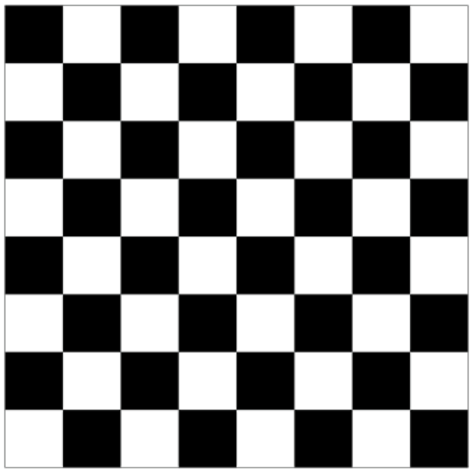
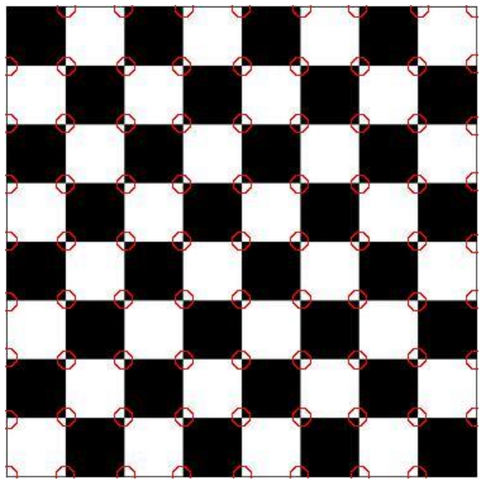
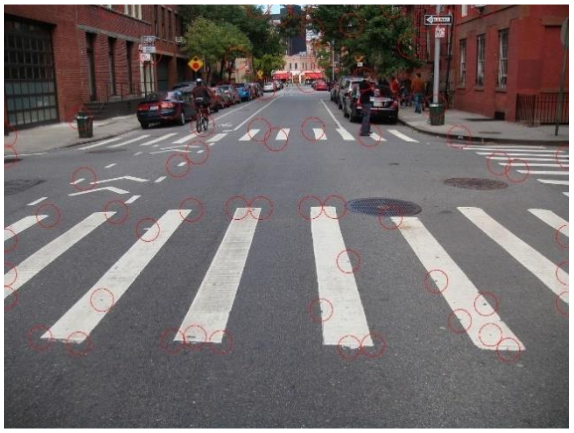
APS106 – Lab #8

Preamble

This week you will create functions to perform some simple, yet powerful, operations for image processing. In the process of this lab, you will build a simplified corner detector that will be able to analyze images and return the location of potential "corners" present within the image.

Input Image	Identified corners indicated by red circles
	
	

As usual, we will not be concerned with any complex theories or mathematics behind these approaches but focus on how these algorithms can be achieved using the tools you have learned in APS106. The focus of these exercises is to practice *using lists, tuples, and loops* with structured data. The lab may seem a little overwhelming in places because we are giving you background information about the problem, but we have broken the problem down into small functions that are well within your capabilities as APS106 students.

Take time to read the instructions carefully, give yourself time to complete the lab, and ask questions in your tutorials and practicals when you get stuck. If you think carefully about each function, you can finish the lab by writing less than 70 lines of code. **Most importantly, have fun**. This lab is designed to show you some of the really cool things you can do with only a little bit of programming experience!

Deliverables

For this lab, you must submit the five functions listed below within a single file named 'lab8.py' to MarkUS by the posted deadline.

Functions to implement for this lab:

- `rgb_to_grayscale`
- `dot`
- `extract_image_segment`
- `kernel_filter`
- `non_maxima_suppression`

Two additional functions are provided within the starter code. You may use these functions to complete and test your code as needed. **You are not expected to modify these two functions**.

- `harris_corner_strength`
- `harris_corners`

Use appropriate variable names and place comments throughout your program.

The name of the source file must be "lab8.py".

Five test cases are provided on MarkUs to help you prepare your solution. **Passing all these test cases does not guarantee your code is correct.** You will need to develop your own test cases to verify your solution works correctly. Your programs will be graded using ten secret test cases. These test cases will be released after the assignment deadline.

