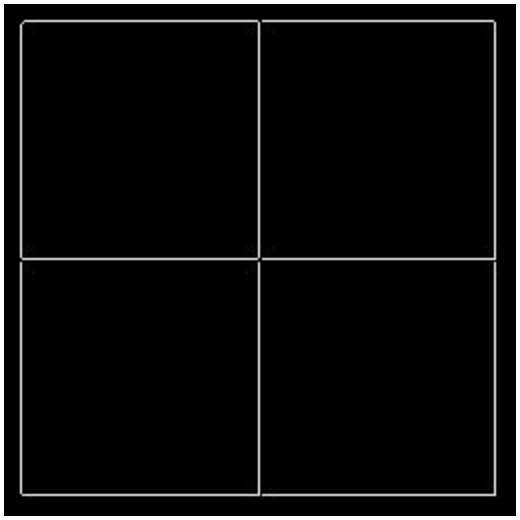
# Problem Set 2: Edges and Lines

## Questions

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

       Edge image:

       `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.

    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` ($\rho$ resolution in pixels) and `theta_res` ($\theta$ resolution in radians), and your function should return three values: the hough accumulator array H, rho ($\rho$) values that correspond to rows of H, and theta ($\theta$) values that correspond to columns of H.

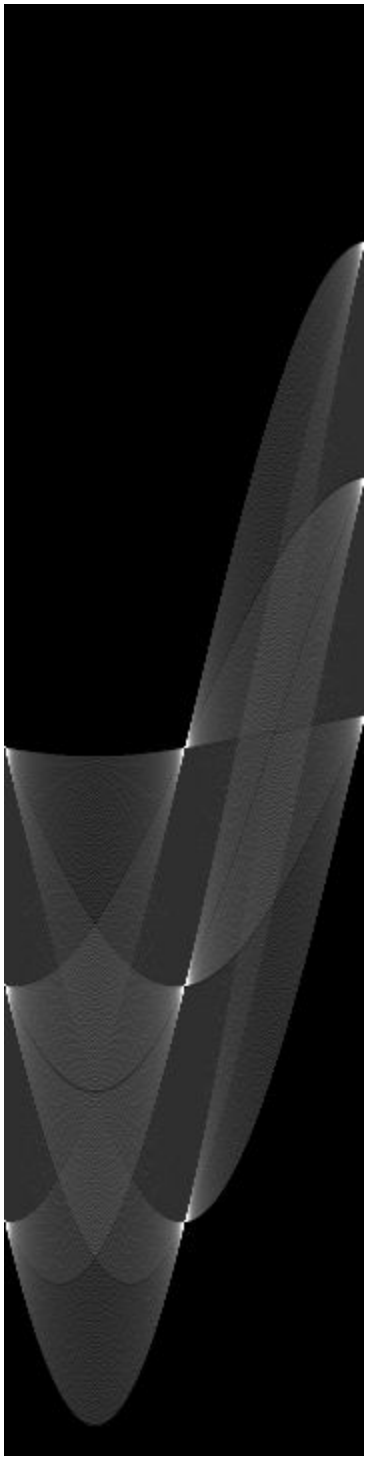
       Apply it to the edge image (img_edges) from question 1:

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

       Or, with one optional parameter specified ($\theta$ resolution = $\pi$/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).



Accumulator ps2-2-a-1.png (color adjusted)

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 $\rho$ = rho[rho_idx] and $\theta$ = 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).

ps2-2-b-1.png

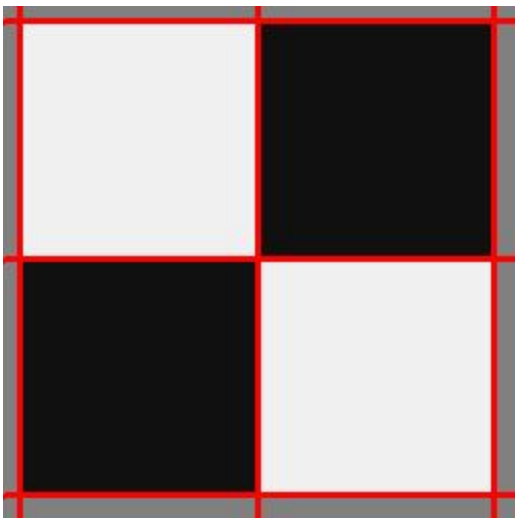
c. Write a function **`hough_lines_draw()`** to draw color lines that correspond to peaks found in the accumulator array. This means you need to look up rho, theta values using the peak indices, and then convert them (back) to line parameters in cartesian coordinates (you can then use

