1. Jupyter Notebook and NumPy Warmup [20pts]

We will make extensive use of Python's numerical arrays (NumPy) and interactive plotting (Matplotlib) in Jupyter notebooks for the course assignments. The first part of this assignment is intended as a gentle warm up in case you haven't used these tools before. Start by reading through the following tutorials:

If you haven't used Jupyter before, a good place to start is with the introductory documentation here:

https://jupyter-notebook.readthedocs.io/en/stable/notebook.html#starting-the-notebook-server
(https://jupyter-notebook.readthedocs.io/en/stable/notebook.html#starting-the-notebook-server)
https://nbviewer.jupyter.org/github/jupyter/notebook/blob/master/docs/source/examples/Notebook/Notebook
(https://nbviewer.jupyter.org/github/jupyter/notebook/blob/master/docs/source/examples/Notebook/Notebook
https://nbviewer.jupyter.org/github/jupyter/notebook/blob/master/docs/source/examples/Notebook/Running%
(https://nbviewer.jupyter.org/github/jupyter/notebook/blob/master/docs/source/examples/Notebook/Running%

This page gives a good introduction to NumPy and many examples of using NumPy along with Matplotlib:

http://www.scipy-lectures.org/intro/numpy/numpy.html (http://www.scipy-lectures.org/intro/numpy/numpy.html)

You should also get comfortable with searching through the documentation as needed

https://docs.scipy.org/doc/numpy-1.13.0/reference/index.html (https://docs.scipy.org/doc/numpy-1.13.0/reference/index.html) https://matplotlib.org/api/_as_gen/matplotlib.pyplot.html (https://matplotlib.org/api/_as_gen/matplotlib.pyplot.html)

NumPy Array Operations

Describe in words what each of each of the following statements does and what the value of `result` will be (i.e. if you were to execute `print(result)`). You should do this with out actually executing the code but instead just looking it and refering to the NumPy documentation.

[1.1]

```
import numpy as np
a = np.arange(5,15)
result = a[::3]
```

It first creates an array range from 5 to 14. Then the indexing jumps three number each time. For example, the first number is 5, then the second is 8(5 + 3), then the third is 11(8 + 3), etc.

result: [5, 8, 11, 14]

[1.2]

```
a = np.arange(1,5)
result = a[::-1]
```

It first creates an array range from 1 to 4. Then the "step" in indexing is '-1' which iterates the number in reverse order.

result: [4, 3, 2, 1]

[1.3]

```
f = np.arange(1840,1860)
g = np.where(f>1850)
result = f[g]
```

It first creates an array range 1840 to 1859. Then np.where return the location of the number which is greater than 1850. So the result shows all the numbers which are greater than 1850.

result: [1851, 1852, 1853, 1854, 1855, 1856, 1857, 1858, 1859]

[1.4]

```
x = np.ones((1,10))
result = x.sum(axis = 1)
```

It first creates an array which contains ten 1s. And then the result is the summation of all the elements in the array.

result: [10.]

NumPy Coding Exercises

Add or modify the code in the cells below as needed to carry out the following steps.

[1.5]

Use **matplotlib.pyplot.imread** to load in a grayscale image of your choice. If you don't have a grayscale image handy, load in a color image and then convert it to grayscale averaging together the three color channels (use **numpy.mean**).

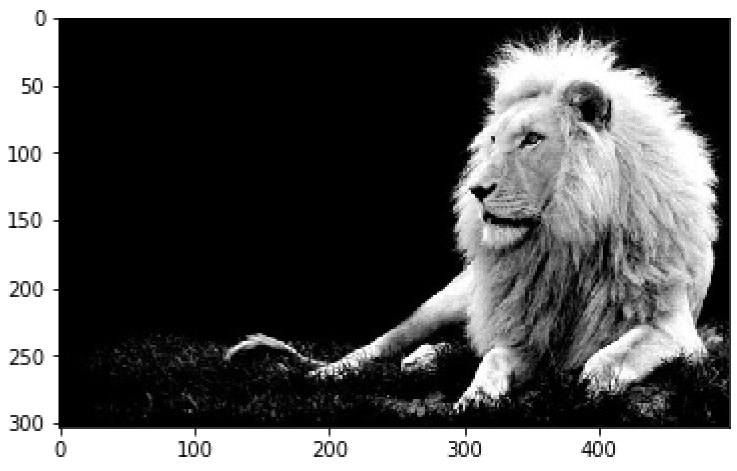
Finally create an array A that contains the pixels in a 100x100 sub-region of your image and display the image in the notebook using the **matplotlib.pyplot.imshow** function.

