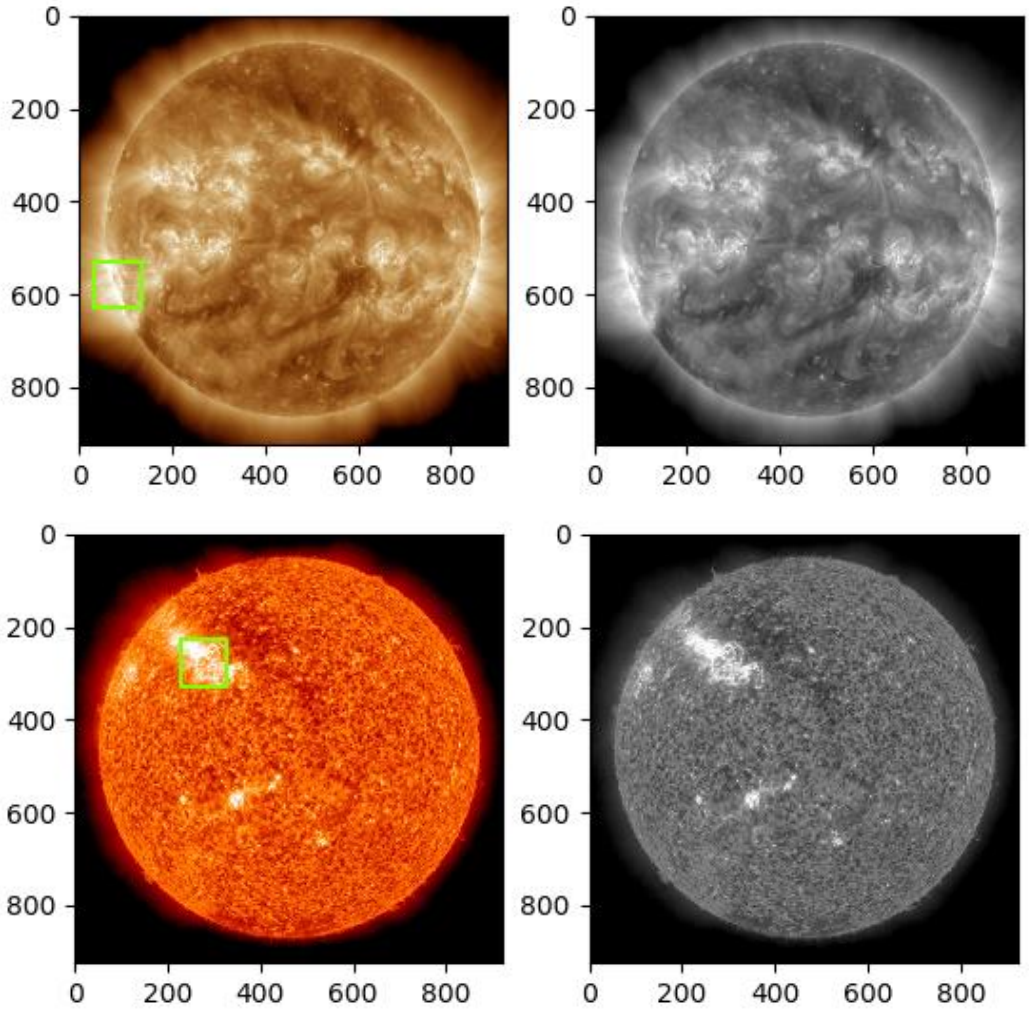
Lab II

Brightest Regions in the Sun



Course: CS 2302

Section: 12:00 p.m. – 1:20 p.m.

Author: Oswaldo Escobedo

Instructor: Dr. Fuentes

TA: Harshavardhini Bagavathyraj

Introduction

The objective of this laboratory is to create a program that can detect and plot a square in the brightest region in the image of a sun. To create the code, we will use python matplotlib and numpy, as well we will manipulate arrays by slicing, use integral images, explore different algorithms, and know more about the use of time complexity, etc. We will use and identify the brightest regions of the following 9 images:

Image 1

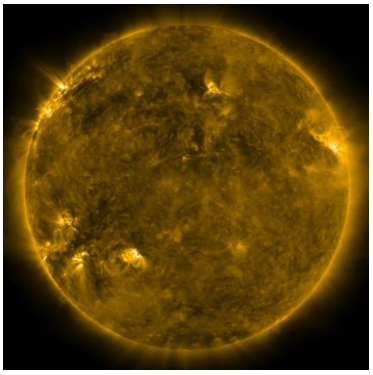


Image 2

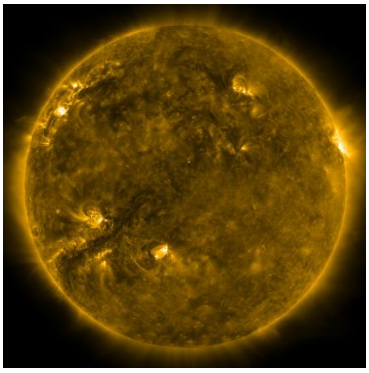


Image 3

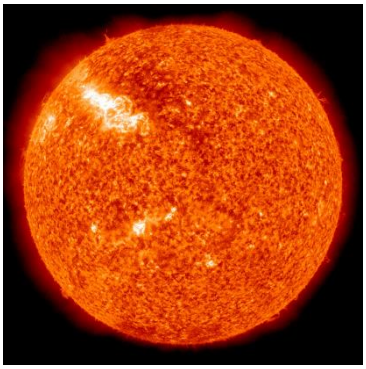


Image 4

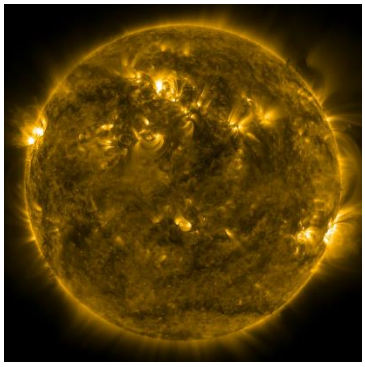


Image 5

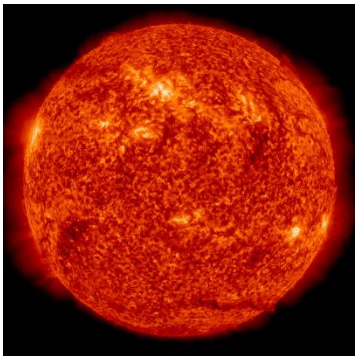


Image 6

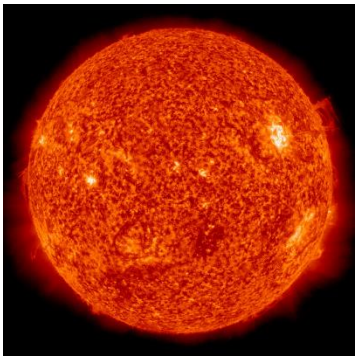


Image 7

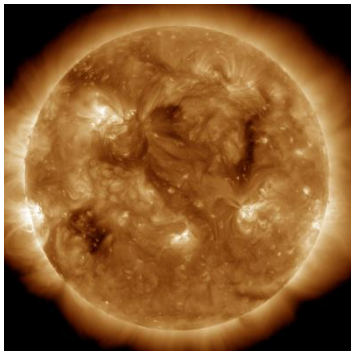


Image 8

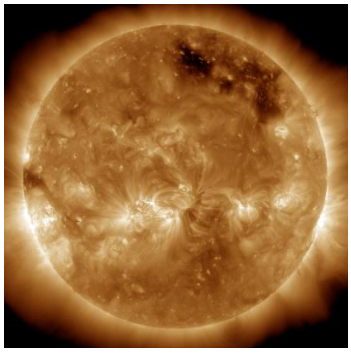
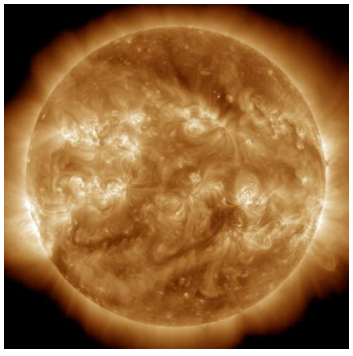


Image 9



Proposed Solution Design and Implementation

The first thing I did was understand the piece of code that Dr. Fuentes gave us. The only method he gave us was read_image (), whose function is to read and return two images, one in color scale and one in grayscale. In addition, in this method the transparent channel of the color image is removed and the color image is converted to grayscale.

In addition to the method, the teacher also gave us a for loop, which has the function of accessing and displaying each image on a Cartesian plane. Before starting the laboratory, I created another file with the same methods, there I dedicated myself to creating the methods and debug code.

```
  File "C:/Users/oesco/Downloads/read_and_show_images.py", line 46,
in <module>
    img_files = os.listdir(img_dir)  # List of files in directory

FileNotFoundError: [WinError 3] The system cannot find the path
specified: '.\\solar images\\'
```

One of the first problems I encountered was that this error appeared to me:

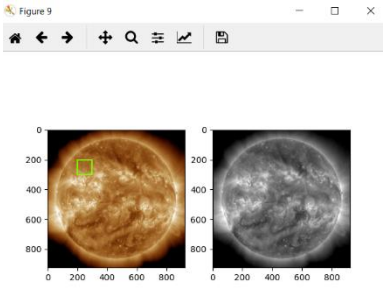
To solve the problem I asked a friend, she told me that this error was due to the folder ". \\ solar images \\" and the Python file "read_and_show_images ()" were not in the same folder. Therefore, I decided to move both files to this folder:

After fixing the error, I executed the program to see what the output was, having done this the images and the name of the file were shown.

