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.

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.

What to submit

Download and unzip: **ps2.zip**

It should have the following structure and contents:

ps2/

- **input/** - input images, videos or other data supplied with the problem set
  - ps2-input0.png
  - ps2-input0-noise.png
  - ps2-input1.png
  - ps2-input2.png
  - ps2-input3.png
- **output/** - directory containing output images and other files that your code generates
  Note: Output images must be stored with following mandatory naming convention:

  ps<problem set #>-<question #>-<part>-<counter>**.png**

  Example: ps2-1-a-1.png (first output image for question 1-a)
- **ps2.py** - main code for completing each part - this will be run by our grading script as:

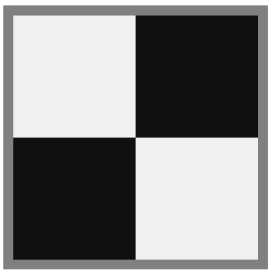
  python ps2.py

  You may include additional .py files with functions, classes, etc. and import them in your main code.
- **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).

Zip it as **ps2.zip**, and submit on T-Square.

Questions

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



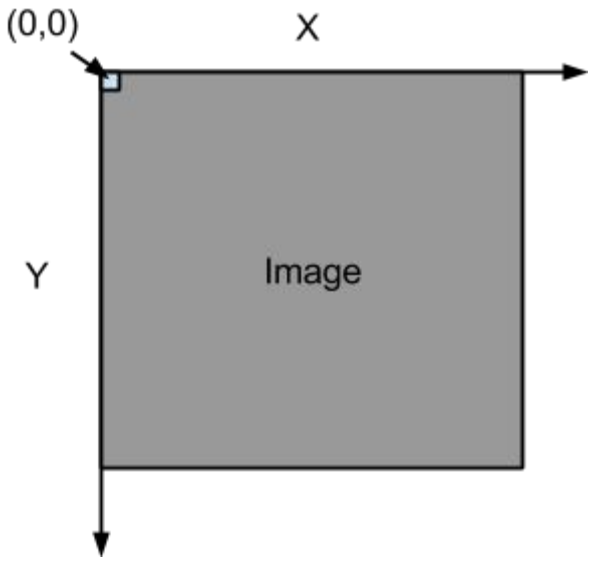
This is a test image for which the answer should be clear, where the "object" boundaries are only lines.

   a. Load the input grayscale image (`input/ps2-input0.png`) as `img` and generate an edge image – which is a binary image with white pixels (1) on the edges and black pixels (0) elsewhere.

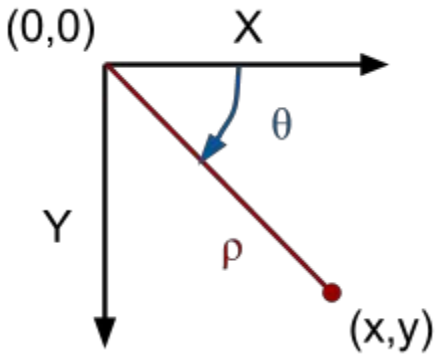
      For reference, look at the documentation for OpenCV in Python and read about edge operators. Use one operator of your choosing – for this image, it probably will not matter much which one is used. If your edge operator uses parameters (like Canny), play with those until you get the edges you would expect to see.

      Output: Store edge image (`img_edges`) as `ps2-1-a-1.png`

2. Implement a Hough Transform method for finding lines. 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: ρ `= x·cos(`θ`) + y·sin(`θ`),` and θ `= atan2(y,x)`.



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 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 that correspond to columns of H.

   Apply it to the edge image (img_edges) from question 1:
   ```
   H, rho, theta = hough_lines_acc(img_edges)
   ```
   Or, with one optional parameter specified (θ resolution = π /180, i.e. 1 radian):
   ```
   H, rho, theta = hough_lines_acc(img_edges, theta_res=pi/180)
   ```

   Function: `hough_lines_acc()`
   Output: Store the hough accumulator array (H) as ps2-2-a-1.png (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 Q (integer >= 1) indicating the (maximum) number of peaks to find, and return indices for up to Q peaks.

   Note that you need to return a Px2 matrix (where P≤Q) where each row is a (rho_idx, theta_idx) pair, where ρ = rho[rho_idx] and θ = theta[theta_idx]. (This could be used for general peak finding.)

