EECS 127/227AT Optimization Models in Engineering
Spring 2019 **Homework 2**

Release date: 9/12/19
Due date: 9/19/19, 23:00 (11 pm)

Please LaTeX or handwrite your homework solution and submit an electronic version.

Submission Format
Your homework submission should consist of a single PDF file that contains all of your answers (any handwritten answers should be scanned) as well as your IPython notebook saved as a PDF.

If you do not attach a PDF "printout" of your IPython notebook, you will not receive credit for problems that involve coding. Make sure that your results and your plots are visible. Assign the IPython printout to the correct problem(s) on Gradescope.

1. Vectors and Orthogonality

Let $\{v_1, ..., v_n\}$ be an orthonormal basis in $\mathbb{R}^n$. Prove for all $x \in \mathbb{R}^n$ that

$$\sum_{i=1}^{n} |\langle x, v_i \rangle|^2 = \langle x, x \rangle.$$

Solution:

$$\begin{aligned}
\sum_{i=1}^{n} |\langle x, v_i \rangle|^2 &= \sum_{i=1}^{n} \langle x, v_i \rangle^2 \\
&= \sum_{i=1}^{n} (x^\top v_i)(x^\top v_i) \\
&= \sum_{i=1}^{n} (x^\top v_i)(v_i^\top x) \\
&= \sum_{i=1}^{n} x^\top (v_i v_i^\top) x \\
&= x^\top (\sum_{i=1}^{n} v_i v_i^\top) x \\
&= x^\top (V V^\top) x \\
&= x^\top I x \\
&= x^\top x \\
\sum_{i=1}^{n} |\langle x, v_i \rangle|^2 &= \langle x, x \rangle.
\end{aligned}$$

If we view x as a discrete signal and $\langle x, x \rangle$ as its energy, then this results shows that changing the basis used for representing x does not change its energy.

EECS 127 HW2 Ex 2 - Introduction to Image Processing in Python

Representation of Images

What is an image? There isn't one exact answer to this question. Depending on who you ask (and what their area of expertise is), you would get a different answer. Someone who specializes in signal processing would consider it a two dimensional signal, whereas someone in optics might consider it a collection of intensity measurements. Both of these answers are equally correct, they are just different ways to interpret the same thing. And, depending on what we want to do with the image, interpreting images in one or several of these ways may be useful.

The two most common paradigms for images (and fittingly the definitions we will use in this class) are as a matrix or a vector of pixel values. This interpretation of images lends itself well to the concepts of this class, as you will see during the course of the semester.

Now, let’s get into how to read, write, and manipulate images using python. Before we start using real images, let's take a look at a "toy" image to better understand how to work with images in python

Toy Image Work

2a) Images as Matrices

In [1]:
```
import numpy as np
import matplotlib.pyplot as plt
%matplotlib inline
```

In [2]:
```
toy_im = np.reshape(np.linspace(0, 255, 9).astype(int), (3,3))
print(toy_im)
```

```
[[  0  31  63]
 [ 95 127 159]
 [191 223 255]]
```

Let's take a closer look into the toy image we have given to you. We created it by equally dividing the range of values from 0-255 into 9 parts, and reshaping it into a 3x3 matrix. In image nomenclature, each entry of the image matrix is referred to as a **pixel**

In the code box below, print out the pixel intensities (values of the matrix) at the top left, center, and bottom right of the image

For full credit, your solution for the center pixel and bottom right pixel must not use hard coded values, but rather properties of the image (so that this code would work for any size image)

In [3]:
```
end_r, end_c = toy_im.shape
top_left_pixel = toy_im[0,0]
center_pixel = toy_im[end_r//2,end_c//2]
bottom_right_pixel = toy_im[end_r-1,end_c-1]

print("Top Left Pixel Value: {}".format(top_left_pixel))
print("Center Pixel Value: {}".format(center_pixel))
print("Bottom Right Pixel Value: {}".format(bottom_right_pixel))
```

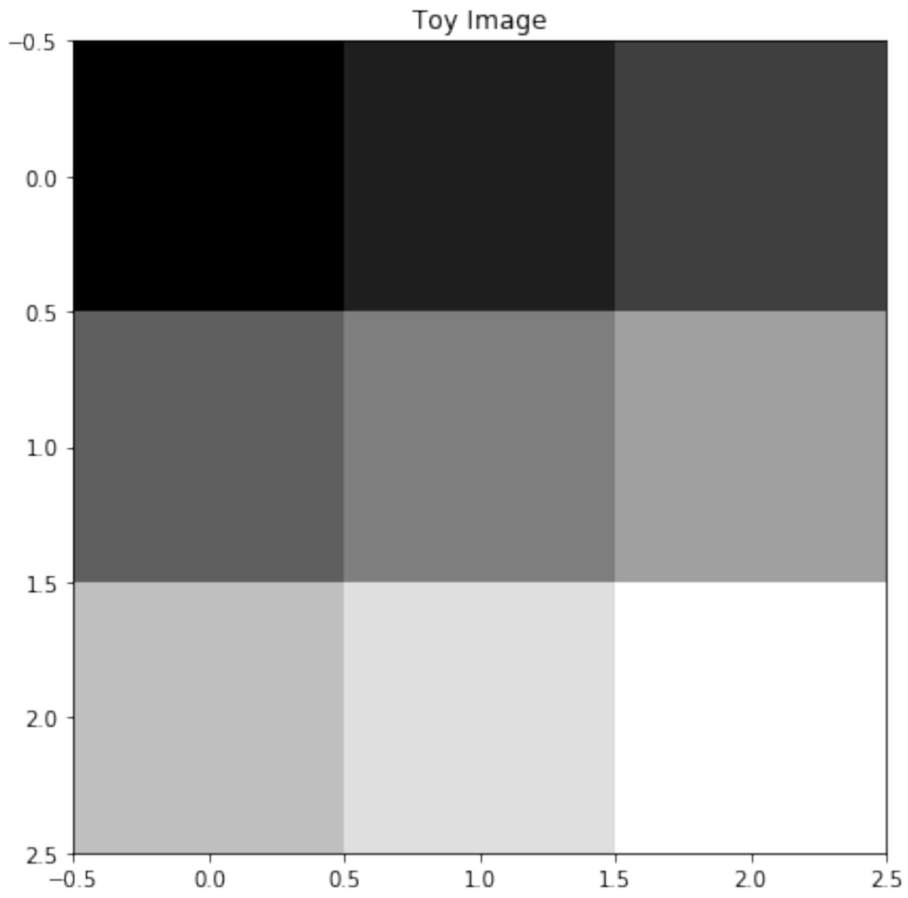
```
Top Left Pixel Value: 0
Center Pixel Value: 127
Bottom Right Pixel Value: 255
```

