EN.601.461/661 – Computer Vision
Spring 2020
Homework #1
Due: 11:59 PM, Sunday, February 23, 2020

Only basic Python, Numpy, and OpenCV functions are allowed, unless otherwise specified. This rule applies in particular with the use of OpenCV. Basic image IO functions such as `cv2.imread`, `cv2.imwrite` etc. are allowed. If you are unsure of an allowable function, please ask on Piazza before assuming! You will get no credit for using a "magic" function to answer any questions where you should be answering them. When in doubt, ASK!

About Piazza usage: Piazza is a great tool for collaboration and questions. However, **never post big chunks of source code as a public post**. Make use of this tool to talk about ideas, concepts and implementation details.

Please use python 3 for the programming problems. You will be provided with template Python files containing functions for you to complete. You can implement your own helper functions if necessary. However, do **NOT** change the input and output signature of the original functions in the instructions. For all functions that take or return images, make sure to handle the actual arrays. Do not pass filenames around.

We will use Gradescope for homework submission and grading. Detailed submission instructions will be released on Piazza soon.

Written Assignment

1) Consider a pinhole camera with perspective projection.

   a. (10 points) Consider a circular disk that lies in a plane parallel to the image plane. The disk does not necessarily lie on the optical axis. What is the shape of the image of the disk? Show your work and derive your equations.

   b. (10 points) In class, we discussed how to find the vanishing point of lines. Suppose I now consider a family of lines that all lie in a single plane. Where on the image would the vanishing points of ALL lines on the plane $Ax+By+Cz+D = 0$ lie? The coordinate frame is located at the pinhole with the z-axis pointing towards the image and the effective focal length is f.

First, start with a simple case where A=C=D=0 and B=1 which defines a plane extending horizontally from the optical center. Write down three different line directions in this plane and work out where they vanish. Include your work in your submission.

Second, do the same with the plane B=C=D=0 and A = 1. Again include your work in your submission.

c. (5 points) Now try to define a general relationship (this is challenging!) that relates the parameters of the plane to the parameters of the vanishing points of all lines that fall in that plane.

Programming Assignment

!!! Clarification on Coordinate Definitions:

For problem 1, in order to follow the definition of orientation used in the slides, we assume the origin is at the bottom left of the image, the x axis points to the right, while the y axis points upwards. Therefore, column indices directly correspond to x coordinates, but conversions on the row indices are needed to get the correct y coordinates.

For problem 2 and 3, simply use the column indices as x coordinates and the row indices as y coordinates. This corresponds to the convention where the origin is at the top left of the image, the x axis points to the right, while the y axis points downwards. This convention is more widely used in the vision community today.

!!! Include design choices and output images in your write-up (put in the same .pdf file with your written assignment):

(1) For each required function, list the main ideas behind your code and the choices you made. Examples include (but are not limited to):

- What is the threshold value you used to get a good binary image?
- What are the kernel values in your Sobel edge detector?

(2) For each required function, if the output is an image, include it in your write-up; if it is a number or a short array, report it in your write-up; if it is a long array (> 10 elements), report the first 10 elements in your write-up.

1) Our goal is to develop a vision system that characterizes two-dimensional objects in images. Given an image such as **two_objects.png**, we would like our vision system to determine how many objects are in the image and to compute their positions and orientations. This information is valuable as it can be used by a robot to sort and assemble manufactured parts.

The task is divided into three parts, a)-c), each corresponding to a Python function. You need to complete the code snippets marked with "`#TODO`" in **p1_object_attributes.py**.

In this problem, you are provided with a driver program in the `main` function. The code can be run with the following command, which specifies the image to process and the thresh_val described in a):

```
python3 p1_object_attributes.py two_objects 128
```

a. (5 points) Write a Python function that converts a gray-level image to a binary one using a threshold value. The binary image should be 255 if intensity >= thresh_val else 0.

```
def binarize(gray_image, thresh_val):
  # TODO
  return binary_image
```

b. (15 points) Implement the sequential labeling algorithm that segments a binary image into connected regions:

```
def label(binary_image):
  # TODO
  return labeled_image
```

Note that you may have to make two passes of the image to resolve possible equivalences in the labels. In the "labeled" output image each object region should be painted with a different gray-level: the *gray-level assigned to an object is its label*. The labeled images can be displayed to check the results produced by your program. Note that your program must be able to produce correct results given any binary image. You can test it on the images given to you.

Note: You can either implement the two-pass version of the algorithm in the notes, or a (somewhat simpler but slower) recursive algorithm as described here: https://courses.cs.washington.edu/courses/cse373/00au/chcon.pdf

c. (15 points) Write a Python function that takes a labeled image and computes a list of object attributes (which in a real application could be used to locate and identify an object).

```
def get_attribute(labeled_image):
  # TODO
  return attribute_list
```

Each element of the attribute list should be a dictionary with the following keys: position, orientation, and roundedness. The position should be a dictionary with keys: 'x' and 'y'. The origin is defined as the bottom left pixel of the image. Please use radian for orientation, and all numbers are floats.