IMPORTANT:

- Do not change the file name or function names
- Do not use input() inside your program
- **Using numpy, scipy, opencv or other image processing packages to complete this lab is strictly prohibited and will result in a grade of zero.**

Introduction (Important information, read carefully)

For this week's lab, you will imagine you are working on a team developing autonomous vehicles. As a new engineer on the team, you have been asked to create some tools to detect and track the movement of certain objects in the vehicle's environment using the video from cameras in the vehicle's sensor system. As a starting point, someone tells you that corner detection can be used as a simple method for object movement detection and tracking. So, you set out, armed with your APS106 programming skills, to tackle this problem…

At this point you may be a little confused. We have not discussed "images" as a data type within the lectures and many of you may be asking questions like:

- What do images look like when stored in a variable?
- How do I get an "image" into my program?

Luckily, both questions are not too difficult to answer, and you already have all the programming tools you need to complete this lab! We will start by answering the first question and then answer the second question a little later.

So how are images stored in computers and our python programs? Computers store all information in binary code, so they 'see' images quite a bit differently than us humans. Rather than seeing abstract elements like colours, shapes, and objects, computers see a grid of numbers. These numbers are referred to as "pixels." The colour or intensity of an image at a location on the image is determined by the value of the pixel at that location (figure 1). Check out this short 2-minute video for a quick overview of how computers see images: https://realpython.com/lessons/how-computers-see-images/. You can also check out this demo to zoom in and out of a low-resolution image to see how pixels can make up a lager image: https://csfieldguide.org.nz/en/interactives/pixel-viewer/.

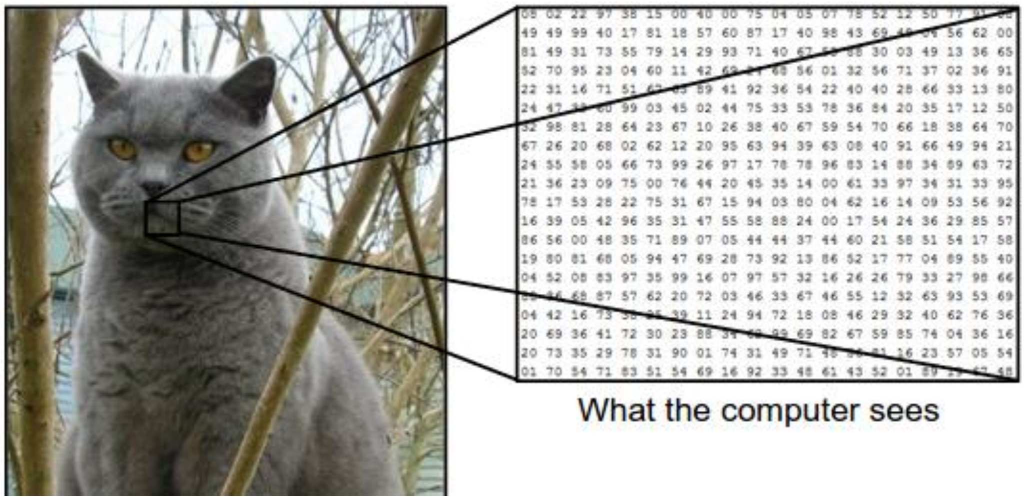


Figure 1. Computers represent images as a two-dimensional grid of pixel values. Image source: https://cs231n.github.io/

So within a computer program, an image is simply an *ordered* collection of numbers. **This means we can store and represent images in python as lists or tuples of integers!** For example, we could represent an image using the following tuple of numbers:

```
pixels = (250, 253, 255, 223, 181, 184, 232, 255,  ...
```

Each one of these numbers is the value of a pixel at a particular location in the image. To specify pixel locations, we use 2-dimensional coordinates along the height and width of the image. By convention, the top left corner pixel is the coordinate (0,0). Increasing the x-coordinate is equivalent to moving from left to right across the image. Increasing the y-coordinate is equivalent to moving from the top to the bottom of the image (figure 2).

