Assignment 1- Machine vision

Task 1: feature extraction algorithm inspired by sift

A. In this task we are supposed to load the image (Assignment_MV_01_image_1.jpg) provided in the desktop and convert the image into a gray scale (using Open CV (2 pts) and convert the type to float32 (1 pt) and resizing the image converting it to double the size of the current image (2 pts).

Please see the code provided in the assignment_1.py file

As we can see in the screenshot below the function named resized_Image is responsible for the task A

```
def main():
    """This is the main function that will run the entire code"""
    print("="*5+"done"+"="*5)

    #setting and assigning image locations
    img_1="./Assignment_MV_01_image_1.jpg" #my first image path
    img_2="./Assignment_MV_01_image_1.jpg" #my second image path

    #load the image using the single channel grey command when color =1 it load the color image instead
    resized_image=single_channel_grey(img_1)

    gaussian_kernels(resized_image)

    print("="*5+"done"+"="*5)

#calling the main
main()
```

Let's see in more detail how it does that,

The function itself take two arguments - The image that you want to perform the operation on and the colour value which is optional and by default it is set to 0 unless we want to apply the operations on a coloured image (for which set color=1).

Let's see how the code actually implements it

Inorder to convert loaded images to greyscale images we can use the function called cvtColor(image, cv2.COLOR_BGR2GRAY) short form for convert colour but it is comparatively slower to a much faster implementation used and Hence we can immediately convert the image to grey scale at the time of loading it when calling the imshow function specifying the color parameter as 0.

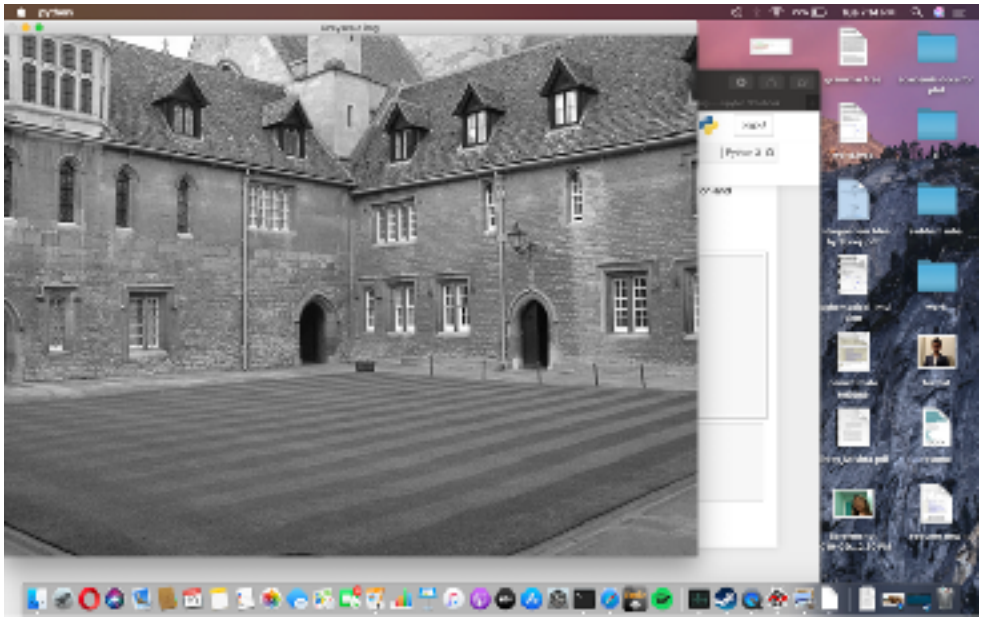
```
image = cv2.imread(img,color)
```

```
we can verify the image is greyscale by
looking at the shape of the image: (768,
1024)
Height of Image: 768 pixels
Width of Image:  1024 pixels
image dtype:  uint8
62
```

We can verify the output by printing out the image variable's shape.

As we can see in the console window we can view important information about the image such as the resolution of the image, the dtype of the image and the printing the very first pixel value of the image.

(since the image has only two channels now it is a grey scale image we can verify the fact by viewing the image using cv2.imshow)



Cv2 makes another window titled as greyscale img display the image that we have.

As we have noticed in the grey scale version of the image that it's dtype is uint 8 meaning that the numpy array which stores all the image values is filled with integers as is the case for the first pixel that has the value of 62.

We will now try to convert this array to a float32 format where the pixels are floating points (decimal values)

```
41    image = np.float32(image)
42    print('image dtype: ',image.dtype)
43    print(image[0][0])
```

The above function float32 does the task for us. Let's try to now look at the very first pixel value now.

```
===now converting the image's dtype to
float===
image dtype:  float32
62.0
```

The pixel value is now changed from 62 to 62.0

Let's now try to resize the image,

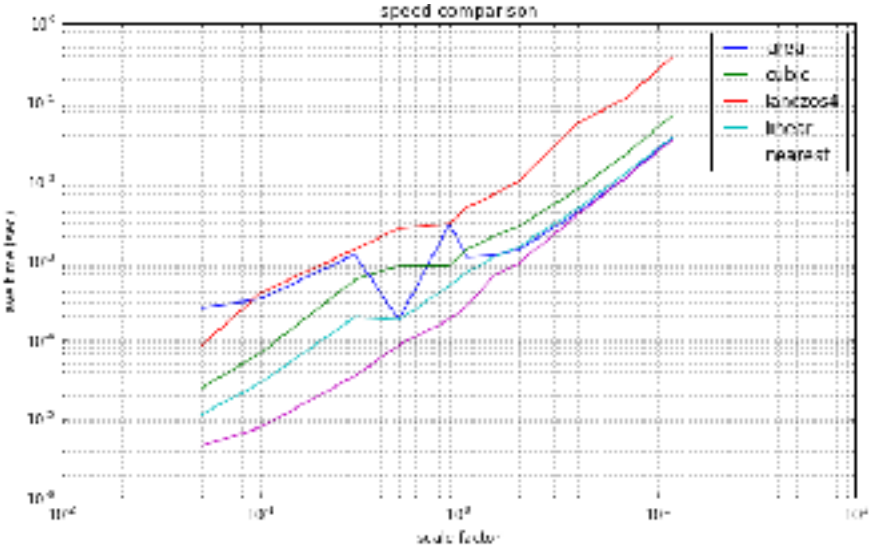
```
scale_factor=2 #this is the sceling factor by default it's 2 because we need to double the image size
dim=(int(original_width*scale_factor),int(original_height*scale_factor)) #dim is always width, height
#resizes the image to double it's size +2 point
#we have used interpolation ===> lanczos4 because it's theoritically the best
#refer- https://stackoverflow.com/questions/3112364/how-do-i-choose-an-image-interpolation-method-emgu-opencv
print("resizing the image.....")
resized_image = cv2.resize(image, dim, interpolation = cv2.INTER_LANCZOS4)
new_height=int(resized_image.shape[0])
new_width=int(resized_image.shape[1])
print('Height of Image:', new_height, 'pixels')
print('Width of Image: ', new_width, 'pixels')
```

