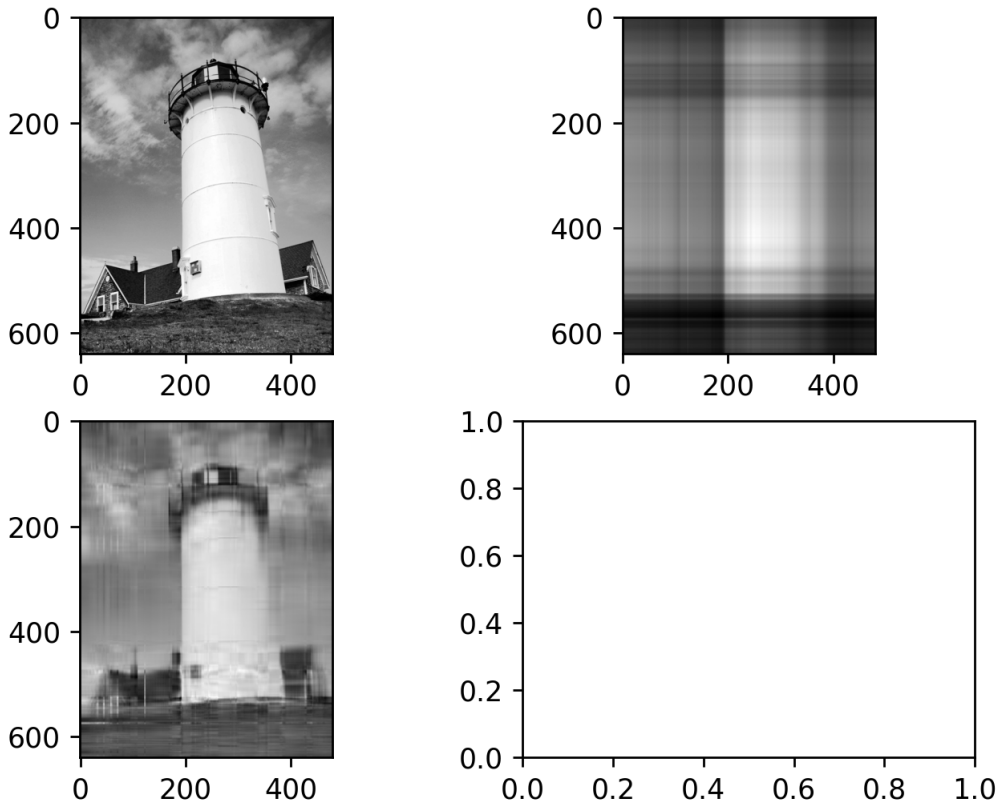
Obviously this image does not look very great, but it gives the general features of the image: it is dark on the bottom and there's a pillar in the middle. Let's see what the rank-10 approximation looks like.

In [19]:
```
# Calculate rank-10 approximation
# This looks just like before, except with 1 replaced by 10
rank_10 = (U[:, 0:10]@S_mat[0:10, 0:10])@Vt[0:10, :]

# Then plot it
#  1st row, 0th column
ax[1,0].imshow(rank_10, cmap='gray')

# Show the figure
fig
```

Figure 4



We can see that we have a more clear version of the picture, but still not perfect. We can continue on this route, and I'll leave you to play around with that on your own. I'll also leave you to calculate the percentage energy on your own.

Note that before we were talking about basis vectors. It's hard to interpret what the basis vectors are in this case, but rest assured they are there!

We are going to use the same technique for the movie recommendation software, more on that in the next lecture!