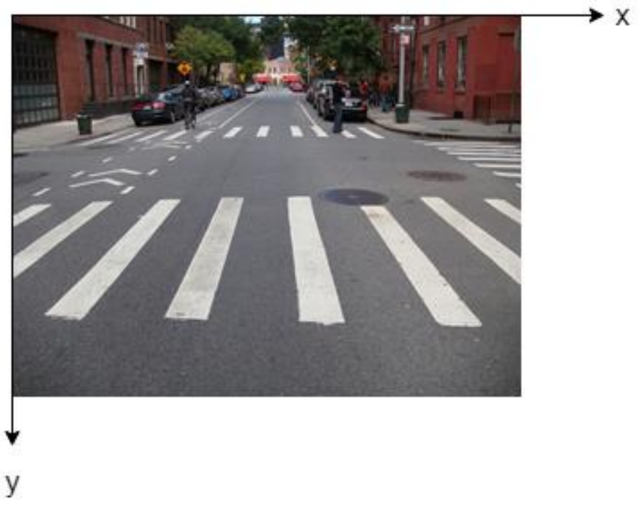
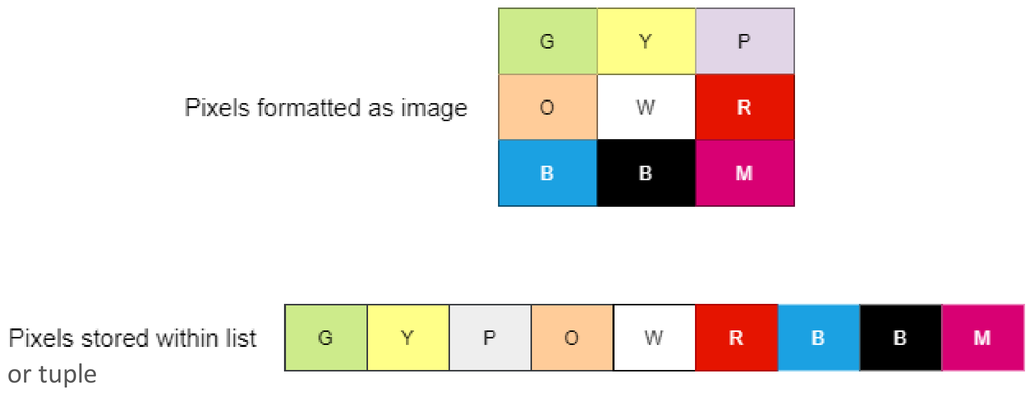


Figure 2. Coordinate axis conventions for images. Note that increasing the y-coordinate results in moving towards the bottom of the image.

For our program this week, you will be storing pixels in a one-dimensional tuple. Pixels will be ordered row-by-row. This means the pixel at coordinate (0,0) (the top left corner pixel) is stored in list index 0, the pixel at coordinate (1,0) (the second pixel from the left on the top row) is stored at index 1, the pixel at coordinate (2,0) is stored at index 2, and so on. Once we reach the end of a row, we move down a row (increase y) and start again from the left of the image. The pixel location and corresponding indices for a 3x3 pixel image are visualized below.



Optional – Loading and Displaying Images in Python

Now we will show you how to answer the question: How do I get an “image” into my program? Note that this component of the lab is for fun and completely optional and will not impact grading. If you choose to skip this part, you will not be able to load your own images into your program nor will you be able to display or save images generated by your program.

We have created a separate file, `lab8_image_utils.py`, which can be downloaded from the course website. Download and save this file in the same folder as your `lab8.py` file. The file contains two functions:

1. `image_to_pixels`
2. `display_image`

The `image_to_pixels` function will load an image specified by a filename and return a three-element tuple. The first element is a list of RGB pixels (described in part 1 below), the second element is the width of the image in pixels, and the third element is the height of the image in pixels.

The `display_image` function can be used to display and optionally save a list of pixels as an image. The first argument passed to this function is a tuple of pixels, the second argument is the width of the image in pixels, and the third argument is the height of the image in pixels. You may additionally pass two optional parameters. The first, `markers`, is a tuple of pixel coordinates. If markers are passed to the function, red circles will be added to the image at that coordinate location. The second optional parameter, `filename`, is a string. If a filename is passed to the function, it will save the image to a file with the specified name.

To use these functions in your lab, simply *uncomment* this line from the top of lab8.py

```
#from lab8_image_utils import image_to_pixels, display_image
```

To use these functions, you will need to install the pillow package. Instructions for installing the package are given on quercus.