As we can see that we need to specify the dimensions of the new resized image that we want our new image to look like which should be the exactly double of that what we are currently having, hence we want to scale our image by a factor of +2, so we set our scaling factor to be 2.

We can verify the new Image size by printing out the resized image's shape and we can see the sizes have been doubled now for each of the axis.

```
resizing the image.....
Height of Image: 1536 pixels
Width of Image:  2048 pixels
62.77438
```

The resizing function internally uses an interpolation method to oversample these images for our assignment we are using the LANCZOS4 technique which is better than most of the other sampling techniques in terms of quality but can take up some time. Depending upon the usage.



(Source- https://stackoverflow.com/questions/3112364/how-do-i-choose-an-image-interpolation-method-emgu-opencv)

But we can use any Interpolation technique available in opencv (please refer to the link provided).

Another interesting fact to note is that our first pixel has now changed to the value 62.77438 because of the interpolation method applied.

(Note- Cv2 imshow cannot view float32 values by default, hence we need to convert it back to the uint8 format to view it, or conversely use matplot lib's imshow function)

B. Create twelve Gaussian smoothing kernels with increasing σ = 2 and plot each of these kernels as image [4 points]. Make sure that the window size is large enough to sufficiently capture the characteristic of the Gaussian. Apply these kernels to the resized input image from subtask A to create a scale-space representation and display all resulting scale-space images [2 points].

We begin by creating a list of all possible sigma values by applying a map function on the range of K's varying from 0 to 11

```
sigma_vals=list(range(0,n))
sigma_vals = list(map(lambda x: 2**(x/2),sigma_vals))
filtered_images=[]
```

We then begin by iterating through all the sigma values and initialising a mesh grid with a fixed window size (in my case I used a window size of 6 that was good enough to fit the image).

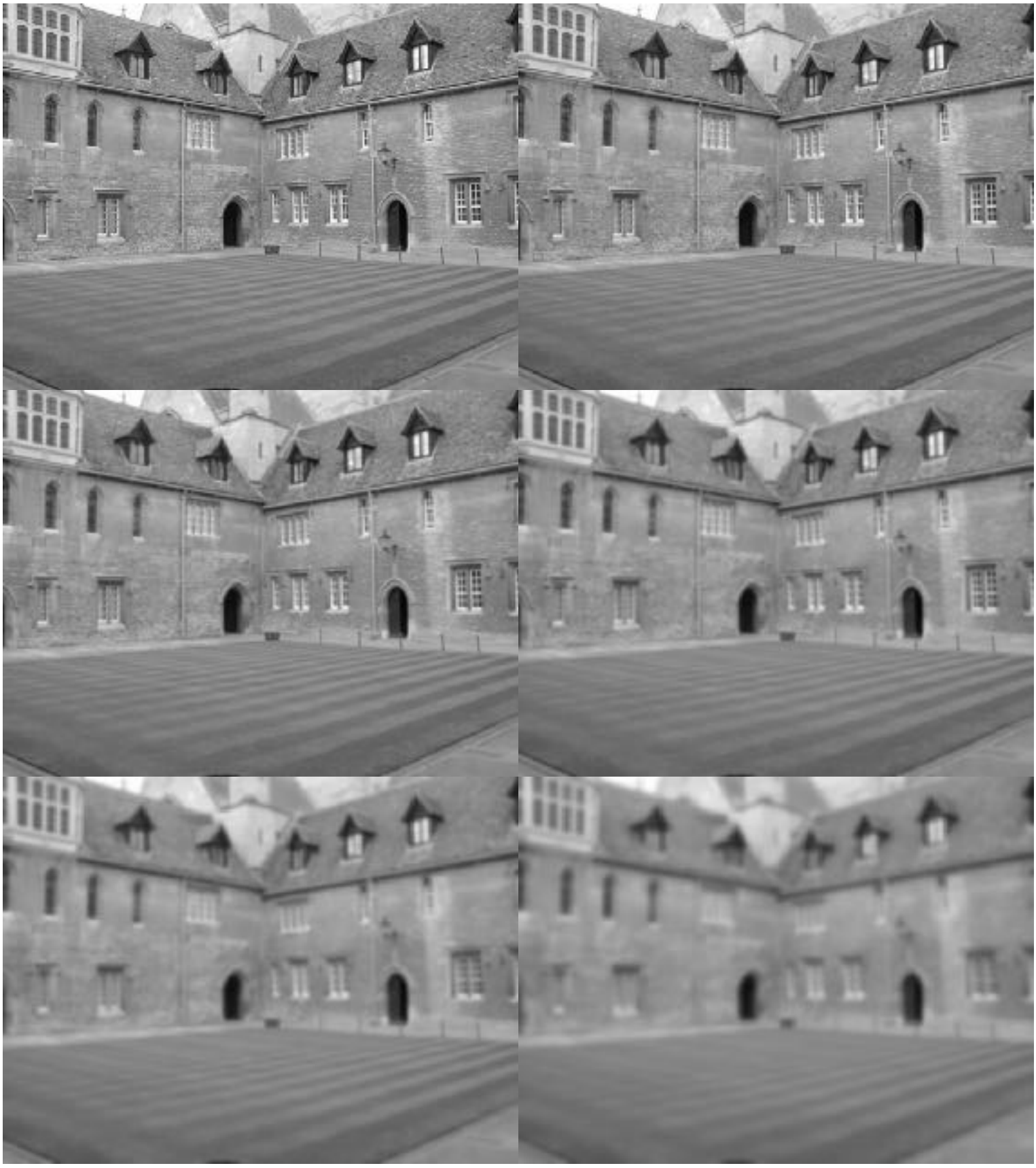
We then start making kernels based on the formula provide to us in the lecture slides for week 3.

