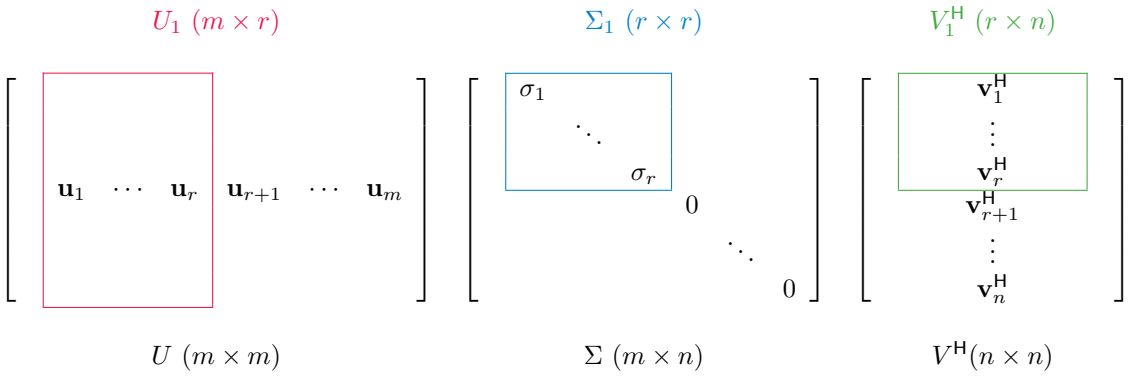
6 The SVD and Image Compression

Lab Objective: *The Singular Value Decomposition (SVD) is an incredibly useful matrix factorization that is widely used in both theoretical and applied mathematics. The SVD is structured in a way that makes it easy to construct low-rank approximations of matrices, and it is therefore the basis of several data compression algorithms. In this lab we learn to compute the SVD and use it to implement a simple image compression routine.*

The SVD of a matrix A is a factorization $A = U\Sigma V^\mathsf{H}$ where U and V have orthonormal columns and Σ is diagonal. The diagonal entries of Σ are called the *singular values* of A and are the square roots of the eigenvalues of $A^\mathsf{H}A$. Since $A^\mathsf{H}A$ is always positive semidefinite, its eigenvalues are all real and nonnegative, so the singular values are also real and nonnegative. The singular values σ_i are usually sorted in decreasing order so that $\Sigma = \text{diag}(\sigma_1, \sigma_2, \dots, \sigma_n)$ with $\sigma_1 \geq \sigma_2 \geq \dots \geq \sigma_n \geq 0$. The columns $\mathbf{u}_i$ of U, the columns $\mathbf{v}_i$ of V, and the singular values of A satisfy $A\mathbf{v}_i = \sigma_i \mathbf{u}_i$.

Every $m \times n$ matrix A of rank r has an SVD with exactly r nonzero singular values. Like the QR decomposition, the SVD has two main forms.

- **Full SVD**: Denoted $A = U\Sigma V^\mathsf{H}$. U is $m \times m$, V is $n \times n$, and Σ is $m \times n$. The first r columns of U span $\mathscr{R}(A)$, and the remaining $n - r$ columns span $\mathscr{N}(A^\mathsf{H})$. Likewise, the first r columns of V span $\mathscr{R}(A^\mathsf{H})$, and the last $m - r$ columns span $\mathscr{N}(A)$.
- **Compact (Reduced) SVD**: Denoted $A = U_1\Sigma_1 V_1^\mathsf{H}$. U_1 is $m \times r$ (the first r columns of U), V_1 is $n \times r$ (the first r columns of V), and Σ_1 is $r \times r$ (the first $r \times r$ block of Σ). This smaller version of the SVD has all of the information needed to construct A and nothing more. The zero singular values and the correpsonding columns of U and V are neglected.



Finally, the SVD yields an *outer product expansion* of A in terms of the singular values and the columns of U and V,

$$A = \sum_{i=1}^{r} \sigma_i \mathbf{u}_i \mathbf{v}_i^H. \tag{6.1}$$

Note that only terms from the compact SVD are needed for this expansion.

Computing the Compact SVD

It is difficult to compute the SVD from scratch because it is an eigenvalue-based decomposition. However, given an eigenvalue solver such as `scipy.linalg.eig()`, the algorithm becomes much simpler. First, obtain the eigenvalues and eigenvectors of A^HA, and use these to compute Σ. Since A^HA is normal, it has an orthonormal eigenbasis, so set the columns of V to be the eigenvectors of A^HA. Then, since $A\mathbf{v}_i = \sigma_i \mathbf{u}_i$, construct U by setting its columns to be $\mathbf{u}_i = \frac{1}{\sigma_i} A\mathbf{v}_i$.

