EN.601.461/661 – Computer Vision
Fall 2020
Homework #1
Due: 11:59 PM, Sunday, October 4, 2020

All solutions (e.g. your code, write-ups, output images) should be zipped up as HW1_yourJHED.zip and submitted on **Gradescope**, where 'yourJHED' is your JHED ID (e.g. ghager1). If you have hand-written the written assignment, please make a scan and attach the .pdf in the submission zip.

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 writing code to answer 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.

Written Assignment (35 pts)

1) Consider a pinhole camera with perspective projection.

    a. (10 points) In class we talked about approximating perspective projection by assuming a constant depth. Let's assume we have a camera with a 10mm lens, a target at a distance of 1 meter, and we're viewing an object that has a depth (distance front to back) of Xmm centered at 1 meter.
        i. How large can we make X with less than 5% difference in the projection between true perspective and the linear approximation?
        ii. We talked about depth of field and blur related to defocus. Let's assume our image sensor has 100 pixels/mm. Can you estimate how large the diameter of the aperture can be to ensure that the entire range of distances you calculated above would be in focus?

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. Intuitively, all of the lines will vanish on a common line (also known as the horizon line). What is the equation for this line as a function of the plane?
   i. Recall a plane can be written in the form $Ax+By+Cz+D = 0$, where (A,B,C) is a unit vector. The coordinate frame is located at the pinhole with the z-axis pointing towards the image; you may assume a unit focal length and disregard that term in the projection equations.
   ii. Work out a simple case where B=C=D=0 and A=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.
   iii. Second, do the same with the plane B=C=D=0 and A = 1. Again include your work in your submission.

c. (5 points) Following on the above, define a general relationship that relates the parameters of the plane to the parameters of the vanishing points of all lines that fall in that plane. This is challenging, but a good change to refresh your geometry and algebra skills in anticipation of the next section of the course.

2) (10 pts) In the slides for class, we showed the Fourier transforms for sin(k x) and cos(k x) but we never looked at the case where there was an offset of the form sin(kx + b) where b is not a multiple of 2π. Show that we can recover b using the atan of the imaginary and real components of the Fourier transform (hint, recall the trig expansion for sin(a + b)).

Programming Assignment (65 pts)

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, to compute the type of shape they are, and to compute their positions and orientations.

   The task is divided into three parts, a and b, each corresponding to a Python function. You need to complete the code snippets marked with "`#TODO`" in **p1n2.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 p1n2.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. (10 points) Write a Python function that takes a labeled image and computes a list of object global shape 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 upper left pixel of the image. Please use radians for orientation, and all numbers are floats.

2) These global attributes are not enough to distinguish e.g. a triangle from a rectangle from a circle. We will now extend the exercise above with a Hough transform method to detect straight lines that are part of a shape.

a. (5 pts) 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 binary image with the edges detected. You should use a derivative of Gaussian filter and threshold the magnitudes. You should make the filter sigma value and threshold parameters:

```
def detect_edges(image, sigma, threshold):
  # TODO
  return edge_image
```

(5 points extra credit: implement hysteresis thresholding, in which case your call will have an upper and a lower threshold)

b. (10 points) Next, you need to use the Hough transform to detect lines and associate them with the objects they came from. For this purpose you can use the opencv function HoughLines. This function will return the coordinates of lines found in the images. You will need to go back and figure out which edge pixels associate with these lines (giving you a line segment) and then associate them with a figure from part 1. That

is, if you have a rectangle, you should find four line segments and associate that fact with the information you computed in part 1 for that rectangle. To do so, add an array of line parameters consisting of angle, distance from origin, and length (in pixels) to the dictionary for each object of part 1. For the length, the simplest approach is to use the line equation, identify edge pixels with a given distance of the line, and count. Adding a fitting step and some connected component analysis will possibly provide better results.

```
def get_edge_attribute(labeled_image, edge_image):
  # TODO
  return attribute_list
```

(Extra Credit (5 points): Do the same to find circles using HoughCircles and add this to your edge attributes).

3) Test your functions.
   a. (5 pts) First, test your functions on a set of shapes in the images **many_objects_1.png** and **many_objects_2.png.** The output should be the list of attributes of each object in the image computed by 1 and 2 above.

   b. (10 pts) Now, use the main program we have provided to load the images in a training data folder we have created. Each image contains one object. Build a data structure that stores the name of the file along with the attributes of the object found in the image. Implement a matching function:

```
def best_match(object_database, test_object)
  # TODO
  return object_name
```

This function will accept the list of objects that were in the training data folder, and return the best matching object. It is up to you to decide how to match objects -- whether by size, roundedness, number of lines, the length of the lines, etc.

Our driver program will call your match function and print and display your match for the images in the test files **many_objects_1.png** and **many_objects_2.png.** Note that for grading purposes, we will also test your functions on held out data as well, so don't overtune your function to just work on our supplied test data!

4) 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.

```
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 your 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
```