Note: We chose the range of values to be from 0-255 as convention, because most grayscale images are represented by 8-bit pixel values (and color, or RGB, images get 8 bits per color per pixel)

Now we have a matrix that represents our image, great. However, this doesn't quite look like an image yet. How could we visualize this matrix as an image? Luckily, matplotlib has a function called `imshow` that does exacly what we want to do. It takes in a matrix as the argument and displays it as an image. "Plot" the image below.

Before running the code below, think about the what the matrix might look like as an image.

In [4]:
```
plt.figure(figsize=(7,7))
plt.imshow(toy_im, cmap='gray', vmin=0, vmax=255)
plt.title("Toy Image")
plt.show()
```



Before we move on, let's understand what these keyword arguments do.

- `cmap`: Specifies the color map `imshow` uses to display the image. By setting it to 'gray', we plot using grayscale rather than another color scheme. This is probably the most common way to visualize matrices, but if you are interested in looking at some of the other colormap options, you can read the matplotlib documentation
- `vmin` and `vmax`: Specifies the range of values used to scale the image. If not provided, `imshow` will automatically scale the image between the min and max of the image. This works well in most cases, but these parameters are useful in case you want to change the contrast of the image.

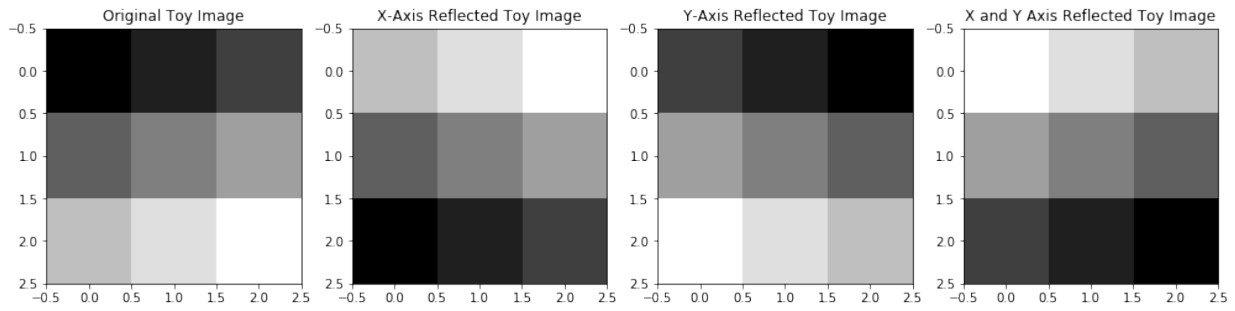
We recommend playing with these parameters in the code block above to get a feel for how they change how the matrix is portrayed.

Reflecting Images

As an exercise of image manipulation, **in the code box below, reflect the `toy_im` across the x-axis (horizontal), y-axis (vertical), and both x and y axes**

Hint: There is a simple solution using array splicing techniques

```
In [5]: x_reflected_im = toy_im[::-1,:]
        y_reflected_im = toy_im[:,::-1]
        xy_reflected_im = toy_im[::-1,::-1]

        plt.figure(figsize=(17,10))
        imgs = [toy_im, x_reflected_im, y_reflected_im, xy_reflected_im]
        titles = ["Original Toy Image", "X-Axis Reflected Toy Image", "Y-Axis
        Reflected Toy Image", "X and Y Axis Reflected Toy Image"]
        for i in range(len(imgs)):
            plt.subplot(1, 4, i+1)
            plt.imshow(imgs[i], cmap='gray')
            plt.title(titles[i])
        plt.show()
```



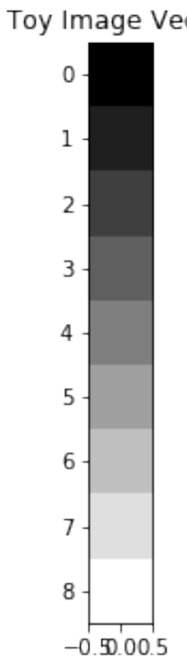
2b) Images as Vectors

Now that we have seen images as a matrix, let's take a look at images as vectors. To go from a matrix to a vector, it is as simple as taking each row of the matrix and stacking them onto each other to get a vector. So, we can represent any $\mathbb{R}^{m \times n}$ matrix as a $\mathbb{R}^{mn}$ vector.

We won't spend too much time on images as vectors in this notebook, but you will see during the rest of the course how this can be a powerful representation that enables us to apply various salient linear algebra concepts to images.

