1

Image Segmentation

Lab Objective: *Graph theory has a variety of applications. A graph (or network) can be represented in many ways on a computer. In this lab we study a common matrix representation for graphs and show how certain properties of the matrix representation correspond to inherent properties of the original graph. We also introduce tools for working with images in Python, and conclude with an application of using graphs and linear algebra to segment images.*
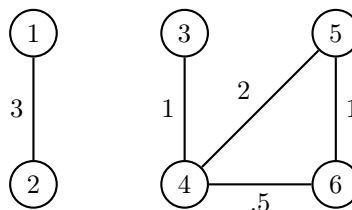
Graphs as Matrices

A *graph* is a mathematical structure that represents relationships between objects. Graphs are defined by $G = (V, E)$, where V is a set of *vertices* (or *nodes*) and E is a set of *edges*, each of which connects one node to another. A graph can be classified in several ways.

- The edges of an *undirected* graph are bidirectional: if an edge goes from node A to node B, then that same edge also goes from B to A. For example, the graphs G_1 and G_2 in Figure 1.1 are both undirected. In a *directed graph*, edges only go one way, usually indicated by an arrow pointing from one node to another. In this lab, we focus on undirected graphs.

- The edges of a *weighted* graph have a weight assigned to them, such as G_2. A weighted graph could represent a collection of cities with roads connecting them: each vertex would represent a city, and the edges would represent roads between the cities. The length of each road could be the weight of the corresponding edge. An *unweighted* graph like G_1 does not have weights assigned to its edges, but any unweighted graph can be thought of as a weighted graph by assigning a weight of 1 to every edge.
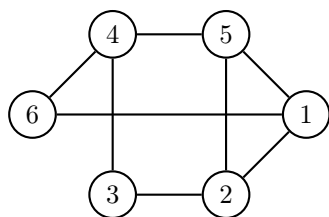
(a) G_1, an unweighted undirected graph.   (b) G_2, a weighted undirected graph.

Figure 1.1

1

Adjacency, Degree, and Laplacian Matrices

For computation and analysis, graphs are commonly represented by a few special matrices. For these definitions, let G be a graph with N nodes and let w_{ij} be the weight of the edge connecting node i to node j (if such an edge exists).

1. The *adjacency matrix* of G is the $N \times N$ matrix A with entries

$$a_{ij} = \begin{cases} w_{ij} & \text{if an edge connects node i and node j} \\ 0 & \text{otherwise.} \end{cases}$$

The adjacency matrices A_1 of G_1 and A_2 of G_2 are

$$A_1 = \begin{bmatrix} 0 & 1 & 0 & 0 & 1 & 1 \\ 1 & 0 & 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 1 & 1 \\ 1 & 1 & 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 1 & 0 & 0 \end{bmatrix}, \qquad A_2 = \begin{bmatrix} 0 & 3 & 0 & 0 & 0 & 0 \\ 3 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 2 & .5 \\ 0 & 0 & 0 & 2 & 0 & 1 \\ 0 & 0 & 0 & .5 & 1 & 0 \end{bmatrix}.$$

   Notice that these adjacency matrices are symmetric. This is always the case for undirected graphs since the edges are bidirectional.

2. The *degree matrix* of G is the $N \times N$ diagonal matrix D whose ith diagonal entry is

$$d_{ii} = \sum_{j=1}^{N} w_{ij}. \qquad (1.1)$$

The degree matrices D_1 of G_1 and D_2 of G_2 are

$$D_1 = \begin{bmatrix} 3 & 0 & 0 & 0 & 0 & 0 \\ 0 & 3 & 0 & 0 & 0 & 0 \\ 0 & 0 & 2 & 0 & 0 & 0 \\ 0 & 0 & 0 & 3 & 0 & 0 \\ 0 & 0 & 0 & 0 & 3 & 0 \\ 0 & 0 & 0 & 0 & 0 & 2 \end{bmatrix}, \qquad D_2 = \begin{bmatrix} 3 & 0 & 0 & 0 & 0 & 0 \\ 0 & 3 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 3.5 & 0 & 0 \\ 0 & 0 & 0 & 0 & 3 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1.5 \end{bmatrix}.$$

   The ith diagonal entry of D is called the *degree* of node i, the sum of the weights of the edges leaving node i.

