Problem Set 2: Template Matching and FFT

September 14, 2021

ASSIGNMENT DESCRIPTION

Description

Problem Set 2 is aimed at introducing basic building blocks of image processing. Key areas that we wish to see you implement are: loading and manipulating images, producing some valued output of images, and comprehension of the structural and semantic aspects of what makes an image. For this and future assignments, we will give you a general description of the problem. It is up to the student to think about and implement a solution to the problem using what you have learned from the lectures and readings. You will also be expected to write a report on your approach and lessons learned.

Learning Objectives

- Use Hough tools to search and find lines and circles in an image.
- Use the results from the Hough algorithms to identify basic shapes.
- Use template matching to identify shapes
- Understand the Fourier Transform and its applications to images
- Address the presence of distortion / noise in an image.

Problem Overview

Methods to be used

Rules

You may use image processing functions to find color channels and load images. Don't forget that those have a variety of parameters and you may need to experiment with them. There are certain functions that may not be allowed and are specified in the assignment's autograder Ed post.

Refer to this problem set's autograder post for a list of banned function calls.

Please do not use absolute paths in your submission code. All paths should be relative to the submission directory. Any submissions with absolute paths are in danger of receiving a penalty!

INSTRUCTIONS

Obtaining the Starter Files:

Obtain the starter code from canvas under files.

Programming Instructions

Your main programming task is to complete the api described in the file **ps2.py**. The driver program **experiment.py** helps to illustrate the intended use and will output the files needed for the writeup. Additionally there is a file **ps2_test.py** that you can use to test your implementation.

Write-up Instructions

Create **ps2_report.pdf** - a PDF file that shows all your output for the problem set, including images labeled appropriately (by filename, e.g. ps2-1-a-1.png) so it is clear which section they are for and the small number of written responses necessary to answer some of the questions (as indicated). For a guide as to how to showcase your results, please refer to the latex template for PS2.

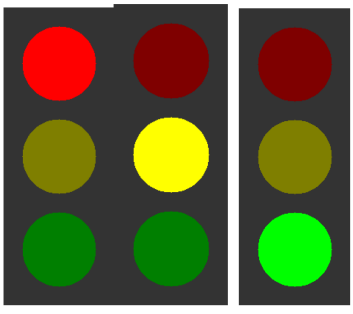
How to Submit

Two assignments have been created on Gradescope: one for the report - **PS2_report**, and the other for the code - **PS2_code** where you need to submit ps2.py and experiment.py.

1. HOUGH TRANSFORMS [10 POINTS]

1.a. Traffic Light

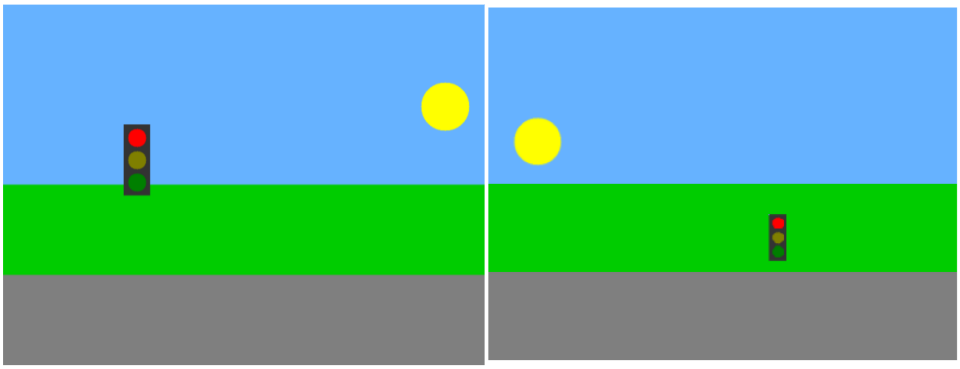
First off, you are given a generic traffic light to detect from a scene. For the sake of the problem, assume that traffic lights are shown as below: (with red, yellow, and green) lights that are vertically stacked. You may also assume that there is no occlusion of the traffic light.



