Problem Set 2: Edges and Lines

Description

This problem set is your first "vision" project where you compute an "answer" – that is, some structural or semantic description as to what is in an image. You'll find edges and objects. And you'll learn that some methods work well for carefully controlled situations and hardly at all when you relax those constraints. Note that Autograded problems will be marked with a (*).

RULES: You may use image processing functions to find edges, such as Canny or other operators. Don't forget that those have a variety of parameters and you may need to experiment with them. ***BUT: YOU MAY NOT USE ANY HOUGH TOOLS.*** For example, you need to write your own accumulator array data structures and code for voting and peak finding.

Please do not use absolute paths in your submission code. All paths should be relative to the submission directory. We recommend using the *os* python library to define paths. Any submissions with absolute paths are in danger of receiving a penalty! Additionally, make sure that these parameters are compatible with all three platforms (windows, mac os, and linux).

Obtaining the Starter Files:

Clone the problem set files:

```
git clone https://github.gatech.edu/omscs6476/ps02.git
```

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.

Write-up Instructions

Create **ps02_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 PS2 results, please refer to the template for PS2 here: PS2 Template **We require PDF only and will not accept a word document or any other format**.

We also provide a LaTeX template for PS2: PS2 LaTeX Template. You can also download the template using git clone: https://git.overleaf.com/7777631qzxhqrdvgmwr

How to submit

To submit your code, in the terminal window run the following command:

```
python submit.py ps02
```

To submit the report and experiment.py, in the terminal window run the following command:

```
python submit.py ps02_report
```

YOU MUST SUBMIT your report separately, i.e., two submissions for the code and the report, respectively. Your last submission before the deadline will be counted for each of the code and the report.

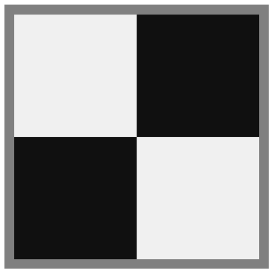
You should see the autograder's feedback in the terminal window. Additionally, you can look at a history of all your submissions at https://bonnie.udacity.com/

Questions

1. Finding Lines with Tool [5 pts]

> For reference, look at the documentation for OpenCV in Python and read about edge operators. Use one operator of your choosing – for this image, it probably will not matter much which one is used. If your edge operator uses parameters (like Canny), play with those until you get the edges you would expect to see. The tool provided `find_edges.py` lets you experiment with the parameters of your edge image and may be very useful for later questions. Use this with a temporary image that we provided for you, ps2-input-flowergirl

For this question we will use input/ps2-input0.png:



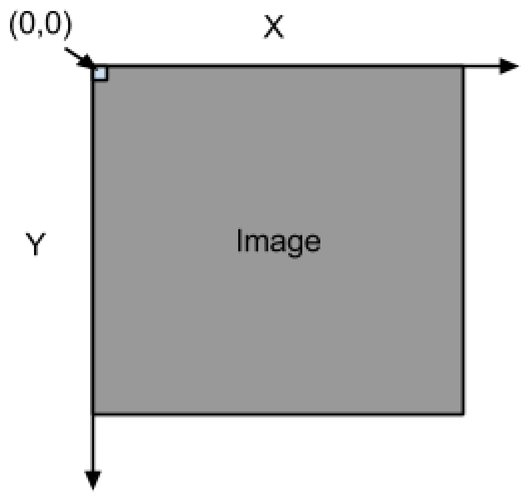
This is a test image for which the answer should be clear, where the "object" boundaries are only lines.

a. Load the input grayscale image at **ps2-input0.png** as img and generate an edge image, which is a binary image with white pixels (1, or 255) on the edges and black pixels (0) elsewhere. For this question, you will create an edge image.

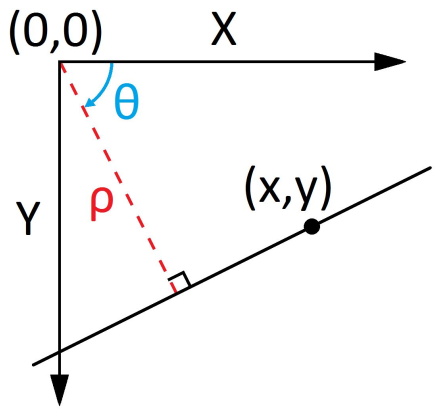
Report: Store edge image (img_edges) as **ps2-1-a-1.png**

2. (*) Implementing Hough Transform Line method [20 pts]

> Note that the coordinate system used is as pictured below, with the origin placed at the upper-left pixel of the image and with the Y-axis pointing downwards.



