Lab 10

Image Segmentation

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

Graph Theory

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 10.1). A *directed graph* has the additional information of an arrow on each edge, meaning the edge is only one way. 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 roads, and weight of an edge would be the length of a road. Such a graph is depicted in Figure 10.2.

Any 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 special 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 10.1. *The* adjacency matrix *is an $N \times N$ matrix whose (i, j)-th entry is*

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

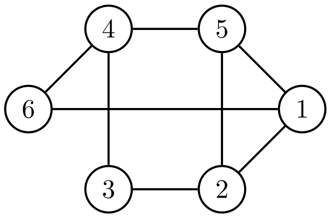

Figure 10.1: An undirected graph that is connected.

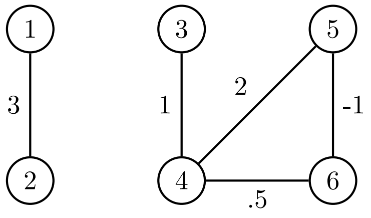

Figure 10.2: A weighted undirected graph that is not connected.

For example, the graph in Figure 10.1 has the adjacency matrix A_1 and the graph in Figure 10.2 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 10.2. *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 just the sum of the weight of each edge that touches 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 10.1 and 10.2 are D_1 and D_2, respectively.

$$D_1 = \begin{pmatrix} 3 & 0 & 0 & 0 & 0 & 0 \\ 0 & 3 & 0 & 0 & 0 & 0 \\ 0 & 0 & 2 & 0 & 0 & 0 \\ 0 & 0 & 0 & 3 & 0 & 0 \\ 0 & 0 & 0 & 0 & 3 & 0 \\ 0 & 0 & 0 & 0 & 0 & 2 \end{pmatrix}. \quad D_2 = \begin{pmatrix} 3 & 0 & 0 & 0 & 0 & 0 \\ 0 & 3 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 3.5 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & -.5 \end{pmatrix}$$

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

Definition 10.3. *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 10.1 and 10.2 are L_1 and L_2, respectively, where

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

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. Your function should return the Laplacian matrix. Hint: You can compute the diagonal of the degree matrix in one line by summing over an axis (see Lab 3). Another hint: Test your function on the graphs in Figures 10.1 and 10.2.

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 10.1 is connected, whereas the graph in Figure 10.2 is not. In applications, 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 (for example, the internet), and for such graphs this approach is not feasible.

Fortunately, there is a better way. Recall that the adjacency matrix of an undirected graph is always symmetric. Thus, it will have real eigenvalues. Surprisingly, a graph is connected if the second smallest eigenvalue of its Laplacian matrix is positive. By second smallest, we mean the second eigenvalue when they are ordered smallest to largest. The eigenvalues of the Laplacian matrix are never negative; thus, if the second one is not zero, the graph is connected. In many applications, the Laplacian matrix is sparse, so by taking advantage of this sparsity, we can cheaply determine if a graph is connected.

Problem 2.

1. Write a function that accepts a symmetric adjacency matrix as an argument and returns the second smallest eigenvalue of the Laplacian matrix. Use the `scipy.linalg` package to compute the eigenvalue. Note: Only return the real part.

2. Here is a function that creates a random symmetric matrix of Boolean values with sparsity determined by the input `c`.

```python
def sparse_generator(n, c):
    ''' Return a symmetric nxn matrix with sparsity determined by c.
    Inputs:
        n (int): dimension of matrix
        c (float): a float in [0,1]. Larger values of c will produce
             matrices with more entries equal to zero.
    '''
    A = np.random.rand(n**2).reshape((n, n))
    A = ( A > c**(.5) )
    return A.T.dot(A)
```

Hint: Test your function on matrices created by `sparse_generator` with inputs $n = 10, 100$ and $c = .25, .5, .95$. What do you notice about the likelihood that a random graph is connected?

Image Segmentation: Second Application of Laplacians

Image segmentation is the process of finding natural boundaries in an image (see Figure 10.3). This is an easy task for humans, who can easily pick out portions of an image that "belong together." In this lab, you will learn one way to program a computer to segment images.