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
```

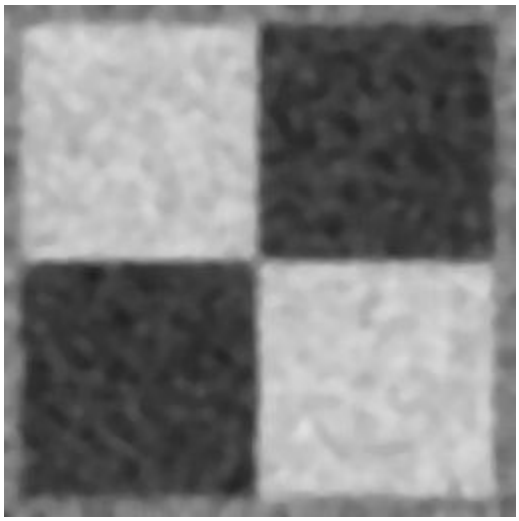
**Function**: `hough_lines_draw()`
**Output**: `ps2-2-c-1.png` - output of `hough_lines_draw()`.

i. 

`ps2-2-c-1.png - my output.`

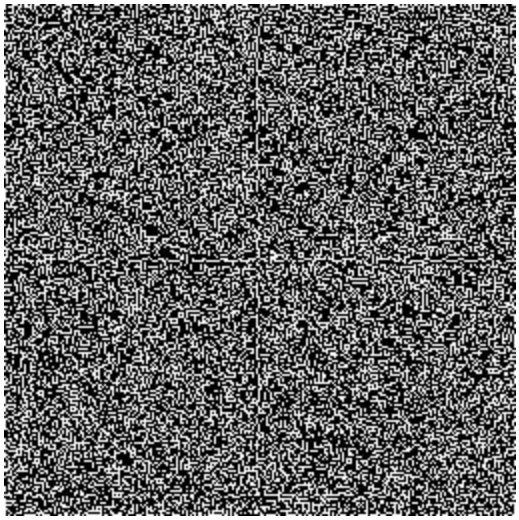
d. What parameters did you use for finding lines in this image?
**Output**: Text response describing your accumulator bin sizes, threshold, and neighborhood size parameters for finding peaks, and why/how you picked those.
   i. To get this image, I used my default bin sizes. Rho was set to every pixel step, and theta went in one degree steps. Rho scaled from 0 to the max size (the diagonal of the image) and theta ranged from 0-180. I then grabbed the top 6 peaks and found the lines I was looking for - no need to threshold at this point. Most of this was determined by trial and error (rho and theta were the first that I chose, and the peak finding was achieved by a simple sort).

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 $\sigma$ at least a few pixels big.
   **Output**: Smoothed image: `ps2-3-a-1.png`

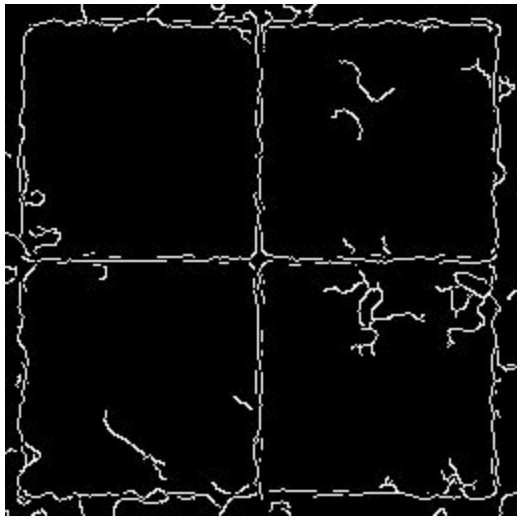


Smoothed image: `ps2-3-a-1.png`

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)

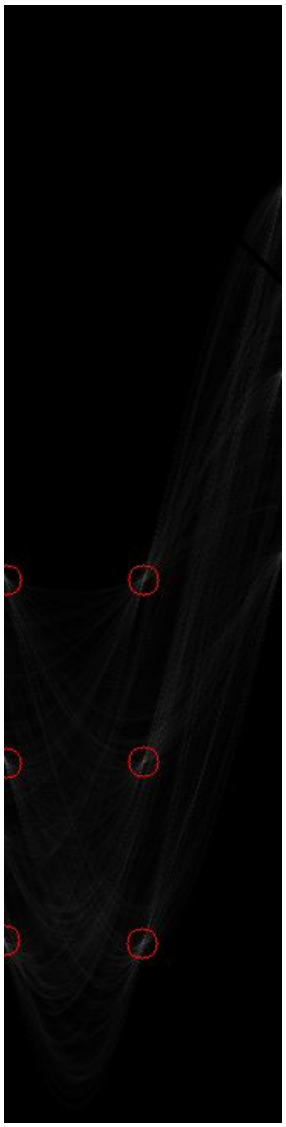


i. Original Canny



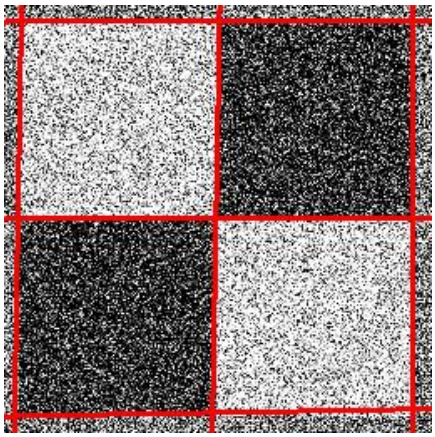