3. The *Laplacian matrix* of G is the $N \times N$ matrix L defined as

$$L = D - A, \qquad (1.2)$$

where D is the degree matrix of G and A is the adjacency matrix of G. For G_1 and G_2, the Laplacian matrices L_1 and L_2 are

$$L_1 = \begin{bmatrix} 3 & -1 & 0 & 0 & -1 & -1 \\ -1 & 3 & -1 & 0 & -1 & 0 \\ 0 & -1 & 2 & -1 & 0 & 0 \\ 0 & 0 & -1 & 3 & -1 & -1 \\ -1 & -1 & 0 & -1 & 3 & 0 \\ -1 & 0 & 0 & -1 & 0 & 2 \end{bmatrix}, \qquad L_2 = \begin{bmatrix} 3 & -3 & 0 & 0 & 0 & 0 \\ -3 & 3 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & -1 & 0 & 0 \\ 0 & 0 & -1 & 3.5 & -2 & -.5 \\ 0 & 0 & 0 & -2 & 3 & -1 \\ 0 & 0 & 0 & -.5 & -1 & 1.5 \end{bmatrix}.$$

> **Problem 1.** Write a function that accepts the adjacency matrix A of a graph G. Use (1.1) and (1.2) to compute the Laplacian matrix L of G.
> (Hint: The diagonal entries of D can be computed in one line by summing A over an axis.)
>
> Test your function on the graphs G_1 and G_2 from Figure 1.1 and validate your results with `scipy.sparse.csgraph.laplacian()`.

Connectivity

A *connected graph* is a graph where every vertex is connected to every other vertex by at least one path. For example, G_1 is connected, whereas G_2 is not because there is no path from node 1 (or node 2) to node 3 (or nodes 4, 5, or 6). The naïve brute-force algorithm for determining if a graph is connected is to check that there is a path from each edge to every other edge. While this may work for very small graphs, most interesting graphs have thousands of vertices, and for such graphs this approach is prohibitively expensive. Luckily, an interesting result from algebraic graph theory relates the connectivity of a graph to its Laplacian matrix.

If L is the Laplacian matrix of a graph, then the definition of D and the construction $L = D - A$ guarantees that the rows (and columns) of L must each sum to 0. Therefore L cannot have full rank, so $\lambda = 0$ must be an eigenvalue of L. Furthermore, if L represents a graph that is **not** connected, more than one of the eigenvalues of L must be zero. To see this, let $J \subset \{1, 2, \ldots, N\}$ such that the vertices $\{v_j\}_{j \in J}$ form a connected component of the graph, meaning that there is a path between each pair of vertices in the set. Next, let $\mathbf{x}$ be the vector with entries

$$x_k = \begin{cases} 1, & k \in J \\ 0, & k \notin J. \end{cases}$$

Then $\mathbf{x}$ is an eigenvector of L corresponding to the eigenvalue $\lambda = 0$.

For example, the example graph G_2 has two connected components.

1. $J_1 = \{1, 2\}$ so that $\mathbf{x}_1 = [1, 1, 0, 0, 0, 0]^\mathsf{T}$. Then

$$L_2 \mathbf{x}_1 = \begin{bmatrix} 3 & -3 & 0 & 0 & 0 & 0 \\ -3 & 3 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & -1 & 0 & 0 \\ 0 & 0 & -1 & 3.5 & -2 & -.5 \\ 0 & 0 & 0 & -2 & 3 & -1 \\ 0 & 0 & 0 & -.5 & -1 & 1.5 \end{bmatrix} \begin{bmatrix} 1 \\ 1 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix} = \mathbf{0}.$$

2. $J_2 = \{3, 4, 5, 6\}$ and hence $\mathbf{x}_2 = [0, 0, 1, 1, 1, 1]^\mathsf{T}$. Then

$$L_2 \mathbf{x}_2 = \begin{bmatrix} 3 & -3 & 0 & 0 & 0 & 0 \\ -3 & 3 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & -1 & 0 & 0 \\ 0 & 0 & -1 & 3.5 & -2 & -.5 \\ 0 & 0 & 0 & -2 & 3 & -1 \\ 0 & 0 & 0 & -.5 & -1 & 1.5 \end{bmatrix} \begin{bmatrix} 0 \\ 0 \\ 1 \\ 1 \\ 1 \\ 1 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix} = \mathbf{0}.$$

In fact, it can be shown that the number of zero eigenvalues of the Laplacian exactly equals the number of connected components. This makes calculating how many connected components are in a graph only as hard as calculating the eigenvalues of its Laplacian.

