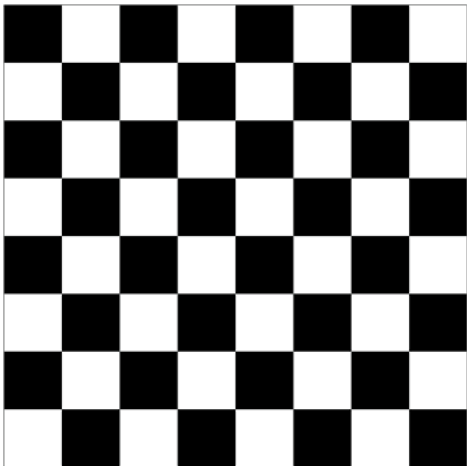
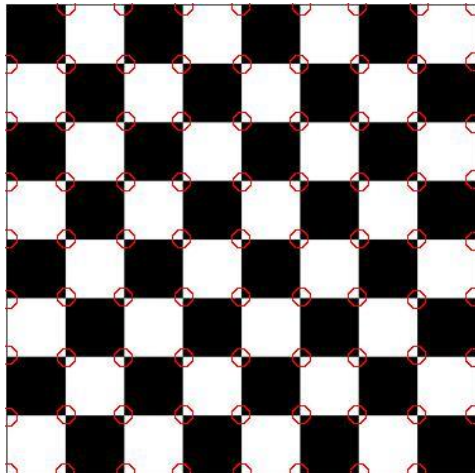




APS106 – Lab #6

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 complex theory or mathematics behind these approaches but focus on how these algorithms can be achieved using the tools you have learned within APS106. The focus of these exercises is to practice *using lists 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.

Make sure you 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 150 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 'lab6.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 "lab6.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

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:

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

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 second question and then answer the first 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 grid is determined by the value of the pixel at that location. 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 larger image: <https://csfieldguide.org.nz/en/interactives/pixel-viewer/>.

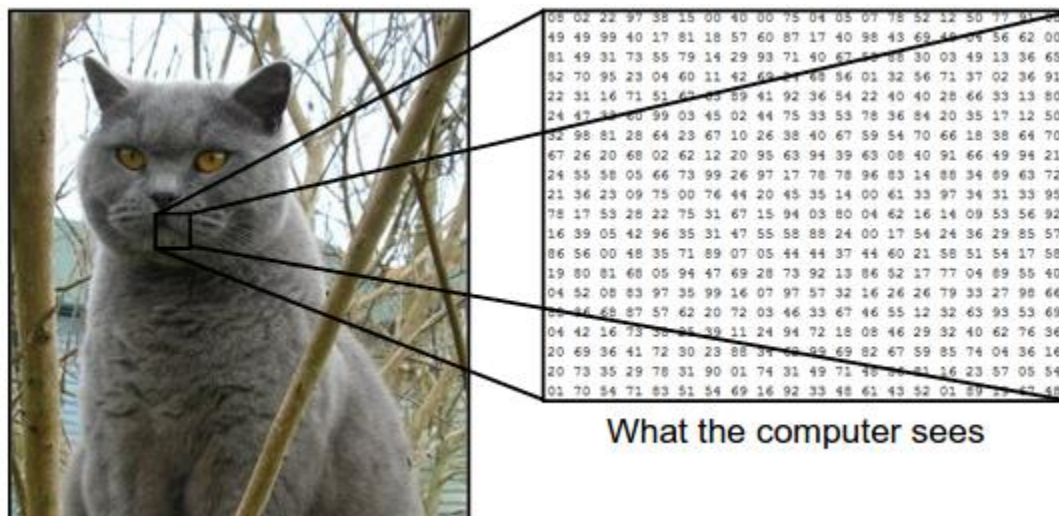


Image source: <https://cs231n.github.io/>

An image is simply an ordered collection of numbers. **This means we can store and represent images in python as lists of integers!** For example, we could represent an image using the following list 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. 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.



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

Pixels formatted as image

G	Y	P
O	W	R
B	B	M

Pixels stored within list

G	Y	P	O	W	R	B	B	M
---	---	---	---	---	---	---	---	---

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, `lab6_image_utils.py`, which can be downloaded from the course website. Download and save this file in the same folder as your `lab6.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 list. The first element is a list of RGB pixels (described in part 1), 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 list 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 list 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 `lab6.py`

```
#from lab6_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 `lab6_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 list 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)
```

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 list of RGB pixels and the function returns a list 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* lists where each pixel is a three-element list. The first element is the red intensity, the second element is the green intensity, and the third element is the blue intensity.

```
pixel_list = [[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:

$$\text{Grayscale value} = 0.3R + 0.59G + 0.11B$$

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 Cases