In the code block below, take the `toy_im` matrix and represent it as a vector

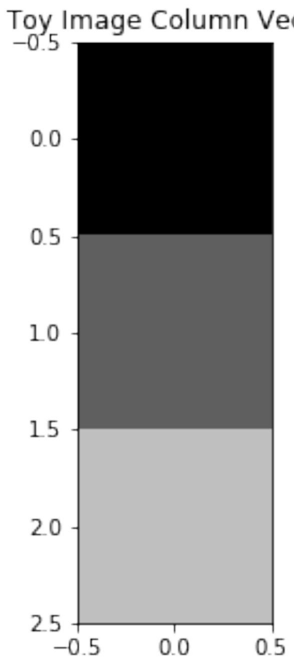
In [6]:
```
#Hint: there is a numpy function that unravels a matrix into a vector
toy_vec = toy_im.ravel()[:]

toy_vec = toy_vec.reshape((9,1))
plt.figure(figsize=(10,5))
plt.imshow(toy_vec, cmap='gray')
plt.title("Toy Image Vector")
plt.show()

#If your code is correct, you should see a vector of grayscale intensi
ties going from
#black at the top and increasingly getting brighter until it is white
at the bottom of the vector
```



Toy Image Column Vector

As an exercise of image manipulation, **in the code box below, display the first column of `toy_im` as a vector**

```
In [7]: im_col = toy_im[:,0,None]

        plt.figure(figsize=(5,5))
        plt.imshow(im_col, cmap='gray', vmin=0, vmax=255)
        plt.title("Toy Image Column Vector")
        plt.show()
```



Note: If did not specify the vmin and vmax in this example, we wouldn't be able to distinguish which column we chose. This is because the `imshow` function would rescale to the min and max of the provided data.

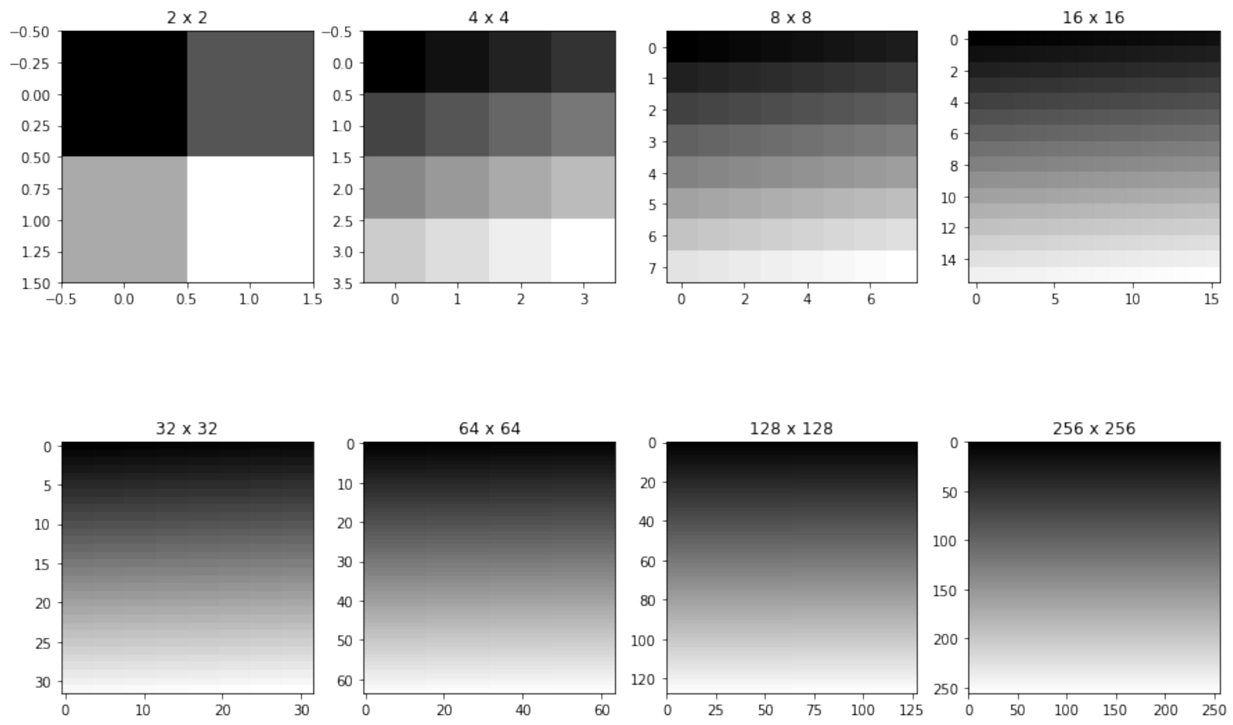
"Continuous" Images

With the work you've done above, you should now be somewhat familiar with the basics of image representation and manipulation in python. However, you may be wondering how these examples would relate to "actual" images, since so far we've just been using this blocky toy example. After all, this toy image isn't what you would first think of when you heard the term "image."

Well, all images have this underlying representation. The only difference is that most images have a much higher **resolution**. What exactly does that mean? Simply put, it means there are more pixels to represent the image. The more pixels we use to represent our image, the more "continuous" our image looks.

Run the code block below to see how a gradient (just like our toy image) can be depicted using varying resolutions

In [8]:
```
plt.figure(figsize=(15,10))
for i in range(1,9):
    plt.subplot(2,4,i)
    dim = 2**i
    plt.title("{} x {}".format(dim, dim))
    plt.imshow(np.reshape(np.linspace(0, 255, dim*dim), (dim,dim)), cm
ap='gray')

plt.show()
```



Real Image Work

Now that we understand how images are represented and manipulated in python, we can now get our hands dirty with a real image.

First, how do we read an image? Well, the answer to that depends on how the image was saved (i.e. its file type). When dealing with raw matrices saved to disk (a `.npy` file), we can use in-built numpy functions to read and write. This is useful if we know we are reading and writing images in python.

