Lab 9

Image Segmentation

Lab Objective: *Understand some basic applications of eigenvalues to graph theory. Learn how to calculate the Laplacian matrix of a graph. Apply the Laplacian matrix to determine connectivity of a graph and segment an image.*

Graph Theory

Graph theory is a branch of mathematics dealing with mathematical structures called graphs. Graphs represent relationships between objects. An *undirected graph* is a set of nodes (or vertices) and edges, where each edge connects exactly two nodes (see Figure 9.1). In a *directed graph*, edges are directional. Each edge only goes one way, usually visualized as an arrow pointing from one node to another. In this lab, we will only consider undirected graphs, which we will simply call graphs (unless we wish to emphasize the fact that they are undirected).

A *weighted* graph is a graph with a weight attached to each edge. For example, a weighted graph could represent a collection of cities with roads connecting them. The vertices would be cities, the edges would be roads, and weight of an edge would be the length of a road. Such a graph is depicted in Figure 9.2.

Any unweighted graph can be thought of as a weighted graph by assigning a weight of 1 to each edge.

Adjacency, Degree, and Laplacian Matrices

We will now introduce three matrices associated with a graph. Throughout this section, assume we are working with a weighted undirected graph with N nodes, and that w_{ij} is the weight attached to the edge connecting node i and node j. We first define the adjacency matrix.

Definition 9.1. *The* adjacency matrix *is an $N \times N$ matrix whose (i,j)-th entry is*

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

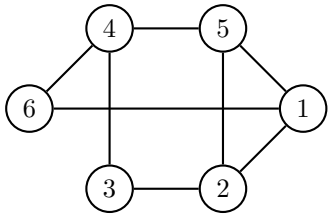

Figure 9.1: An undirected graph that is connected.

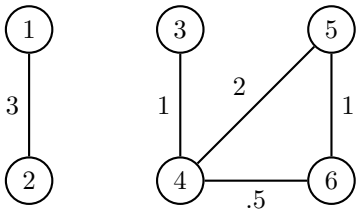

Figure 9.2: A weighted undirected graph that is not connected.

For example, the graph in Figure 9.1 has the adjacency matrix A_1 and the graph in Figure 9.2 has the adjacency matrix A_2, where

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

Notice that these adjacency matrices are symmetric. This will always be the case for undirected graphs.

The second matrix is the degree matrix.

Definition 9.2. *The* degree matrix *is an $N \times N$ diagonal matrix whose (i,i)-th entry is*

$$\sum_{j=1}^{N} w_{ij}.$$

This quantity is the sum of the weight of each edge leaving node i.

We call the (i,i)-th entry of the degree matrix the *degree* of node i. As an example, the degree matrices of the graphs in Figures 9.1 and 9.2 are D_1 and D_2, respectively.

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

Finally, we can combine the degree matrix and the adjacency matrix to get the Laplacian matrix.

Definition 9.3. *The* Laplacian matrix *of a graph is*

$$D - A$$

where D is the degree matrix and A is the adjacency matrix of the graph.

For example, the Laplacian matrix of the graphs in Figures 9.1 and 9.2 are L_1 and L_2, respectively, where

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

In this lab we will learn about graphs by studying their Laplacian matrices. While the Laplacian matrix seems simple, we can learn surprising things from its eigenvalues.

Problem 1. Write a function that accepts the adjacency matrix of a graph as an argument and returns the Laplacian matrix. Test your function on the graphs in Figures 9.1 and 9.2.

Hint: You can compute the diagonal of the degree matrix in one line by summing over an axis (see Lab 2 and NumPy Visual Guide).

Connectivity: First Application of Laplacians

A *connected graph* is a graph where every vertex is connected to every other vertex by at least one path. The graph in Figure 9.1 is connected, whereas the graph in Figure 9.2 is not. It is often important to know if a graph is connected. A naive approach to determine connectivity of a graph is to search every possible path from each vertex. While this works for very small graphs, most interesting graphs will have thousands of vertices, and for such graphs this approach is not feasible.

Instead of the naive approach, we can use an interesting result from algebraic graph theory. This result relates the connectivity of a graph to its Laplacian.

The Laplacian of any graph always has at least one zero eigenvalue. Why is this true? If L is the Laplacian matrix of a graph, then the rows of L must sum to 0. (Think about how L was created.) Since this is true, 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 will be zero. To see this, let $J \subset \{1 \dots N\}$ such that the vertices $\{v_j\}_{j \in J}$ form a connected component of the graph. Define a vector $\mathbf{x}$ such that

$$\mathbf{x}_j = \begin{cases} 1, & j \in J \\ 0, & j \notin J \end{cases}$$

Then $\mathbf{x}$ is an eigenvector of L corresponding to the eigenvalue $\lambda = 0$. (Look at the Laplacian matrix in Figure 9.2 and consider the product $L_2\mathbf{x}$.) In other words, for each connected component, 0 appears at least once as an eigenvalue.

