**Instructions**

This lab has three parts. For each part, implement the outlined functions in `lab2.py` and run them in the notebook `lab2.ipynb`. For the Q&A in Part 4, write your answers directly into the notebook. Expected outputs to each cell in the notebook are supplied as reference but your outputs do not need to match exactly.

Upload to LumiNUS your completed `lab2.py` and `lab2.ipynb` by zipping them into a file named AXXX1_AXXX2.zip, where AXXX is the student number of the group members. Submit one file per group. Missing files, incorrectly formatted code that does not run, etc. will be penalized.

Minh Hoang is the TA in charge of this lab. Please post any questions onto the LumiNUS forum or attend one of the FAQ sessions during the regularly scheduled Lab session on 17.09 or 21.09. Please note that this second FAQ session will take place during recess week!

**Objective**

This lab will cover edge detection using the Canny edge detector and line and circle detection from these edges using the Hough Transform (see Figure 1). We will apply the concept of Hough transform for reflective and rotational symmetry detection in Lab 3.

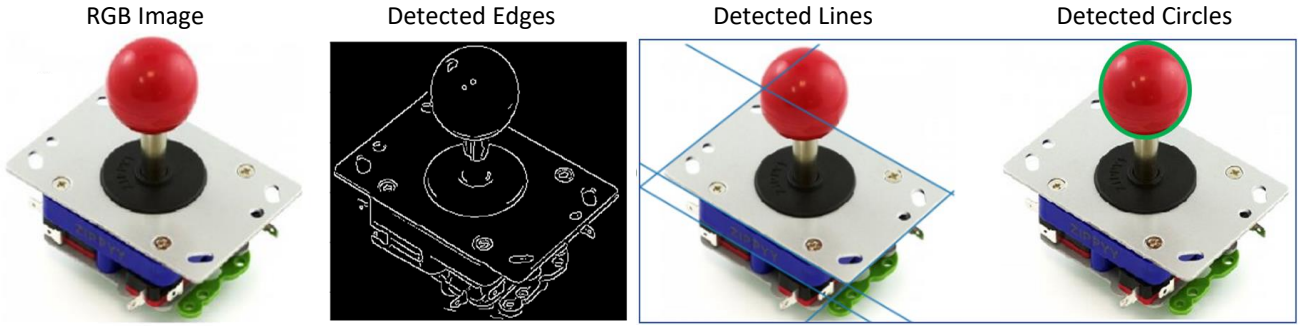
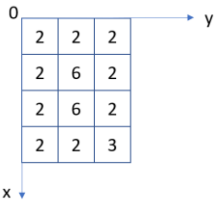



Figure 1. *From the original RGB image, detect the edges using the Canny Edge Detector (Part 1) before applying the Hough Transform to detect lines (Part 2) and circles (Part 3).*




Note: We will use the coordinate convention illustrated on the right where the top left corner is the origin, row# is x (1st coordinate) and column# is y (2nd coordinate). For instance, in the diagram at the right, the image shape is (4,3), the intensity value at origin ($x_0$, $y_0$) = (0,0) is 2, and the intensity value at point ($x_{12}$, $y_{12}$) = (3, 2) is 3.

## Part 1: Canny Edge Detection (30%)

This part detects the edges which will be fed as inputs to the Hough transform in Parts 2 and 3. To implement Canny Edge detection, perform the following steps

1. Pre-processing by blurring the image:
   - `make_gaussian_kernel()` takes in two parameters: kernel size `ksize` and Gaussian standard deviation `sigma` and generate the corresponding Gaussian kernel
   - `cs4243_filter()` is provided for you and applies your generated Gaussian kernel
2. `estimate_gradients()` estimates the gradient magnitude $G_M$ and gradient orientation $G_\theta$ by applying horizontal and vertical Sobel filters to generate the gradient components $G_X$ and $G_Y$. Be aware that our coordinate convention is rotated from Lab 1 and the lecture notes.
3. Perform non-maximum suppression on the threshold gradients $G_M$ to obtain a single-pixel wide gradients $G_1$.