The key is to sort the singular values and the corresponding eigenvectors in the same manner. In addition, it is computationally inefficient to keep track of the entire matrix Σ since it is a matrix of mostly zeros, so we need only store the singular values as a vector $\boldsymbol{\sigma}$. The entire procedure for computing the compact SVD is given below.

Algorithm 6.1

1: **procedure** COMPACT_SVD(A)
2: $\boldsymbol{\lambda}, V \leftarrow \text{eig}(A^HA)$ ▷ Calculate the eigenvalues and eigenvectors of A^HA.
3: $\boldsymbol{\sigma} \leftarrow \sqrt{\boldsymbol{\lambda}}$ ▷ Calculate the singular values of A.
4: $\boldsymbol{\sigma} \leftarrow \text{sort}(\boldsymbol{\sigma})$ ▷ Sort the singular values **from greatest to least**.
5: $V \leftarrow \text{sort}(V)$ ▷ Sort the eigenvectors **the same way** as in the previous step.
6: $r \leftarrow \text{count}(\boldsymbol{\sigma} \neq 0)$ ▷ Count the number of nonzero singular values (the rank of A).
7: $\boldsymbol{\sigma}_1 \leftarrow \boldsymbol{\sigma}_{:r}$ ▷ Keep only the positive singular values.
8: $V_1 \leftarrow V_{:,:r}$ ▷ Keep only the corresponding eigenvectors.
9: $U_1 \leftarrow AV_1/\boldsymbol{\sigma}_1$ ▷ Construct U with array broadcasting.
10: **return** $U_1, \boldsymbol{\sigma}_1, V_1^H$

Problem 1. Write a function that accepts a matrix A and a small error tolerance `tol`. Use Algorithm 6.1 to compute the compact SVD of A. In step 6, compute r by counting the number of singular values that are greater than `tol`.

Consider the following tips for implementing the algorithm.

- The Hermitian A^H can be computed with `A.conj().T`.
- In step 4, the way that $\boldsymbol{\sigma}$ is sorted needs to be stored so that the columns of V can be sorted the same way. Consider using `np.argsort()` and fancy indexing to do this, but remember that by default it sorts from least to greatest (not greatest to least).
- Step 9 can be done by looping over the columns of V, but it can be done more easily and efficiently with array broadcasting.

Test your function by calculating the compact SVD for random matrices. Verify that U and V are orthonormal, that $U\Sigma V^H = A$, and that the number of nonzero singular values is the rank of A. You may also want to compre your results to SciPy's SVD algorithm.

```
>>> import numpy as np
>>> from scipy import linalg as la

# Generate a random matrix and get its compact SVD via SciPy.
>>> A = np.random.random((10,5))
>>> U,s,Vh = la.svd(A, full_matrices=False)
>>> print(U.shape, s.shape, Vh.shape)
(10, 5) (5,) (5, 5)

# Verify that U is orthonormal, U Sigma Vh = A, and the rank is correct.
>>> np.allclose(U.T @ U, np.identity(5))
True
>>> np.allclose(U @ np.diag(s) @ Vh, A)
True
>>> np.linalg.matrix_rank(A) == len(s)
True
```

Visualizing the SVD

An $m \times n$ matrix A defines a linear transformation that sends points from $\mathbb{R}^n$ to $\mathbb{R}^m$. The SVD decomposes a matrix into two rotations and a scaling, so that any linear transformation can be easily described geometrically. Specifically, V^H represents a rotation, Σ a rescaling along the principal axes, and U another rotation.