In fact, it can be rigorously proven that the number of zero eigenvalues of the Laplacian exactly *equals* the number of connected components. If we can solve for the eigenvalues of L, this makes it simple to calculate how many connected components are in the graph.

L is always a positive semi-definite matrix when all weights in the graph are positive, so all of its eigenvalues are greater than or equal to 0. The second smallest eigenvalue of L is known as the *algebraic connectivity* of the graph. It is clearly 0 for non-connected graphs. For connected graphs, the algebraic connectivity can give us useful information about the sparsity or "connectedness" of a graph. A higher algebraic connectivity indicates that the graph is more strongly connected.

Problem 2. Compute the number of connected components in a graph. Write a function that accepts the adjacency matrix of a graph and returns two arguments: the number of connected components, and the algebraic connectivity of the graph (second smallest eigenvalue of the Laplacian).

Use the `scipy.linalg` package to compute the eigenvalues. Note that this package will return complex eigenvalues (with negligible imaginary parts). Keep only the real parts. Your function should also accept a tolerance value, such that all eigenvalues less than this value are assumed to be zero. This should default to `tol=1e-8`.

Image Segmentation: Second Application of Laplacians

Image segmentation is the process of finding natural boundaries in an image (see Figure 9.3). This is a simple task for humans, who can easily pick out portions of an image that "belong together." In this lab, you will learn one way for a computer to segment images using graph theory.

The algorithm we will present comes from a paper by Jianbo Shi and Jitendra Malik in 2000 ([**Shi2000**]). They segment an image using the following steps.

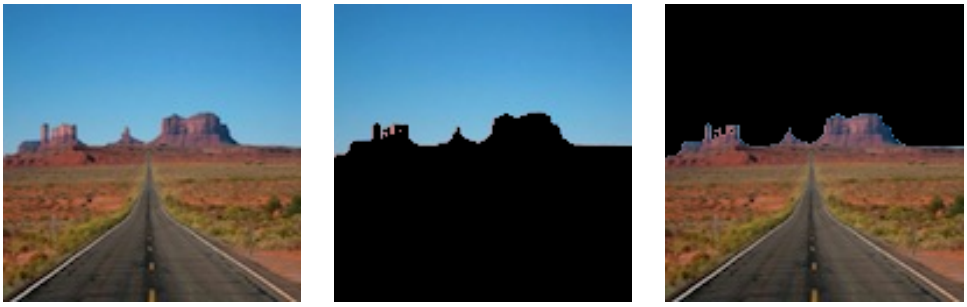

Figure 9.3: An image and its segments.

1. **Represent the image as a weighted graph of pixels.** An image is made up of individual pixels, each having a brightness and a location. (Here, *brightness* is equivalent to the grayscale value of the pixel.) To define a graph representing the image, we let each pixel be a vertex. The weight of the edges between two pixels is determined by their distance apart and their similarity in brightness. We define the graph so that two similar pixels (i.e., with similar brightness and location) will be connected by a strong edge.

2. **Calculate the Laplacian.** We calculate the adjacency and degree matrices of the graph representing the image, and use these to obtain the Laplacian.

3. **Choose the best cut.** The graph we have created is connected, but not all edges have the same weight. Dissimilar pixels will be connected by edges that have a low weight. We can split the graph into two connected components by "cutting" it along the low-weight edges. These components are the image segments we are looking for. (We can also cut an image multiple times to segment it into more than two pieces.) As we will see later, Shi and Malik's algorithm uses spectral information (eigenvectors) of the Laplacian to minimize the weight of the cut edges.

Defining the Graph and Adjacency Matrix

We now define the graph that represents an image. In our $M \times N$ image, the associated graph will have MN nodes, one representing each pixel. We let w_{ij} be the weight of the edge connecting pixels i and j, and define

$$w_{ij} = \begin{cases} \exp(-\frac{|I(i)-I(j)|}{\sigma_I^2} - \frac{d(i,j)}{\sigma_d^2}) & \text{for } d(i,j) < r \\ 0 & \text{otherwise,} \end{cases} \tag{9.1}$$

where

- $d(i,j)$ is the Euclidean distance between pixel i and pixel j
- $|I(i) - I(j)|$ is the difference in brightness of pixels i and j
- r, σ_I and σ_d are constants that we choose