The program will write the images to files and print their properties. Test your function on the images **many_objects_1.png** and **many_objects_2.png.**

2) Your task here is to implement Hough transform to find circles in an image. We provide **data/coins.png** as an example image. You need to complete the code snippets marked with "`#TODO`" in **p2_hough_circles.py**. In this problem, you need to implement your own driver program to load images and save results.

   a. (10 points) First you need to find the locations of edge points in the image. Complete function `detect_edges`. The input to the function is a 2 dimensional uint8 array representing a grayscale image. The output should be a 2 dimensional float array where the intensity at each point is proportional to the edge magnitude. The output array should have the same size as the input array (pad input image with zero when necessary).

      Use Sobel masks to implement the edge detector. You may **NOT** use existing edge detectors in OpenCV. However, you can compare your output to the output of those detectors for verification. (In fact, this is a good idea to test your code!)

```
def detect_edges(image):
  # TODO
  return edge_image
```

   b. (15 points) Next, you need to implement the Hough Transform for circle detection. Complete the function `hough_circles`. The first input argument of the function is the edge magnitude map you got from your edge detector. The second argument is a threshold on the edge magnitude. Thresholding your edge magnitude map with this value gives a boolean map indicating whether each pixel is an edge point or not. Experiment with different threshold values so only strong edges are kept.

      We represent a circle with its center coordinates (*x*, *y*) and its radius *r*. Therefore, your accumulator array should be 3 dimensional. We assume that the *x* and *y* ranges are the same as the input image and the resolutions are both 1 pixel. The radius range and resolution are specified by the third input argument of the function, which is an array of possible radius values, e.g. [20, 21, … , 39, 40].

The function should return the thresholded edge image and the accumulator array. Experiment with different edge threshold. In your driver program, save the output edge image to **output/coins_edges.png** when you are satisfied with the result.

```
def hough_circles(edge_image, edge_thresh, radius_values):
  # TODO
  return thresh_edge_image, accum_array
```

c. (8 points) To find circles in the image, scan through the accumulator array looking for parameter values that have high votes. Complete function `find_circles`. Here again, a threshold must be used to distinguish strong circles.

The function should return a list of 3-tuples where each element (*r*, *y, x*) represents the radius and the center coordinates of a circle found by your program.

Besides, draw these circles on a copy of the original image using OpenCV function `cv2.circle` (with `color=(0, 255, 0)` and `thickness=2`). Return the resulting image as well. Experiment with different vote threshold. In your driver program, save the output image to **output/coins_circles.png** when you are satisfied with the result.

```
def find_circles(image, accum_array, radius_values,
                 hough_thresh):
  # TODO
  return circles, circle_image
```

3) Your task here is to implement normalized cross correlation for simple template matching. We provide **data/face.png** and **data/letter.png** as templates, while **data/king.png** and **data/text.png** are images to be matched. You need to complete the code snippets marked with "`#TODO`" in **p3_template_matching.py**. In this problem, you need to implement your own driver program to load images and save results.

a. (10 points) Implement normalized cross-correlation in function `normxcorr2`.

The function should assume that input images are 2-dimensional arrays where each element is a floating number between 0.0 and 1.0. If you load the images as 3-channel color images in your driver program, don't forget to convert them to grayscale and map the values to the correct range before passing them to the function.

When dealing with image boundaries, there are several commonly used styles: “full”, “valid”, and “same” (see this page for more explanation). To make things simpler, here we use the “valid” style and do not pad the search image. In other words, calculation is performed only at locations where the template is fully inside the search image.

The function should return a 2-dimensional float array representing the correlation map of matching scores.

```
def normxcorr2(template, image):
  # TODO
  return scores
```

b. (5 points) Use your normalized cross-correlation function to find where the face in **face.png** appears in **king.png**.

Complete function `find_matches`. This function should take a template image and a search image, both as a uint8 color image. Ignore the `thresh` argument for now. Make a copy of the input images, do the pre-processing described in (a), and call `normxcorr2` you just implemented to compute the matching scores. After you have the scores, find the best match and determine where in the original image it corresponds to. Return a 2-tuple (x, y) representing the coordinates of the upper left corner of the matched region.

In addition, visualize the matching result by making a copy of the original search image and drawing a box around the matched region using OpenCV function `cv2.rectangle` (with `color=(0, 255, 0)` and `thickness=2`). You can assume that the template size is the same as the region it’s expected to be matched. Therefore, you can use the height and width of the template image as the rectangle size. Return the resulting image as well. In you driver program, save the output image to **output/king.png**.

```
def find_matches(template, image, thresh=None):
  # TODO
  return coords, match_image
```

c. (5 points) Use your normalized cross-correlation function to find all occurrences of **letter.png** in **text.png**.

Extend your `find_matches` function so it has exactly the same behavior as in (b) when `thresh=None` but finds multiple matches when given a threshold. To be specific, when given a threshold, the function should return a list of 2-tuples representing all matches together with the visualization result. Experimenting with different threshold values. In you driver program, save the output image to **output/text.png** when you are satisfied with the results.

```
def find_matches(template, image, thresh=None):
  # TODO
  return coords, match_image
```