Usage examples:

```
# load an image
pixels, width, height = image_to_pixels('crosswalk.jpg')

# display an image
display_image(pixels, width, height)

# display and save image
display_image(pixels, width, height, filename= 'my_image.jpg')

# display image with markers at locations (9,0) and (4,2)
display_image(pixels, width, height, markers = ((0,9),(4,2)))

# display and save image with markers at locations (9,0) and (4,2)
display_image(pixels, width, height, markers=((0,9),(4,2)), filename=
'my_image.jpg')
```

Testing Your Functions without Loading Images

The file `lab8_test_cases.py` provides RGB pixel values and expected function outputs the example snippet below for four different 32x32 images. You may use these samples to help you test your code. The 32x32 images can also be downloaded from quercus.

```
# rgb_pixels is a nested tuple of RGB pixel values

# convert to grayscale
grayscale_pixels = rgb_to_grayscale(rgb_pixels)
```

```
# apply blur filter
blur_kernel = ((1/9, 1/9, 1/9),
               (1/9, 1/9, 1/9),
               (1/9, 1/9, 1/9))
blurred_pixels = kernel_filter(grayscale_pixels, 32, 32, blur_kernel)

# apply vertical edge filter
vedge_kernel = ((-1, 0, 1),
                (-2, 0, 2),
                (-1, 0, 1))
vertical_edge_pixels = kernel_filter(grayscale_pixels,32,32,vedge_kernel)

# calculate harris corners
threshold = 0.1
harris_corners_result = harris_corners(grayscale_pixels, 32, 32, threshold)

# suppress non-maxima corners
min_dist = 8
non_maxima_result = non_maxima_suppression(harris_corners_result, min_dist)
```

To help you debug errors from the MarkUs testcases, we have also provided a file, `lab8_markus_for_students.py` which will simulate the 5 MarkUs testcases on your computer. You need to save this file in the same folder as the code you are writing (i.e. lab8.py) and execute it. It will import your lab8.py and run the same tests as MarkUs. Hopefully, this means that the problem of tests seeming to behave different will disappear. It should give you the same behaviour as MarkUs and so help you to track down your bugs.

Part 1 – RGB to Grayscale Conversion

For this part of the lab, you will complete the `rgb_to_grayscale` function that converts a RGB (colour) image into grayscale (black and white). The input to the function is a tuple of RGB pixels and the function returns a tuple of corresponding grayscale pixels.

RGB Image	Grayscale Image

The image on the left is in RGB format. In this format, each pixel is a combination of different intensities of red, green, and blue pixels. RGB images are represented by *nested* tuples where each pixel is a three-element tuple. The first element is the red intensity, the second element is the green intensity, and the third element is the blue intensity.

```
pixel_tuple = ((R0, G0, B0), (R1, G1, B1), (R2, G2, B2), ...
```

Grayscale images, on the other hand, are made up of only one value per pixel. An RGB pixel can be converted to a grayscale pixel using the following equation:

$$Grayscale\ value = 0.3\,R + \ 0.59G + 0.11B$$

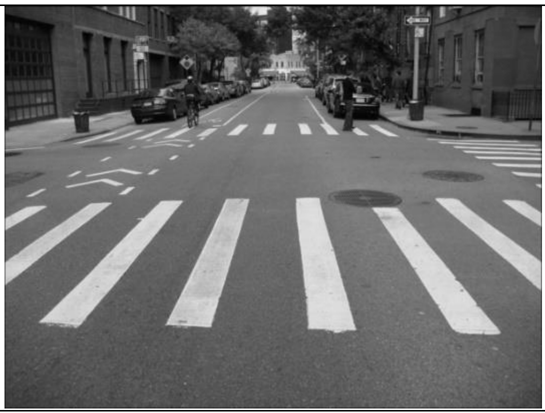
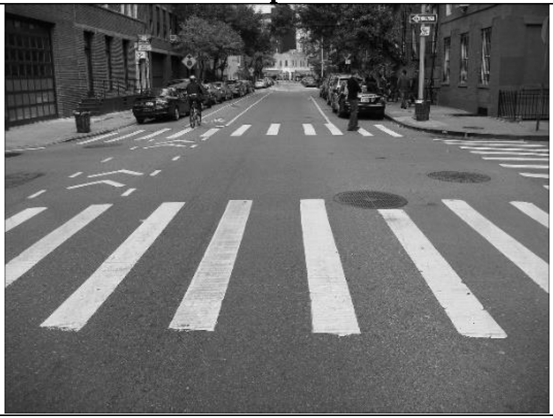
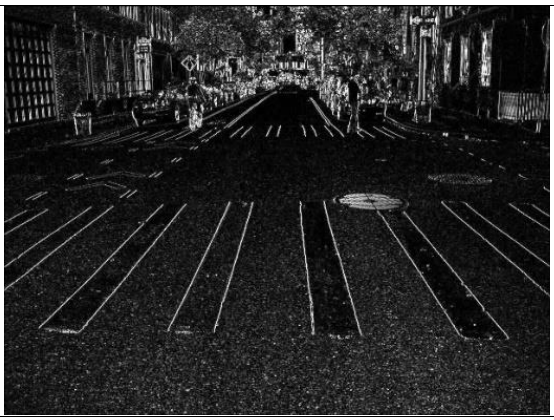
where R, G, and B are the red, green, and blue pixel intensities of the RGB pixel, respectively. All grayscale pixels computed by this function should be **rounded to the nearest integer**.