- `non_maximum_suppression_interpol()` interpolates neighbouring pixel values to find the local maxima as described in lecture. If a given pixel's intensity is larger than both interpolated neighbours, keep as an edge pixel; otherwise, discard.
- `non_maximum_suppression()` skips interpolation and simply uses the nearest available neighbouring pixel values, with the choice in neighbouring pixel depending on the orientation $G_\theta$ (see Fig. 2 on right).

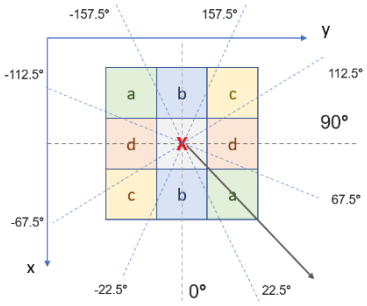



Figure 2. *Non-maximum suppression without interpolation partitions the orientation into ranges. For example,* $22.5^o \leq G_\theta < 67.5^o$ *uses pixels a as the neighbouring pixels for comparison,* $-22.5^o \leq G_\theta < 22.5^o$ *uses pixels b as neighbours, etc.*

4. `double_thresholding()` thresholds the single-pixel wide gradients $G_1$ with a high threshold $\tau_H$ and a low threshold $\tau_L$ to generate the strong edges $E_S = G_1 > \tau_H$ and weak gradients $E_w = \tau_H > G_1 > \tau_L$. Note that $E_S$ and $E_w$ are now binary images (0 for background, 1 for edge). The thresholds $\tau_H$ and $\tau_L$ should be set adaptively based on the gradient magnitudes present in $G_1$, i.e. $\tau_H = \min G_1 + f_H \cdot (\max G_1 - \min G_1)$, and $\tau_L = \min G_1 + f_L \cdot (\max G_1 - \min G_1)$ where $f_H$ and $f_L$ are given as inputs.
5. `edge_linking()` performs a naïve version of edge linking by checking all edge pixels of $E_W$ to see if they have a strong edge in the 8-pixel neighbourhood. Make multiple passes, where in each pass, linked weak pixels are removed from the weak pixel set and added to the strong pixel set. Stop when no more weak pixels get linked or when the maximum number of passes is reached. The resulting strong pixel set will be a complete (or approximate) edge image. For efficiency, you are strongly recommended to vectorise your implementation.

**Part 2: Line Detection with Hough Transform (20%)**

Use the detected edges from Part 1 to detect straight lines. Recall the normal form of a line representation with parameters $\rho$ and θ, the normal line length and angle.

1. Quantize the parameter space for $\rho$ and $\theta$ based on the min and max value for that parameter and an interval size. The min and max of $\rho$ spans from $-d$ to $d$, where $d$ is the length of the image diagonal, while $\theta$ spans from 0 to $\pi$. The interval will be user-defined and determines the quantization or bin coarseness in the accumulator array. Index the quantized spaces for $\rho$ and $\theta$ with $r$ and $t$ respectively. When mapping from $(\rho, \theta)$ to $(r, t)$ treat the lower bound of the interval range as inclusive and upper bound as exclusive. When mapping from $(r, t)$ to $(\rho, \theta)$, apply the interval lower limit of the interval. For example, if the interval for $\theta$ is $\frac{\pi}{4}$, the quantization is split from $\left[0, \frac{\pi}{4}\right), \left[\frac{\pi}{4}, \frac{\pi}{2}\right) \left[\frac{\pi}{2}, \frac{3\pi}{4}\right), \left[\frac{3\pi}{4}, \pi\right]$ and $\theta_r$ associated with $r = 1$ is $\frac{\pi}{4}$. As default, use an interval of 1 pixel and $\frac{\pi}{180}$ for $\rho$ and $\theta$ respectively.
2. Create a 2D accumulator array A of dimension $(R \times T)$ and initialize all entries to 0, where R and T are total number of bins for $\rho$ and $\theta$ based on the chosen quantization for step 1.
3. `hough_vote_lines()` iterates through each pixel in the edge image and casts votes for possible lines passing through that pixel. For a pixel $i$ located at $(x_i, y_i)$, solve $\rho_r = x_i \cos \theta_r + y_i \sin \theta_r$ for every quantized value $\theta_r$ that $\theta$ can take based on your quantization in step 1 by mapping the $r$ values back. You may optionally include T+1 to cover the upper bound of the last bin. Find the corresponding index $(r, t)$ in the accumulator array and increment A$[r, t]$.