The algorithm we will present comes from a paper by Jianbo Shi and Jitendra Malik in 2000 ([**Shi2000**]). Their idea is to represent an image as a weighted graph as follows. To a computer, an *image* is a collection of *pixels*. Each pixel has a brightness and coordinates describing its location in the image. To define a graph representing this image, we let every pixel be a vertex. Two pixels are connected if the distance between their coordinates is small (less than r). The weight of the edge connecting two pixels is related to their similarity in brightness, where a low

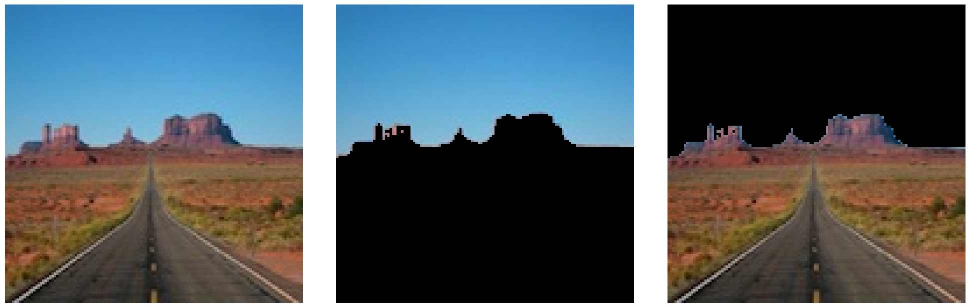

Figure 10.3: An image and its segments.

weight means they are very different.

After defining this graph, we will segment the image by "cutting" (or removing) edges with low weights, which represent lines of high contrast in the image. The "cut" is the total weight of the edges removed. Thus, to segment an image, we wish to minimize the "cut." We can "cut" an image multiple times to segment an image into more than two pieces.

Now let us define the adjacency matrix of the graph associated to an image. Since an $M \times N$ image has $M \times N$ pixels, the adjacency matrix will be $(M \times N) \times (M \times N)$. After choosing a radius r and some constants σ_I and σ_d, we define the adjacency matrix to be $W = (w_{ij})$, where

$$w_{ij} = \begin{cases} \exp\left(-\frac{|I(i)-I(j)|}{\sigma_I^2} - \frac{d(i,j)}{\sigma_d^2}\right) & \text{for } d(i,j) < r \\ 0 & \text{otherwise,} \end{cases} \tag{10.1}$$

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.

Notice that W will be sparse as long as r is small. Figure 10.4 shows what the adjacency matrix looks like for a $4x4$ image when $r = 1.2$.

Computing the Adjacency Matrix

In this section, we will write a function that accepts an image and constants `radius`, `sigma_I`, and `sigma_d`. The function will return the adjacency matrix defined in (10.1) and a list of the diagonals of the corresponding degree matrix. Some helper functions are given to help create the adjacency and diagonal matrices. The input `filename` should be the filename of the image that you want to segment. The function `getImage` will return a 2-D array of brightness values associated with the image. Flatten the array using `img.flatten`.

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

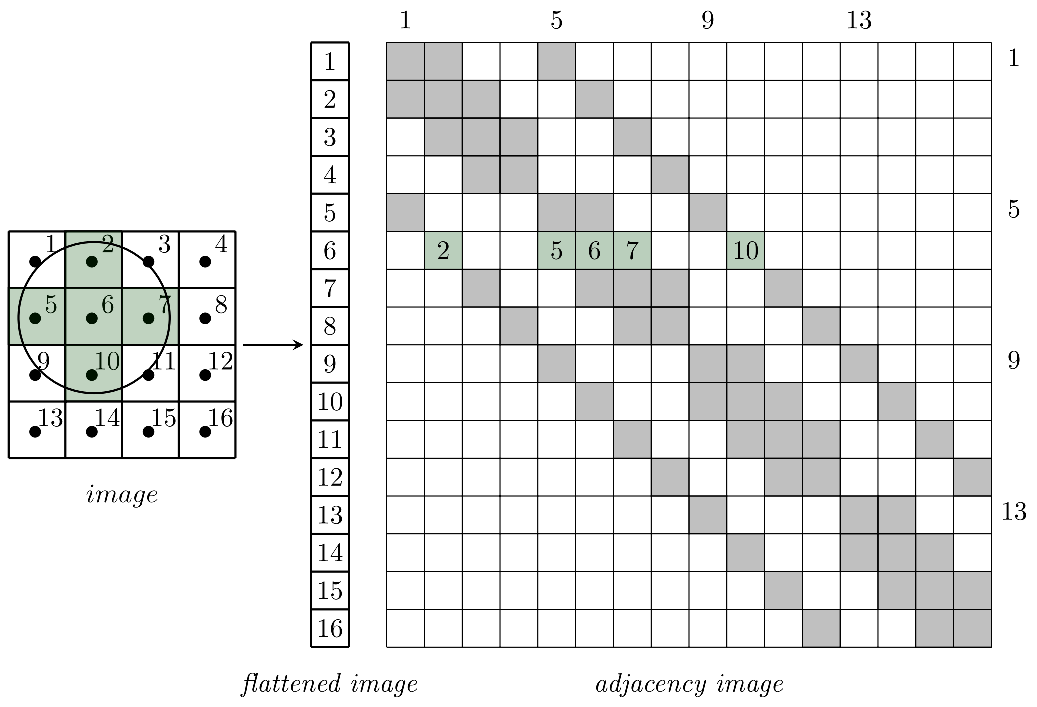

Figure 10.4: The grid at left represents a 4×4 (or $M \times N$) image, which has 16 pixels. Thus the adjacency matrix at right is 16×16 (or $(M \times N) \times (M \times N)$). We have not calculated the weights in the adjacency matrix, but we have shaded the nonzero entries. For example, the 6^{th} row corresponds to the 6^{th} pixel. Within that row, entries are nonzero if they correspond to pixels that are at most 1.2 away from pixel 6. (These nonzero elements correspond to the `indices` and `distances` returned by the `getNeighbors` function given the `pixel` 6 and `radius` 1.2.)

