CS2302 - Data Structures

Spring 2020

Lab 2

Due Friday, February 14, 2020

In solar astronomy, it is important to detect and monitor the brightest regions in the Sun because they are related to important phenomena that can affect life on Earth. For this lab you will implement several algorithms to find these regions.

In computer vision, a grayscale image with m rows and n columns is usually represented by an m by n array of integers I, where $I[r,c]$ is the intensity of the pixel in row r and columns c. Normally, we use 0 to signal a black pixel and 255 to signal a white one. Similarly, a color image is represented by an m by n by 3 array of integers I, where $I[:,:,0], I[:,:,1]$, and $I[:,:,2]$ represent the red, green, and blue channels in the image, respectively. The figure shows a solar image in color and gray level.

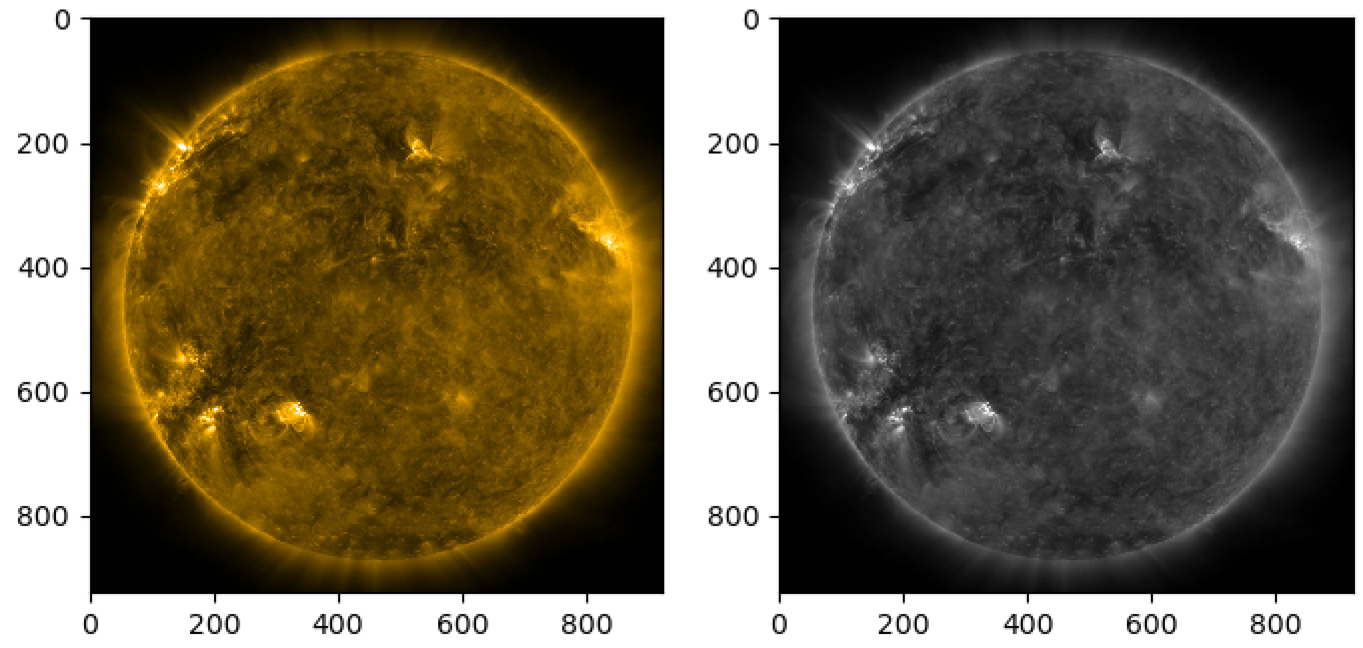


Figure 1: Solar image in the ultraviolet wavelength, displayed in false color and black and white.

Your task for this lab is the following:

1. Write a function that receives an image I and integers (r,c,h,w) and displays I and a rectangle with upper-left corner(r,c), height h and width w. Hint: this can be done using the same instruction we used for plotting in lab 1. This function will be used by your other functions. Notice that the first index in an array represents the row, while the first coordinate in a point represents the x value, or column.

2. Write a function that finds the brightest pixel in a solar image. and displays a rectangle around it.

3. Write a function that receives a gray level image and integers h and w and returns the coordinates of the top-left corner of the brightest region in the image that has size h by w (that is, the region in the image where the pixels have the largest sum). You will implement four different versions of this functions, as described below. Also, display a rectangle around the region using your function from the first item.

1 Region Sum Algorithms and Implementations

1.1 Naive Algorithm

This algorithm consists of simply adding all the pixel values for every region. Let m be the number of rows, n the number of columns, h the height of the region and w the width of the region. There are $(m-h+1)\times(n-w+1)$ regions to consider, and adding all the elements in a single region takes wh operations, so the total running time is $O((m-h+1)(n-w+1)\,w\,h)$, which is approximately equal to $O(mnwh)$.

Version 1.1

Use for loops to add the elements in every region. To do this you'll need 4 nesting levels (rows, columns, height, width).

Version 1.2

Eliminate the two innermost loops by using the built-in sum function in numpy and using array slicing. This function will still run in $O(mnwh)$, but it should be much faster that the previous one.