Thus, the pixel at img(r,c) corresponds to the (x,y) coordinates (r,c), i.e., x = c and y = r. This pixel should vote for line parameters (ρ , θ), where: $\rho = x \cdot \cos(\theta) + y \cdot \sin(\theta)$.



This has the effect of making the positive angular direction *clockwise* instead of *counterclockwise* in the usual mathematical convention. Theta (θ) = zero still points in the direction of the positive X-axis.

For the following parts, you may use the OpenCV function cv2.HoughLines() as a reference. Here is a tutorial that shows how to use the function and interpret its output. You may use the above mentioned function for comparing final outputs, but the code for your functions **must be your own**.

a. Write a function **`hough_lines_acc()`** that takes a binary edge image and computes the Hough Transform for lines, producing an accumulator array. Note that your function should have two optional parameters: rho_res (ρ resolution in pixels) and theta_res (θ resolution in degrees converted to radians), and your function should return three values: the hough accumulator array H, rho (ρ) values that correspond to rows of H, and theta (θ) values one for each column of H.

   Apply it to the edge image (img_edges) from question 1:

   ```
   H, rho, theta = hough_lines_acc(img_edges, rho_res, theta_res)
   ```

   Code: implement `hough_lines_acc()`

Report: place the hough accumulator array (H), as **ps2-2-a-1.png** in the writeup (note: write a normalized uint8 version of the array so that the minimum value is mapped to 0 and maximum to 255).

b. Write a function **hough_peaks()** that finds indices of the accumulator array (here, line parameters) that correspond to local maxima. It should take an additional parameter `hough_threshold` indicating the minimum pixel value that the function will consider when inspecting the array. Also this function used `nhood_delta` which determines the area where the non-maxima suppression occurs. This parameter is a tuple ($\Delta r,\ \Delta c)$ that will define an area of shape $2 * \Delta r \times 2 * \Delta c$. Any other peaks within this area will not be counted as part of the result.

Note that you need to return a Qx2 array (where Q is the number of peaks found in H) each row is a (rho_id, theta_id) pair. (This could be used for general peak finding.)

Call your function with the accumulator from the step above to find the best peaks, with no redundancies:

```
peaks = hough_peaks(H, hough_threshold, nhood_delta)
```

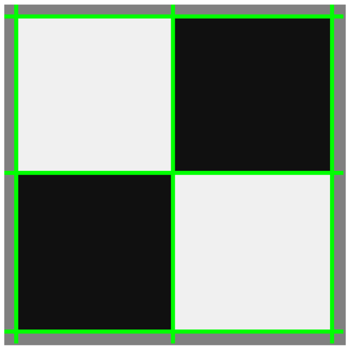
Code: implement hough_peaks()
Report: place your image of the accumulator array with peaks CLEARLY HIGHLIGHTED, as **ps2-2-b-1.png** in the writeup

c. Write a function **hough_lines_draw()** (located in experiment.py) to draw color lines that correspond to peaks found in the accumulator array. This means you need to look up the rho and theta values using the peak indices, and then convert them (back) to line parameters in cartesian coordinates (you can then use regular line-drawing functions).

Use this to draw lines on the original grayscale (not edge) image:

```
hough_lines_draw(img_in, peaks, rho, theta)
```

The image should look something like the image below. The lines should extend to the edges of the image (aka infinite lines). You might get lines at the boundary of the image too depending upon the edge operator you selected (but those really shouldn't be there).



Code: implement `hough_lines_draw()`
Report: place the original grayscale image with lines drawn, as **ps2-2-c-1.png**

3. Noisy Line Finding [10 pts]

Now we're going to add some noise.