The first task I did was to create a method that would plot a figure of size m by n in the brightest region in the image. When I finished the method I found an error, since every time I executed the code, the figure and the image came out separately, so I investigated how to solve the error. When I finished doing the research, I found useful information, but none of them helped me a lot, so I proceeded to analyze the function of ax.plot(). My approach was the following, I decided to show the image first and then plot the figure, why? Simple, because by doing this the figure would overlap the image and therefore, it will plot on top of the image. And the exit was this, my approach worked as expected.

```
ax1.imshow(img)
ax1.plot(p[:,0],p[:,1],linewidth=1.5,color='chartreuse')
```

```
r = 200
c = 200
h = 100
w = 100
display_img_and_rectangle(ax1,img,r,c,h,w)
plt.show()
```



Then, I decided to do the find_brightest_pixel () method, which will serve as an introduction and help us understand the lab's task. The task is simple, the brightest pixel must be found and we are going to use the grayscale image because it will be much simpler to use this one than the color one.

```
In [10]: runfile('C:/Users/oesco/Desktop/
SPRING 2020/read_and_show_images.py',
wdir='C:/Users/oesco/Desktop/SPRING 2020')
20110101_000001_1024_0171_c.png
(145, 207)
20110102_000001_1024_0171_c.png
(856, 358)
20110301_002957_1024_0304_c.png
(228, 227)
20120218_001237_1024_0171_c.png
(398, 207)
20120218_001345_1024_0304_c.png
(415, 218)
20120817_001444_1024_0304_c.png
(728, 359)
20130301_000019_1024_0193_c.png
(262, 282)
20131225_001431_1024_0193_c.png
(64, 406)
20140404_001019_1024_0193_c.png
(61, 532)
```

Note: Keep in mind that in a grayscale image the value of 0 in a pixel means that it is completely black and a value of 255 is a completely white pixel. To implement the code, it must be taken into account that each pixel

of the image will be read, so two for loops will be used. After analyzing the task of this method I realized that the algorithm that we are going to use is exactly the same as when we want to get the maximum or minimum element from an array or list, so I decided to use that algorithm. After executing the algorithm this is what the output:

After having done the main thing and understanding the concept of what we had to do, I decided to start the third method called find_brightest_region(), which has the task of returning the coordinates (row,col or x,y) of the top left corner of the brightest region of size m by n of the solar images. In this method we will use the grayscale image, since it only has a single channel and its pixel intensities or range is from 0 (black / darker) to 255 (white / brighter). In addition, this method will have four versions, one more effective and faster than the previous one.

It should be noted that the following are region sum algorithms:

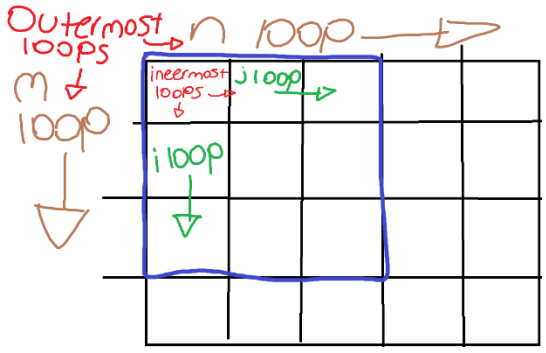
1.1 Naive Algorithm

This algorithm consists of adding each and every one of the pixel values of each region. In which m will represent the rows, n the columns, h the height of the region and w the width of the region. As the laboratory pdf mentions there will be (m-h+1) x (n-w+1) regions in total, and adding the elements of each region will take wh operations. Due to the concept of this algorithm it will be one of the slowest to process the image and, therefore, its time complexity will be O(mnwh).

Version 1.1 (Slowest version)

In this first version of the naive algorithm, all pixel values of each region in the image will be added and will have 4 levels: rows, columns, height and width. My approaches to solve this problem were the following:

1. To solve this method, we need to use an algorithm similar to the one we use to obtain the maximum value of an array/list. This is because the purpose of this method is to identify and obtain the brightest region of the image.
2. In addition, we will also use the algorithm used to add all the elements of an array / list. This is because we want to obtain the sum of all the elements of the current region.
3. The two inner loops will have the function of visiting and adding all the elements of each region. Also, this is where the maximum value (or brightest region) and the coordinates of the brightest region will be updated.
4. The two outermost loops will have the function of moving the region in the image. Therefore, the first for loop will move the region down, while the second for loop will move the region to the right.



The image on the right is the representation of my approximation of point number 3 and 4.

Version 1.2

In this second version of the naive algorithm the two innermost for loops will be removed and replaced with the built in funcion np.sum() and we will use array slicing. So now the algorithm of the second version will change a lot. Although, because I am a beginner in python, I decided to investigate more about how slicing is works and I also did research the use of the np.sum () method. After finishing searching the purpose of these functions, I found that:

np.sum() has the function to sum all the elements of a 1D, 2D and even a 3D array.

Sicing is used to get a subarray (of n dimensions) from an array (of n dimensions).