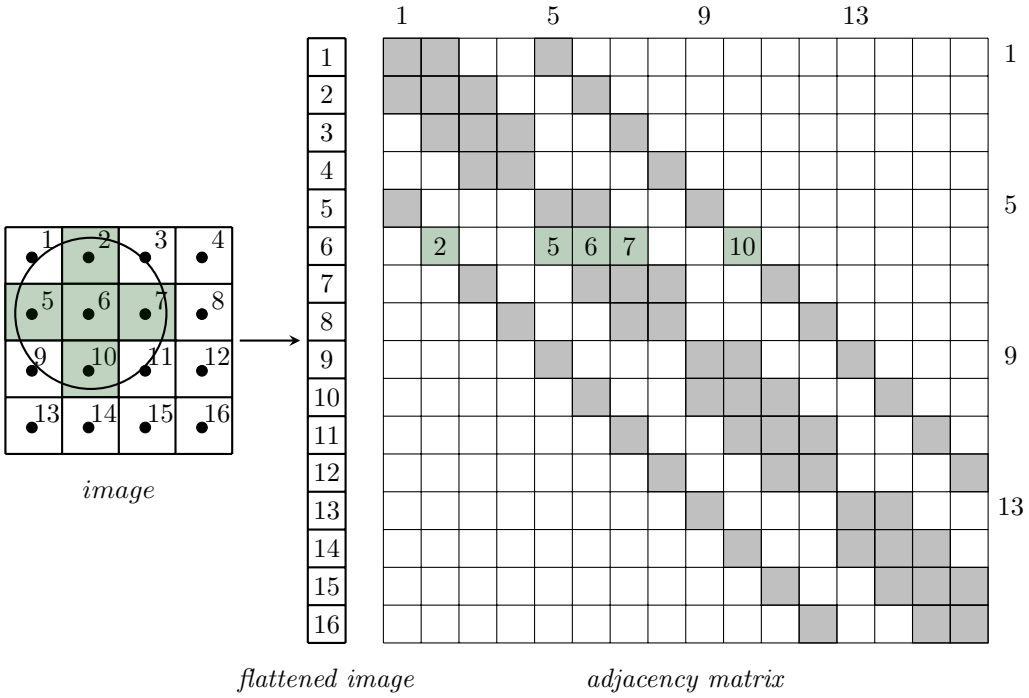

Figure 9.4: The grid on the left represents a 4×4 (or $M \times N$) image with 16 pixels. At right is the corresponding 16×16 (or $(MN) \times (MN)$) adjacency matrix with all nonzero entries shaded. For example, in the 6^{th} row, entries 2, 5, 6, 7, and 10 are nonzero because those pixels are within radius r of pixel 6 (here $r = 1.2$).

With this definition, pixels that are farther apart than radius r will never be connected. Pixels within r will be strongly connected if they are similar in brightness ($|I(i) - I(j)|$ is small) and close together ($d(i, j)$ is small). Highly contrasting pixels ($|I(i) - I(j)|$ is large) will be weakly connected. This gives us a graph with the properties that we intuitively want.

The adjacency matrix W is $(MN) \times (MN)$ and has w_{ij} as its ijth entry. W will be sparse as long as r is small. Figure 9.4 shows a visualization of the adjacency matrix for a 4×4 image when $r = 1.2$.

Computing the Adjacency Matrix

We will now write a function to compute the adjacency matrix for a given image. The function will accept a filename and constants `radius`, `sigma_I`, and `sigma_d`, and return the adjacency matrix and the diagonal of the corresponding degree matrix.

The basic approach is straightforward. For each pixel in the image, compare it to every other pixel, use (9.1) to calculate the weight of the edge between them, and fill in the weight in the adjacency matrix. This section will discuss how to implement this efficiently in Python.

First, load an image and convert it to grayscale. The included function `getImage` helps handle color images. It accepts a filename and returns a 3-D representation

of the image and a 2-D representation of the brightness values of the image.

It will be useful to flatten the $M \times N$ image. This converts it into a 1-D array of pixels of length MN, essentially giving each pixel an index. We can use `img.flatten` to flatten the array `img`.

```
>>> A = np.array([[1,2],[3,4]])
>>> A.flatten()
array([1,2,3,4])
```

Next, initialize empty adjacency and degree matrices to fill in. As in Figure 9.4, the adjacency matrix will be sparse, so initialize it as a sparse matrix `W`. Use the sparse matrix type `lil_matrix`, which is optimized for filling in a matrix one entry at a time. Since we only need to store the diagonal of the degree matrix, initialize this diagonal as a regular 1-dimensional NumPy array `D`.