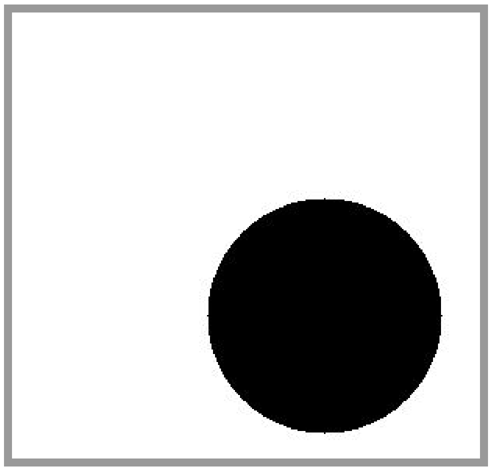
a. Use **ps2-input0-noise.png**, which is the same scene as before, but with added noise. Compute a modestly smoothed version of this image by using a Gaussian filter. Make σ at least a few pixels big.
Report: place the Smoothed image, as **ps2-3-a-1.png** in the writeup

b. Using an edge operator of your choosing, create a binary edge image for both the original image (ps2-input0-noise.png) and the smoothed version above.
Report: place two edge images: **ps2-3-b-1.png** (from original), **ps2-3-b-2.png** (from smoothed) in the writeup

c. Now apply your Hough method to the smoothed version of the edge image. Your goal is to adjust the filtering, edge finding, and Hough algorithms to find the lines as best you can in this test case.
Report: place in the writeup:
- Hough accumulator array image with peaks clearly highlighted, as **ps2-3-c-1.png**
- Intensity image (original one with the noise) with lines drawn on them, as **ps2-3-c-2.png**

4. (*) Finding Circles [20 pts]

For this question we will use input/ps2-input0-circle.png:



a. Now write a circle finding version of the Hough transform. For the first part, we wish for you to implement the single-point method.

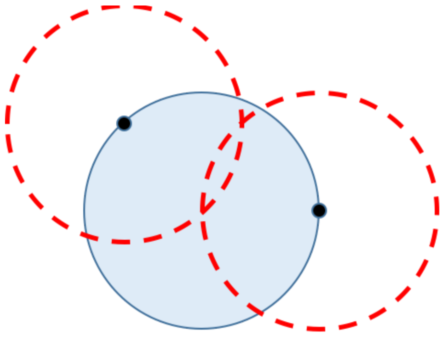
Please refer to the lecture on Detecting Hough Circles. The basic idea is that the Hough transform can be used to determine the parameters of a circle when a number of points that fall on the perimeter are known. A circle with radius R and center (a, b) can be described with the parametric equations x = a + R cos(θ) y = b + R sin(θ) When the angle θ sweeps through the full 360 degree range the points (x, y) to trace the perimeter of a circle. If an image contains many points, some of which fall on perimeters of circles, then the job of the search program is to find parameter triplets (a, b, R) to describe each circle so that the center of the radius can be determined.

If you find your arrays getting too big (hint, hint) you might try to make the range of radii very small to start with and see if you can find one size circle. Then, maybe try the different sizes.

For the following parts, you may use the OpenCV function cv2.HoughCircles() for reference (here is a tutorial on how to use it). You may use the above mentioned function for comparing final outputs, but the code for your functions **must be your own**.

i. Implement **`hough_circles_acc()`** to compute the accumulator array for a given radius of the circle above. Using the original image test_circle.png (monochrome) as above, smooth it, find the edges, and try calling your function with radius = 75:

```
H = hough_circles_acc(img_orig, img_edges, 75, False)
```



```
For every edge pixel (x,y):
   For every θ:
      a = x – r cos(θ)
      b = y + r sin(θ)
      H[a, b] += 1
   end
end
```

The main idea is that for every edge pixel you will create circles centered in this position. Given that we know the radius to be used, your algorithm should implement a for loop similar to the one shown above. This should return an accumulator H of the same size as the supplied image. Each *pixel* value of the accumulator array should be proportional to the likelihood of a circle of the given radius being present (centered) at that location. Find circle centers by using the same peak finding function.

Code: implement `hough_circles_acc()`
Report: place in the writeup:
- Edge image, as **ps2-4-a-1.png**
- Hough Accumulator image with peaks highlighted, as **ps2-4-a-2.png**

b. Next, we will update our circle finding version of the Hough transform. For the second part, we wish for you to implement the point-plus gradient method.

Please refer to the lecture on Hough Transform for Circles. Using the gradient direction will reduce the amount of votes you need to calculate. Just like for lines, you can determine the center with gradients. If you know the radius size, given a gradient, there is only possible size circle that can be portrayed. Therefore, you can test for a center point tangential to the gradient. The point-plus method has voting for only one line.

i. Update the function `hough_circles_acc()` using the point-plus gradient method discussed in the lectures. Using the original image (monochrome) as above, smooth it, find the edges, and try calling your function with radius = 75:

```
H = hough_circles_acc(img_orig, img_edges, 75, True)
```

Code: implement updated `hough_circles_acc()`
Report: place in the writeup the original monochrome image with the circles drawn in color, as **ps2-4-b-1.png**

5. (*) Coins and Pens [20pts]

For this question, please use **ps2-input1.png**

a. This image has objects in it whose boundaries are circles (coins) or lines (pens). For this question, you are still focused on finding lines. Load/create a monochrome version of the image (you can pick a single color channel, or use a built-in color-to-grayscale conversion function). Create the smoothed edge image as you did in previous questions. Apply your Hough algorithm to the edge image to find lines along the pens. Draw the lines in color on the original monochrome (i.e., NOT the edge) image. The lines can be drawn to extend to the edges of the original monochrome image.