$$g_\sigma = \frac{1}{2\pi\sigma^2}\exp -\frac{x^2+y^2}{2\sigma^2}$$

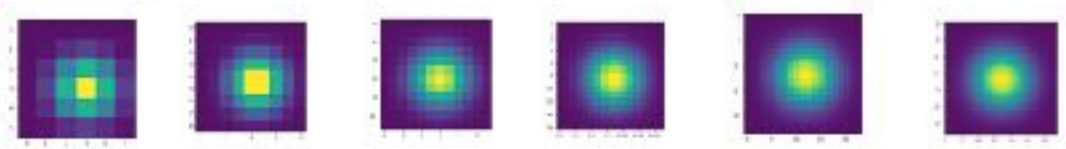
We then apply these kernels created for each sigma value on top of the original image (by default we are also saving the original image and the kernel representation in the drive **so you can turn it off by commenting the code**)

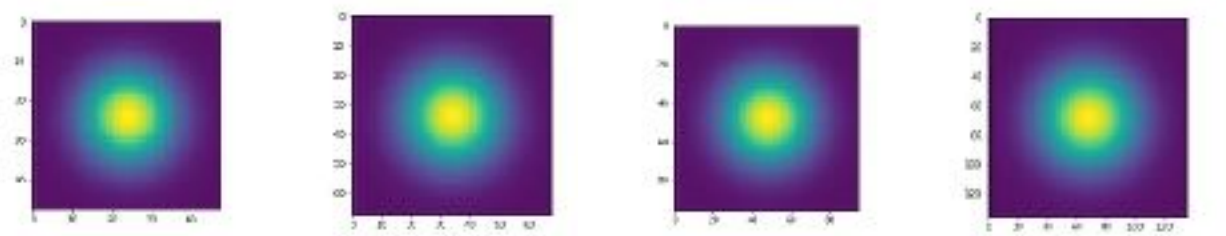
From the below given kernel visualization it is clear that the window size is sufficient enough.

We cant't see much of the difference in the two but because of rescaling to fit in this pdf, for more sigma values we can see the image is blurring even after rescaling (as evident in the next page)



The below is the kernel function plot.



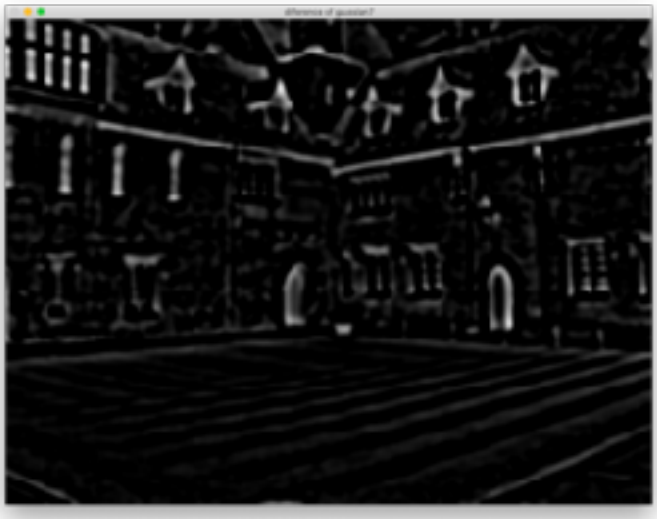


We can see that as we increase sigma the blurring effect increases.

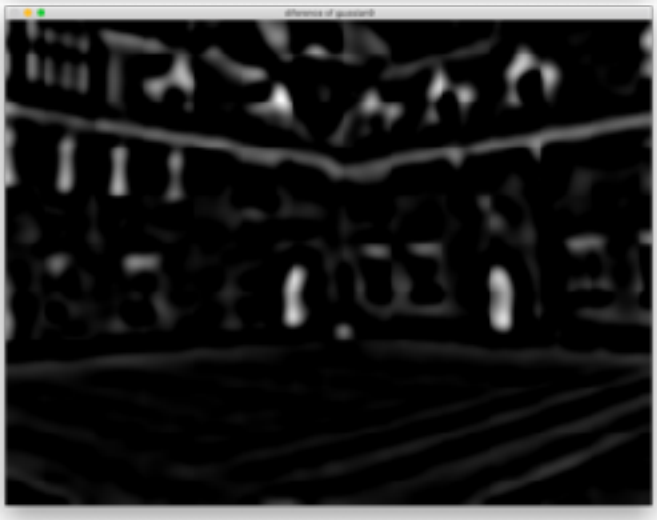
C. Send Use the scale-space representation from subtask B to calculate difference of Gaussian images at all scales. Display all resulting DoG images [3 points].