Smoothed Canny

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`



ps2-3-c-1.png

- Intensity image (original one with the noise) with lines drawn on them: ps2-3-c-2.png

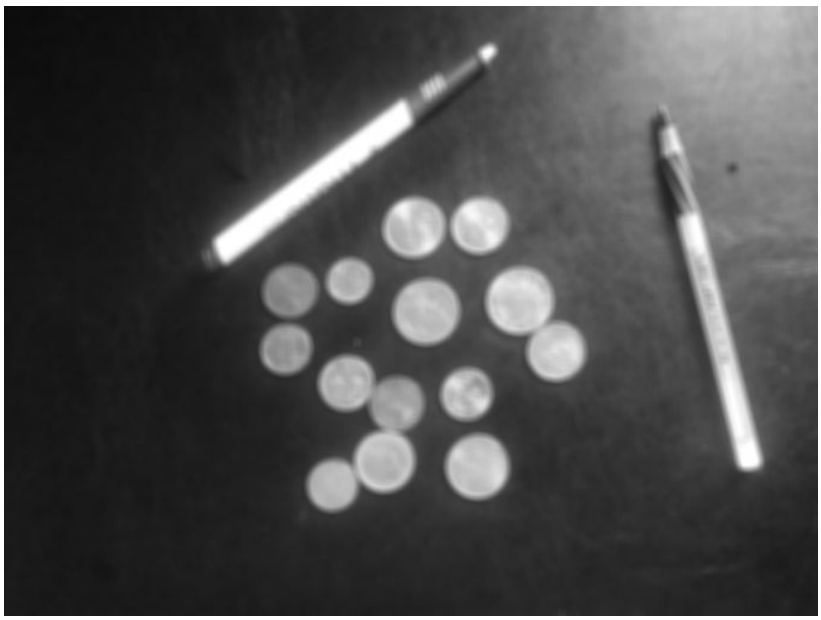


ps2-3-c-2.png

- Text response: Describe what you had to do to get the best result you could.

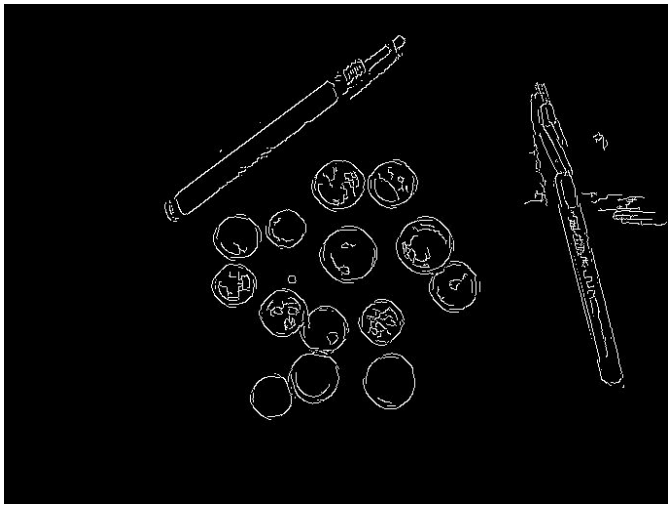
I did a lot of changing parameters to get this response. I tried a number of different Gaussian kernels, using both size and sigma changes. I also changed the Canny thresholds a number of times - and each time, the bottom edge or the top edge simply would not get found. After a number of back and forth changes, I decided to add a median filter in order to try to get a smoother image, and it worked. Gaussian kernel - 7x7, sigma = 5, median filter of 7, Canny parameters - 20, 100.

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 $\sigma$ at least a few pixels big.
   **Output**: Smoothed monochrome image: `ps2-4-a-1.png`



ps2-4-a-1.png

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

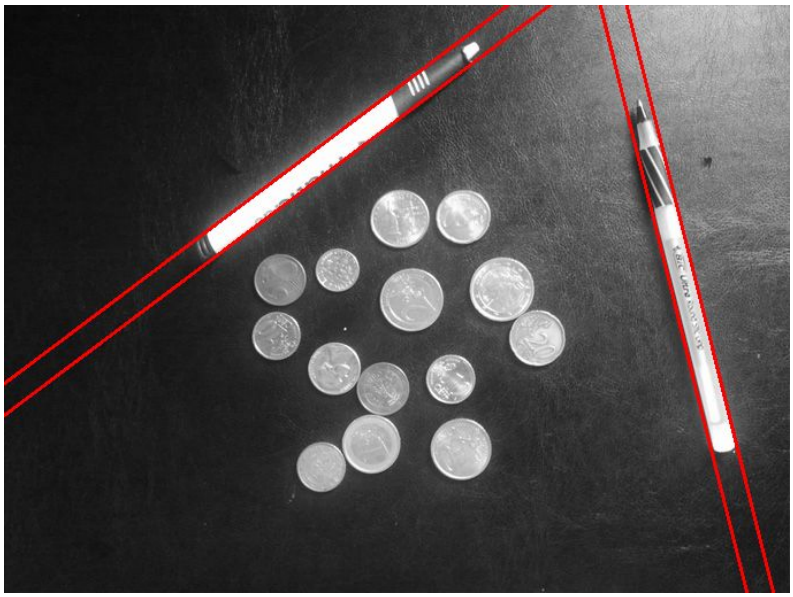


ps2-4-b-1.png

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

ps2-4-c-1.png cropped

- Original monochrome image with lines drawn on it: ps2-4-c-2.png



ps2-4-c-2.png

- Text response: Describe what you had to do to get the best result you could.