Report: place in the writeup
- Hough accumulator array image with peaks highlighted, as **ps2-5-a-1.png**
- Original monochrome image with lines drawn on it, as **ps2-5-a-2.png**

b. Using **hough_circles_acc()**, compute the accumulator array for a radius=23.

   This should return an accumulator H of the same size as the supplied image. Each *pixel* value of the accumulator array should be proportional to the likelihood of a circle of the given radius being present (centered) at that location. Find circle centers by using the same peak finding function:

   ```
   centers = hough_peaks(H)
   ```

   Report: place in the writeup the original monochrome image with the circles drawn in color, as **ps2-5-b-1.png**

c. Implement a function **find_circles()** that searches for circles within a given (inclusive) radius range, and returning circle centers along with their radii, e.g., for 20 to 50:

   ```
   circles = find_circles(img_orig, img_edges, radii, hough_threshold,
                                  nhood_delta)
   ```

   Note, you will be awarded full points for finding **all** the circles.

   Code: implement `find_circles()`
   Report: place in the writeup the Original monochrome image with the circles drawn in color, as **ps2-5-c-1.png**

6. Clutter Images of Lines [10 pts]

   More realistic images. Now that you have Hough methods working, we're going to try them on images that have *clutter*--visual elements that are not part of the objects to be detected. Use: **ps2-input2.png**

   a. Apply your line finder. Use whichever smoothing filter and edge detector that seems to work best for finding all pen edges. Don't worry (until 6b) about whether you are finding a few lines as well. You output image should show the minimum amount of lines that include the pen edges length-wise.
      Report: place the original image with the Hough lines drawn on them, as **ps2-6-a-1.png**

   b. Find only the lines that are the *boundaries* of the pen (2 lines for each pen). Three operations you need to try are: better thresholding in finding the lines (look for stronger edges); checking the minimum length of the line; and looking for nearby parallel lines.
      Report: place the original image with the new Hough lines drawn, as **ps2-6-b-1.png**

   c. Now Find circles on the same clutter image of **ps2-input2.png**. Apply your circle finder using the gradient method. Use a smoothing filter that you think seems to work best in terms of finding all the coins. You will be awarded full points for finding **all** the coins without overlapping circles or false positives.
      Report: place Smoothed image you used with the circles drawn on them, as **ps2-6-c-1.png**

7. Discussion [10 pts]

   Report: Answer the questions below in the writeup.

a. For each of the methods, what sorts of parameters did you use for finding lines in an image? Circles? Did any of the parameters radically change? (Describe your accumulator bin sizes, threshold, and neighborhood size parameters for finding peaks, and why/how you picked those.)
b. What differences do you see when finding the edges in a noisy vs. regular image?

c. Is there any perceived difference in time or computational cost between the single-point and the point-plus gradient method implementations?

d. For question 5, were you able to find all the circles? Describe what you had to do to find circles.

e. In question 6, for a cluttered image, you likely found lines that are not the boundaries of the pen. What problems did you face to overcome this?

f. Likewise in question 6, there may have been false positives for circles. Did you find such false positives? How would/did you get rid of them? If you did these steps, mention where they are in the code by file, line no., and also include brief snippets

8. CHALLENGE PROBLEMS [5 pts]

For the Challenge problems, we will ask you to improve your algorithm for its application in the real world. In this challenge, please pick one of the following options to implement:
a. Computer vision is often used to detect objects and extract meaning from an image. For this, please extract information from the image to calculate the value of the US currency by implementing a new function that evaluates the circle radius (and any other information you would need). Here are some test images you can use US coins.
b. The line finder is currently detecting the presence of any line. However, we want you to detect and draw only the boundaries of the pen as a line segment. Implement a new draw method that does this and draws to the extent of the pen.

Describe in detail what you did to accomplish your results in the report.