4. `find_peak_params()` performs non-maximum suppression similar to Lab 1 and finds the local maxima in A to return the associated $r, t$. This function is provided for you. It has as inputs a threshold on accumulator values and a window size when searching for local maxima.

**Part 3: Circle Detection with Hough Transform (20%)**

Use the detected edges from Part 1 to detect circles. A circle is parameterized by center coordinates $(a, b)$ and radius $r_c$. The steps are analogous to Part 2, with a few changes to improve efficiency.

1. Quantize the parameter space in the same way as Part 2. The limits of $a$ and $b$ span from 0 to H and 0 to W respectively, with (H, W) being the dimension of the image (we are only interested in circles whose centres lie within the image frame). The limits of $r_c$ range from $r_{min}$ to $r_{max}$ which are either user-defined or set to defaults of 3 and the image diagonal length respectively. Use an interval of 1 pixel for all 3 parameters.
2. Create a 3D accumulator array A of size $(R \times H \times W)$ based on the quantization of step 1.
3. `hough_vote_circles()` iterates through each pixel on the edge image. For every candidate radius value r, an edge pixel will cast a vote for circles of radius r, centred at itself.
   - A naïve way to cast votes on a perimeter around pixel $i$ located at $(x_i, y_i)$, would be to iterate through each bin of $r$ and each value θ in the range from 0 to 2π and solve for $a = x_i\theta + cos\ \theta$ and $b = y_i + sin\ \theta$.
   - The naïve formulation is redundant because it is always the same circle of votes cast relative to $(x_i, y_i)$ for some fixed radius $r$. The relative votes can therefore be solved for in advance, e.g. using `skimage.draw.circle_perimeter` and added directly to the accumulator array A. This process is illustrated in Fig. 3 below.
4. Use the provided `find_peak_params()` to detect the local maxima.