Next we want to initialize the adjacency and degree matrices. As can be seen in Figure 10.4, the adjacency matrix will be sparse, so we initialize it as a sparse matrix `W`. The sparse matrix type `lil_matrix` is optimized for building a matrix one entry at a time, which is what we will do. On the other hand, we only need to compute the diagonal of the degree matrix, so we initialize this diagonal as a regular 1-dimensional NumPy array `D`.

Iterate through each pixel in the image, we initialize the corresponding row of the adjacency matrix. The sum of the entries of this row will be the corresponding entry in `D`.

The function `getNeighbors` returns two flat arrays: `indices` and `distances` (code is provided in the spec file). The array `indices` contains the indices of those pixels within `radius` of the input `pixel`. The array `distances` contains the corresponding distances of those pixels from the input `pixel` (all entries of `distances` will be at most `radius`). According to (10.1), the array `indices` contains exactly the indices of the nonzero entries of the current row of `W`. See Figure 10.4 for an example. [1]

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

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. Return the adjacency matrix and the diagonal matrix for the corresponding image. When computing each row of the adjacency matrix, your computation should use (10.1) and take exactly one line. Hint: Vectorize

Test your function on the image `dream.png`.

Minimizing the 'Cut'

As was mentioned before, we are trying to minimize the 'cut', or the total weight of the edges we remove to create segments. Let L be the Laplacian of the adjacency matrix defined in 10.1 and let D be the degree matrix. Shi and Malik proved that we can minimize the 'cut' by finding the second smallest eigenvalue of $D^{-1/2}LD^{-1/2}$. Note that $D^{-1/2}$ refers not to matrix, but element-wise, exponentiation. Because both D and L will be symmetric matrices, all eigenvalues of $D^{-1/2}LD^{-1/2}$ will be real, and so it makes sense to ask for the second smallest one.

The eigenvector associated to the second smallest eigenvalue will have $M \times N$ entries, some positive and some negative. Reshape this vector to be an $M \times N$ mask defining two segments, segments that correspond to the positive and negative values of the eigenvector. Shi and Malik proved that these are the segments we desire.

Once we have a mask of True-False values, we multiply it by the image entrywise. This zeros out the pixels in the matrix corresponding to the `False` entries in the mask and leaves the pixels corresponding to `True` entries unaffected. 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` described in this section. Test your function on the image `dream.png`. Use the function `displayPosNeg` to plot your images. Your segments should look like the segments in Figure 10.5 (the original image is on the left). Here are some hints:

1. Once you have defined $D^{-1/2}$, convert D and $D^{-1/2}$ into sparse matrices using `scipy.sparse.spdiags`.
2. You should NOT use your solution to Problem 1 to calculate the Laplacian matrix because in this problem we are working with sparse matrices, and you have already computed the degree matrix.
3. Multiply sparse matrices with `A.dot(B)`.
4. Use `scipy.sparse.eigs` to calculate the eigenvector. Set the keyword

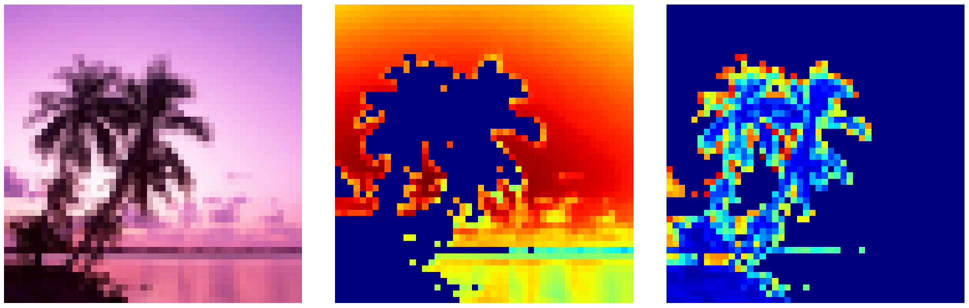

Figure 10.5: Segments of `dream.png`

`which = "SR"`.

5. Reshape the mask to be an $M \times N$ matrix