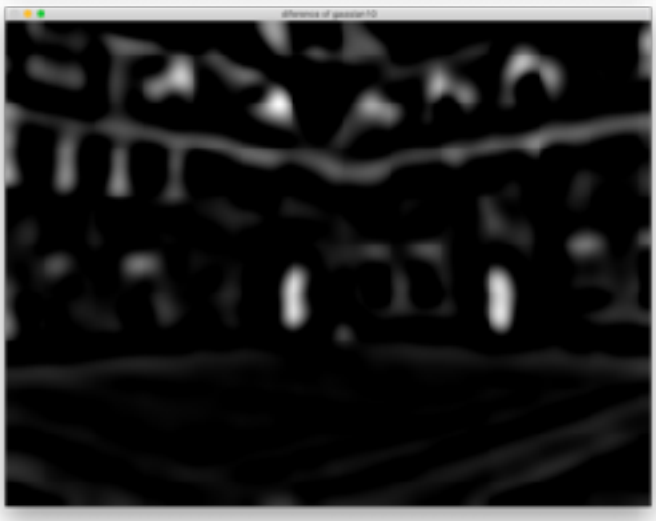
We will now take the difference of the gaussian blur images produced in the previous task. Meaning we will iterate through all the images in the list and then take the difference of second image with the first image.


difference of gaussian6


difference of gaussian7


difference of gaussian8


difference of gaussian9


difference of gaussian10

D. Find key-points by thresholding all DoGs from subtask C using a threshold of T = 10. Suppress non- maxima in scale-space by making sure that the key-points have no neighbours, both in space as well as in scale, with higher values [3 points]. The resulting key-points should comprise three coordinates (x, y, σ), two spatial and the scale at which they were detected.

We will now iterate through all the images in the dog images calculated in the subpart c of the task1, we initialise a loop to traverse through all images across all the coordinated in each image checking if the point is greater than the set threshold and also making sure that the point itself is greater than the neighbourhood of 1 pixels around. This check will help us ensure that we only select the points greater than a set threshold. Finally we store all the points in the list key_points which contains tuples of the cordite and the the image index from which it found the threshold point.

We can now see the total of all the threshold points found in each of the image by calculating the length of this list.

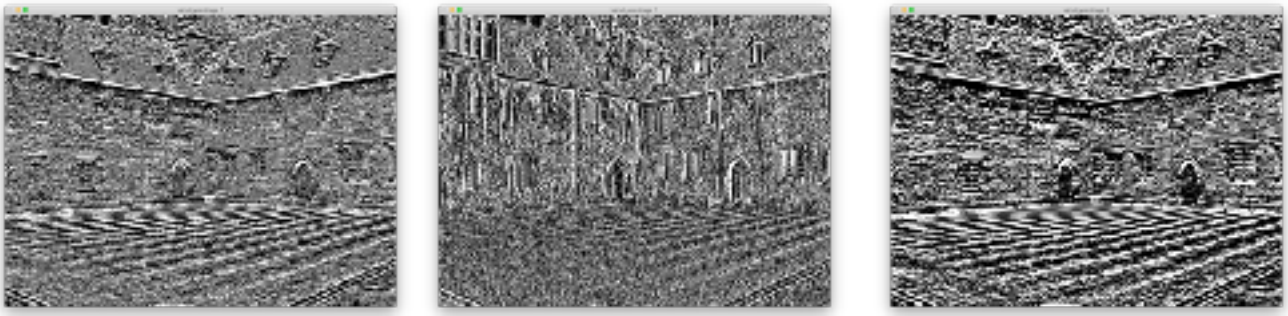
```
this is the length of my all key points 7875
```

(Note - we are not visualising the key points as it is not mentioned in the question)

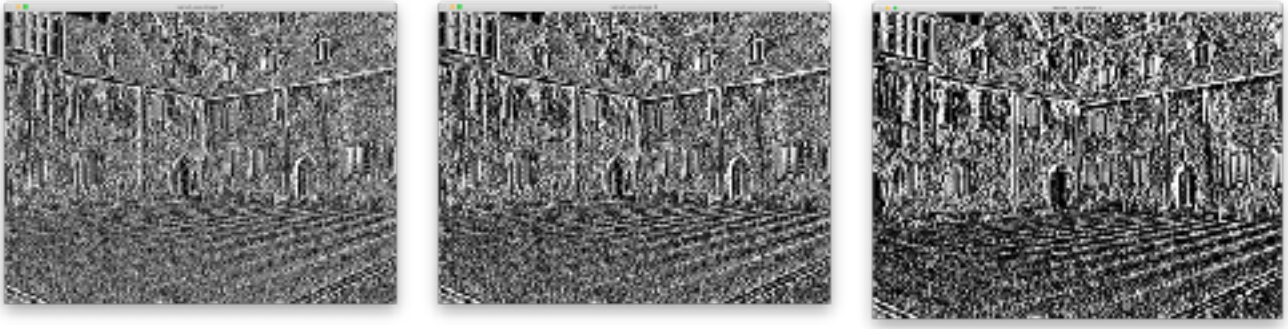
E. Calculate derivatives of all scale-space images from subtask B using the kernels

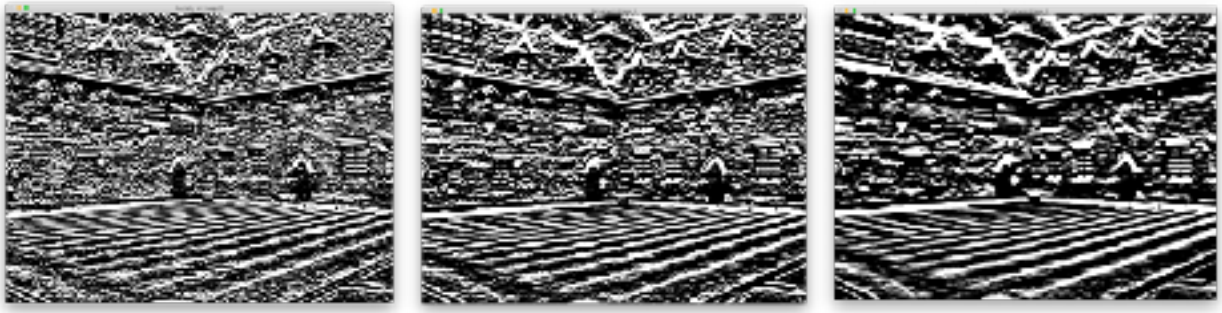
dx = (1 0 –1) and dy = (1 0 –1) . T Display the resulting derivative images at all scales [4 points]

We begin by initialising both the filter [1,0,-1] and it's transpose, then we iterate through all the images and sigmas that we calculated before in the previous subpart B, we can apply this filter to all our scale spaced images by calling the function filter2d, once we apply the filter we will view each of the image under these 2 new filters and save them in the dictionary with their corresponding sigma values.