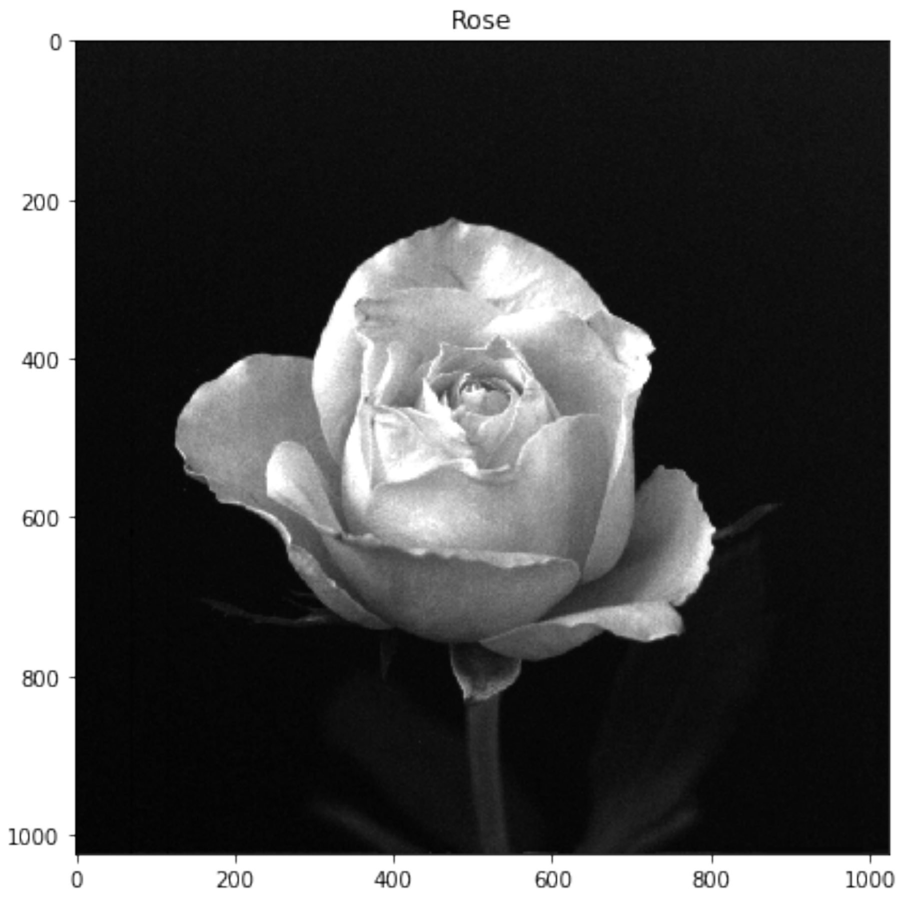
There are plenty of other file formats we can (and probably will during the semester) use, such as `.tif`, `.png`, `.bmp`, `.jpeg`, etc. These file types will also contain the image as a matrix, but add on top of that some metadata (headers) and compression according to some standard. For these types of files, python has external library support, and we will use those libraries to read these files.

For this notebook, we will use matplotlib functions `imread` and `imsave`. With the support of the PIL library (which is included in your ee127 conda env), these functions are capable of reading a variety of file types.

2c) Reading Images

Using the matplotlib `imread` function, read in the image we've provided, named `rose1024.tiff`

In [9]:
```
im = plt.imread("rose1024.tiff")

plt.figure(figsize=(7,7))
plt.imshow(im, cmap='gray')
plt.title("Rose")
plt.show()
```

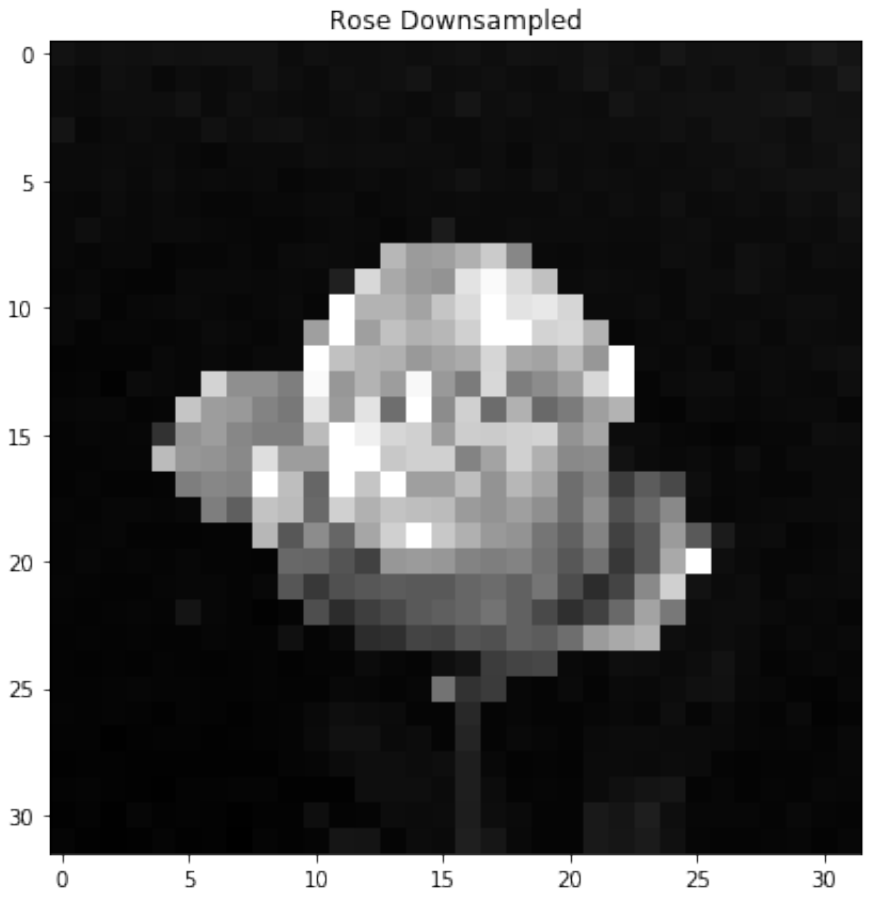


2d) Real Image Manipulation

In the codebox below, downsample the image by a factor of 32 in each dimension to pixelate it