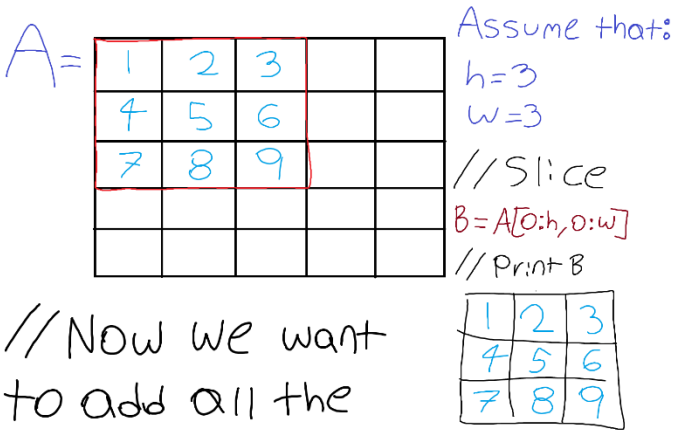
Having done this, I realized a very important pattern that would help me solve this method. After analyzing what I had to do, I reached the following conclusions:

1. The two for loops will remain the same as the version 1.1.
2. The algorithm for finding and updating the maximum (or brightest) region will remain the same as in version 1.1.
3. When I researched, I discovered that with slicing notation we can create a sub-2d array from a 2d array. To create a sub 2d array, we have to take into account the following syntax:

   A[start row : end row, start column : end column]

   If we apply this in the method we will create a sub-2d array or, in other words, we will create a region of size m by n.

4. After finishing our region, we will proceed to make this the parameter of the np.sum () method so that this method helps us to add all the elements of the region.

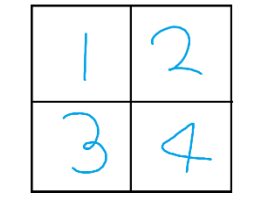


By making these approximations we will be able to replace the function of the two for loops that we must remove. This would decrease the time it takes to process the images, although the complexity time would remain the same.

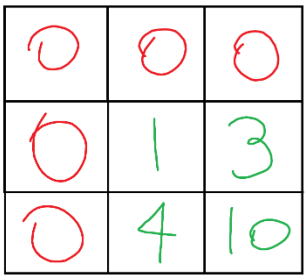
1.2 The Summed Area Table (Integral Image) Algorithm

This algorithm consists on using an auxiliary array to speed up the process. To do this we must make our grayscale image become an integral image, which uses the principles of cumulative sum. In this integral image a row and column of zeros will be added, which will make the array increase in size. In addition, when calculating the sum of the region in which we are, only 4 elements of the integral image will be accessed. Because of this we will no longer have a complexity time of O (wh), but of O (1), since we are accessing directly and not iterating through each element. The formula that will be used to compute the sum of a region will be the following:

$$R = A - B - C + D$$



Original image



Integral image

Version 2.1

In this version the algorithm mentioned above will be used, so we will then create a method called integral_images (), which will have the function of creating the integral image, while the method of version 2.1 will be dedicated to doing the same as in 1.2 , only now instead of using slicing notation and the np.sum () method, four elements of the integral image should be accessed to compute the sum of the region and we will use the formula R = A - B - C +D.

However, despite the explanations of this algorithm I got confused, since I still didn't understand the concept, but I started to investigate how to create the integral image. After, doing this I arrived to these conclussions:

1. We must use and apply the function of img.cumsum () on axis 0 and 1, which will help us create the integral image.
2. To add a row and a column of zeros to an array, we must use the np.insert () method, which will help us add an additional row and column. In addition, in one of its parameters allows us to choose what value we want to be present in the new row and column.
3. The two for loops will remain the same as the previous version. This because we need to visit every region in the image.
4. As already mentioned, the only thing that will change will be the operations to compute the sum of a region. So instead of adding all the elements, now we are going acces four accesses of the current region.

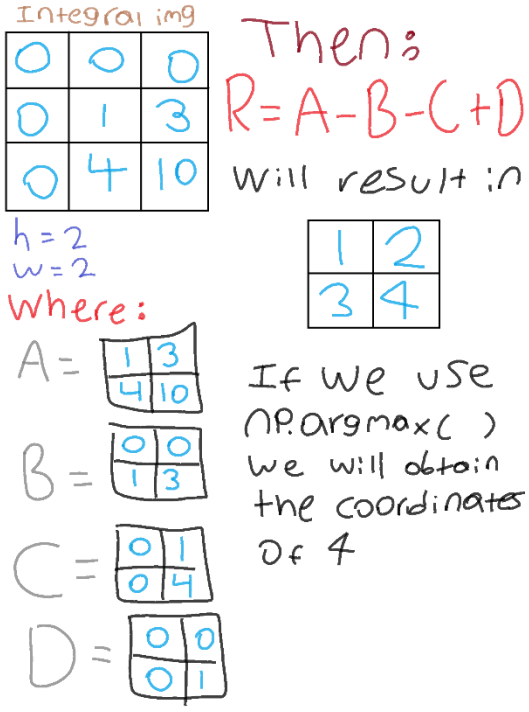
   Those four array accesses are:

   A = Bottom right corner
   B = Upper right corner
   C = Bottom left corner
   D = Upper left corner

R = A - B - C +D, where R represents the sum of the region.

Version 2.2 (faster version)

In this method, we are asked to eliminate the two for loops we have. So the task will now be to replace the function of these loops by using slicing notation and four arrays that are obtained from the integral image will be used. I must say that this method was the most complicated of all, since I didn't understand how to solve it. To clarify my doubts I went with some friends to ask them if they could explain version 2.2 to me, but they were also having problems with that method, then I went with Dr. Fuentes and I got even more confused, after going with the TA's they helped me a lot but still I still had a lot of confusion, it wasn't until after a meeting that I had with some classmates that we solve that method, in which we came to a different approach than Dr. Fuentes. The approach we reached is closely related to the way we solved version 2.1, only now it would be large scale.