```
>>> rgb_to_grayscale([[3,67,90], [249, 255, 0], [49, 150, 128]])
[50, 225, 117]
```

For additional test cases, see the Testing Your Functions without Loading Images section.

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 learning and artificial intelligence algorithms. By the end of this part, you will be able to achieve some of the image transformations shown below.

Filter	Output Image	
None		

Blur			
Sharpen			
Vertical Edge Detect			

The mathematical theory behind these filters is beyond the scope of APS106. The exercises here will focus on how these this filtering procedure can be implemented using the lists and looping tools you have acquired in the course thus far.

Part 2.1 Problem Description

Each individual pixel value in the filtered images is a weighted sum of pixels from of an $N \times N$ (N pixels high & N pixels wide) segment of the original input image. The weights for this sum are defined within a grid, referred to as a kernel.

To illustrate the filtering process, we will examine a simple kernel called the blurring filter as an example. This 3×3 kernel has the following weights:

1/9	1/9	1/9
1/9	1/9	1/9
1/9	1/9	1/9

Using this kernel, the output pixel at position (x,y) is computed using the following four-step procedure:

1. Extract the 3×3 image segment from the input image centred around the position (x,y)
2. Multiply each pixel from this segment with the corresponding weight in the kernel
3. Sum the products from step 2
4. Truncate the sum from 3 into an integer

To better understand this, we will work out an example with the following 7x7 input image:

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

Now let's compute the output pixel value at position (1,1):

1. Extract the 3x3 image segment around the pixel at position (1,1)

4	87	233
2	59	235
17	99	230

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

4	87	233	×	1/9	1/9	1/9	=	4/9	87/9	233/9
2	59	235		1/9	1/9	1/9		2/9	59/9	235/9
17	99	230		1/9	1/9	1/9		17/9	99/9	230/9

3. Sum the products from step 2

$$\frac{4 + 87 + 233 + 2 + 59 + 235 + 17 + 99 + 230}{9} \cong 107.33$$

4. Truncate the sum to an integer

$$\text{int}(107.33) = 107$$

The resultant pixel at location (1,1) in the output image would be 107. Looking at the equation in step 3, you will see that this kernel is just computing the average value of a pixel and its eight

neighbours! We would then repeat this process for the other pixels. The output pixel values for the entire 7x7 image would be:

0	0	0	0	0	0	0
0	107	183	230	224	210	0
0	104	167	214	208	200	0
0	92	137	180	192	186	0
0	77	95	127	138	129	0
0	61	65	93	116	120	0
0	0	0	0	0	0	0

Looking at that output, you are probably wondering why all the pixels at the edge of our output image are zero, even though the average of the input pixels in these areas of the input image are not zero! The reason for these zeros is for these edge pixels, we cannot extract a full 3x3 window around the pixels because the 3x3 window would extend beyond the border of the image. So, for simplicity, we will set the output to zero for all pixels in edge regions where a full NxN window cannot be extracted.

If this example was unclear, 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

If we convert the kernel weights and image segments from step 2 of the four-step procedure into vectors, steps 2 and 3 become the dot product operation from linear algebra. In the next part of the lab, you will write a function to perform this dot product operation.

The `dot` function accepts two list, 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`

In this part of the lab, you will write the `extract_image_segment` function. This function will be used to complete step 1 of the four-step output pixel calculation procedure. This function accepts the following input arguments:

- `img` – A list of grayscale image pixel values
- `width` – The width of the input image
- `height` – The height of the input image
- `centre_coordinate` – A two-element list 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 list. For this lab, you may assume that an $N \times N$ window around the centre coordinate will always exist. That is, 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` list contains the pixel values for 7x7 image used in the example in part 2.1.

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

>>> 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`

In this part, you will complete the `kernel_filter` function. This function takes the following as input parameters:

- `img` – A list of grayscale image pixel values
- `width` – The width of the input image
- `height` – The height of the input image
- `kernel` – A nested list defining the $N \times N$ two-dimensional kernel weights. Each element of the list 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.

For sample test cases, see the Testing Your Functions without Loading Images section.

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		

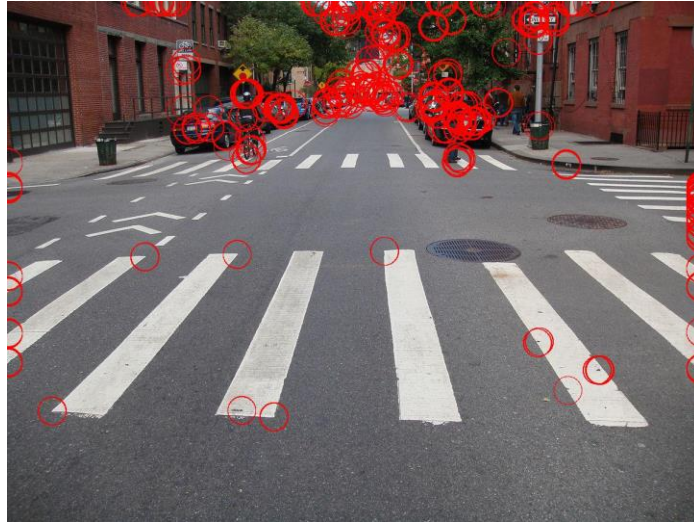
Sharpen		0	-1	0		Enhance image edges
		-1	5	-1		
		0	-1	0		
Vertical Sobel		-1	0	1		Detect vertical edges in images
		-2	0	2		
		-1	0	1		
Horizontal Sobel		-1	-2	-1		Detect horizontal edges in images
		0	0	0		
		1	2	1		
5x5 Gaussian	1/256	4/256	6/256	4/256	1/256	Like blur, with greater weight given to pixels closest to the centre
	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	

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 list.

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 the image 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.



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 list 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`,

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

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

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 .