((x-coordinate, y-coordinate), 'color')

It is your goal to find a way to determine the state of each traffic light and position in a scene. Position is measured from the center of the traffic light. Given that this image presents symmetry, the position of the traffic light matches the center of the yellow circle.
Complete your python ps2.py such that traffic_light_detection returns the traffic light center

coordinates (x, y) ie (col, row) and the color of the light that is activated ('red', 'yellow', or 'green'). Read the function description for more details.
Testing:
A traffic light scene that we will test will be randomly generated, like in the following pictures and examples in the github repo.



Functional assumptions:
For the sake of simplicity, we are using a basic color scheme, but assume that the scene may have different color objects and backgrounds [relevant for part 2 and 3]. The shape of the traffic light will not change, nor will the size of the individual lights relative to the traffic light. Size range of the lights can be reasonably expected to be between 10-30 pixels in radius. There will only be one traffic light per scene, but its size and location will be generated at random (that is, a traffic light could appear in the sky or in the road–no assumptions should be made as to its logical position). While the traffic light will not be occluded, the objects in the background may be.

Code:

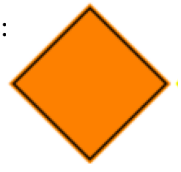
Complete traffic_light_detection(img_in, radii_range)

Report:

Place the coordinates using cv2.putText before saving the output images. Input: scene_tl_1.png. Output: ps2-1-a-1.jpg [5]

1.b. Construction sign one per scene [5 points]

Now that you have detected a basic traffic light, see if you can detect road signs. Below is the construction sign that you would see in the United States (apologies to those outside the United States).
Implement a way to recognize the signs:



'construction'

Similar to the traffic light, you are tasked with detecting the sign in a scene and finding the (x, y) i.e (col, row) coordinates that represent the **polygon's centroid**.
Functional assumptions:
Like above, assume that the scene may have different color objects and backgrounds. The size and location of the traffic sign will be generated at random. While the traffic signs will not be occluded, objects in the background may be.

Code:

Complete the following functions. Read their documentation in ps2.py for more details.

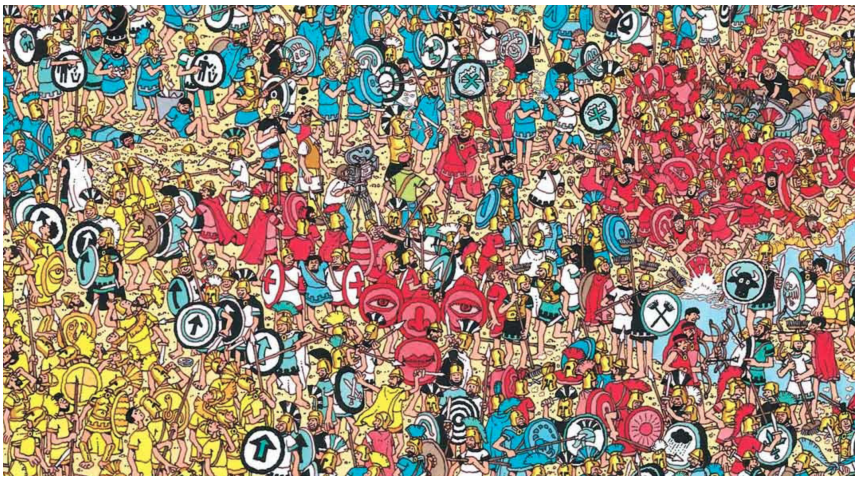
- construction_sign_detection(img_in)

Report:

Place the coordinates using c2.putText before saving the output images. Input: scene_constr_1.png. Output: ps2-1-b-1.jpg [5]

2. Template Matching [30 points]

Template matching is a common image processing technique that is used to find small parts of an image that match with the template image. In this section, we will learn how to perform template matching using different metrics to establish matching.
We will try to retrieve the traffic light and the traffic sign in Part 1, by using their templates. In addition, We have the image of a Waldo and another image where Waldo is hidden.