Problem 2. Write a function that accepts a 2×2 matrix A. Generate a 2×200 matrix S representing a set of 200 points on the unit circle, with x-coordinates on the top row and y-coordinates on the bottom row (recall the equation for the unit circle in polar coordinates: $x = \cos(\theta)$, $y = \sin(\theta)$, $\theta \in [0, 2\pi]$). Also define the matrix

$$E = \left[\begin{array}{c|c|c} \mathbf{e}_1 & \mathbf{0} & \mathbf{e}_2 \end{array}\right] = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix},$$

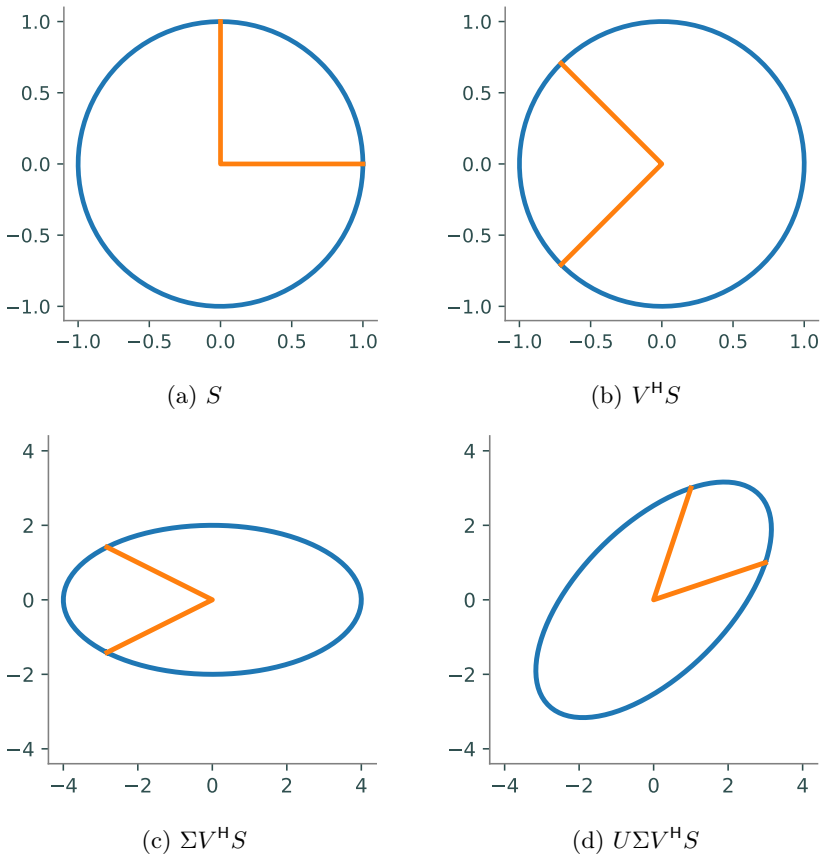
so that plotting the first row of S against the second row of S displays the unit circle, and plotting the first row of E against its second row displays the standard basis vectors in $\mathbb{R}^2$.

Compute the full SVD $A = U\Sigma V^\mathsf{H}$ using `scipy.linalg.svd()`. Plot four subplots to demonstrate each step of the transformation, plotting S and E, $V^\mathsf{H}S$ and $V^\mathsf{H}E$, $\Sigma V^\mathsf{H}S$ and $\Sigma V^\mathsf{H}E$, then $U\Sigma V^\mathsf{H}S$ and $U\Sigma V^\mathsf{H}E$.

For the matrix

$$A = \begin{bmatrix} 3 & 1 \\ 1 & 3 \end{bmatrix},$$

your function should produce Figure 6.1.
(Hint: Use `plt.axis("equal")` to fix the aspect ratio so that the circles don't appear elliptical.)



(a) S (b) $V^{\mathsf{H}}S$

(c) $\Sigma V^{\mathsf{H}}S$ (d) $U\Sigma V^{\mathsf{H}}S$

Figure 6.1: Each step in transforming the unit circle and two unit vectors using the matrix A.