A Laplacian matrix L is always a positive semi-definite matrix when all weights in the graph are positive, meaning that its eigenvalues are each nonnegative. The second smallest eigenvalue of L is called the *algebraic connectivity* of the graph. It is clearly 0 for non-connected graphs, but for a connected graph, the algebraic connectivity provides useful information about its sparsity or "connectedness." A higher algebraic connectivity indicates that the graph is more strongly connected.

Problem 2. Write a function that accepts the adjacency matrix A of a graph G and a small tolerance value `tol`. Compute the number of connected components in G and its algebraic connectivity. Consider all eigenvalues that are less than the given `tol` to be zero.

Use `scipy.linalg.eig()` or `scipy.linalg.eigvals()` to compute the eigenvalues of the Laplacian matrix. These functions return complex eigenvalues (with negligible imaginary parts); use `np.real()` to extract the real parts.

Images as Matrices

Computer images are stored as arrays of integers that indicate pixel values. Most $m \times n$ grayscale (black and white) images are stored in Python as a $m \times n$ NumPy arrays, while most $m \times n$ color images are stored as 3-dimensional $m \times n \times 3$ arrays. Color image arrays can be thought of as a stack of three $m \times n$ arrays, one each for red, green, and blue values. The datatype for an image array is `np.uint8`, unsigned 8-bit integers that range from 0 to 255. A 0 indicates a black pixel while a 255 indicates a white pixel.

Use `imageio.imread()` to read an image from a file and `imageio.imwrite()` to save an image. Matplotlib's `plt.imshow()` displays an image array, but it displays arrays of floats between 0 and 1 more cleanly than arrays of 8-bit integers. Therefore it is customary to scale the array by dividing each entry by 255 before processing or showing the image. In this case, a 0 still indicates a black pixel, but now a 1 indicates pure white.

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

>>> image = imread("dream.png")       # Read a (very) small image.
>>> print(image.shape)                # Since the array is 3-dimensional,
(48, 48, 3)                           # this is a color image.

# The image is read in as integers from 0 to 255.
>>> print(image.min(), image.max(), image.dtype)
0 254 uint8

# Scale the image to floats between 0 and 1 for Matplotlib.
>>> scaled = image / 255.
>>> print(scaled.min(), scaled.max(), scaled.dtype)
0.0 0.996078431373 float64

# Display the scaled image.
>>> plt.imshow(scaled)
>>> plt.axis("off")
```

A color image can be converted to grayscale by averaging the RGB values of each pixel, resulting in a 2-D array called the *brightness* of the image. To properly display a grayscale image, specify the keyword argument `cmap="gray"` in `plt.imshow()`.

```python
# Average the RGB values of a colored image to obtain a grayscale image.
>>> brightness = scaled.mean(axis=2)        # Average over the last axis.
>>> print(brightness.shape)                 # Note that the array is now 2-D.
(48, 48)

# Display the image in gray.
>>> plt.imshow(brightness, cmap="gray")
>>> plt.axis("off")
```

Finally, it is often important in applications to flatten an image matrix into a large 1-D array. Use `np.ravel()` to convert a $m \times n$ array into a 1-D array with mn entries.

```python
>>> import numpy as np
>>> A = np.random.randint(0, 10, (3,4))
>>> print(A)
[[4 4 7 7]
 [8 1 2 0]
 [7 0 0 9]]