Impact of first kernel (above) and the impact of second kernel (below)





Impact of first kernel (above) and the impact of second kernel (below)

Impact of first kernel (above) and the impact of second kernel (below)

F. Calculate the gradient length mqr and gradient direction θqr for the 7 × 7 grid of points $(q, r) \in \{x + 3\ k\sigma \mid k = -3, \ldots, 3\} \times \{y + 3\ k\sigma \mid k = -3, \ldots, 3\}$ sampled around 22 each key-point (x, y) and using the appropriate scale σ determined in subtask D and the correct gradient images from subtask E [4 points]. Also calculate a Gaussian weighting function $wqr = e^{-(q^2+r^2)/(9\sigma^2/2)}/(9\pi\sigma^2/2)$ for each of the grid points [1 point]. Now create a 36-bin orientation histogram vector h and accumulate the weighted gradient lengths $wqrmqr$ for each grid point (q, r) where the gradient direction θqr falls into this particular bin [3 points]. Use the maximum of this orientation histogram to determine the orientation of the key-point [1 point].

In this section we iterate through all the key points in the image, then for every key point we calculate a q,r defined in the k1 and k2 range of (-3,3) inclusive.

$$(q,r) \in \left\{x + \frac{3}{2}k\sigma \mid k = -3,..,3\right\} \times \left\{y + \frac{3}{2}k\sigma \mid k = -3,..,3\right\}$$

For this function we will use our q and r coordinates to compute the weighted gaussian function defined as

$$w_{qr} = e^{-(q^2+r^2)/(9\sigma^2/2)}/(9\pi\sigma^2/2)$$

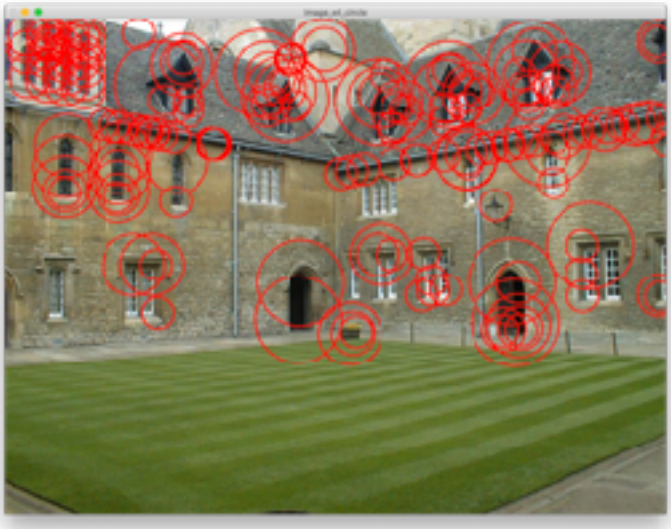
We can now calculate gx and gy using the formula to compute the theta value as explained in the discussion forums.

$$m = \sqrt{g_x^2 + g_y^2}$$

$$\theta = \text{atan2}(g_y, g_x)$$

G. Draw all the key-points into the input image using a circle with 3σ radius and a line from the key-point centre to the circle radius to indicate the orientation (see example for a single key-point on the right). Display the resulting output image with all the key-points [3 points].

From the key points calculated in the subtask D we will now try to plot circles around the original resized image, the circle radius denotes the intensity value of the key point. If radius is greater than 20 pixels only then it will plot.
(this is just to reduce the total number of circles that are to be marked on the image, keeping this threshold will help us to see bold points and avoid over clogging)



TASK 2 (Image matching, *17 points*): In this task you will implement a correlation-based area matching algorithm to find a patch extracted from one image in a second image.

A. Download the input image files Assignment_MV_01_image_1.jpg (the same as in the previous task) and Assignment_MV_01_image_2.jpg from Canvas. Load both files and convert them into a single channel grey value image [*2 points*]. Make sure the data type is float32 to avoid any rounding errors [*1 point*].

We will begin the task by calling the function by calling the function convert_grey_scale_task_2 and providing the locations of both the images that we want to convert into grey scale

```
image_1,image_2=convert_grey_scale_task_2(img_1,img_2)
```

Since the default color parameter is set to be 0, hence both the images will be loaded in a single grey channel by the mired function denoted below

```
#load our images in the grey scale format with only one channel
image_1= cv2.imread(image_1,color)
image_2 = cv2.imread(image_2,color)
```

We can see the size of both the images in the console output (since both of them have the same size), we can then try to see the first pixel value in both the images and which are 62 and 65 respectively.

We can convert this image into float32 format as shown and see that the pixel values are now changed to decimal numbers.

```
size of the first image: (768, 1024)
image 1 pixel 1 value :  62
Height of Image: 768 pixels
Width of Image:  1024 pixels
size of the second image: (768, 1024)
image 1 pixel 1 value :  65
Height of Image: 768 pixels
Width of Image:  1024 pixels
==========
===now converting the image_1 dtype to
float===
image dtype:  float32
62.0
===now converting the image_2 dtype to
float===
image dtype:  float32
65.0
are both images same :  False
360 210 430 300
=====done=====
```

Since both the images visually look same and have the same shape, let's check if they are actually the same or not ?

We can do so by comparing the two arrays that we got when we converted the image into single channel greyscale image by calling function array_equal function in numpy which will element wise compare the two arrays and only return True if all the elements are same at the same index in both the arrays. Which is why we get a False value (see the image below).

```
print("are both images same : ", np.array_equal(image_1,image_2))
return image_1,image_2
```

B. The window on the 1st floor above the arch on the left wing is in a rectangle with the image coordinates ((360,210), (430,300)) in the first input image. Draw a rectangle around this window in the input image and display it [*1 point*]. Now cut out the image patch only containing the window and display it as image [*2 points*].