The approach was the following:

1. First, the integral image will be obtained, and then divided into four equal parts, each one will be given its respective variable name:

   A = Bottom right corner array

   B = Upper right corner array

   C = Bottom left corner array

   D = Upper left corner array

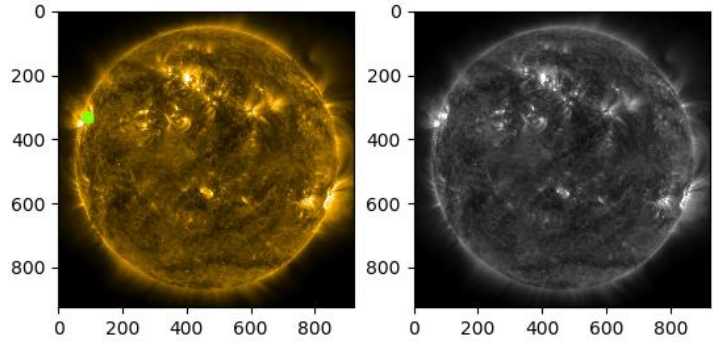
   R = A – B – C + D, where R will represent the sum of the entire image.

2. After doing this operation, the method np.argmax () will be used, which has the function to return the location of the maximum number in an array. Next, we will use the modulist operator and integer division to obtain the index of the row and column where the maximum value is in the array.

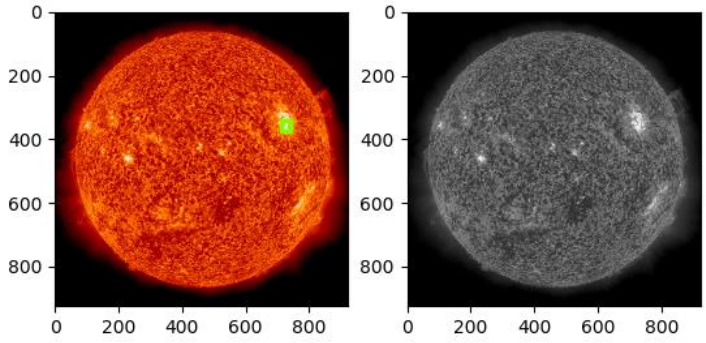
Experimental Results

First of all we will start showing what were the outputs using the different dimensions that we were asked for:

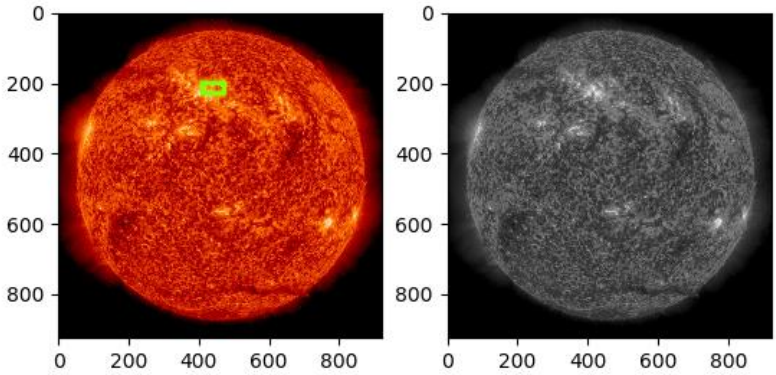
20 x 20 dimension



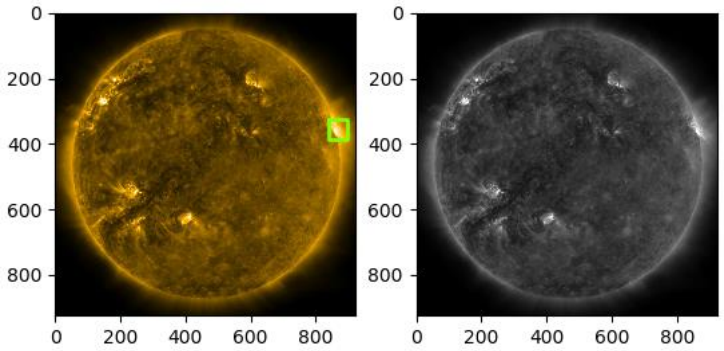
30 x 30 dimension



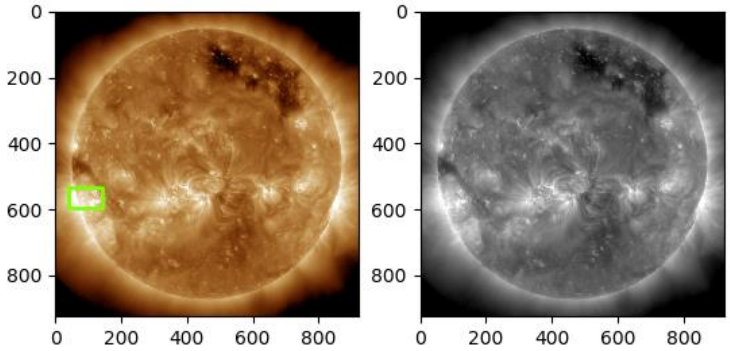
30 x 60 dimension



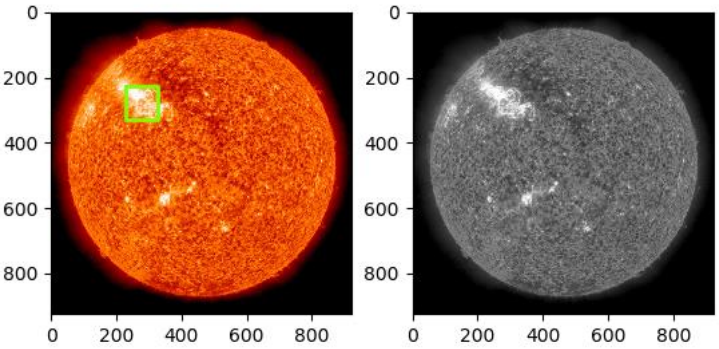
60 x 60 dimension



60 x 100 dimension



100 x 100 dimension



After analyzing the images and their regions I came to the conclusion that my methods were working, but I found the following patterns:

1. The brightest regions will depend on the size of the region.

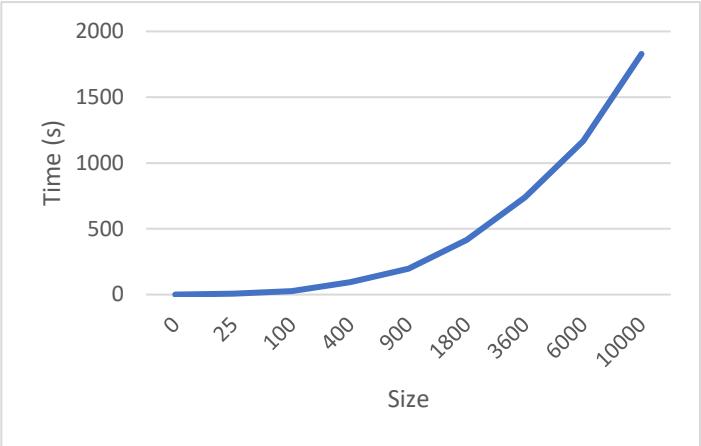
2. If there are two equally bright regions in an image, the method will only identify the first.

Therefore, we will measure the running time and time complexity for each version of the find_brightest_pixel() method we need to use the python time library. In which, we are going to use the time.time(), which will return the time in seconds. And then we will use the input sizes and time to make a graphical representation of the time complexity of the versions, this will help us understand and analyze the behavior of the method.

```
start = t.time()
brightest_region_v11(ax1,img,img_gl,h,w)
end = t.time()
print(end-start)
```

Version 1.1

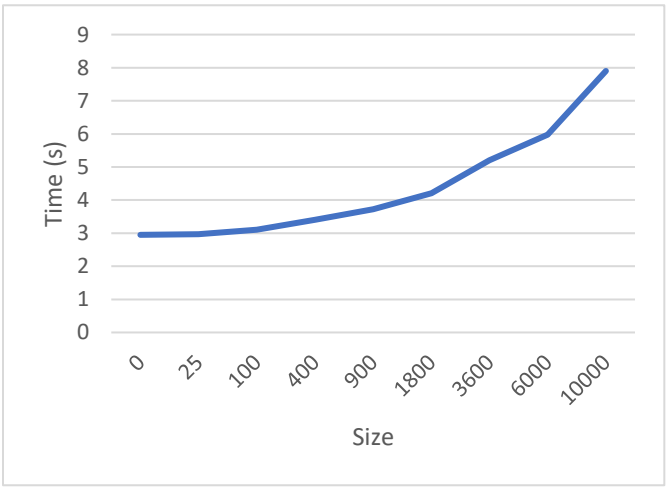
Input Size	Time
20x20	94.007 s
30x30	197.788 s
30x60	415.767 s
60x60	737.98 s
60x100	1167.087 s
100x100	1828.77 s



As you can see, this version was the slowest. Therefore, it caused that in order to obtain a region of 100x100 we needed to wait approximately 30 minutes per image.

Version 1.2

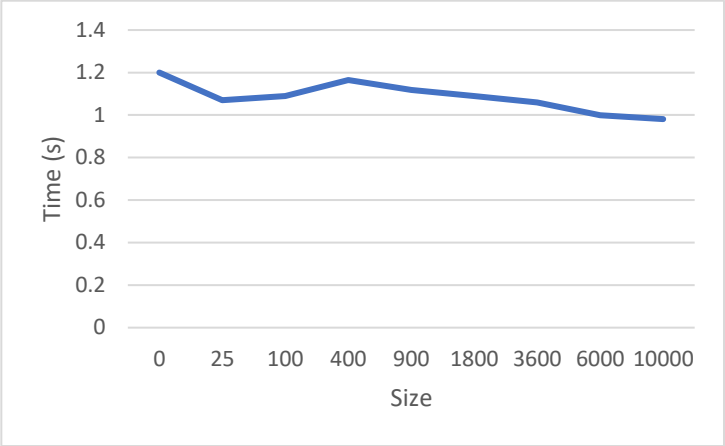
Input Size	Time
20x20	3.40 s
30x30	3.72 s
30x60	4.21 s
60x60	5.20 s
60x100	5.97 s
100x100	7.901 s



The second version was much faster than its predecessor and that makes it stand out too much, since even despite continuing to use two for loops, it manages to complete the task of obtaining the brightest region of 100 by 100 in less than 8 seconds.