# Unravel the 2-D array (by rows) into a 1-D array.
>>> np.ravel(A)
array([4, 4, 7, 7, 8, 1, 2, 0, 7, 0, 0, 9])

# Unravel a grayscale image into a 1-D array and check its size.
>>> M,N = brightness.shape
>>> flat_brightness = np.ravel(brightness)
>>> M*N == flat_brightness.size
True
>>> print(flat_brightness.shape)
(2304,)
```

Problem 3. Define a class called `ImageSegmenter`.

1. Write the constructor so that it accepts the name of an image file. Read the image, scale it so that it contains floats between 0 and 1, then store it as an attribute. If the image is in color, compute its brightness matrix by averaging the RGB values at each pixel (if it is a grayscale image, the image array itself is the brightness matrix). Flatten the brightness matrix into a 1-D array and store it as an attribute.

2. Write a method called `show_original()` that displays the original image. If the original image is grayscale, remember to use `cmap="gray"` as part of `plt.imshow()`.

ACHTUNG!

Matplotlib's `plt.imread()` also reads image files. However, this function automatically scales PNG image entries to floats between 0 and 1, but it still reads non-PNG image entries as 8-bit integers. To avoid this inconsistent behavior, always use `imageio.imread()` to read images and divide by 255 when scaling is desired.

Graph-based Image Segmentation

Image segmentation is the process of finding natural boundaries in an image and partitioning the image along those boundaries (see Figure 1.2). Though humans can easily pick out portions of an image that "belong together," it takes quite a bit of work to teach a computer to recognize boundaries and sections in an image. However, segmenting an image often makes it easier to analyze, so image segmentation is ongoing area of research in computer vision and image processing.

Figure 1.2: The image `dream.png` and its segments.

There are many ways to approach image segmentation. The following algorithm, developed by Jianbo Shi and Jitendra Malik in 2000 [?], converts the image to a graph and "cuts" it into two connected components.

Constructing the Image Graph

Let G be a graph whose vertices are the mn pixels of an $m \times n$ image (either grayscale or color). Each vertex i has a brightness $B(i)$, the grayscale or average RGB value of the pixel, as well as a coordinate location $X(i)$, the indices of the pixel in the original image array.

Define w_{ij}, the weight of the edge between pixels i and j, by

$$w_{ij} = \begin{cases} \exp\left(-\frac{|B(i)-B(j)|}{\sigma_B^2} - \frac{\|X(i)-X(j)\|}{\sigma_X^2}\right) & \text{if } \|X(i) - X(j)\| < r \\ 0 & \text{otherwise,} \end{cases} \tag{1.3}$$

where r, σ_B^2 and σ_X^2 are constants for tuning the algorithm. In this context, $\|\cdot\|$ is the standard *euclidean norm*, meaning that $\|X(i) - X(j)\|$ is the physical distance between vertices i and j, measured in pixels.

With this definition for w_{ij}, pixels that are farther apart than the radius r are not connected at all in G. Pixels within r of each other are more strongly connected if they are similar in brightness and close together (the value in the exponential is negative but close to zero). On the other hand, highly contrasting pixels where $|B(i) - B(j)|$ is large have weaker connections (the value in the exponential is highly negative).
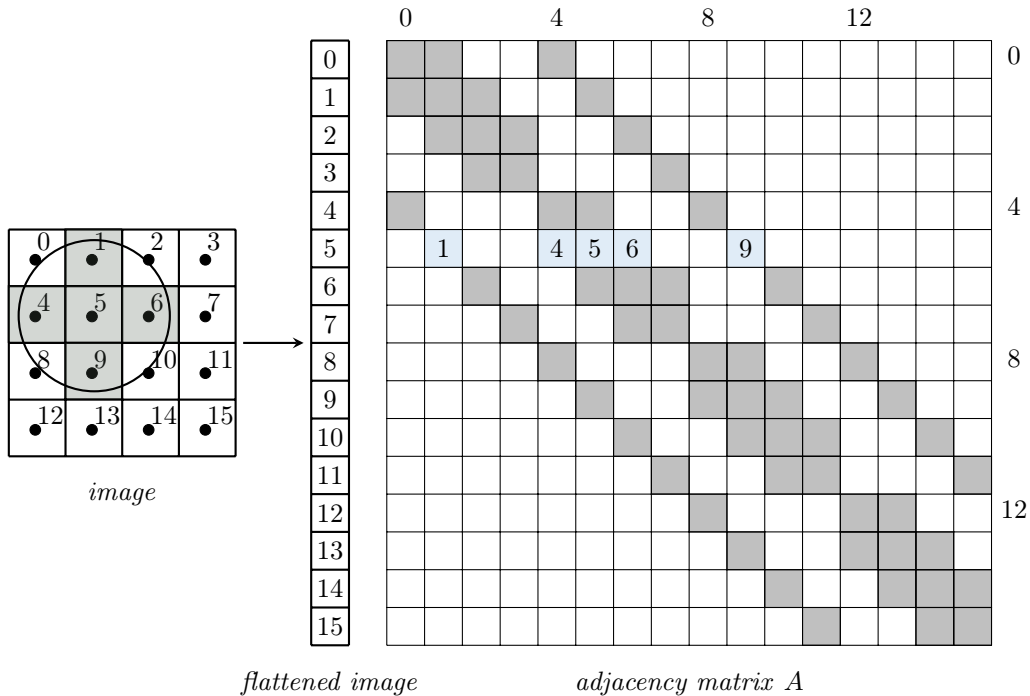

Figure 1.3: The grid on the left represents a 4×4 ($m \times n$) image with 16 pixels. On the right is the corresponding 16×16 ($mn \times mn$) adjacency matrix with all nonzero entries shaded. For example, in row 5, entries 1, 4, 5, and 6, and 9 are nonzero because those pixels are within radius $r = 1.2$ of pixel 5.

Since there are mn total pixels, the adjacency matrix A of G with entries w_{ij} is $mn \times mn$. With a relatively small radius r, A is relatively sparse, and should therefore be constructed and stored as a sparse matrix. The degree matrix D is diagonal, so it can be stored as a regular 1-dimensional NumPy array. The procedure for constructing these matrices can be summarized in just a few steps.

1. Initialize A as a sparse $mn \times mn$ matrix and D as a vector with mn entries.

2. For each vertex i ($i = 0, 1, \ldots, mn - 1$),

   (a) Find the set of all vertices J_i such that $\|X(i) - X(j)\| < r$ for each $j \in J_i$. For example, in Figure 1.3 $i = 5$ and $J_i = \{1, 4, 5, 6, 9\}$.

   (b) Calculate the weights w_{ij} for each $j \in J_i$ according to (1.3) and store them in A.

   (c) Set the ith element of D to be the sum of the weights, $d_i = \sum_{j \in J_i} w_{ij}$.

The most difficult part to implement efficiently is step 2a, computing the neighborhood J_i of the current pixel i. However, the computation only requires knowing the current index i, the radius r, and the height and width m and n of the original image. The following function takes advantage of this fact and returns (as NumPy arrays) both J_i and the distances $\|X(i) - X(j)\|$ for each $j \in J_i$.