Using the SVD for Data Compression

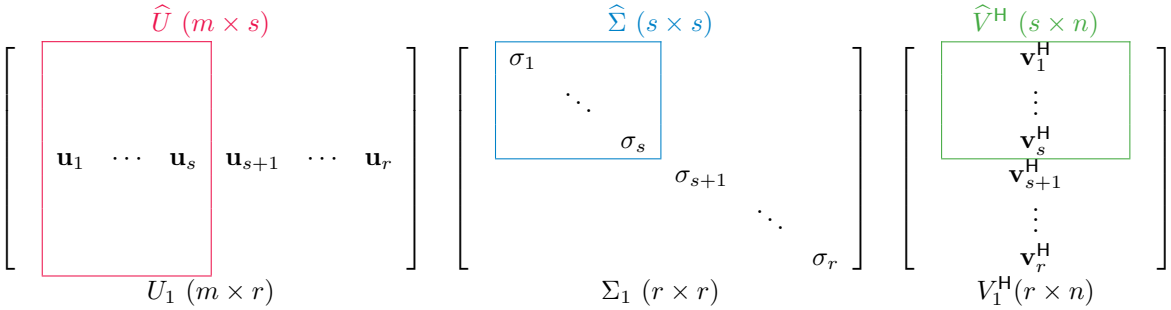
Low-Rank Matrix Approximations

If A is a $m \times n$ matrix of rank $r < \min\{m, n\}$, then the compact SVD offers a way to store A with less memory. Instead of storing all mn values of A, storing the matrices U_1, Σ_1 and V_1 only requires saving a total of $mr + r + nr$ values. For example, if A is 100×200 and has rank 20, then A has $20,000$ values, but its compact SVD only has total $6,020$ entries, a significant decrease.

The *truncated SVD* is an approximation to the compact SVD that allows even greater efficiency at the cost of a little accuracy. Instead of keeping all of the nonzero singular values, the truncated SVD only keeps the first $s < r$ singular values, plus the corresponding columns of U and V. In this case, (6.1) becomes

$$A_s = \sum_{i=1}^{s} \sigma_i \mathbf{u}_i \mathbf{v}_i^{\mathsf{H}}.$$

More precisely, the truncated SVD of A is $A_s = \widehat{U}\widehat{\Sigma}\widehat{V}^{\mathsf{H}}$, where $\widehat{U}$ is $m \times s$, $\widehat{V}$ is $n \times s$, and $\widehat{\Sigma}$ is $s \times s$. The resulting matrix A_s has rank s and is only an approximation to A, since $r - s$ nonzero singular values are neglected.



The beauty of the SVD is that it makes it easy to select the information that is most important. Larger singular values correspond to columns of U and V that contain more information, so dropping the smallest singular values retains as much information as possible. In fact, given a matrix A, its rank-s truncated SVD approximation A_s is the *best rank s approximation* of A with respect to both the induced 2-norm and the Frobenius norm. This result is called the *Schmidt, Mirsky, Eckhart-Young theorem*, a very significant concept that appears in signal processing, statistics, machine learning, semantic indexing (search engines), and control theory.

Problem 3. Write a function that accepts a matrix A and a positive integer s.

1. Use your function from Problem 1 or `scipy.linalg.svd()` to compute the compact SVD of A, then form the truncated SVD by stripping off the appropriate columns and entries from U_1, Σ_1, and V_1. Return the best rank s approximation A_s of A (with respect to the induced 2-norm and Frobenius norm).

2. Also return the number of entries required to store the truncated form $\widehat{U}\widehat{\Sigma}\widehat{V}^{\mathsf{H}}$ (where $\widehat{\Sigma}$ is stored as a one-dimensional array, not the full diagonal matrix). The number of entries stored in NumPy array can be accessed by its `size` attribute.

   ```
   >>> A = np.random.random((20, 20))
   >>> A.size
   400
   ```

3. If s is greater than the number of nonzero singular values of A (meaning $s > \text{rank}(A)$), raise a `ValueError`.

Use `np.linalg.matrix_rank()` to verify the rank of your approximation.

Error of Low-Rank Approximations

Another result of the Schmidt, Mirsky, Eckhart-Young theorem is that the exact 2-norm error of the best rank-s approximation A_s for the matrix A is the $(s+1)$th singular value of A:

$$\|A - A_s\|_2 = \sigma_{s+1}. \tag{6.2}$$

This offers a way to approximate A within a desired error tolerance ε: choose s such that σ_{s+1} is the largest singular value that is less than ε, then compute A_s. This A_s throws away as much information as possible without violating the property $\|A - A_s\|_2 < \varepsilon$.

Problem 4. Write a function that accepts a matrix A and an error tolerance ε.

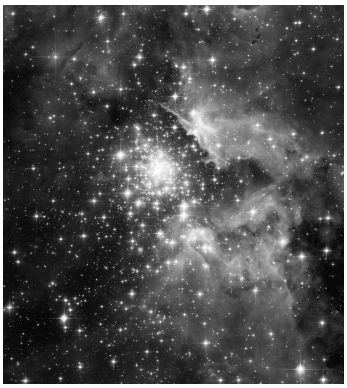
1. Compute the compact SVD of A, then use (6.2) to compute the lowest rank approximation A_s of A with 2-norm error less than ε. Avoid calculating the SVD more than once. (Hint: `np.argmax()`, `np.where()`, and/or fancy indexing may be useful.)
2. As in the previous problem, also return the number of entries needed to store the resulting approximation A_s via the truncated SVD.
3. If ε is less than or equal to the smallest singular value of A, raise a `ValueError`; in this case, A cannot be approximated within the tolerance by a matrix of lesser rank.