In this section we will plot a rectangular boundary across the window to do so we need to know the coordinates of both the diagonals points that make up that rectangle. In the PDF thankfully we are provided with the coordinates to draw the image.

Which are (360,210), (430,300)) representing each of the diagonal point respectively, so let's try to plot the rectangles.

```
#make a rectangle(image,start diagnol ,end diag,color of the line, thickness )

cv2.rectangle(image, coordinates[0], coordinates[1], (127,50,127), thickness)
cv2.imshow("Rectangled image", image)

#this will show the bounded boxed image
cv2.waitKey(0)
cv2.destroyAllWindows()
```

The rectangle function takes the image that you want to draw upon, the diagonal coordinates of the rectangle, the color of the boundary and then finally the thickness of the line in pixels

The thickness denotes how many pixel boundary we have to surround inside the diagonal coordinates of our rectangle (notice in the image below as I increase my thickness less pixels are visible)

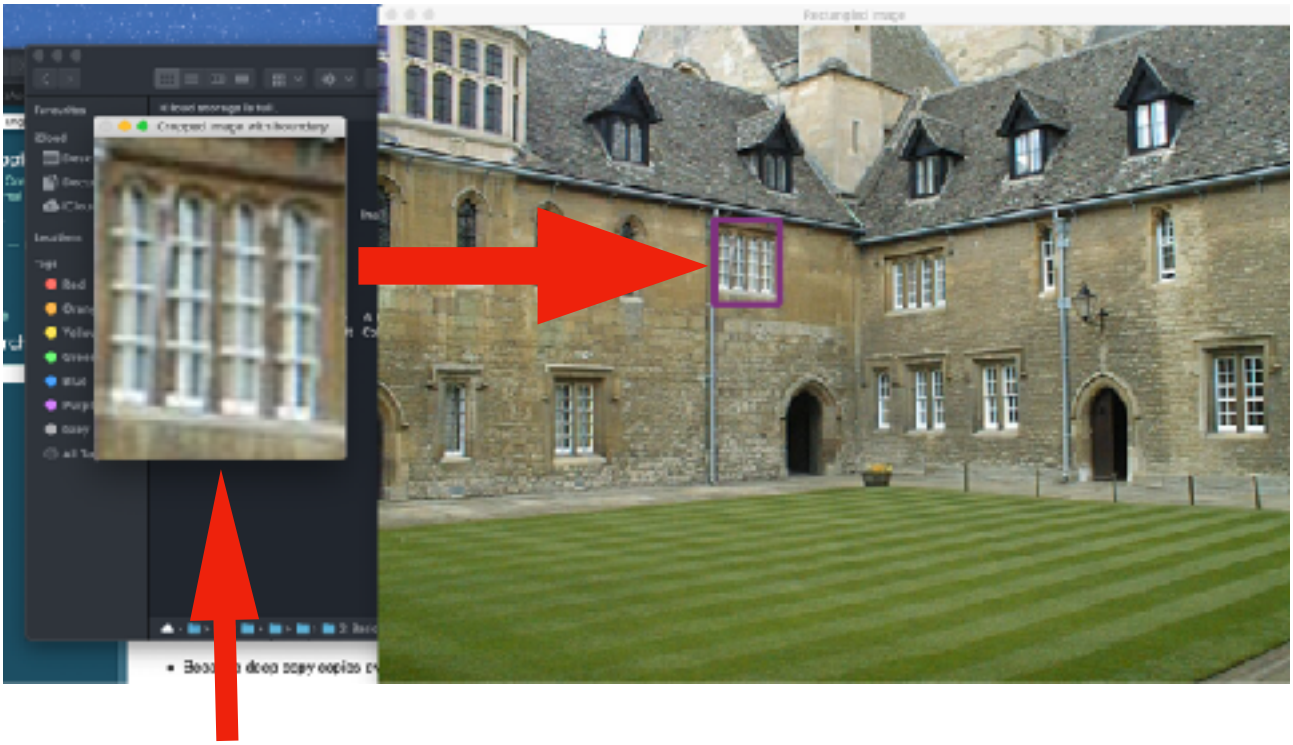
The above three images indicate the values of various thicknesses, tk=-1 highlights everything inbound the diagonal points in the region (tk=5 vs tk= 15, tk=-1)

We will now try to crop the section of the window as indicated by the boundary on the top.

If we now try to crop the selected pixel values we will also take the coloured boundary box that we had originally highlighted to avoid to take this coloured boundary we will set our thickness to 0 and operate on a fresh original_image of the house that we operated upon.

```
# we now set thickness to 0 as we don't want the highlight section to take any further margin over the cut out
thickness=0 #
print("thick: ",thickness)

#when thickness=0 we are only display the window but when give thickness we can also view the boundary
start_x=coordinates[0][0]-thickness
start_y=coordinates[0][1]-thickness
end_y=coordinates[1][1]+thickness
end_x=coordinates[1][0]+thickness
print(start_x,start_y,end_x,end_y)
cropped = original_image[start_y:end_y, start_x:end_x] #y trimming followed by x trimming
#numpy sees them differntly rows become my y and cols become my x

cv2.imshow("Cropped Image with boundary", cropped)
cv2.waitKey(0)
cv2.destroyAllWindows()
```

To crop a section of the image we are basically slicing the numpy array.
An interesting thing to note is that in numpy my row slicing becomes the height slicing of the image axes and the column slicing becomes the width slicing of the image coordinates.