We will attempt to find Waldo by matching the Waldo template with the image using the following techniques:

- Sum of squared differences: tm_ssd
- Normalized sum of squared differences: tm_nssd
- Correlation: tm_ccor
- Normalized correlation: tm_nccor

We use the sliding window technique for matching the template. As we slide our template pixel by pixel, we calculate the similarity between the image window and the template and store this result in the top left pixel of the result. The location with maximum similarity is then touted a match for the template.

Code:

Complete the template matching function. Each method is called for a different metric to determine the degree to which the template matches the original image. You'll be testing on the traffic signs used in Part 1, and Suggestion : For loops in python are notoriously slow. Can we find a vectorized solution to make it faster?

Report:

Pick the best of the 4 methods to display in the report.
Input: scene_tl_1.png. Output: ps2-2-a-1.jpg [5]
Input: scene_constr_1.png. Output: ps2-2-b-1.jpg [5]
Input: waldo1.png. Output: ps2-2-c-1.jpg [5]

Text:

2d. What are the disadvantages of using Hough based methods in finding Waldo? Can template matching be generalised to all images? Explain Why/Why not. Which method consistently performed the best, why? [15]

3. Fourier Transform

In this section we will use the Fourier Transform to compress an image. The Fourier transform is an integral signal processing tool used in a variety of domains and converts a signal into individual spectral components (sine and cosine waves). Another way of thinking about this is that it converts a signal from the time domain to the frequency domain. While signals like audio are a 1-dimensional signal, we will apply the Fourier transform to images as a 2-dimensional signal. For more information on the Fourier Transform, lectures 2C-L1 and 2C-L2 provide a good overview.

1-Dimensional Fourier Transform

The Fourier transform can be computed in two different algorithmic ways: Discrete and Fast. In big O notation, the Discrete Fourier Transform (DFT) can be computed in $O(n^2)$ time while the Fast Fourier Transform can be computed in $O(n\log(n))$ time. In this assignment, we will implement the Discrete Fourier Transform.

The Discrete Fourier Transform can be defined as

$$F(k) = \sum_{x=0}^{x=N-1} f(x) e^{\frac{-i2\pi kx}{N}}$$

One way to calculate the Fourier Transform is a dot product between a coefficient matrix and the signal. Given a signal of length n, we define the coefficient matrix ($n \times n$) as

$$M_n(\omega) = \begin{bmatrix} 1 & 1 & 1 & 1 & ... & 1 \\ 1 & \omega & \omega^2 & \omega^3 & ... & \omega^{n-1} \\ 1 & \omega^2 & \omega^4 & \omega^6 & ... & \omega^{2(n-1)} \\ 1 & \omega^3 & \omega^6 & \omega^9 & ... & \omega^{3(n-1)} \\ 1 & \omega^j & \omega^{2j} & \omega^{3j} & ... & \omega^{(n-1)j} \\ . & . & . & . & . & . \\ . & . & . & . & . & . \\ . & . & . & . & . & . \\ 1 & \omega^{n-1} & \omega^{2(n-1)} & \omega^{3(n-1)} & ... & \omega^{(n-1)(n-1)} \end{bmatrix}$$

where j represents each row and ω is $e^{\frac{-i2\pi}{N}}$. The vector resulting from $M_n(\omega) \cdot f(x)$ is now your Fourier transformed signal! To compute the inverse of the fourier transform, ω is $e^{\frac{i2\pi}{N}}$ and $f(x) = \frac{1}{N} M_n(\omega) \cdot F(x)$.

Code:

Complete the following functions following the process above. Numpy matrix operations can be used to simplify the calculation but np.fft functions are not allowed in this section.

- dft(x)
- idft(x)

Report: No writeup for this section

2-Dimensional Fourier Transform

Now that we have computed the Fourier Transform for a 1-dimensional signal, we can do the same thing for a 2-dimensional image. Remember that the 2-dimensional Fourier transform simply applies the one-dimensional transform on each row and column.