This one was all about the correct filtering again. This time I was able to get the noise down successfully with just a Gaussian kernel ( 7x7 sigma = 11) After that, Canny with thresholds of 10,50 reduced the image to the point where the hough lines were perfect. This was a pretty quick one, with minimal changes to get things lined up.

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)
      ```

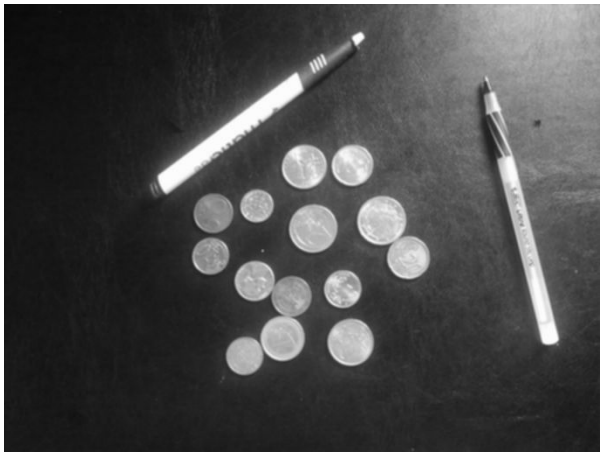
      This should return an accumulator H of the same size as the supplied image. Each *pixel* value of the accumulator array should be proportional to the likelihood of a circle of the given radius being present (centered) at that location. Find circle centers by using the same peak finding function:

      ```
      centers = hough_peaks(H, 10)
      ```

      **Function**: `hough_circles_acc()` (`hough_peaks()` should already be there)

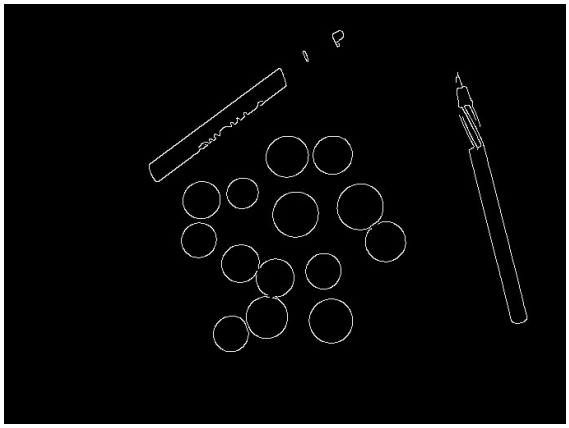
**Output**:

- Smoothed image: `ps2-5-a-1.png` (this may be identical to `ps2-4-a-1.png`)



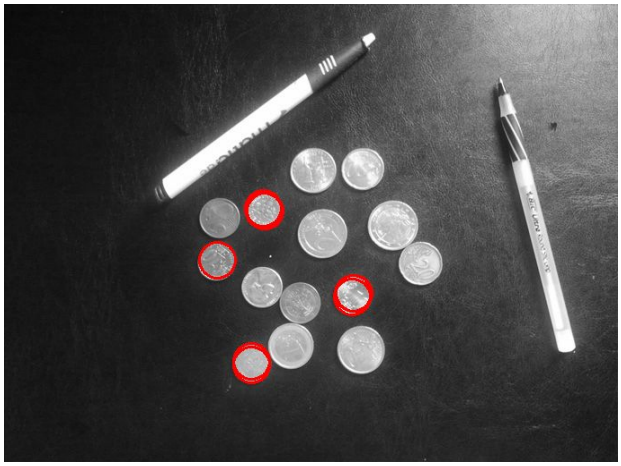
`ps2-4-a-1.png`

- Edge image: `ps2-5-a-2.png` (this may be identical to `ps2-4-b-1.png`)



`ps2-4-b-1.png`

- Original monochrome image with the circles drawn in color: `ps2-5-a-3.png`

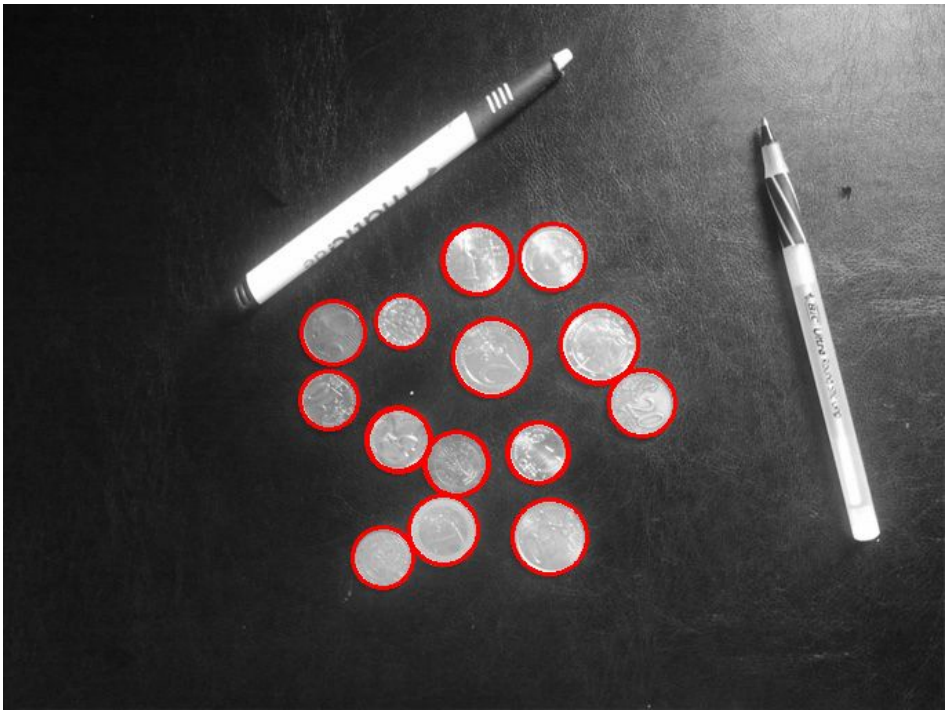