Version 2.1

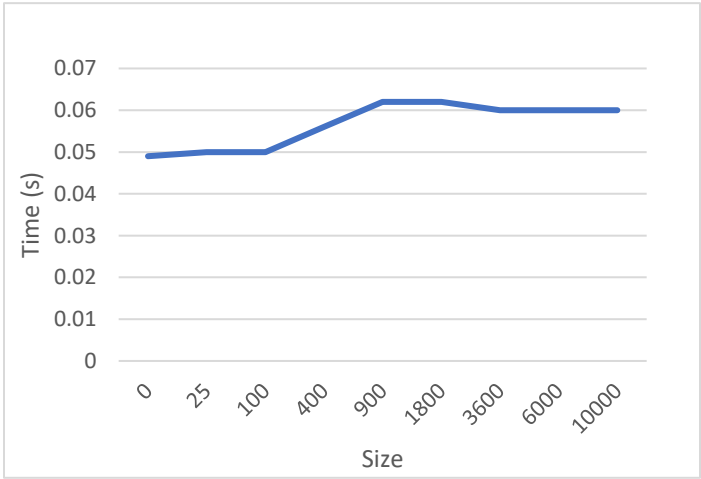
Input Size	Time
20x20	1.165 s
30x30	1.118 s
30x60	1.09 s
60x60	1.06 s
60x100	0.999 s
100x100	0.981 s



For the third version, we obtained a slightly peculiar graph, since if you observe, the behavior of the function remains almost static, because the function does not grow or decrease. This may be because this function is much more effective than the previous one, since the method will complete the task at the same time, regardless of size. For example, the time it takes for the method to calculate the brightest region of 10x10 will be the same as if we were to calculate a region of 100x100.

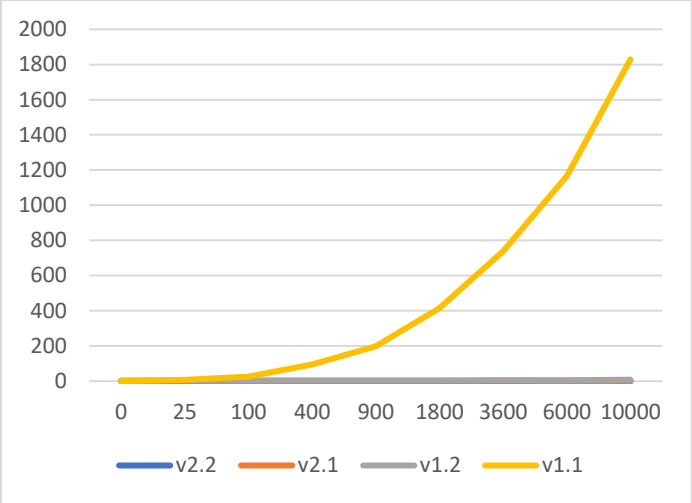
Version 2.2

Input Size	Time
20x20	0.056 s
30x30	0.062 s
30x60	0.062 s
60x60	0.06 s
60x100	0.06 s
100x100	0.06 s



As for the final and most optimal version, there is not much to talk about, as it would be the same behavior as the last version, only that the task would be completed ten times faster than the previous one.

And finally, I decided to create a graph in which were the behavior of the four versions and thus be able to compare which is the least effective.



And as you can see in this graph, the version that has less effectiveness is 1.1, why? If you analyze well you will see that this function will grow very fast, while 1.2 is just starting to grow but it is taking time, which means that this version is much faster than its predecessor. In regards to the functions of 2.1 and 2.2, they cannot be observed, since their performance is so effective that they remain in a straight line regardless of the input size.

Conclusions

In conclusion, this project helped me to really understand why it is important that we optimize our code as much as we can. Because as we saw in the results of the experiment of the four versions, we saw that the worse the time complexity an algorithm has, the longer it would take to complete a task. For example, to process the image in version 1.1 (O(mnwh)) we had to wait aprox. 5 minutes for an input of 20x20, while version 2.2 (O(mw)), which is more effective, we had to wait a couple of seconds, for the image to be processed. Not only that, but we also learned more about image processing, how you can plot over an image, I learned that one of the most effective algorithms for obtaining the sum of a region in an image is to convert it to an integral image, I could also practice the slicing notation and much more. Without a doubt, this has been one of the funniest projects I've done, but at the same time it has been one of the most stressful and challenging ones due to the immense amount of time I had to wait to get the output of version 1.1 and also version 2.2 gave me many problems since I am used to solving these problems using for loops.