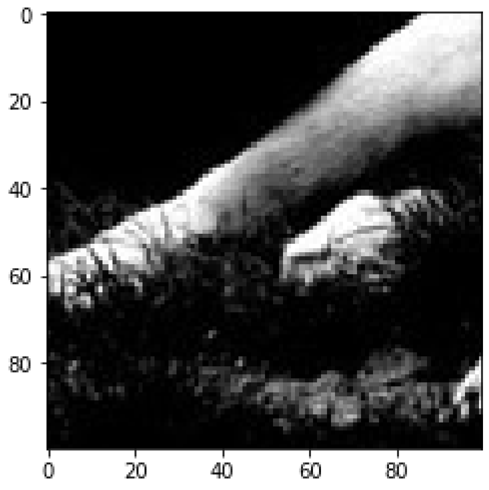
HINT: When loading an image with **imread** it is important to example the data type of the returned array. Depending on the image it may be that `I.dtype = uint8` or `I.dtype = float32`. Integer values range in [0..255] while floating point values for an image will be in [0..1]. A simple approach is to always convert images to floats, this will avoid much confusion and potential bugs later on.

In [1]:

```
import numpy as np
import matplotlib.pyplot as plt

#load an image
I = plt.imread('https://vignette.wikia.nocookie.net/grayscale/images/4/47/Lion.p
ng')

#display the shape of the array and data type
print("I.shape=",I.shape,"\nI.dtype=",I.dtype)

#convert to float data type and scale to [0..1] if necessary
if (I.dtype == np.uint8):
    I = I.astype(float) / 256

#I.dtype should now be float

#if your image is color (shape HxWx3), convert to grayscale by averaging togethe
r R,G,B values
I = np.mean(I, axis = 2)

#display the image in the notebook using a grayscale colormap
plt.imshow(I,cmap=plt.cm.gray)

#force matplotlib to go ahead and display the plot now
plt.show()

#select out a 100x100 pixel subregion of the image
A = I[200:300:, 200:300:]

#display the selected subregion
plt.imshow(0.1 * A, cmap=plt.cm.gray)
plt.show()
```

```
I.shape= (303, 497, 3)
I.dtype= float32
```





[1.6]

In the cell below, describe what happens if you comment out the `plt.show()` lines?

How does the visualization of `A` change if you scale the brightness values (i.e. `plt.imshow(0.1*A,cmap=plt.cm.gray)`)?

Explain what is happening, referring to the **matplotlib** documentation as necessary (https://matplotlib.org/api/_as_gen/matplotlib.pyplot.html (https://matplotlib.org/api/_as_gen/matplotlib.pyplot.html))

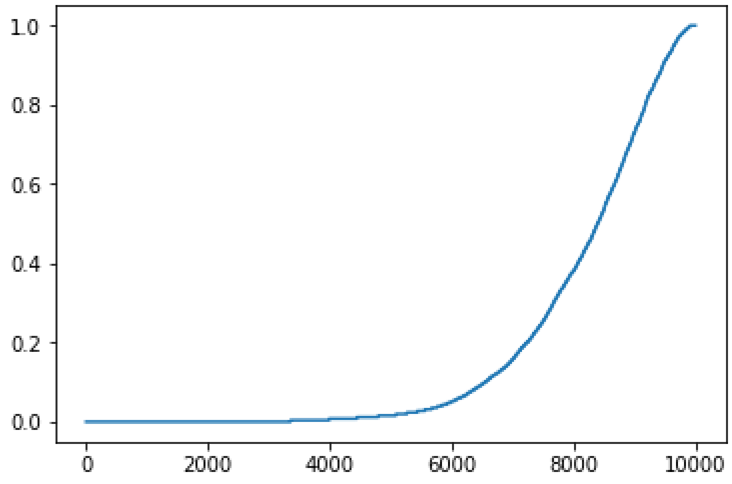
There are two plt.show() lines. Both image A and image I will not display if I comment out plt.show() lines. Visualization of A will not change if I scale the brightness values, because the changes of brightness do not depend on the scaling. And cmap=plt.cm.gray makes the image to be gray color.