`ps2-5-a-3.png`

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`



`ps2-5-b-1.png`

- Text response: Describe what you had to do to find circles.

It was a challenge to find all of these circles. I made a lot of changes to the way I was filtering (note how different it is from the previous section). Noise in my edge image threw my Hough accumulator off a good bit. I implemented neighborhood voting at this time, to try to get better results. But, in the end, it came down to getting the Canny parameters and the filtering done correctly. Once that was done, I was able to grab the top peaks and it worked out great. I ran this with a minimum radius of 15 and a maximum radius of 50. I think I could have cut it down some, but I wanted to grab everything. Gaussian kernel (3x3 sigma = 16) + Canny (120, 500) got me the great wire frame I needed. After that, I grabbed the top twenty peaks. I know there's some overlap there, but I didn't de-dupe as it gave a great image.

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

i. The problems here are that there are a lot of lines in the image! Since we really only care about finding the pens, we don't want things like the text box or the edges of the book to show up. Simply put, there's too much valid noise.

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`

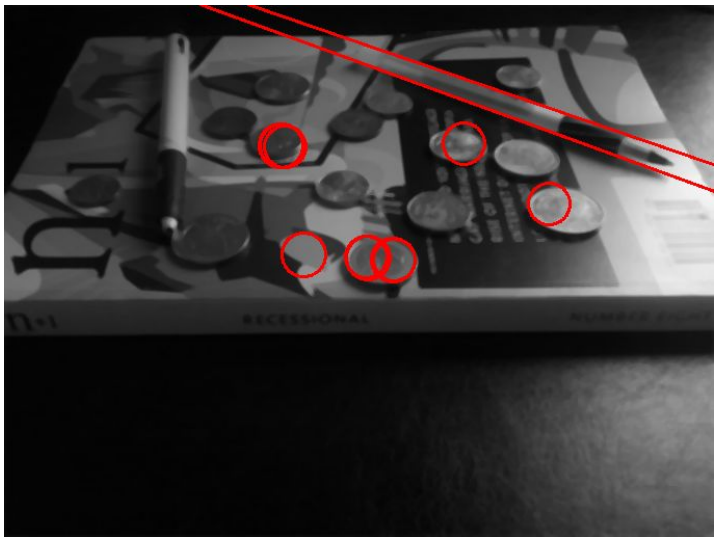
b. Are there any false positives? How would/did you get rid of them?
**Output**: Text response (if you did these steps, mention where they are in the code by file, line no., and also include brief snippets)

i. There are no false positives in my output, but rather a lot of false negatives. This is due to the aggressive filtering. I cut down a lot of the image by:

```
blur = cv2.GaussianBlur(noise_img,(13,13),1)
blur = cv2.medianBlur(blur, 5)
blur_img_edges = cv2.Canny(blur, 130, 120)
```

While this removed most of the false positives, it revealed a lot of false negatives. The image simply didn't have enough to find many of the coins. I ran this with a range of radii from 15-50. It took a good while to run, but the background is simply too noisy for me to find everything in it.

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
    i. This reminds me a lot of issues in Computational Photography. If we could calculate a legitimate homograpy to map the circle on to, I imagine we could do the same to the Hough space. Additionally, we could try using ellipses equations instead of circles. This seems to be a viable option given this paper: http://hci.iwr.uni-heidelberg.de/publications/dip/2002/ICPR2002/DATA/07_3_20.PDF

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).