This is similar to doing the opposite of the continuous gradient example above. By downsampling the image, we are using less pixels to represent it, which lowers its resolution and leads to a pixelated effect that you may be familiar with.

```
In [10]: im_low_res = im[::32, ::32]

         plt.figure(figsize=(7,7))
         plt.imshow(im_low_res, cmap='gray')
         plt.title("Rose Downsampled")
         plt.show()
```



2e) Writing Image FIles

Using the matplotlib `imsave` function, save the downsampled image as a new file, and try to open it up from your file manager

Make sure you specify a color map when writing to disk. Also, make sure to specify the file extension in the file name.

```
In [11]: plt.imsave("downsampled_rose.tiff", im_low_res, cmap='gray')
```

3. Representation of a graph as a matrix: the adjacency matrix

In this exercise, we are interpreting a graph as a matrix. Then, we show that finding a path between two nodes can be done with a matrix multiplication. This can be useful to quickly compute several shortest paths, or to re-compute shortest paths when the weights on the links of the graph only change slightly.

We are given a graph as a set of vertices $V = \{1, ..., N\}$, with edges $E \subseteq V \times V$. We assume that the graph is undirected (without arrows), meaning that $(i, j) \in E$ implies $(j, i) \in E$.

We define the adjacency matrix of the graph A by:

$$A_{ij} = \begin{cases} 1 & \text{if } (i,j) \in E, \\ 0 & \text{otherwise.} \end{cases}$$

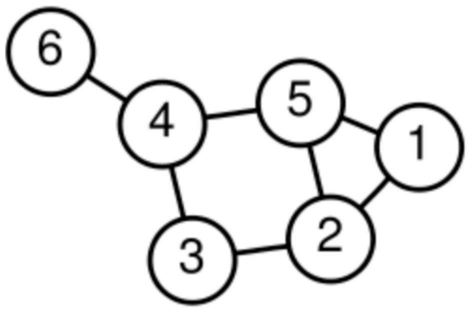


Figure 1: Example of an undirected graph.

(a) Form the adjacency matrix for the graph shown in Figure 1.

Solution: The adjacency matrix for the graph is

$$A = \begin{bmatrix} 0 & 1 & 0 & 0 & 1 & 0 \\ 1 & 0 & 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 1 & 1 \\ 1 & 1 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 1 \end{bmatrix}.$$

(b) Turning to a generic graph, show that the adjacency matrix A is symmetric.

Solution: A is symmetric, by definition of the edge set E:

- If $A_{i,j} = 1$ then there is an edge between i and j. Because the graph is undirected; there is an edge between j and i. Therefore $A_{j,i} = 1 = A_{i,j}$.
- If $A_{i,j} = 0$, then $A_{j,i} \neq 1$ (because $A_{j,i} = 1 \implies A_{i,j} = 1$), so $A_{j,i} = 0 = A_{i,j}$.

Now let's show that if there is a path between $i \in V$ and $j \in V$, then there exists $n \in \mathbb{N}$ such that $e_i^\top A^n e_j \neq 0$. We define e_i as the vector of length N where the ith entry is 1 and all other entries are 0.

Let's proceed by induction. Define the following property:

$$P(n): \text{ there is a path of length } n \text{ between } i \text{ and } j \implies e_i^\top A^n e_j \neq 0 \tag{1}$$

And show that $P(1)$ is true and that $\forall n > 1,\ P(n-1) \implies P(n)$. This will show that $\forall n \geq 1$, $P(n)$ is true.

To get some intuition about the exercise that you will now solve, it is strongly recommended to run the iPython notebook "Adjacency matrix.ipynb". You do **NOT** need to submit the iPython notebook to gradescope.

Let assume that there is a path – denoted p – between i and j. We denote the length of the path p as n.

(c) Show that if $n = 1$ – meaning that there is a path of length 1 between i and j – then $e_i^\top A^n e_j \neq 0$. State what is the value of $e_i^\top A^n e_j$.

Solution: $e_i^\top A e_j = A_{i,j}$ If $n = 1$ then the path is only one link. This link must begin at i and finish at j. So the link is (i, j). Therefore there is a link between i and j: $A_{i,j} = 1$.

So, if $n = 1$, $e_i^\top A e_j = 1$.

This shows $P(1)$.

Now let's assume that the property is true for $n - 1$, with $n > 1$. Let show that it is true for n.

(d) Show that $e_i^\top A^n e_j = \sum\limits_k (e_i^\top A e_k)(e_k^\top A^{n-1} e_j)$. This equation means that any path of length n between i and j is the combination of a link between i and a vertex k and a path of length $n-1$ from k to j.

Solution:

- $A^n = AA^{n-1} = AIA^{n-1}$.
- $I = \sum_k e_k e_k^\top$.
- So $A^n = A \sum_k e_k e_k^\top A^{n-1} = \sum_k A e_k e_k^\top A^{n-1}$.
- Then, $e_i^\top A^n e_j = e_i^\top \sum_k A e_k e_k^\top A^{n-1} e_j = \sum\limits_k (e_i^\top A e_k)(e_k^\top A^{n-1} e_j)$.