[1.7]

Write code in the cell below which (a) puts the values of `A` into a singe 10,000-dimensional column vector `x`, (b) sorts the entries in `x`, and (c) visualizes the contents of the sorted vector `x` by using the **matplotlib.pyplot.plot** function

In [2]:

```
x = A.reshape(10000, 1)
x = np.sort(x, axis = 0)
plt.plot(x)
plt.show()
```

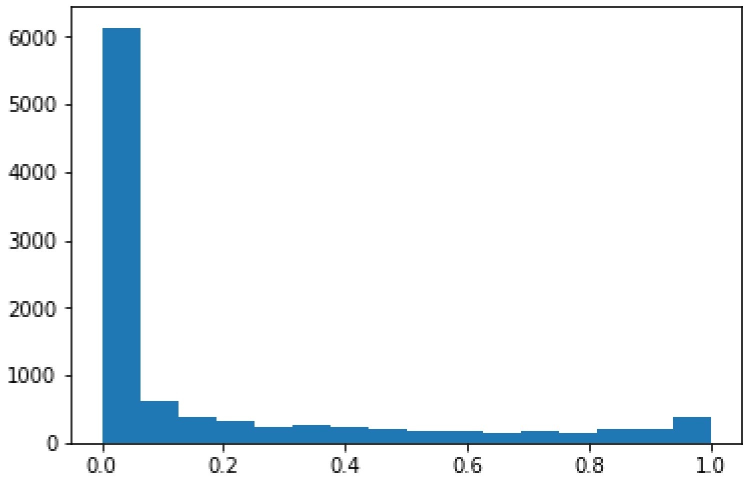


[1.8]

Display a figure showing a histogram of the pixel intensities in `A` using **matplotlib.axes.hist**. Your histogram should have 16 bins. You will need to convert `A` to a vector in order for the histogram to display correctly (otherwise it will show 16 bars for each row of A)

In [3]:

```
y = A.flatten()
plt.hist(y, bins = 16)
plt.show()
```



[1.9]

Create and display a new (binary) image the same size as `A`, which is white wherever the intensity in `A` is greater than a threshold specified by a variable `t`, and black everywhere else. Experiment in order to choose a value for the threshold which makes the image roughly half-white and half-black. Also print out the percentage of pixels which are black for your chosen threshold.

In [4]:

```
tempsort = sorted(y)
threshold = tempsort[4999]
temp = A.copy()
temp[temp <= threshold] = 0.0
temp[temp > threshold] = 1.0
black_perc = len(temp[temp == 0.0])/len(temp)
white_perc = len(temp[temp == 1.0])/len(temp)

print("threshold is: ", threshold, "\n", "percentage of black is: ", black_perc,
 "\n",
      "percentage of white is: ", white_perc)
```

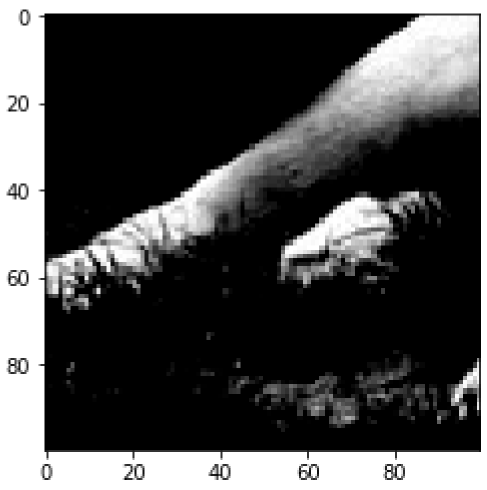
```
threshold is:  0.0156863
 percentage of black is:  50.8
 percentage of white is:  49.2
```

[1.10]

Generate a new grayscale image, which is the same as A, but with A's mean intensity value subtracted from each pixel. After subtracting the mean, set any negative values to 0 and display the result.