```python
def get_neighbors(index, radius, height, width):
    """Calculate the flattened indices of the pixels that are within the given
    distance of a central pixel, and their distances from the central pixel.

    Parameters:
        index (int): The index of a central pixel in a flattened image array
            with original shape (radius, height).
        radius (float): Radius of the neighborhood around the central pixel.
        height (int): The height of the original image in pixels.
        width (int): The width of the original image in pixels.

    Returns:
        (1-D ndarray): the indices of the pixels that are within the specified
            radius of the central pixel, with respect to the flattened image.
        (1-D ndarray): the euclidean distances from the neighborhood pixels to
            the central pixel.
    """
    # Calculate the original 2-D coordinates of the central pixel.
    row, col = index // width, index % width

    # Get a grid of possible candidates that are close to the central pixel.
    r = int(radius)
    x = np.arange(max(col - r, 0), min(col + r + 1, width))
    y = np.arange(max(row - r, 0), min(row + r + 1, height))
    X, Y = np.meshgrid(x, y)

    # Determine which candidates are within the given radius of the pixel.
    R = np.sqrt(((X - col)**2 + (Y - row)**2))
    mask = R < radius
    return (X[mask] + Y[mask]*width).astype(np.int), R[mask]
```

To see how this works, consider Figure 1.3 where the original image is 4×4 and the goal is to compute the neighborhood of the pixel $i = 5$.

```python
# Compute the neighbors and corresponding distances from the figure.
>>> neighbors_1, distances_1 = get_neighbors(5, 1.2, 4, 4)
>>> print(neighbors_1, distances_1, sep='\n')
[1 4 5 6 9]
[ 1.  1.  0.  1.  1.]