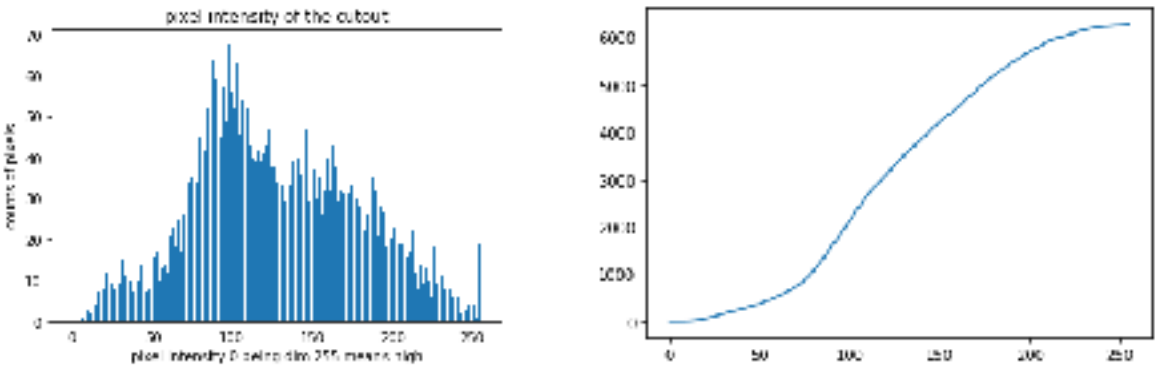


We can now see the cut out image by calling the imshow over the cropped.

The above image highlights a full resolution image with highlighted rectangle and the cut-out image without the highlighted boundary on the left.

C. Calculate the mean and standard deviation of the cut-out patch from subtask B [*2 points*]. Go through all positions in the second input image and cut out a patch of equal size [*2 points*]. Also calculate mean and standard deviation and from this the cross-correlation between the two patches [*3 points*]. Create and display an image of all cross-correlations for all potential positions in the second image [*2 points*]. Find the position with maximum cross-correlation and draw a rectangle around this position in the second input image. Display the result [*2 points*].

To visualise better let's plot the intensity values of the points in the cropped patch that we exported.



On the left we can see the histogram denoting the intensity values and on the right we have have a cumulative density function of the patch, we can see the plot has a mean intensity between 100-150 range.

Which can be calculated using the standard numpy functions,

```
#get the mean and the std of the cutout
mean=np.mean(cropped_image)
std=np.std(cropped_image)

print("my mean value of the cut out is : {} and my std dev is : {}".format(mean,std))

#let's cut out all possible patches in the second image
```

```
my mean value of the cut out is : 136.79137566137567 and my std dev is :
46.14895849382982
```

Now we will traverse through all the possible positions in the image (patch wise) to extract necessary frames.

To do so we are freezing a point of origin that is the top left corner, meaning, I will first traverse horizontally and then at the end will traverse vertically.

This is a nested loop where the limit on width is the max_image width- patch width to look, similarly maximum height is max image height- patch height. This will ensure that our patch does not exceed the image frame.

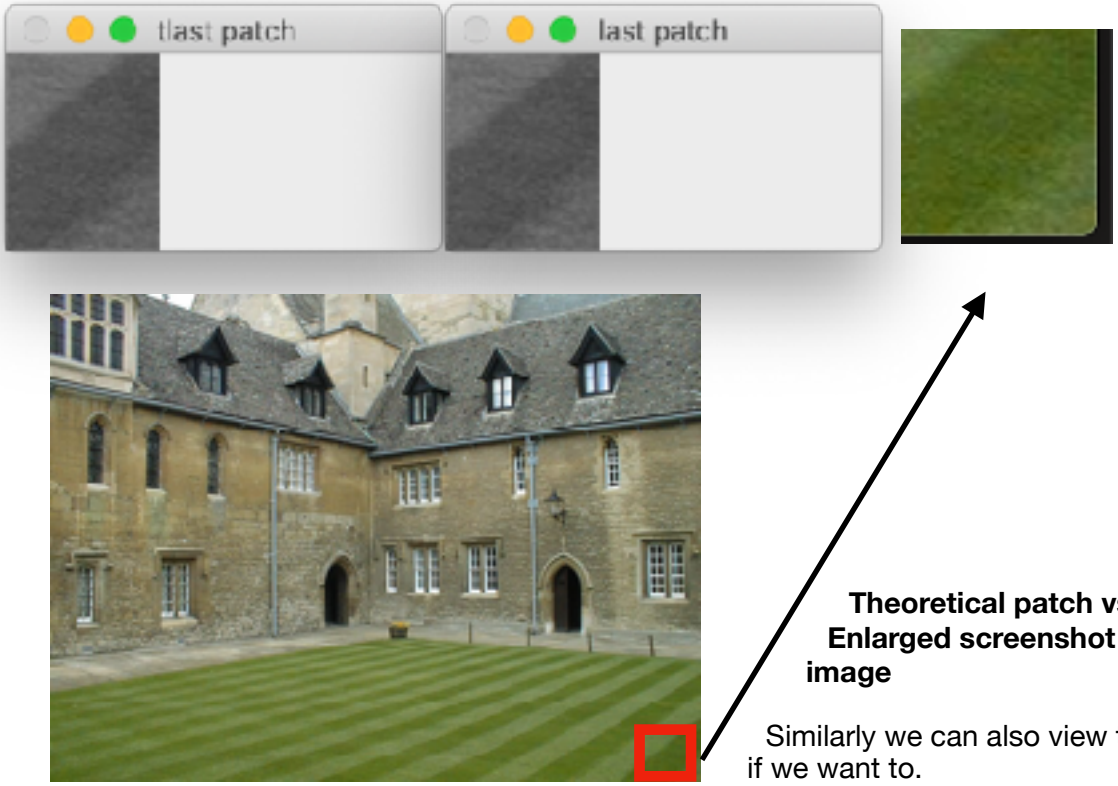
We can view information about this patch by uncommenting the piece of code

```
#patch_mean=np.mean(patch)
#patch_std=np.std(patch)
#result = cv2.matchTemplate(image_2, patch, cv2.TM_CCOEFF)
#print(result.shape)
```

But since there are too many patches formed for this image, we will comment the code as it drastically affects the performance or execution of the code.

Some important information about the patch shape and theoritically last patch coordinate can be seen by this section of code

We can see the last patch is very similar to the actual patch in the image and theoretical patch coordinates that we estimated it to have.



Theoretical patch vs last patch vs Enlarged screenshot of actual image

Similarly we can also view the first patch, if we want to.

This ensures that our code is traversing through all the possible patches in the image.