In [5]:

```
temp = A.copy()
temp = temp - np.mean(temp)
temp[temp < 0] = 0.0
plt.imshow(temp, cmap=plt.cm.gray)
plt.show()
```



[1.11]

Let `y` be a column vector: `y = [1, 2, 3, 4, 5, 6]` so that `y.shape = (6,1)`. Reshape the vector into a matrix `z` using the **numpy.array.reshape** and (**numpy.array.transpose** if necessary) to form a new matrix `z` whose first column is `[1, 2, 3]`, and whose second column is `[4, 5, 6]`. Print out the resulting array `z`

In [6]:

```
y = np.array([1,2,3,4,5,6])
y = y.reshape(2,3)
z = y.transpose()
print(z)
```

```
[[1 4]
 [2 5]
 [3 6]]
```

[1.12]

Find the minimum value of `A`, if there are multple entries with the same minimum value it is fine to return the first one. Set `r` to be the row in which it occurs and `c` to be the column. Print out `r`, `c`, and `A[r,c]`

In [7]:

```
index = np.where(A == np.min(A))
r = index[0][0]
c = index[1][0]
print(" r = ", r, '\n', 'c = ', c)
print("A[r,c] = ", A[r, c])
```

```
 r =  0
 c =  0
A[r,c] =  0.0
```

[1.13]

Let v be the vector: `v = [1,8,8,2,1,3,9,8]`. Using the unique function, compute and print the total number of unique values that occur in `v`.

In [8]:

```
v = np.array([1,8,8,2,1,3,9,8])
result = len(np.unique(v))
print(result)
```

```
5
```

2. Averaging Images [40pts]

In this exercise you will write code which loads a collection of images (which are all the same size), computes a pixelwise average of the images, and displays the resulting average.

The images below give some examples that were generated by averaging "100 unique commemorative photographs culled from the internet" by Jason Salavon. Your program will do something similar.

image.png

Download the images provided on the Canvas course website for this assignment `averageimage_data.zip`. There are two sets, `set1` and `set2`. Notice that they are all the same size within a single set.

[2.1]

Write a function in the cell below that loads in one of the sets of images and computes their average. You can use the **os.listdir** to get the list of files in the directory. As you load in the images, you should compute an average image on the fly. Color images are represented by a 3-dimensional array of size (HxWx3) where the third dimension indexes the red, green and blue channels. You will want to compute a running average of the red, green and blue slices in order to get your final average color image.

You should encapsulate your code in a function called **average_image** that takes the image directory as an input and returns the average of the images in that directory. Your function should implement some error checking. Specifically your function should skip over any files in the directory that are not images (**plt.imread** will thrown an **OSError** if the file is not an image). It should ignore images that are not color images. Finally, it should also skip any images which are not the same height and width as the first color image you load in.

In [9]:

```
#
# these are the only modules needed for problem #2
#
import numpy as np
import os
import matplotlib.pyplot as plt
```

In [10]:

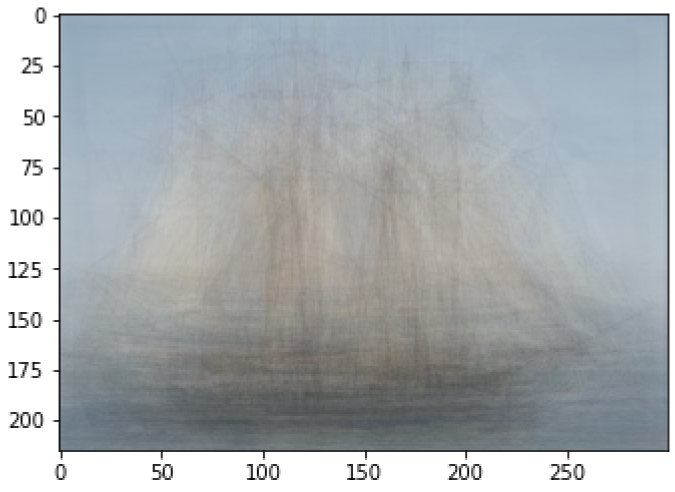
```
def average_image(dirname):
    """
    Computes the average of all color images in a specified directory and return
s the result.

    The function ignores any images that are not color images and ignores any im
ages that are not
    the same size as the first color image you load in

    Parameters
    ----------
    dirname : str
        Directory to search for images

    Returns
    -------
    numpy.array (dtype=float)
        HxWx3 array containing the average of the images found

    """

    #[your code here]
    base_shape = (0,0,0)
    count = 0
    first_image = 0

    for file in os.listdir(dirname):
        filename = os.path.join(dirname,file)
        if os.path.isfile(filename):
            try:
                #load an image
                I = plt.imread(filename)

                #convert to float data type and scale to [0..1] if necessary
                if (I.dtype == np.uint8):
                    I = I.astype(float) / 256

                #load first image and setup
                if first_image == 0:
                    first_image = 1
                    base_shape = I.shape
                    Iaverage = np.zeros([I.shape[0], I.shape[1], 3])
                    continue

                if (len(I.shape)!= 3):
                    continue

                if I.shape != base_shape:
                    continue

                Iaverage = np.add(Iaverage, I)
                count += 1

            except OSError:
                    continue

    Iaverage = np.divide(Iaverage, count)
    return Iaverage
```