This function should be close to identical to the function from Problem 3, but with the extra step of identifying the appropriate s. Construct test cases to validate that $\|A - A_s\|_2 < \varepsilon$.

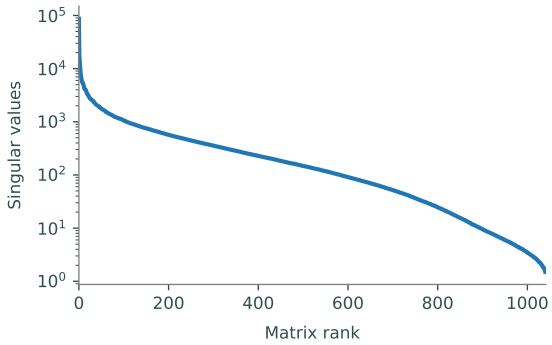
Image Compression

Images are stored on a computer as matrices of pixel values. Sending an image over the internet or a text message can be expensive, but computing and sending a low-rank SVD approximation of the image can considerably reduce the amount of data sent while retaining a high level of image detail. Successive levels of detail can be sent after the inital low-rank approximation by sending additional singular values and the corresponding columns of V and U.

Examining the singular values of an image gives us an idea of how low-rank the approximation can be. Figure 6.2 shows the image in `hubble_gray.jpg` and a log plot of its singular values. The plot in 6.2b is typical for a photograph—the singular values start out large but drop off rapidly. In this rank 1041 image, 913 of the singular values are 100 or more times smaller than the largest singular value. By discarding these relatively small singular values, we can retain all but the finest image details, while storing only a rank 128 image. This is a **huge** reduction in data size.



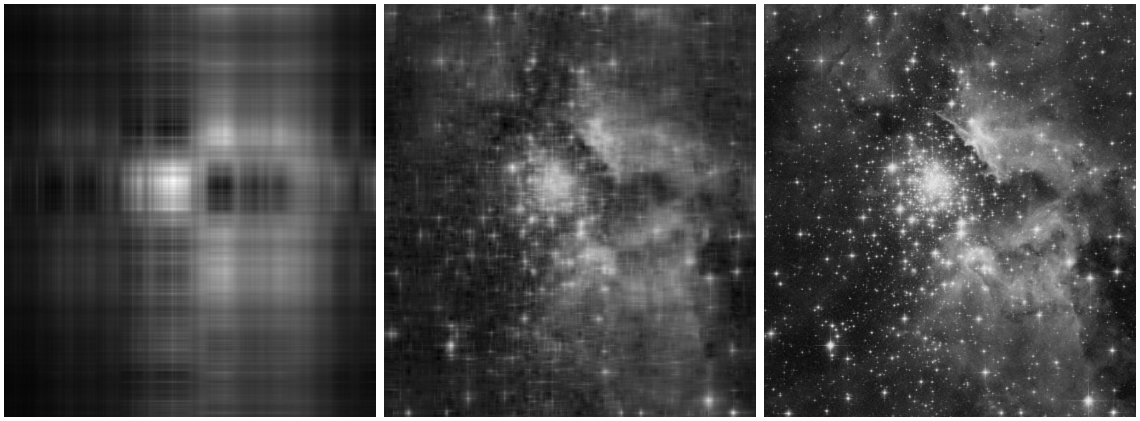
(a) NGC 3603 (Hubble Space Telescope).



(b) Singular values on a log scale.

Figure 6.2

Figure 6.3 shows several low-rank approximations of the image in Figure 6.2a. Even at a low rank the image is recognizable. By rank 120, the approximation differs very little from the original.