Appendix

```
#   Course: CS 2302
#   Assignment: Lab II
#   Author: Oswaldo Escobedo
#   Instructor: Dr. Fuentes
#   TA: Harshavardhini Bagavathyraj
#   Date of Last Modification: 02/14/2020
#   Purpose of the Program: to identify the
#   brightest region in a solar image.

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

def read_image(imagefile):
    # Reads image in imagefile and returns color and gray-level images
    #
    img = (plt.imread(img_dir+file)*255).astype(int)
    img = img[:,:,:3]  # Remove transparency channel
    img_gl = np.mean(img,axis=2).astype(int)
    return img, img_gl

def integral_image(img):
    S = img.cumsum(axis=0).cumsum(axis=1)   # Creates an integral image.
    S = np.insert(S,0,0,axis=0)   # Adds a row of zeros into the array.
    S = np.insert(S,0,0,axis=1)   # Adds a column of zeros into the array.
    return S

def integral_regions(S,h,w):
```

```
    m = len(S)
    n = len(S[0])
    A = S[:m-h,:n-w]   # First array (region) of size m by n
    B = S[:m-h,w:]    # Second array
    C = S[h:,:n-w]   # Third array
    D = S[h:,w:]   # Fourth array
    R = A-B-C+D   # We compute the sum of the entire image by accesing four arrays.
    return R

def display_img_and_rectangle(ax1,img,r,c,h,w):

    p0 = [r,c]   # Upper left corner point.
    p1 = [r,c+h]   # Upper right corner point.
    p2 = [r+w,c+h]   # Bottom right corner point.
    p3 = [r+w,c]   # Bottom left corner point.
    p = np.array([p0,p1,p2,p3,p0])   # Creates the given shape of size m by n.
    ax1.imshow(img)
    ax1.plot(p[:,0],p[:,1],linewidth=1.5,color='chartreuse')

def find_brightest_pixel(I):
    max_pix = 0
    for m in range(len(I)):
        for n in range(len(I)):
            if I[m][n] > max_pix:
                max_pix = I[m][n]
                r = n
                c = m
    return r,c

def brightest_region_v11(ax,orig_img,gray_img,h,w):
```

```
    r = 0
    c = 0
    max_region = 0
    for m in range(0,len(gray_img)-h+1):   # for loop m moves the region vertically.
        for n in range(0,len(gray_img)-w+1):   # for loop n moves the region horizontally.
            sum_region = 0
            for i in range(m,h+m):   # for loop i and j visits every element in
                for j in range(n,w+n):   # the region of size h by w.
                    sum_region += gray_img[i][j]
            if sum_region > max_region:
                max_region = sum_region
                r = n
                c = m
    display_img_and_rectangle(ax,orig_img,r,c,h,w)

def brightest_region_v12(ax,orig_img,gray_img,h,w):
    r = 0
    c = 0
    max_region = 0
    for m in range(0,len(gray_img)-h+1):
        for n in range(0,len(gray_img)-w+1):
            sum_region = np.sum(gray_img[m:m+h,n:n+w])   # Creates a 2D array (region),
            if sum_region > max_region:           # using slicing, of size h by w.
                max_region = sum_region
                r = n
                c = m
    display_img_and_rectangle(ax,orig_img,r,c,h,w)

def brightest_region_v21(ax,orig_img,gray_img,h,w):
    S = integral_image(gray_img)
```

```
    r = 0
    c = 0
    max_region = 0
    for m in range(0,len(S)-w):
        for n in range(0,len(S[0])-h):
            sum_region = S[m+w][n+h]-S[m][n+h]-S[m+w][n]+S[m][n]   # We compute the sum of the
            if sum_region > max_region:                # current region by accesing four elements.
                max_region = sum_region
                r = n
                c = m
    display_img_and_rectangle(ax,orig_img,r,c,h,w)

def brightest_region_v22(ax,orig_img,gray_img,h,w):
    S = integral_image(gray_img)
    R = integral_regions(S,h,w)
    indices = np.argmax(R)   # We locate the position of the max element in R.
    r = indices%len(R)   # Operation to know the row index position of the max element.
    c = indices//len(R)   # Operation to know the column index position of the max element.
    display_img_and_rectangle(ax,orig_img,r,c,h,w)

img_dir = '.\\solar images\\' # Directory where imagea are stored

img_files = os.listdir(img_dir)  # List of files in directory

if __name__ == "__main__":

    plt.close('all')

    for file in img_files:
        print(file)
```

```
if file[-4:] == '.png': # File contains an image
    img, img_gl = read_image(img_dir+file)
    fig, (ax1, ax2) = plt.subplots(nrows=1, ncols=2)

    ax1.imshow(img)                 #Display color image
    ax2.imshow(img_gl,cmap='gray')   #Display gray-leval image

    plt.show()

    h = 100
    w = 100

    '''
    Uncomment to test each version of the method brightest_region()
    '''

    #print(find_brightest_pixel(img_gl))
    #brightest_region_v11(ax1,img,img_gl,h,w)
    #brightest_region_v12(ax1,img,img_gl,h,w)
    #brightest_region_v21(ax1,img,img_gl,h,w)
    brightest_region_v22(ax1,img,img_gl,h,w)
```