We have shown that $e_i^\top A^n e_j = \sum\limits_k (e_i^\top A e_k)(e_k^\top A^{n-1} e_j)$.

(e) Show that $\forall k, (e_i^\top A e_k)(e_k^\top A^{n-1} e_j) \geq 0$.

Solution:

- $e_i^\top A e_k = A_{i,k} \geq 0$ (by definition of $A_{i,k}$).
- $e_k^\top A^{n-1} e_j = \sum\limits_l (e_k^\top A e_l)(e_l^\top A^{n-2} e_j)$. You can do an induction here to show that $e_k^\top A^{n-1} e_j \geq 0$.
- One can also see that if $e_k^\top A^{n-1} e_j < 0$ this will imply that $\exists i, j$ such that $A_{i,j} < 0$. Which is false.

Let's denote l the first node achieved in the path p from i.

(f) Explain why $e_l^\top A^{n-1} e_j \neq 0$.

Solution: If l is the first node achieved in the path p from i, there exists a $\tilde{p}$ between l and j of length $n-1$. We can apply the property $P(n-1)$ for the path between l and j. This gives that $e_l^\top A^{n-1} e_j \neq 0$.

(g) State the value of $e_i^\top A e_l$.

Solution: $e_i^\top A e_l = A_{i,l} = 1$ because there is a link between i and l (the first link of the path p begins at i and arrives at l).

(h) Conclude that $e_i^\top A^n e_j \neq 0$.

Solution:

$e_i^\top A^n e_j = \sum_k (e_i^\top A e_k)(e_k^\top A^{n-1} e_j) = (e_i^\top A e_l)(\mathrm{e}_l^\top A^{n-1} e_j) + \sum_{k \setminus l} (e_i^\top A e_k)(e_k^\top A^{n-1} e_j)$.

But we know, $\forall k, (e_i^\top A e_k)(e_k^\top A^{n-1} e_j) \geq 0$. So $\sum_{k \setminus l} (e_i^\top A e_k)(e_k^\top A^{n-1} e_j) \geq 0$.

Also, $e_i^\top A e_l = A_{i,l} = 1$ and $\mathrm{e}_l^\top A^{n-1} e_j \neq 0$, so $(e_i^\top A e_l)(\mathrm{e}_l^\top A^{n-1} e_j) > 0$.

So: $e_i^\top A^n e_j > 0$. Which gives that: $e_i^\top A^n e_j \neq 0$.

(i) Have you shown that there is a path between i and j if and only if $\exists n : e_i^\top A^n e_j \neq 0$?

Solution: We only have shown:

$$\text{There is a path between } i \text{ and } j \implies \exists n : e_i^\top A^n e_j \neq 0$$

We have not shown that:

$$\text{There is a path of length between } i \text{ and } j \iff \exists n : e_i^\top A^n e_j \neq 0$$

If you enjoyed this exercise, feel free to read:

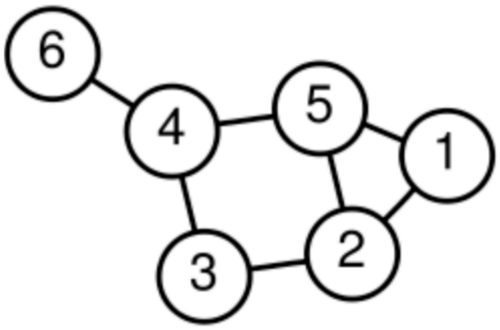
- Algebraic Graph Theory, C.Godsil, G.Royle
- Algebraic Graph Algorithms, P.Sankowski

```
In [1]: import numpy as np
```

1. The Adjacency Matrix

In this notebook we will explore some properties and uses of the adjacency matrix of an undirected graph $G = (V, E)$.

To begin, fill out the adjacency matrix of the below graph:



```
In [2]: #TO DO: Fill out the adjacency matrix:
        A = np.array([[0, 1, 0, 0, 1, 0],
                      [1, 0, 1, 0, 1, 0],
                      [0, 1, 0, 1, 0, 0],
                      [0, 0, 1, 0, 1, 1],
                      [1, 1, 0, 1, 0, 0],
                      [0, 0, 0, 1, 0, 0]])
```

Symmetry of the Adjacency Matrix

It is a fact (that you will show in the homework) that the adjacency matrix of any undirected graph is symmetric.

Let's verify that this is true for the above graph:

```
In [3]: np.prod(A.T == A) == True
Out[3]: True
```

2.1. Length of the shortest path between vertices 1 and 6

We will see in the homework that a path p of length n exists between vertices i and j in an undirected graph G if and only if $e_i^\top A^n e_j$ is nonzero, where A is the adjacency matrix of G and e_i signifies the standard basis vector that is 1 at index i and 0 elsewhere (e_j defined similarly).

This fact can be directly applied to find the shortest path between two vertices i and j. Just loop through increasing values for $n \in \mathbb{N}$ until you find the first n such that $e_i^\top A^n e_j \neq 0$. This n is the length of the shortest path.

```
In [4]: e1 = np.array([1,0,0,0,0,0]).T
        e6 = np.array([0,0,0,0,0,1]).T
```

```
In [5]: B = A
        for i in range(10):
            print(e1.T @ B)
            if (e1.T @ B @ e6)!=0:
                print("Length of the shortest path between 1 and 6 is: " + str
        (i+1))
                print("Value of e1.T A^" + str(i+1) + " e6: " + str(e1.T @ B @
        e6))
                break
            B = B @ A
```

```
[0 1 0 0 1 0]
[2 1 1 1 1 0]
[2 4 2 2 4 1]
Length of the shortest path between 1 and 6 is: 3
Value of e1.T A^3 e6: 1
```

2.2. Length of the shortest path using eigen-decomposition of A

You may have noticed that the above method uses one matrix multiplication per iteration of n, which can be expensive. We can instead use the eigen-decompostiion of A to avoid this.