# Increasing the radius from 1.2 to 1.5 results in more neighbors.
>>> neighbors_2, distances_2 = get_neighbors(5, 1.5, 4, 4)
>>> print(neighbors_2, distances_2, sep='\n')
[ 0  1  2  4  5  6  8  9 10]
[ 1.41421356  1.          1.41421356  1.          0.          1.
  1.41421356  1.          1.41421356]
```

Problem 4. Write a method for the `ImageSegmenter` class that accepts floats r defaulting to 5, σ_B^2 defaulting to .02, and σ_X^2 defaulting to 3. Compute the adjacency matrix A and the degree matrix D according to the weights specified in (1.3).

Initialize A as a `scipy.sparse.lil_matrix`, which is optimized for incremental construction. Fill in the nonzero elements of A one row at a time. Use `get_neighbors()` at each step to help compute the weights.

(Hint: Try to compute and store an entire row of weights at a time. What does the command `A[5, np.array([1, 4, 5, 6, 9])] = weights` do?)

Finally, convert A to a `scipy.sparse.csc_matrix`, which is faster for computations. Then return A and D.

Use `blue_heart.png` to test A and D, saved as `HeartMatrixA.npz` and `HeartMatrixD.npy` datafiles.

Segmenting the Graph

With an image represented as a graph G, the goal is to now split G into two distinct connected components by removing edges from the existing graph. This is called *cutting* G, and the set of edges that are removed is called the *cut*. The cut with the least weight will best segment the image.

Let D be the degree matrix and L be the Laplacian matrix of G. Shi and Malik [?] proved that the eigenvector corresponding to the second smallest[1] eigenvalue of $D^{-1/2}LD^{-1/2}$ can be used to minimize the cut: the indices of its positive entries are the indices of the pixels in the flattened image which belong to one segment, and the indices of its negative entries are the indices of the pixels which belong to the other segment. In this context $D^{-1/2}$ refers to element-wise exponentiation, so the (i,j)th entry of $D^{-1/2}$ is $1/\sqrt{d_{ij}}$.

Because A is $mn \times mn$, the desired eigenvector has mn entries. Reshaping the eigenvector to be $m \times n$ allows it to align with the original image. Use the reshaped eigenvector to create a boolean mask that indexes one of the segments. That is, construct a $m \times n$ array where the entries belonging to one segment are `True` and the other entries are `False`.

```
>>> x = np.arange(-5,5).reshape((5,2)).T
>>> print(x)
[[-5 -3 -1  1  3]
 [-4 -2  0  2  4]]

# Construct a boolean mask of x describing which entries of x are positive.
>>> mask = x > 0
>>> print(mask)
[[False False False  True  True]
 [False False False  True  True]]

# Use the mask to zero out all of the nonpositive entries of x.
>>> x * mask
array([[0, 0, 0, 1, 3],
       [0, 0, 0, 2, 4]])
```

---

[1]Both D and L are symmetric matrices, so all eigenvalues of $D^{-1/2}LD^{-1/2}$ are real, and therefore "the second smallest one" is well-defined.

Problem 5. Write a method for the `ImageSegmenter` class that accepts an adjacency matrix A as a `scipy.sparse.csc_matrix` and a degree matrix D as a 1-D NumPy array. Construct an $m \times n$ boolean mask describing the segments of the image.

1. Compute the Laplacian L with `scipy.sparse.csgraph.laplacian()` or by converting D to a sparse diagonal matrix and computing $L = D - A$ (do not use your function from Problem 1 unless it works correctly and efficiently for sparse matrices).

2. Construct $D^{-1/2}$ as a sparse diagonal matrix using D and `scipy.sparse.diags()`, then compute $D^{-1/2}LD^{-1/2}$. Use `@` or the `dot()` method of the sparse matrix for the matrix multiplication, **not** `np.dot()`.

3. Use `scipy.sparse.linalg.eigsh()` to compute the eigenvector corresponding to the second-smallest eigenvalue of $D^{-1/2}LD^{-1/2}$. Set the keyword arguments `which="SM"` and `k=2` to compute only the two smallest eigenvalues and their eigenvectors.

4. Reshape the eigenvector as a $m \times n$ matrix and use this matrix to construct the desired boolean mask. Return the mask.

Multiplying the boolean mask component-wise by the original image array produces the *positive segment*, a copy of the original image where the entries that aren't in the segment are set to 0. Computing the *negative segment* requires inverting the boolean mask, then multiplying the inverted mask with the original image array. Finally, if the original image is a $m \times n \times 3$ color image, the mask must be stacked into a $m \times n \times 3$ array to facilitate entry-wise multiplication.

```
>>> mask = np.arange(-5,5).reshape((5,2)).T > 0
>>> print(mask)
[[False False False  True  True]
 [False False False  True  True]]

# The mask can be negated with the tilde operator ~.
>>> print(~mask)
[[ True  True  True False False]
 [ True  True  True False False]]

# Stack a mask into a 3-D array with np.dstack().
>>> print(mask.shape, np.dstack((mask, mask, mask)).shape)
(2, 5) (2, 5, 3)
```

Problem 6. Write a method for the `ImageSegmenter` class that accepts floats r, σ_B^2, and σ_X^2, with the same defaults as in Problem 4. Call your methods from Problems 4 and 5 to obtain the segmentation mask. Plot the original image, the positive segment, and the negative segment side-by-side in subplots. Your method should work for grayscale or color images.

Use `dream.png` as a test file and compare your results to Figure 1.2.