We now fill in `W` according to our definition in Equation 9.1. Each pixel in the image corresponds to one row in `W`. For each pixel:

- Find its neighbors (the pixels in the image that are within distance r of it)
- Use 9.1 to calculate the weights connecting the pixel to each of its neighbors
- Fill in these weights in `W`, leaving the rest of the values in the row as zeros. [1]

The sum of the entries of a row in `W` will be the corresponding entry in `D`.

You may choose to use the provided helper function `getNeighbors` to find the neighbors of a given pixel. The function accepts the index of a pixel in the flattened image, along with a value for r and the original image dimensions. It returns two flat arrays: `indices` and `distances`. The array `indices` contains the indices of the neighbor pixels within distance r of the input pixel. The array `distances` contains the corresponding distances of those pixels from the input pixel. Using Figure 9.4 as an example, with the inputs 6, 1.2, 4, and 4 the outputs would be `indices = array([2,5,6,7,10])` and `distances = array([1,1,0,1,1])`. Try running the function with different inputs to build intuition about what it does.

Finally, convert `W` to the sparse matrix type `csc_matrix`, which is faster for computations. Then return `W` and `D`.

Problem 3. Write the function `adjacency` described in this section. Accept an image a filename and constants `radius`, `sigma_I`, and `sigma_d`. Return the corresponding sparse adjacency matrix `W` and the diagonal of the degree matrix `D`. Use (9.1) to compute the weights in the adjacency matrix. For speed, try to compute an entire row of `W` at once, instead of filling in `W` entry by entry.

[1] Note that `W` will be a symmetric matrix. We could potentially speed up this algorithm by taking advantage of this fact.

Minimizing the 'Cut'

As stated earlier, the goal is to split the image into two segments, while minimizing the weight of the edges that are 'cut' in the corresponding graph. Let L be the Laplacian of the adjacency matrix defined in 9.1 and let D be the degree matrix. Shi and Malik proved that using the second smallest eigenvalue of $D^{-1/2}LD^{-1/2}$, we can minimize the 'cut' . Both D and L will be symmetric matrices, so all eigenvalues of $D^{-1/2}LD^{-1/2}$ will be real, therefore the second smallest one is well-defined. (Note that $D^{-1/2}$ refers not to matrix, but element-wise, exponentiation.)

The eigenvector associated to the second smallest eigenvalue is the key to segmenting the image. This eigenvector will have MN entries. Shi and Malik proved that the indices of its *positive* entries are the indices of the pixels in the flattened image which belong in one segment. Likewise, the indices of its *negative* entries are the indices of the pixels which belong in the other segment.

To compute the segments, reshape this eigenvector as an $M \times N$ array, and set the positive entries to True and the negative entries to False. We can multiply this True-False mask entry-wise by the image. This zeros out the pixels in the image corresponding to the `False` entries in the mask, without affecting the pixels corresponding to `True` entries. We can negate the mask using the tilde operator, which lets us compute the other segment of the image. Finally we return the two segments.

Problem 4. Write the function `segment` to segment an image. The function should accept an image filename and return both of the segments. You should call the code you wrote in Problem 3. Use sparse matrices where possible.

Test on the image `dream.png`. Your segments should look like the segments in Figure 9.5 (the original image is on the left).

Hints:

1. After defining $D^{-1/2}$, convert D and $D^{-1/2}$ into sparse matrices using `scipy.sparse.spdiags`.
2. Since are now dealing with sparse instead of dense matrices, you shouldn't use your solution to Problem 1 to calculate the Laplacian.
3. Multiply sparse matrices with `A.dot(B)`.
4. Use `scipy.sparse.eigsh` to calculate the eigenvector. This is a sparse eigenvalue solver optimized for symmetric matrices. Set the keyword `which = "SM"` to return the smallest eigenvalues.
5. The provided function `displayPosNeg` can be used to plot your images.

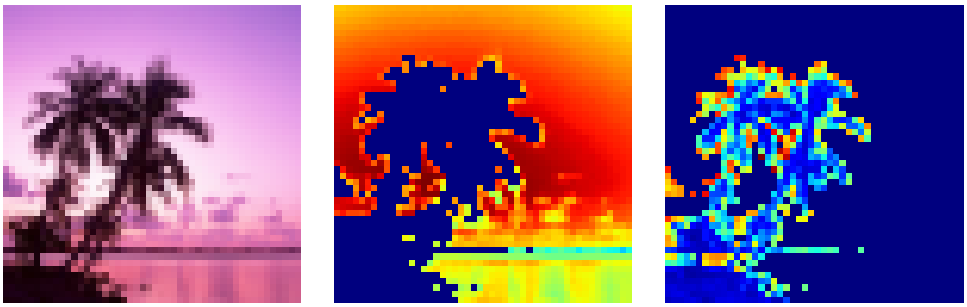

Figure 9.5: Segments of `dream.png`