That is, we use $A = P\Lambda P^\top$ where the columns of P are the eigenvectors of A and Λ is the diagonal matrix of A's eigenvalues. (Note: we can always do this because A is symmetric). Then, instead of calculating A^n directly, we instead do

$$A^n = (P\Lambda P^\top)^n = P\Lambda^n P^\top.$$

Now, instead of multiplying an arbitrary matrix A, which in general takes time $O(n^{2.38})$, we multiply the diagonal matrix Λ, which takes only $O(n)$ time.

In [6]:
```
values, vectors = np.linalg.eig(A)
for i in range(10):
    B = vectors @ np.diag(values**(i+1)) @ vectors.T
    if (e1.T @ B @ e6)> 0.1:
        print("Lenght of the shortest path between 1 and 6 is: " + str
(i+1))
        print("Value of e1.T A^" + str(i+1) + " e6: " + str(e1.T @ B @
e6))
        break
```

```
Lenght of the shortest path between 1 and 6 is: 3
Value of e1.T A^3 e6: 0.9999999999999987
```

Wave-front of Breadth-First Search

We can also use the adjacency matrix A to perform breadth-first search.

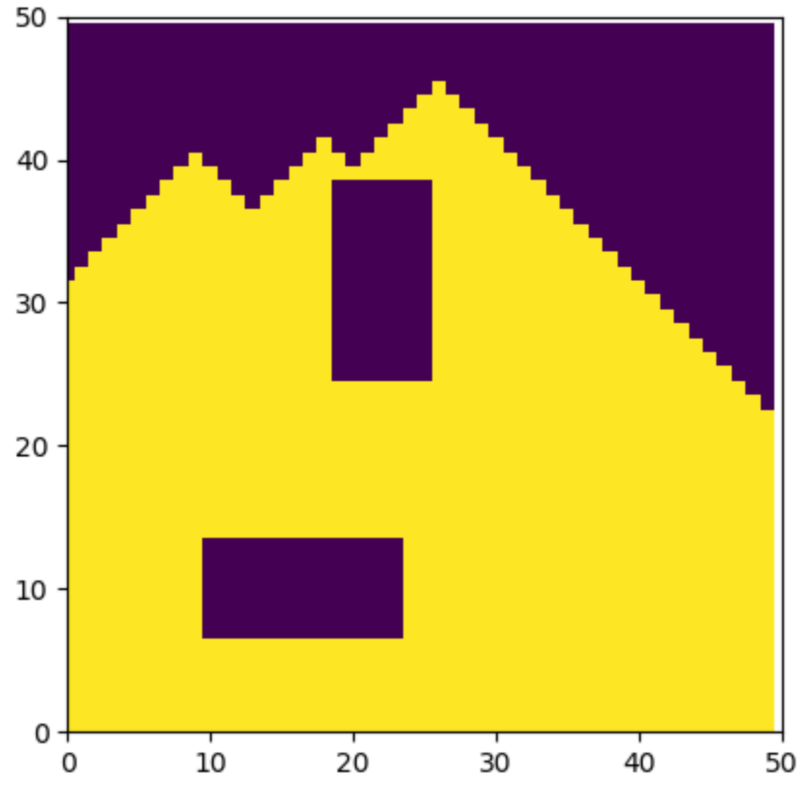
Let v_t represent the nodes visited after t steps of breadth-first search, where $(v_t)_j \neq 0$ if and only if node j was visited at the t-th step. We start with $v_0 = e_i$ where i is the starting node of the BFS.

Notice that in general, when v is elementwise nonnegative then Av is also elementwise nonnegative and nonzero in exactly the indices corresponding to vertices adjacent to those with nonzero indices in v (convince yourself of this). That is, multiplication by A performs exactly the function of taking the next step in the BFS, i.e. $v_{t+1} = Av_t$.

In other words, $A^n e_i$ is a vector that is nonzero at exactly the nodes a distance of n from node i.

In [7]:
```
m = 50
n = m**2
A = np.zeros((n, n))

for k in range(n):
    A[k][k]= 1
    i = k%m
    j = k//m
    if i+1<m:
        A[k][i+1 + m*j] = 1
    if i>0:
        A[k][i-1 + m*j] = 1
    if j+1<m:
        A[k][i + m*(j+1)] = 1
    if j>0:
        A[k][i + m*(j-1)] = 1

# add obstables
for i in range((int) (2*np.sqrt(m))):
    for j in range((int) (np.sqrt(m))):
        A[:, ((int) ((m+1) * np.sqrt(m))) + (i + m*j)] = np.zeros(n)

for i in range((int) (np.sqrt(m))):
    for j in range((int) (2*np.sqrt(m))):
        A[:, ((int) (1.35 * m * np.sqrt(m) * np.sqrt(np.sqrt(m)))) + (
i + m*j)] = np.zeros(n)
```

In [8]:
```
%matplotlib notebook

nb_frame = 100
e1 = np.zeros((n,nb_frame))
e1[320][0] = 1

for i in range(nb_frame):
    if i == 0:
        continue
    e1[:,i] = e1[:,i-1].T @ A

import matplotlib.pyplot as plt
import matplotlib.animation as animation

fig = plt.figure()
ax = plt.axes(xlim=(0, m), ylim=(0, m))

im=plt.imshow((e1[:,1].T).reshape(m,m), interpolation='nearest')

# animation function.  This is called sequentially
def animate(i):
    im.set_array((e1[:,i].T >= 1).reshape(m,m))
    return [im]

ani = animation.FuncAnimation(fig, animate, frames=nb_frame)

plt.show()
```


0
10
20
30
40
50
0
10
20
30
40
50

4. Gram-Schmidt

Any set of n linearly independent vectors in $\mathbb{R}^n$ could be used as a basis for $\mathbb{R}^n$. However, certain bases could be more suitable for certain operations than others. For example, an orthonormal basis could facilitate solving linear equations.