[2.2]

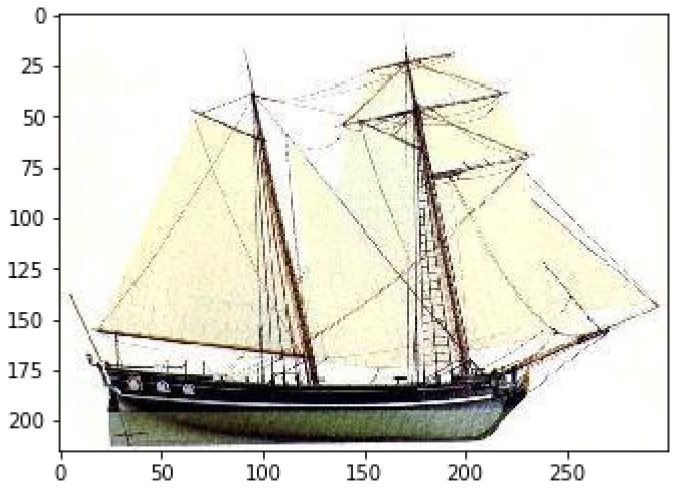
Write code below which calls your **average_image()** function twice, once for each set of images. Display the resulting average images. Also display a single example image from each set for comparison

In [11]:

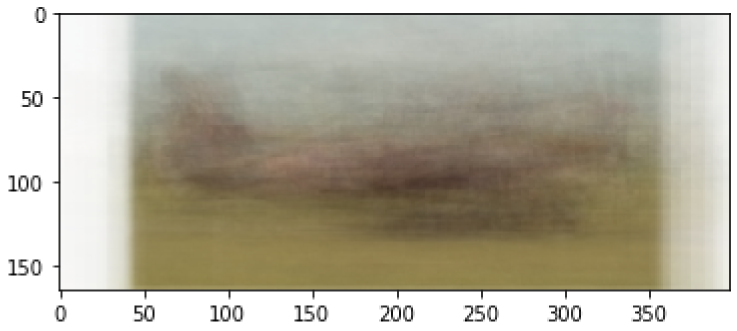
```
Iav = average_image("averageimage_data/set1")
plt.imshow(Iav)
plt.show()

I = plt.imread("averageimage_data/set1/im01.jpg")
plt.imshow(I)
plt.show()

Iav = average_image("averageimage_data/set2")
plt.imshow(Iav)
plt.show()

I = plt.imread("averageimage_data/set2/im01.jpg")
plt.imshow(I)
plt.show()
```



0
25
50
75
100
125
150
175
200
0
50
100
150
200
250



0
25
50
75
100
125
150
175
200
0
50
100
150
200
250



0
50
100
150
0
50
100
150
200
250
300
350

0
50
100
150
0
50
100
150
200
250
300
350

[2.3]

Provide a description of the appearance of the average images. Give an explanation as to why the average image does not look like the individual example images.

The average images are blurred and it looks like a mixed version of all the images. Because the RGB value of every pixel in the average image is calculated from summing all the RBG value of all the images in the file and then divided by the total number of images. So the RGB value of every pixel in the average image is differe from those in each single image in the file.

3. Color sensor demosaicing [40pts]

As discussed in class, there are several steps to transform raw sensor measurements into nice looking images. These steps include Demosaicing, White Balancing and Gamma Correction. In this problem we will implement the demosaicing step. (see Szeliski Chapter 2.3) In the assignment data directory on Canvas there is a zip file containing raw images from a Canon 20D camera as well as corresponding JPEG images from the camera (`*.JPG`). The raw image files (`*.CR2`) have been converted to 16-bit PGM images (`*.pgm`) using David Coffin's dcraw program to make it easy to load them in as arrays using the supplied code below **read_pgm**

image.png

Bayer RGGB mosaic.