Sample Test Case

Input	Expected Output
`((3,67,90), (249, 255, 0), (49, 150, 128))`	`(50, 225, 117)`

Part 2 – Kernel Filtering

In this part of the lab, you will implement a function to apply two-dimensional filters to *grayscale images*. This filtering approach is called convolutional filtering and it is the building block of many modern deep machine learning and artificial intelligence algorithms. By the end of this part, your program will be able to produce the example image transformations shown below.

Original Image	Blurred
	
Sharpened	**Vertical Edge Detection**
	

The mathematical theory behind these filters is beyond the scope of APS106; instead, the exercises here will focus on how these this filtering procedure can be implemented using the data types and looping tools you have acquired in the course thus far. In the next part, we will walk through the algorithm you need to implement. Then you will write three functions to perform these filtering operations.

Part 2.1 Problem Description (important information, read carefully)

Each individual pixel value in the output image is a weighted sum of pixels from a two-dimensional $N \times N$ (N pixels high & N pixels wide) segment of the input image. The weights for the sum are also defined using a two-dimensional grid, referred to as a kernel.

As an example, a 3×3 "Gaussian blurring" kernel has the following weights:

1/16	1/8	1/16
1/8	1/4	1/8
1/16	1/8	1/16

To illustrate the filtering procedure, we will examine an example below using this kernel. The pixel values at each *(x,y)* coordinate of the *output* image are computed using the following four-step procedure:

1. Extract the $N \times N$ image segment (group of pixels) from the input image centred around the position *(x,y)*. In this example, $N = 3$.
2. Multiply each pixel value from the $N \times N$ segment with the kernel weight in the corresponding location.
3. Sum the products from step 2.
4. Truncate or *round down* the sum from step 3 to the nearest integer.

We will illustrate this procedure with the following 7×7 input image example. Each number in the table represents a pixel value.

4	87	233	245	227	209	190
2	59	235	246	229	219	200
17	99	230	220	211	210	201
46	58	196	165	201	179	150
82	63	41	169	190	188	145
99	55	54	55	74	23	12
45	55	56	45	155	145	156

To compute the output pixel value at coordinate (1,1):

1. Extract the 3×3 image segment around the pixel at position (1,1). Remember that the top left pixel is at position (0,0). The pixel value at position (1,1) in the table below is shown in bold and the 3×3 segment is shown by the red border.

4	87	233	245	227	209	190
2	**59**	235	246	229	219	200
17	99	230	220	211	210	201
46	58	196	165	201	179	150
82	63	41	169	190	188	145
99	55	54	55	74	23	12
45	55	56	45	155	145	156

2. Multiply each pixel from this segment with the corresponding weight in the kernel

4	87	233
2	59	235
17	99	230

×

1/16	1/8	1/16
1/8	1/4	1/8
1/16	1/8	1/16

=

4/16	87/8	233/16
2/8	59/4	235/8
17/16	99/8	230/16

3. Sum each of the products from step 2

$$\frac{4}{16}+\frac{87}{8}+\frac{233}{16}+\frac{2}{8}+\frac{59}{4}+\frac{235}{8}+\frac{17}{16}+\frac{99}{8}+\frac{230}{16}=97.875$$

4. Truncate the sum to an integer

```
int(97.875) = 97
```

The resultant pixel at location (1,1) in the output image would be 97. We would then repeat this process for the other pixels. The output pixel values for the entire 7×7 image are shown in the table below. We encourage you to try and compute the pixel value at location (2,1) by hand to make sure you have a sound understanding of the algorithm.