(a) Rank 2 (b) Rank 20 (c) Rank 120

Figure 6.3

Grayscale images are stored on a computer as 2-dimensional arrays, while color images are stored as 3-dimensional arrays—one layer each for red, green, and blue arrays. To read and display images, use `imageio.imread()` and `plt.imshow()`. Images are read in as integer arrays with entries between 0 and 255 (`dtype=np.uint8`), but `plt.imshow()` works better if the image is an array of floats in the interval $[0, 1]$. Scale the image properly by dividing the array by 255.

```
>>> from imageio import imread
>>> from matplotlib import pyplot as plt

# Send the RGB values to the interval (0,1).
>>> image_gray = imread("hubble_gray.jpg") / 255.
>>> image_gray.shape            # Grayscale images are 2-d arrays.
(1158, 1041)
>>> image_color = imread("hubble.jpg") / 255.
>>> image_color.shape           # Color images are 3-d arrays.
(1158, 1041, 3)

# The final axis has 3 layers for red, green, and blue values.
>>> red_layer = image_color[:,:,0]
>>> red_layer.shape
(1158, 1041)

# Display a gray image.
>>> plt.imshow(red_layer, cmap="gray")
>>> plt.axis("off")             # Turn off axis ticks and labels.
>>> plt.show()

# Display a color image.
>>> plt.imshow(image_color)     # cmap=None by default.
>>> plt.axis("off")
>>> plt.show()
```

Problem 5. Write a function that accepts the name of an image file and an integer s. Use your function from Problem 3, to compute the best rank-s approximation of the image. Plot the original image and the approximation in separate subplots. In the figure title, report the difference in number of entries required to store the original image and the approximation (use `plt.suptitle()`).

Your function should be able to handle both grayscale and color images. Read the image in and check its dimensions to see if it is color or not. Grayscale images can be approximated directly since they are represented by 2-dimensional arrays. For color images, let R, G, and B be the matrices for the red, green, and blue layers of the image, respectively. Calculate the low-rank approximations R_s, G_s, and B_s separately, then put them together in a new 3-dimensional array of the same shape as the original image.
(Hint: `np.dstack()` may be useful for putting the color layers back together.)

Finally, it is possible for the low-rank approximations to have values slightly outside the valid range of RGB values. Set any values outside of the interval $[0, 1]$ to the closer of the two boundary values.
(Hint: fancy indexing and/or `np.clip()` may be useful here.)

To check, compressing `hubble_gray.jpg` with a rank 20 approximation should appear similar to Figure 6.3b and save $1,161,478$ matrix entries.

Additional Material

More on Computing the SVD

For an $m \times n$ matrix A of rank $r < \min\{m, n\}$, the compact SVD of A neglects last $m - r$ columns of U and the last $n - r$ columns of V. The remaining columns of each matrix can be calculated by using Gram-Schmidt orthonormalization. If $m < r < n$ or $n < r < m$, only one of U_1 and V_1 will need to be filled in to construct the full U or V. Computing these extra columns is one way to obtain a basis for $\mathscr{N}(A^H)$ or $\mathscr{N}(A)$.

Algorithm 6.1 begins with the assumption that we have a way to compute the eigenvalues and eigenvectors of $A^H A$. Computing eigenvalues is a notoriously difficult problem, and computing the SVD from scratch without an eigenvalue solver is much more difficult than the routine described by Algorithm 6.1. The procedure involves two phases:

1. Factor A into $A = U_a B V_a^H$ where B is bidiagonal (only nonzero on the diagonal and the first superdiagonal) and U_a and V_a are orthonormal. This is usually done via *Golub-Kahan Bidiagonalization*, which uses Householder reflections, or *Lawson-Hanson-Chan bidiagonalization*, which relies on the QR decomposition.
2. Factor B into $B = U_b \Sigma V_b^H$ by the QR algorithm or a divide-and-conquer algorithm. Then the SVD of A is given by $A = (U_a U_b)\Sigma(V_a V_b)^H$.

For more details, see Lecture 31 of [TB97] or Section 5.4 of *Applied Numerical Linear Algebra* by James W. Demmel.

Animating Images with Matplotlib

Matplotlib can be used to animate images that change over time. For instance, we can show how the low-rank approximations of an image change as the rank s increases, showing how the image is recovered as more ranks are added. Try using the following code to create such an animation.

```
from matplotlib import pyplot as plt
from matplotlib.animation import FuncAnimation

def animate_images(images):
    """Animate a sequence of images. The input is a list where each
    entry is an array that will be one frame of the animation.
    """
    fig = plt.figure()
    plt.axis("off")
    im = plt.imshow(images[0], animated=True)

    def update(index):
        plt.title("Rank {} Approximation".format(index))
        im.set_array(images[index])
        return im,              # Note the comma!

    a = FuncAnimation(fig, update, frames=len(images), blit=True)
    plt.show()
```

See `https://matplotlib.org/examples/animation/dynamic_image.html` for another example.