(a) Given a matrix $A \in \mathbb{R}^{n\times n}$, it could be represented as a multiplication of two matrices

$$A = QR,$$

where Q is a unitary matrix (its columns form an orthonormal basis for $\mathbb{R}^n$) and R is an upper-triangular matrix. For the matrix A, describe how Gram-Schmidt process could be used to find the Q and R matrices, and apply this to

$$A = \begin{bmatrix} 3 & -3 & 1 \\ 4 & -4 & -7 \\ 0 & 3 & 3 \end{bmatrix}$$

to find a unitary matrix Q and an upper-triangular matrix R.

Solution: Let a_i and q_i denote the columns of A and Q, respectively. Using Gram-Schmidt, we obtain an orthogonal basis q_i for the column space of A.

$$\begin{aligned} p_1 = a_1, \quad q_1 &= \frac{p_1}{\|p_1\|_2} \\ p_2 = a_2 - (a_2^\top q_1)q_1, \quad q_2 &= \frac{p_2}{\|p_2\|_2} \\ p_3 = a_3 - (a_3^\top q_1)q_1 - (a_3^\top q_2)q_2, \quad q_3 &= \frac{p_3}{\|p_3\|_2} \\ &\vdots \end{aligned}$$

Rearranging terms, we have

$$a_1 = r_{11}q_1 \tag{2a}$$
$$a_i = r_{i1}q_1 + \cdots + r_{ii}q_i, \quad i = 2, ..., n, \tag{2b}$$

where each q_i has unit norm, and $r_{ij}q_j$ denotes the projection of a_i onto the vector q_j for $j \neq i$. Stacking a_i horizontally into A and rewriting (2a-b) in matrix notation, we obtain $A = QR$. For the given matrix, we have

$$A = \begin{bmatrix} 0.6 & 0 & -0.8 \\ 0.8 & 0 & 0.6 \\ 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} 5 & -5 & -5 \\ 0 & 3 & 3 \\ 0 & 0 & -5 \end{bmatrix}.$$

Note that an equivalent factorization is $A = (-Q)(-R)$.

(b) Given an invertible matrix $A \in \mathbb{R}^{n\times n}$ and an observation vector $b \in \mathbb{R}^n$, the solution to the equality

$$Ax = b$$

is given as $x = A^{-1}b$. For the matrix $A = QR$ from part (a), assume that we want to solve

$$Ax = \begin{bmatrix} 8 \\ -6 \\ 3 \end{bmatrix}.$$

By using the fact that Q is a unitary matrix, find $\bar{b}$ such that

$$Rx = \bar{b}.$$

Then, given the upper-triangular matrix R and $\bar{b}$ in part (c), find the elements of x <u>sequentially</u>.

Solution: We note that $Q^{-1} = Q^T$.

$$\begin{aligned} Ax &= b \\ QRx &= b \\ Q^\top QRx = Rx &= Q^\top b. \end{aligned}$$

Thus

$$\bar{b} = Q^\top b = \begin{bmatrix} 0 \\ 3 \\ -10 \end{bmatrix}.$$

Given R and $\bar{b}$, we can find x by back-substitution:

$$\begin{bmatrix} 5 & -5 & -5 \\ 0 & 3 & 3 \\ 0 & 0 & -5 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} 0 \\ 3 \\ -10 \end{bmatrix} \implies x_3 = 2 \implies x_2 = -1 \implies x_1 = 1 \implies x = \begin{bmatrix} 1 \\ -1 \\ 2 \end{bmatrix}.$$

(c) Describe how your solution in the previous problem is akin to Gaussian elimination in solving a system of linear equations.

Solution: In Gaussian elimination, we first perform forward elimination, which reduces the matrix A to an upper triangular matrix for the later backward substitution. In this approach, computing $Q^\top b$ is equivalent to forward elimination, followed by the exact same backward substitution.

(d) Given an invertible matrix $B \in \mathbb{R}^{n \times n}$ and an observation vector $c \in \mathbb{R}^n$, find the computational cost of finding the solution z to the equation $Bz = c$ by using the QR decomposition of B. Assume that Q and R matrices are available, and adding, multiplying, and dividing scalars take one unit of "computation".

As an example, computing the inner product $a^\top b$ is said to be $O(n)$, since we have n scalar multiplications total – one for each $a_i b_i$. Similarly, matrix vector multiplication is $O(n^2)$, since matrix vector multiplication can be viewed as computing n inner products. The computational cost for inverting a matrix in $\mathbb{R}^n$ is $O(n^3)$, and consequently, the cost grows rapidly as the set of equations grows in size. This is why the expression $A^{-1}b$ is usually not computed by directly inverting the matrix A. Instead, the QR decomposition of A is exploited to decrease the computational cost.

Solution: We count the number of operations in back substitution. Solving the initial equation

$$r_{nn} x_n = \bar{b}_n$$

takes 1 multiplication. Solving each subsequent equation takes one more multiplication and one more addition than the previous. In total, we have $1 + 3 + 5 + \cdots$ of operations, which is on the order of $O(n^2)$.

Thus, matrix multiplication and back substitution are both $O(n^2)$. Given the QR decomposition of A, we can solve $Ax = b$ in $O(n^2)$ time.