The raw image has just one value per pixel. The sensor is covered with a filter array that modifies the sensitivity curve of each pixel. There are three types of filters: "red", "green", and "blue", arranged in the following pattern repeated from the top left corner:

```
R G . . .
G B
.
.
.
```

Your job is to compute the missing color values at each pixel to produce a full RGB image (3 values at each pixel location). For example, for each "green" pixel, you need to compute "blue" and "red" values. Do this by interpolating values from adjacent pixels using the linear interpolation scheme we described in class.

In [12]:

```
#
# these are the only modules needed for problem #3
#
import numpy as np
import matplotlib.pyplot as plt

#
# this function will load in the raw mosaiced data stored in the pgm file
#
def read_pgm(filename):
    """
    Return image data from a raw PGM file as a numpy array
    Format specification: http://netpbm.sourceforge.net/doc/pgm.html

    """
    infile = open(filename,'r',encoding="ISO-8859-1")

    # read in header
    magic = infile.readline()
    width,height = [int(item) for item in infile.readline().split()]
    maxval = infile.readline()

    # read in image data and reshape to 2D array, convert 16bit to float
    image = np.fromfile(infile, dtype='>u2').reshape((height, width))
    image = image.astype(float)/65535.
    return image
```

[3.1]

Implement a function `demosaic` which takes an array representing the raw image and returns a standard color image. To receive full credit, you should implement this using NumPy indexing operations like you practiced in the first part of the assignment. You should not need any for loops over individual pixel locations. You can accomplish this by either using array subindexing or alternately by using the imfilter function with the appropriate choice of filter.

In [13]:

```
def demosaic(I):
    """
    Demosaic a Bayer RG/GB image to an RGB image.

    Parameters
    ----------
    I : numpy.array  (dtype=float)
        RG/GB mosaic image of size  HxW

    Returns
    -------
    numpy.array (dtype=float)
    HxWx3 array containing the demosaiced RGB image

    """

    #[enter your code here]

    red = np.zeros(I.shape)
    red[::2, ::2] = I[::2, ::2]

    blue = np.zeros(I.shape)
    blue[1::2, 1::2] = I[1::2, 1::2]

    green = np.zeros(I.shape)
    green[::2, 1::2] = I[::2, 1::2]
    green[1::2, ::2] = I[1::2, ::2]

    red[::2, 1:-1:2] = (red[::2, :-2:2] + red[::2, 2::2])/2 #row 0,2,4 column 1,
3,5
    red[1:-1:2, ::2] = (red[:-2:2, ::2] + red[2::2, ::2])/2 #row 1,3,5 column 2,
4,6
    red[1:-1:2, 1:-1:2] = (red[:-2:2, :-2:2] + red[:-2:2, 2::2] +
    red[2::2, :-2:2] + red[2::2, 2::2])/4
    red[:, -1] = red[:, -2] #rightmost column
    red[-1, :] = red[-2, :] #bottom row

    blue[2::2, 1::2] = (blue[1:-2:2, 1::2] + blue[3::2, 1::2])/2 #row 3,5,7 colu
mn 1,3,5
    blue[1::2, 2::2] = (blue[1::2, 1:-2:2] + blue[1::2, 3::2])/2 #row 2,4,6 colu
mn 3,5,7
    blue[2::2, 2::2] = (blue[1:-2:2, 1:-2:2] + blue[1:-2:2, 3::2] +
    blue[3::2, 1:-2:2] + blue[3::2, 3::2])/4
    blue[0, :] = blue[1, :] # top row
    blue[:, 0] = blue[:, 1] # leftmost column

    green[1:-2:2, 1:-2:2] = (green[0:-2:2, 1:-2:2] + green[2::2, 1:-2:2] +
    green[1:-1:2, 0:-3:2] + green[1:-1:2, 2:-1:2])/4  #cross
    green[2::2, 2::2] = (green[1:-2:2, 2::2] + green[3::2, 2::2] +
    green[2::2, 1:-2:2] + green[2::2, 3::2])/4 #cross

    green[0, 2::2] = (green[0, 1:-2:2] + green[0, 3::2])/2 #row 1
    green[0, 0] = green[0, 1] #topleft corner
    green[-1, 1:-2:2] = (green[-1, 0:-2:2] + green[-1, 2::2])/2 #last row
    green[-1, -1] = green[-1, -2] #bottomright corner
    green[2::2, 0] = (green[1:-2:2, 0] + green[3::2, 0])/2 #first column
    green[1:-2:2, -1] = (green[0:-2:2, -1] + green[2::2, -1])/2 #last column
    result = np.stack((red, green, blue), axis = -1)
    return result
```