   Call your function with the accumulator from the step above to find up to 10 strongest lines:
   ```
   peaks = hough_peaks(H, 10)
   ```

   Function: `hough_peaks()`

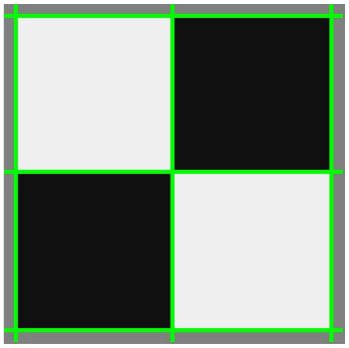
Output: `ps2-2-b-1.png` - like above, with peaks highlighted (you can use drawing functions).

c. Write a function **`hough_lines_draw()`** to draw color lines that correspond to peaks found in the accumulator array. This means you need to look up rho, 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. The lines should extend to the edges of the image (aka infinite lines):

```
hough_lines_draw(img_out, peaks, rho, theta)  # img_out is 3-channel
```

Function: `hough_lines_draw()`
Output: `ps2-2-c-1.png` - output of `hough_lines_draw()`.
It should look something like this:



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).

d. What parameters did you use for finding lines in this image?
Output: Text response describing your accumulator bin sizes, threshold, and neighborhood size parameters for finding peaks, and why/how you picked those.

3. Now we're going to add some noise.
   a. Use `ps2-input0-noise.png` - same image as before, but with noise. Compute a modestly smoothed version of this image by using a Gaussian filter. Make σ at least a few pixels big.
   Output: Smoothed image: `ps2-3-a-1.png`

   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.
   Output: Two edge images: `ps2-3-b-1.png` (from original), `ps2-3-b-2.png` (from smoothed)

   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.
   Output:
   - Hough accumulator array image with peaks highlighted: `ps2-3-c-1.png`

- Intensity image (original one with the noise) with lines drawn on them: `ps2-3-c-2.png`
- Text response: Describe what you had to do to get the best result you could.

4. For this question 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), and compute a modestly-smoothed version of this image by using a Gaussian filter. Make σ at least a few pixels big.
   Output: Smoothed monochrome image: `ps2-4-a-1.png`

   b. Create an edge image for the smoothed version above.
   Output: Edge image: `ps2-4-b-1.png`

   c. 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 edge) image. The lines can be drawn to extend to the edges of the original monochrome image.
   Output:
   - Hough accumulator array image with peaks highlighted: `ps2-4-c-1.png`
   - Original monochrome image with lines drawn on it: `ps2-4-c-2.png`
   - Text response: Describe what you had to do to get the best result you could.

5. Now write a circle finding version of the Hough transform. You can implement either the single point method or the point plus gradient method. **WARNING: This part may be hard!!! Leave extra time!**

   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**.

   a. Implement **`hough_circles_acc()`** to compute the accumulator array for a given radius. Using the same original image (monochrome) as above (`ps2-input1.png`), smooth it, find the edges (or directly use edge image from 4-b above), and try calling your function with radius = 20:

   ```
   H = hough_circles_acc(img_edges, 20)
   ```

   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, 10)
   ```

   Function: `hough_circles_acc()` (`hough_peaks()` should already be there)
   Output:
   - Smoothed image: `ps2-5-a-1.png` (this may be identical to `ps2-4-a-1.png`)

- Edge image: `ps2-5-a-2.png` (this may be identical to `ps2-4-b-1.png`)
- Original monochrome image with the circles drawn in color: `ps2-5-a-3.png`

b. Implement a function **`find_circles`**`()` that combines the above two steps, searching for circles within a given (inclusive) radius range, and returning circle centers along with their radii:

```
centers, radii = find_circles(img_edges, (20, 50))
```

Function: `find_circles()`
Output:
- Original monochrome image with the circles drawn in color: `ps2-5-b-1.png`
- Text response: Describe what you had to do to find circles.

6. 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 other lines as well.
   Output: Smoothed image you used with the Hough lines drawn on them: `ps2-6-a-1.png`

   b. Likely, the last step found lines that are not the boundaries of the pens. What are the problems present?
   Output: Text response

   c. Attempt to find only the lines that are the *boundaries* of the 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.
   Output: Smoothed image with new Hough lines drawn: `ps2-6-c-1.png`

7. Finding circles on the same clutter image (`ps2-input2.png`).
   a. Apply your circle finder. Use a smoothing filter that you think seems to work best in terms of finding all the coins.
   Output: the smoothed image you used with the circles drawn on them: `ps2-7-a-1.png`

   b. Are there any false positives? How would/did you get rid of them?
   Output: Text response (if you did these steps, mention where they are in the code by file, line no., and also include brief snippets)

8. Sensitivity to distortion. There is a distorted version of the scene at `ps2-input3.png`.
   a. Apply the line and circle finders to the distorted image. Can you find lines? Circles?
   Output: Monochrome image with lines and circles (if any) found: `ps2-8-a-1.png`

   b. What might you do to fix the circle problem?
   Output: Text response describing what you might try

   c. EXTRA CREDIT: Try to fix the circle problem (**THIS IS HARD**).
   Output:

- Image that is the best shot at fixing the circle problem, with circles found: `ps2-8-c-1.png`
- Text response describing what tried and what worked best (with snippets).