Code:

You may use the functions from the last sections but np.fft functions are not allowed.

- dft2(x)
- idft2(x)

Report: No writeup for this section

4. Using the Fourier Transform for Compression [15 points]

Compression is a useful tool in all types of signal processing but especially useful for images. Lets say you take a picture of your dog that is 10 mb but want to reduce the file size and yet still maintain the quality of the image. One method of doing this is to convert the image into the frequency domain and keeping only the most dominant frequencies. For this section, we will

implement image compression and pseudocode for the algorithm is shown below.

Input: an image of shape $n \times n$ (img_bgr) and threshold percentage (t);
for *channel* $= b, g, r$ **do**
    Select channel of img_bgr as image. Convert the image to frequency domain;
    Sort all the values of frequency from greatest to least into a 1-dimensional array of length (n^2);
    Find the threshold value at the index calculated by tn^2;
    Mask the frequency image to keep all values greater than the threshold value;
    Convert the masked frequency image back into pixel values with the inverse Fourier transform;
    Set the channel of the new image to the masked filtered version;
    Set the channel of the frequency domain image to the masked version;
end
return the filtered image and the masked frequency domain image (each should have 3 channels)

Algorithm 1: Image Compression with Fourier Transform

To visualize the masked frequency image, be sure to shift all the low frequencies to the center of the image with np.fft.fftshift. Additionally, take $20 * log(abs(x))$ to properly visualize the frequency domain image.

Code:

- compress_img_fft(x)

This functions are used in: compression_runner()

Report:

Use threshold percentages of 0.1, 0.05, and 0.001. Display both the resulting image and the frequency domain image in the report for each.

Outputs: ps2-4-a-1.jpg, ps2-4-a-1-freq.jpg, ps2-4-a-2.jpg, ps2-4-a-2-freq.jpg, ps2-4-a-3.jpg, ps2-4-a-3-freq.jpg [15 points, 2.5 each]

5. Filtering with the Fourier Transform [35 points]

Now that we have seen how the Fourier Transform can be used for compression, we will now use the Fourier Transform as a low-pass filter. A low-pass filter similarly keeps all the low frequencies within the image and 0's out all the high frequency components. We will follow the process shown in lecture video 2C-L1-14 by first converting the image to the frequency domain, masking the spectral image with a circle of radius r, and converting the image back to pixel color

values. The pseudocode for the algorithm is given below:

Input: an image (img) and circle of radius r ;
for *channel* = *r*, *g*, *b* **do**
  Convert the image to frequency domain;
  Shift the spectral frequencies so that all low frequencies are in the center of the image (use np.fft.fftshift);
  Mask the frequency image with a circle of radius r, keeping all the low frequencies and removing all the high frequencies;
  Undo the phase shift with np.fft.ifftshift;
  Convert the masked frequency image back into pixel values with the inverse Fourier transform;
  Set the channel of the new image to the low pass filtered version;
  Set the channel of the frequency domain image to the masked version;
end
return the filtered image and the masked frequency domain image (each should have 3 channels)

Algorithm 2: Low Pass Filter with Fourier Transform

Code:

- low_pass_filter(img_bgr, r)

This function is used in: low_pass_filter_runner()
Report: Use radii of 100, 50, and 10. Display both the resulting image and the frequency domain image in the report for each.
Outputs: ps2-5-a-1.jpg, ps2-5-a-1-freq.jpg, ps2-5-a-2.jpg, ps2-5-a-2-freq.jpg, ps2-5-a-3.jpg, ps2-5-a-3-freq.jpg [15 points, 2.5 each]
5-b What are the differences between compression and filtering? How does this change the resulting image? [10 points]
5-c Given an image corrupted with salt and pepper pepper noise, what filtering method can effectively reduce/remove this noise? Also explain your choice of filtering method. Show some examples. [10 points]