0	0	0	0	0	0	0
0	97	195	233	225	212	0
0	100	184	215	210	202	0
0	88	145	181	192	185	0
0	69	91	131	152	141	0
0	61	60	84	105	98	0
0	0	0	0	0	0	0

Looking at that output, you may be wondering why all the pixels at the edge of our output image are zero. The reason for these zeros is that for these edge pixels we cannot extract a full 3×3 window around the pixels because the 3×3 window would extend beyond the border of the image. So, for simplicity, we set the output to zero for all pixels in edge regions where a full $N \times N$ window cannot be extracted.

For more examples, check out this great interactive tool that shows you how output image pixels are computed using segments of the input image and the kernel: https://setosa.io/ev/image-kernels/.

In the next parts of the lab, you will write three functions to implement this kernel filtering process. We will begin by writing two helper functions to complete steps 1, 2, and 3 from the four-step procedure outlined above and finish by writing a function to perform these steps for all pixels in the image.

Part 2.2 Dot Product – dot function

If we convert the kernel weights and image segments into vectors, steps 2 and 3 become the dot product operation from linear algebra. Here, you will write a function to perform this dot product operation.

The `dot` function accepts two tuples, each representing a vector, as inputs. You may assume the lists are of equal length and only contain numerical values. The function should compute and return the dot product of the two vectors as a float.

Part 2.3 Extracting NxN Image Segments – extract_image_segment function

The `extract_image_segment` function will be used to complete step 1 of the four-step procedure by extracting and returning a $N \times N$ segment of pixels from a tuple of pixels representing an image. This function accepts the following input arguments:

- `img` – A tuple of grayscale image pixel values
- `width` – The width of the input image
- `height` – The height of the input image

- `centre_coordinate` – A two-element tuple containing the coordinate specifying the centre of the segment to extract
- `N` – Integer specifying the width and height of the segment to extract, N will always be a positive, odd integer

The function should extract the $N \times N$ segment of pixels centred around the `centre_coordinate` and return the pixel values contained within the segment as a tuple. You may assume that the centre coordinate is **not** within an edge region where the full window is undefined.

Sample Testcases

For each testcase, assume that the `img` variable refers to the pixel values for the 7×7 image used in the example in part 2.1.

Extract 3x3 window centred at (4,1)

```
>>> extract_image_segment(img, 7, 7, (4,1), 3)
(245, 227, 209, 246, 229, 219, 220, 211, 210)
```

Extract 5x5 window centred at (2,2)

```
>>> extract_image_segment(img, 7,7, (2,2), 5)
(4, 87, 233, 245, 227, 2, 59, 235, 246, 229, 17, 99, 230, 220, 211,
46, 58, 196, 165, 201, 82, 63, 41, 169, 190)
```

Part 2.4 Put it Together – kernel_filter function

The `kernel_filter` function takes the following as input parameters:

- `img` – A tuple of grayscale image pixel values
- `width` – The width of the input image
- `height` – The height of the input image
- `kernel` – A nested tuple defining the $N \times N$ two-dimensional kernel weights. Each element of the tuple is a row of kernel weights. N must be an odd integer.

The function should return a list of grayscale pixel values representing the filtered image. The output image should be the same size (height and width) as the input. All pixels in the edge region of the output should be set to zero. **Hint:** as part of this function, you will need to determine the size of the edge region where a full $N \times N$ window cannot be extracted.

Here are a few test kernels you may want to further experiment with.

Name	Kernel			Description
Blur	1/9	1/9	1/9	Blur images by taking average of 3x3 neighbourhoods
	1/9	1/9	1/9	
	1/9	1/9	1/9	
Sharpen	0	-1	0	Enhance image edges
	-1	5	-1	
	0	-1	0	
Vertical Sobel	-1	0	1	Detect vertical edges in images
	-2	0	2	
	-1	0	1	

Horizontal Sobel	-1 | -2 | -1 0 | 0 | 0 1 | 2 | 1	Detect horizontal edges in images
5x5 Gaussian	1/256 | 4/256 | 6/256 | 4/256 | 1/256 4/256 | 16/256 | 24/256 | 16/256 | 4/256 6/256 | 24/256 | 36/256 | 24/256 | 6/256 4/256 | 16/256 | 24/256 | 16/256 | 4/256 1/256 | 4/256 | 6/256 | 4/256 | 1/256	Average with greater weight given to pixels closest to the centre

Part 3 – Harris Corner Computations

There is no code to write for this part of the assignment. Instead, we will describe the output of the `harris_corners` function which you will be used as the input to the `non_maxima_suppression` function in the next part.