1.2 The Summed Area Table (Integral Image) Algorithm

This algorithm uses an auxiliary array to speed-up the computation. Given an image I of size m by n, we compute its *integral image* S, of size $(m+1)$ by $(n+1)$. $S[r,c]$ is the sum of all the pixels above and to the left of $I[r-1,c-1]$. Formally:

$$S[r,c] = \sum_{r'=0}^{r-1}\sum_{c=0}^{c-1} I[r',c']$$

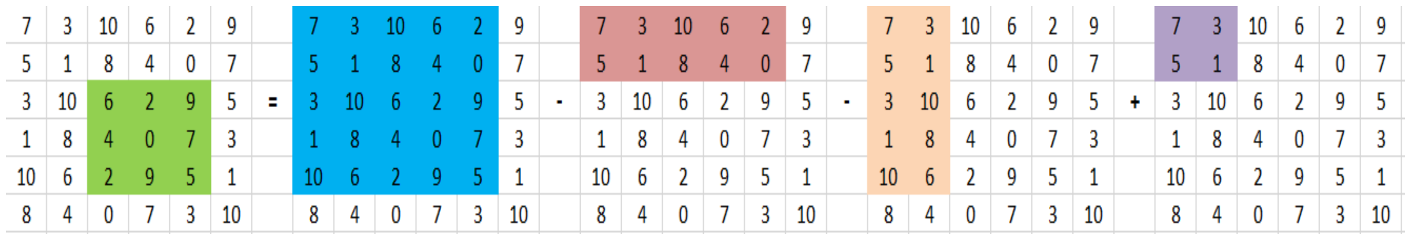
For example, suppose I is the following array:

```
[[ 7  3 10  6  2  9]
 [ 5  1  8  4  0  7]
 [ 3 10  6  2  9  5]
 [ 1  8  4  0  7  3]
 [10  6  2  9  5  1]
 [ 8  4  0  7  3 10]]
```

Then S is given by:

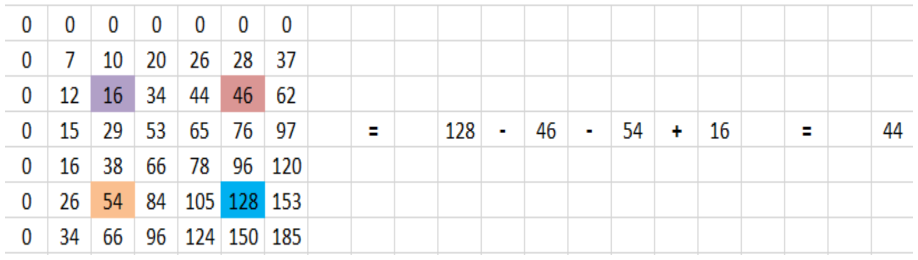
```
[[  0   0   0   0   0   0   0]
 [  0   7  10  20  26  28  37]
 [  0  12  16  34  44  46  62]
 [  0  15  29  53  65  76  97]
 [  0  16  38  66  78  96 120]
 [  0  26  54  84 105 128 153]
 [  0  34  66  96 124 150 185]]
```

Then, for any given region in the original image I, we can compute its sum in $O(1)$ rather than $O(wh)$ time by accessing four elements in the integral images S. Then the running time for finding the sums of all regions is $O(mn)$. The following example shows how we can compute the sum of the pixels in a region of interest (colored green) in constant time, assuming the integral image has already been computed.



7	3	10	6	2	9		7	3	10	6	2	9		7	3	10	6	2	9		7	3	10	6	2	9		7	3	10	6	2	9
5	1	8	4	0	7		5	1	8	4	0	7		5	1	8	4	0	7		5	1	8	4	0	7		5	1	8	4	0	7
3	10	6	2	9	5	=	3	10	6	2	9	5	-	3	10	6	2	9	5	-	3	10	6	2	9	5	+	3	10	6	2	9	5
1	8	4	0	7	3		1	8	4	0	7	3		1	8	4	0	7	3		1	8	4	0	7	3		1	8	4	0	7	3
10	6	2	9	5	1		10	6	2	9	5	1		10	6	2	9	5	1		10	6	2	9	5	1		10	6	2	9	5	1
8	4	0	7	3	10		8	4	0	7	3	10		8	4	0	7	3	10		8	4	0	7	3	10		8	4	0	7	3	10

Using the integral image, no summations are required, just four array accesses:



0	0	0	0	0	0	0													
0	7	10	20	26	28	37													
0	12	16	34	44	46	62													
0	15	29	53	65	76	97	=		128	-	46	-	54	+	16		=		44
0	16	38	66	78	96	120													
0	26	54	84	105	128	153													
0	34	66	96	124	150	185													

Version 2.1

Use for loops to compute the sum of all rectangular regions, but each sum should be computed by four accesses to the integral image.

Version 2.2

Eliminate the for loops by performing the operations using four arrays obtained from the integral image by slicing. This should be the fastest implementation. Feel free to ask for help if you find this difficult.

2 Submission

As usual, write a report describing your work. Compute and report the running times of each of the four implementations using regions of the following sizes:

1. 20×20
2. 30×30
3. 30×60
4. 60×60
5. 60×100
6. 100×100

Use plots and/or tables to illustrate your results. Do not take into consideration the time required to read the image files or to plot the results to the screen. As starting point, use the program `read_and_show_images.py`, provided in the class webpage. Use the solar images in `lab2imgs.7z` or `lab2imgs.zip`.