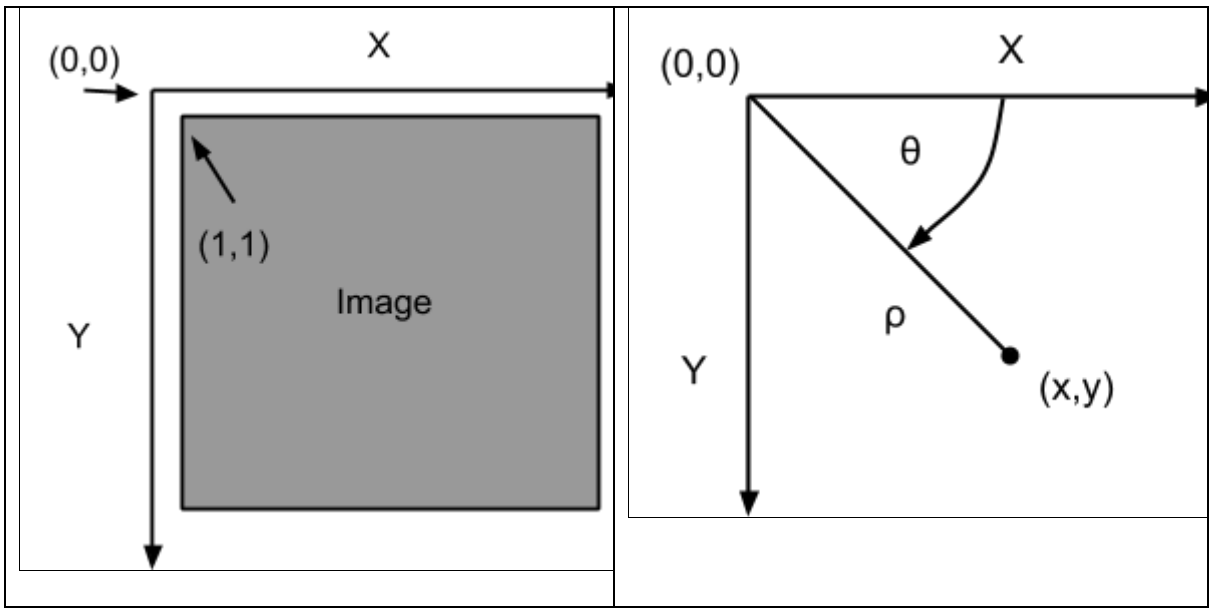
IVP Lab Assignment 3

Q1. Implement a Hough Transform method for finding lines (you can use the inbuilt edge operator functions). Note that the coordinate system used is as pictured below with the origin placed one pixel above and to the left of 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 counter-clockwise in the usual convention. Theta (θ) = zero still points in the direction of the positive X-axis.

a) Write a function `hough_lines_acc` that computes the Hough Transform for lines and produces an accumulator array. Your code should try to conform to the specifications of the hough function in Matlab: http://www.mathworks.com/help/images/ref/hough.html
Note that it has also two optional parameters RhoResolution and Theta, and returns three values - the hough accumulator array H, theta (θ) values that correspond to columns of H and rho (ρ) values that correspond to rows of H.
You can apply it to the edge image (`img_edges`) of 'ps1.png' as:

```
[H, theta, rho] = hough_lines_acc(img_edges);
```

Or, with one optional parameter specified (θ = integers `-90` to `89`, i.e. `180` values including `0`):

```
[H, theta, rho] = hough_lines_acc(img_edges, 'Theta', -90:89);
```

b) Write a function `hough_peaks` that finds indices of the accumulator array (here line parameters) that correspond to local maxima. Your code should try to conform to the specifications of the Matlab function `houghpeaks`:
http://www.mathworks.com/help/images/ref/houghpeaks.html
Note that you need to return a Qx2 matrix with row indices (here rho) in column 1, and column indices (here theta) in column 2. (This could be used for other peak finding purposes as well.) Call your function with the accumulator from the step above to find up to 10 strongest lines:

```
peaks = hough_peaks(H, 10);
```

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).

Q2. (a) To find the Harris corners you need to compute the gradients in both the X and Y directions. These will probably have to be lightly filtered using a Gaussian to be well behaved. You can do this either the "naive" way - filter the image and then do simple difference between left and right (X gradient) or up and down (Y gradient) - or you can take an analytic derivative of a Gaussian in X or Y and use that filter. The scale of the filtering is up to you. You may play with the size of the Gaussian as it will interact with the window size of the corner detection. Try your code on both transA and simA. To display the output, adjoin the two gradient images(X and Y) to make a new, twice as wide, single image (the "gradient-pair"). Since gradients have negative and positive values, you'll need to produce an image that is gray for 0.0 and black is negative and white is positive.

Now write code to compute the Harris response value for the image. The only design decisions are the size of the window (the sum), the windowing function that controls the weights, and the value of α in the Harris scoring function. Remember you can check you code on the checkerboard images (though those might use a different optimal window size than the real ones). Apply to transA, transB, simA, and simB.

Finally you can find some corner points. To do this requires two steps: thresholding and non-maximal suppression. You'll need to choose a threshold value that eliminates points that don't seem to be plausible corners. And for the non-maximal suppression, you'll need to choose a radius (could be size of a side of a square window instead of a circle radius) over which a pixel has to be a maximum. Write a function to threshold and do non-maximal suppression on the Harris output.

(b) Now that you have keypoints for both image pairs, we can compute descriptors. You will be glad to know that we do not expect you to write your own SIFT descriptor code. Instead you'll use a MATLAB package called VLFeat and for Python, the SIFT or SURF classes in OpenCV. Please check out the supplemental document for instructions on using SIFT in VLFeat/MATLAB or OpenCV/Python. The standard use of a SIFT library consists of you just providing an image and the library does its thing: finds interest points at various scales and computes descriptors at each point. We're going to use the library code only to compute the orientation histogram descriptors for the interest points you have already detected from previous step. To do so, you need to provide a scale setting and an orientation for each feature point as well as the gradient magnitude and angle for each pixel. The scale we'll fix to 1.0. The orientation (gradient direction) needs to be computed from the gradient images you got from before. So, Write the function to compute the angle for the set of interest points you found above. Now call the SIFT descriptor code. You need to pass in each keypoint location along with its scale and orientation (This process is covered in more detail in the accompanying supplemental document). Once we have the descriptors, we need to match them. This is what is called *putative* matches. Given keypoints in two images, get the best matches. Both VLFeat and OpenCV have functions for computing matches (e.g. for VLFeat it's called vl_ubcmatch.) You will call those and then you will make an image that has both the A and B version adjoined and that draws lines from each keypoint in the left to the matched keypoint in the right. We'll call this new image the putative-pair-image.

(c) You now have keypoints, descriptors and their putative matches. What remains is RANSAC. To do this for the translation case is easy. Using the matched keypoints for transA and transB, randomly select one of the putative matches. This will give you an offset (a translation in X and Y) between the

two images. Find out how many other putative matches agree with this offset (remember, you may have to account for noise, so "agreeing" means within some tolerance). This is the *consensus* set for the selected first match. Find the best such translation - the one with the biggest consensus set. So, write the code to do the translational case on transA and transB. Draw the lines on the adjoined images of the biggest consensus set.

For the other image pair (simA and simB) we need to compute a transformation matrix which allows for translation, rotation and scaling. We can represent this transform with a matrix with six unknowns. Each match gives two equations - so you need to pick three matches to solve. Write code to apply RANSAC by randomly picking two matches, solving for the transform, and determining the consensus set. Draw the lines on the adjoined images for the biggest consensus set.