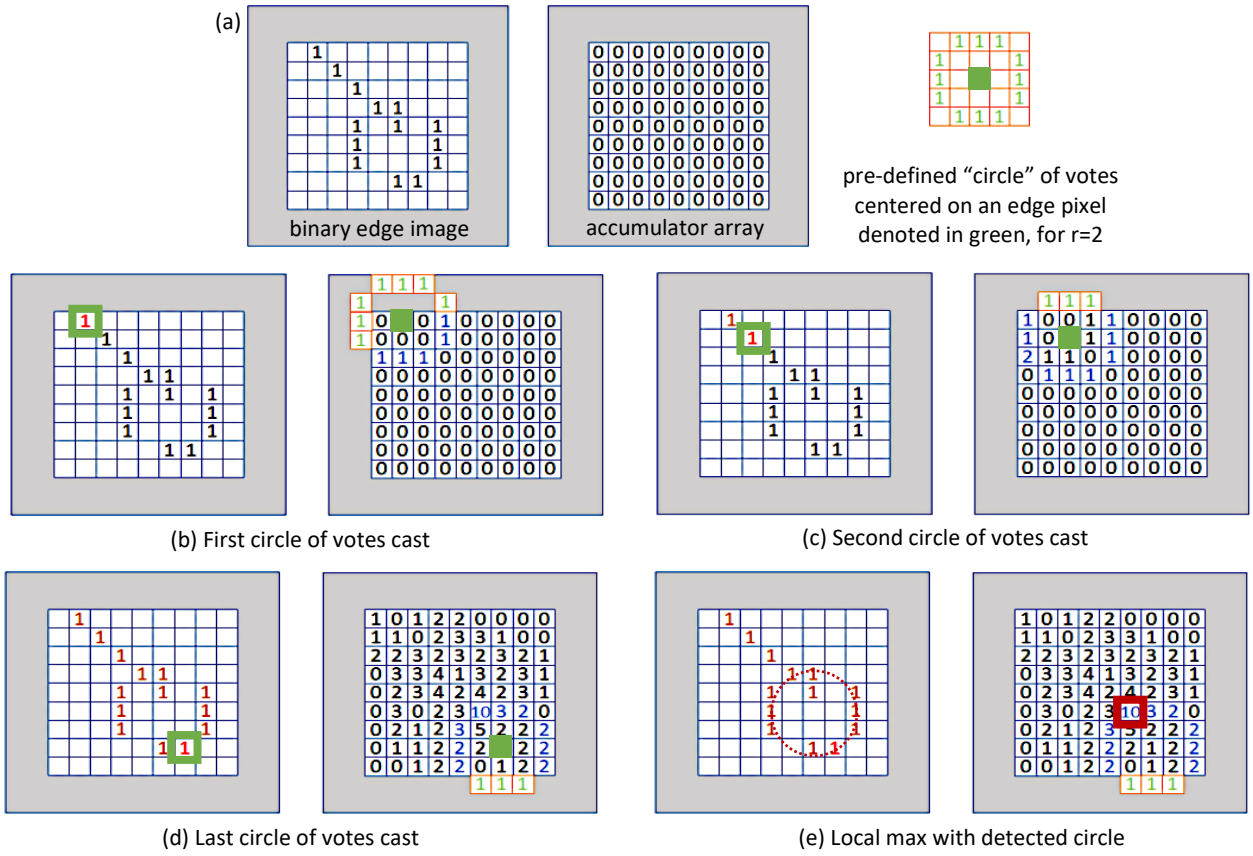


*Figure 3. (a) Given an edge image and an accumulator array, we can generate a pre-defined (quantized) circle of votes. (b) – (d) This vote circle can be added to relative to each edge pixel in the image. The final detected circle is based on the local maximum in the accumulator array. Note the padding in the accumulator array denoted by the grey shading to accommodate out-of-range votes.*

**Part 4: Exploratory Questions (30%)**

- **Weighted Voting (5%)**: Apply your `hough_vote_circles()` to the image `coins2.png`. Are you able to find all the circles? What adjustments can you make to the votes cast to ensure that circles of all sizes are detected? Hint: Run `find_peak_params()` and visualize the bin counts. Each bin is associated with the number of edge pixels that cast a vote. How is this affected by the circle size?
- **Leveraging gradients (5%):** Implement `hough_vote_circles_grad()` that leverages the gradient orientation information to reduce the number of votes cast, as described in lecture. In this case, each edge pixels does not cast a circle of votes, but two votes along the gradient orientation, i.e. for an edge pixel $(x_i,y_i)$ with gradient orientation $\theta_i$, the associated votes are $a_1 = x_i + r\cos\theta_i$, $b_1 = y_i + r\sin\theta_i$ and $a_2 = x_i - r\cos\theta_i$ , $b_2 = y_i - r\sin\theta_i$. Weight your votes accordingly similar to the previous question.
- **Interval / Bin Size (10%)**:
  - Test your implementation of `hough_vote_circles_grad()` on the image `penny_farthing.png`. Comment on your results. Are you able to find any circles? To make this part more intuitive, you can also set the weight of each vote to 1 so that you can determine the number of edge pixels which cast a vote in that bin.
  - The default interval setting of $r$, $a$, and $b$ is 1. Repeat the previous step with interval sizes of 3 and 5 for all three parameters. Again, comment on your results and explain the difference in result when the interval was set to 1. What do you think will happen if the interval size is set to a large value like 10 or 20?
- **Gaussian Sigma (10%):** Consider the images `flowers.png` and `cells.png`. Apply your edge detection algorithm from Part 1 with varying input values of `sigma` (1, 3, and 5) for `make_gaussian_kernel`. Perform circle detection for each sigma and comment on the quality of the edge and circle detection. What impact does changing `sigma` have on detecting the circles and why? Note that you may need to tune the parameters of `double_thresholding()` and `find_peak_params()`, but your answer should explain the impact of the various `sigma`.