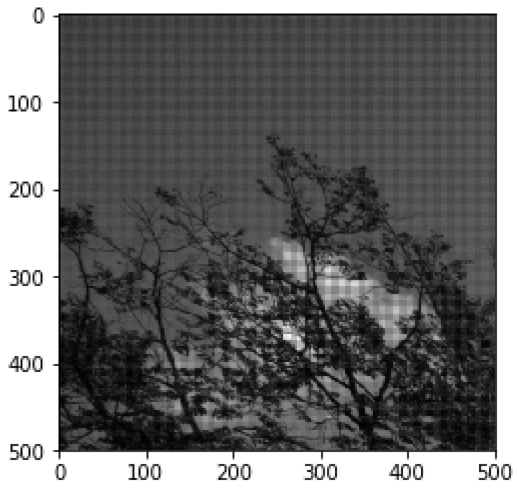
[3.2]

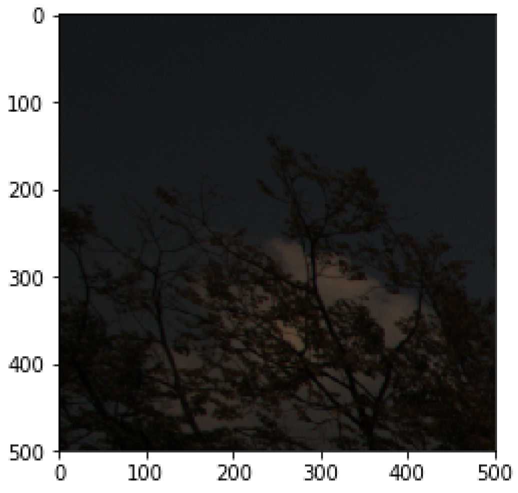
Write code and comments below that demonstrate the results of your demosaic function using `IMG_1308.pgm`. You are encouraged to include multiple examples for illustration. Since the images are so large, work with just the upper-left 500x500 pixel sub-block for illustrations.

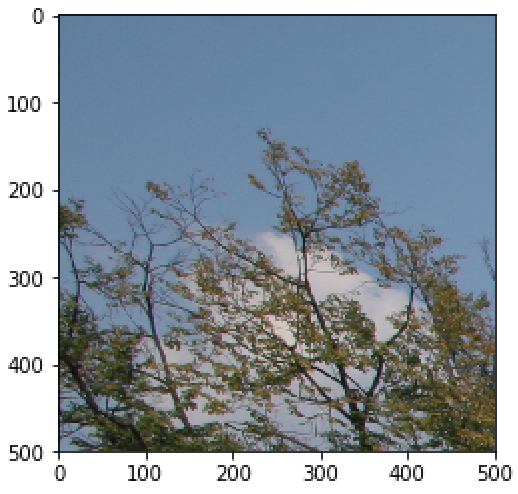
You should display: (a) the original raw image with a grayscale colormap, (b) the resulting RGB image after demosaicing, (c) the corresponding part of the provided JPG file from the camera

In [14]:

```
Iraw = read_pgm("demosaic/IMG_1308.pgm")
new_Iraw = Iraw[:500, :500]
plt.imshow(new_Iraw, cmap=plt.cm.gray)
plt.show()

RGB = demosaic(new_Iraw)
plt.imshow(RGB)
plt.show()

JPG = plt.imread("demosaic/IMG_1308.JPG")
new_JPG = JPG[:500, :500]
plt.imshow(new_JPG)
plt.show()
```







[3.3]

The correctly demosaiced image will appear darker than the JPG version provided. Provide an explanation of why this is the case based on your reading about the digital camera pipeline.

Because in order to get nice looking JPG images, we need more steps which include White Balancing, Gamma Correction and so on. Now we only implement the first step.