In order to speed up the process of matching and to compare all the extracted patches with the cropped patch from the first image, we will now use inbuilt opencv functions.

```
#some other stuff to make sure every thing is working right (approximately)
theoritical_lastpatch=image_2[768-cut_out_height:,1024-cut_out_width:]
print(theoritical_lastpatch.shape)
print(last_patch.shape)
cv2.imshow("tlast patch", theoritical_lastpatch)
cv2.waitKey(0)
cv2.imshow("last patch", last_patch)
cv2.waitKey(0)
print(np.array_equal(last_patch,theoritical_lastpatch))
cv2.destroyAllWindows()
```

In total we had 646813 number of patches in our list and computing the best match from the cropped image can take some time , hence we will now see opencv's implementation of matching.

The below code load the rgb version of the image and converts both the matching template and the image upon which we want to achieve matching as grey scale values as it is the prerequisite, this won't change anything for coloured image (in terms of location identified)

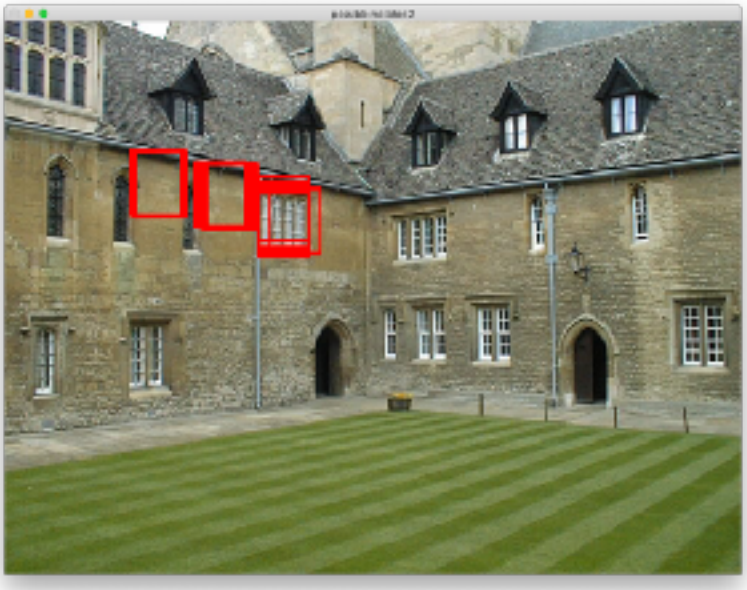
```
img_rgb = cv2.imread(img_2)
img_gray = cv2.cvtColor(img_rgb, cv2.COLOR_BGR2GRAY)
template = cv2.cvtColor(cropped_image,cv2.COLOR_BGR2GRAY)

w, h = template.shape[::-1]

res = cv2.matchTemplate(img_gray,template,cv2.TM_CCOEFF_NORMED)
threshold = 0.55
loc = np.where( res >= threshold)
for pt in zip(*loc[::-1]):
    cv2.rectangle(img_rgb, pt, (pt[0] + w, pt[1] + h), (0,0,255), 2)

cv2.imshow("possible matches"+str(len(loc)),img_rgb)
cv2.waitKey(0)
cv2.destroyAllWindows()
```

We are also setting a threshold and a comparison metric across which we compare.

The more is the threshold the less number of patches are identified as potential match and higher is the confidence.

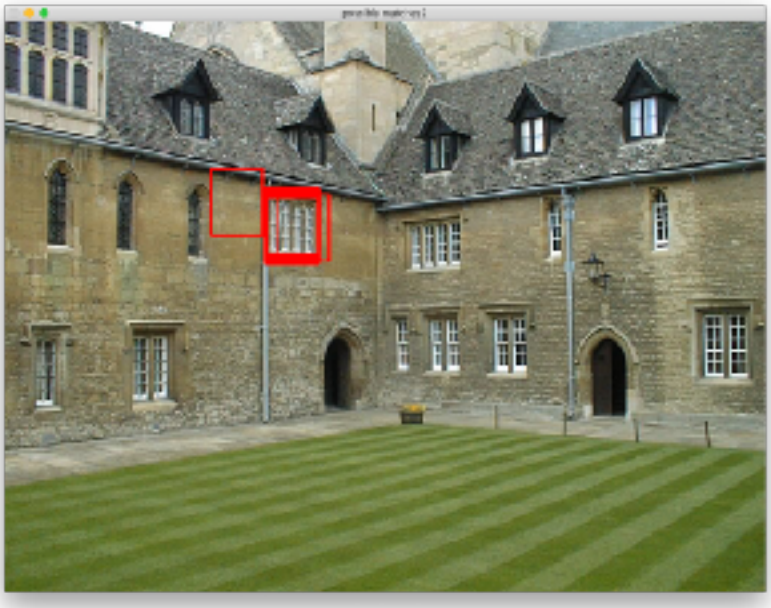
(for a more exhaustive list of matching techniques refer - [open_cv_doc])

The below graph indicates all possible matches that surpass the set threshold cross coif value to be 0.55.



As we can see on the image on the left that there are many false positives that fall into the match section however increasing the threshold will lower them.

Threshold =0.55 (see the image above) & Threshold =0.58 (see the image below)



However in the last section of this part we are asked to find the maximum cross section patch, which is quite easy with the min max function that gives us the patch coordinates(the diagonal points), In addition an extra boundary of 50 pixels is added so that the patch is clearly visible and does not get foreshadowed with the rectangle.

We can now see another patch of blue color and a text beside saying the best match while also highlighting the potential matches

```
extra_border=50
bottom_right = (top_left[0] + w, top_left[1] + h) #get the diagnal 2 point value
cv2.rectangle(img_rgb, top_left, bottom_right, (200,10,20), 2) #plot the bounding box

cv2.putText(img_rgb, 'best match', (bottom_right[0]-extra_border,bottom_right[1]+extra_border),
            cv2.FONT_HERSHEY_COMPLEX, 1, (200,10,20), 2)

cv2.imshow('best match', img_rgb)
cv2.waitKey(0)
cv2.destroyAllWindows()
```