The Harris Corner Detector algorithm is an algorithm to infer the location of corners within an image. A detailed description of the algorithm and its derivation is beyond the scope of this assignment. If you want to learn more about the algorithm, there are several tutorials available on the internet. Briefly, the algorithm takes a grayscale image as an input and computes a "corner strength" for each pixel in the image. The algorithm then identifies the locations of all pixels with a corner strength greater than a user-defined threshold. These locations are then sorted according to the corner strength value from greatest to smallest. Our `harris_corners` function outputs these sorted corner locations as a tuple.

If you examine the returned values from this function, however, you will find that it often returns many pixel locations which are very close to each other (see all the overlapping red circles in Figure 3 below). Many of the locations returned are identifying the same corner. Ideally, we would like our corner detector to only identify one pixel location per corner. In the final part of this lab, you will implement a function that will limit the number of corners returned by the algorithm within subregions of the image.

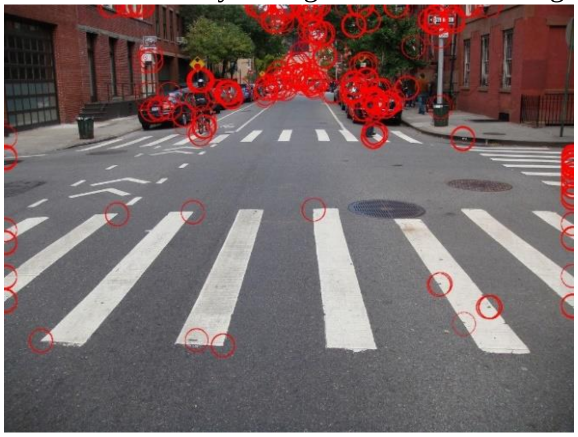


Figure 3. Output of Harris Corners without Non-Maxima Suppression. Each red circle indicates a detected corner.

Part 4 – Non-Maxima Suppression

In this part of the lab, you will complete the `non_maxima_suppression` function. This function takes the following input parameters:

- `corners` – A tuple of corner locations, sorted from strongest to weakest, as returned by `harris_corners`
- `min_distance` – A float defining the minimum allowed distance between corners returned by this function

This function filters the list of corners to remove any corners that are within a specified distance to corner with a greater corner strength. This filtering is achieved using the following algorithm:

1) Initialize an empty list of unsuppressed corners, *F*
2) For each potential corner *i* in the *corners* list:
   a) Calculate the Euclidean distances between *i* and all corners within *F*
   b) Add *i* to *F, unless* any of the Euclidean distances from 2a are less than *min_distance,*
3) Convert *F* to a tuple and return

The Euclidean distance between two pixels *i* and *j* with locations (x_i, y_i) and (x_j, y_j), can be computed as

$$\left(\left(x_i - x_j\right)^2 + \left(y_i - y_j\right)^2\right)^{\frac{1}{2}}$$

Let's work through an example to see how this works. Suppose our input list of potential corner coordinates is `((5,6), (15,6), (17,5))` and `min_distance = 10.`

In step 1, we create an empty list of unsuppressed corners

```
F = []
```

Then we start to iterate through all the corners of our input list. The first coordinate *i* is `(5,6)`. In step 2a, we need to calculate the distance between *i* and all the coordinates in *F* and check if any of these are less than `min_distance`. Since *F* is empty, there are no distances to calculate and therefore, there are no distances less than `min_distance`. So, we add *i* to *F*.

```
F = [(5,6)]
```

Now we return to step 2 and set *i* to the next coordinate `(15,6)` and then calculate the distances between *i* and all the points in *F*. The Euclidean distance between `(5,6)` and `(15,6)` is exactly 10. Since 10 is not less than `min_distance`, *i* is added to *F*.

```
F = [(5,6), (15,6)]
```

Now we repeat the process for the final coordinate `(17,5)`. The distances between `(17,5)` and the points within *F* are 12.04 and 2.24. Since 2.24 is less than `min_distance`, we do not add `(17,5)` to *F*. Now that we have iterated though every point, we convert *F* to a tuple and return.

Conclusion

That's it, you've used your programming super powers to make a corner detector program. Congratulations! You have come a long way since your simple one-or-two-line programs in the first week of the course. Take a moment to reflect on everything you have learned about programming in the short amount of time since the beginning of term and what you have just been able to accomplish in this lab.