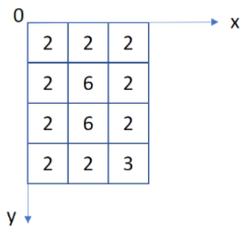
Instructions
This lab has four parts. For each part, implement the outlined functions in `lab3.py` and run them in the notebook `lab3.ipynb`. Exploratory questions are embedded directly at the end of the sections; write your answers directly into the notebook. Expected outputs are supplied as reference, but your outputs do not need to match exactly.

Upload to LumiNUS your completed `lab3.py` and `lab3.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 22.10 or 26.10.

Objective
This lab covers keypoint detection and feature descriptors. We will explore the Harris Corner Detector, a simplified version of the SIFT descriptor and two applications for keypoint matching: image stitching and symmetry detection.

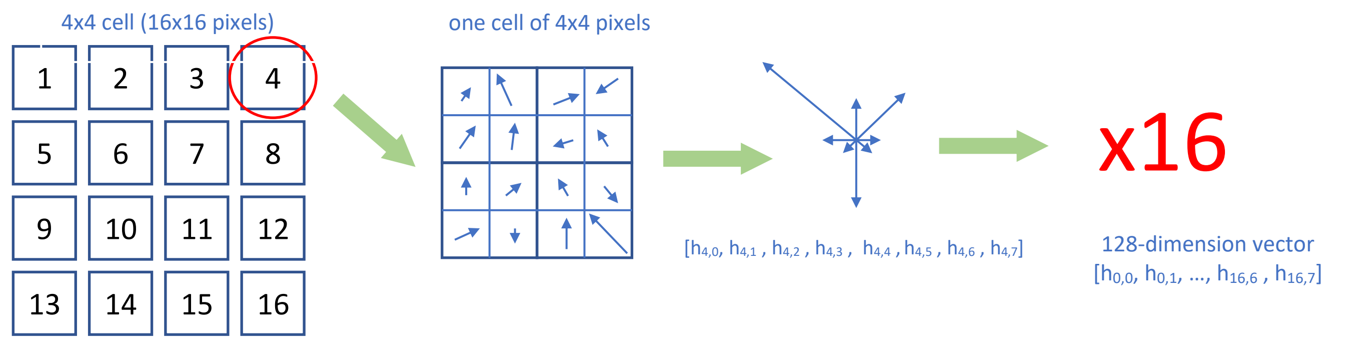
Unfortunately, we need to change the coordinate convention (x- and y-axis are swapped compared to Lab 2, see Figure on the right) to stay consistent with the conventions of the OpenCV functions which are used in this lab.



Part 1: Keypoint Detection, Description, and Matching (35%)
This part covers foundations of keypoint descriptors and matching which will be used in later parts. We provide the wrapper `describe_keypoints` which calls either the naïve descriptor or the simplified version of SIFT.

- `harris_corners()` detects Harris corner keypoints. For each pixel in an imsage,
  1. Compute image gradient I_x, I_y and compose the second moment matrix H. Note that the elements A, B, and C of matrix H are obtained by summing the elements in some window W:
     i. $H = \begin{bmatrix} A & B \\ B & C \end{bmatrix}$ $\quad A = \sum_{(x,y)\in W} I_x^2 \quad B = \sum_{(x,y)\in W} I_x I_y \quad C = \sum_{(x,y)\in W} I_y^2$

     Consider an efficient implementation of the window-wise summation based on convolution (i.e., do not use for-loops). Weight the elements uniformly in the window for the summation. Hint: Consider using built-in functions `skimage.filters.sobel_v`, `sobel_h` and `scipy.ndimage.filters.convolve`.
  2. Compute the corner response $R = det(H) - k\left(tr(H)\right)^2$ with $k = 0.04$.
  3. Find local maxima using the provided non-maximum suppression; the local maxima in the response map corresponds to the keypoints.
- `naive_descriptor()` computes a local descriptor based on the normalized pixel intensity, i.e. $F(x,y) = \frac{I(x,y)-\mu}{\sigma+0.0001}$. $I(x,y)$ is the intensity at a keypoint located at (x,y) and μ and σ are the mean and standard deviation of the pixel intensities in a patch of 5x5 centered at (x,y). The 0.0001 term in the numerator improves numerical stability.
- `simple_sift()` is a simplified version of the full SIFT descriptor which is not rotationally invariant. For each keypoint, take a 16×16 patch of pixels around the keypoint and compute the image gradient magnitude and orientation at each pixel. For efficiency, consider precomputing the image gradients for the entire image in advance. Divide the patch into a 4×4 grid of cells, where each cell has $4 \times 4 = 16$ pixels (see Figure 1a).



(a) 16x16 patch is partitioned into a 4x4 array of cells; each cell is 4x4 pixels. Note the numbers do not correspond to indices.

(b) Histogram of gradient orientations from each cell; the 8 bin histogram forms an 8-dim. feature vector. Concatenating this vector from all 16 cells in the patch forms a 128d feature.

Figure 1. Schematic visualization of estimating our simplified SIFT descriptor.

For each cell, create a histogram of the gradient orientations with 8 bins of 45° each (e.g., 0° to 44° for bin 0, 45° to 89° for bin 1, etc.). Each pixel's contribution to the histogram is weighted by the pixel's gradient magnitude and the weight from a Gaussian kernel of 16×16 centered on the keypoint coordinate. Use the provided `make_gaussian_kernel`. Concatenate the histograms from each cell to form a $4\times4\times8=128$-dimension vector (see Figure 1b). Normalize the vector to unit length by dividing by the vector's magnitude [link].

- `top_k_matches()` finds the k top matches between two feature descriptor sets $F_1 = \{F_{11}, F_{12} \dots F_{1i} \dots F_{1M}\}$ and $F_2 = \{F_{21}, F_{22} \dots F_{2j} \dots F_{2N}\}$. For each descriptor in F_1, find the k descriptors in F_2 with the smallest Euclidean distance. Give the output as a list of length-2 tuples. For each tuple, the first element is the descriptor index i from F_1 and the second element is a k-long list of length-2 tuples, with indices j in F_2 and the Euclidean distance $D(F_{1i}, F_{2j})$. Hint: consider using `scipy.spatial.distance.cdist()`.
- `ratio_test_match()` checks for F_{1i} the ratio between the top 2 matches F_{2a} and F_{2b}. If $\frac{D(F_{1i},F_{2a})}{D(F_{1i},F_{2b})} < \tau$, then F_{1i} and F_{2a} are a match. The ratio threshold τ is given as an argument.

Part 2: Image Stitching (25%)

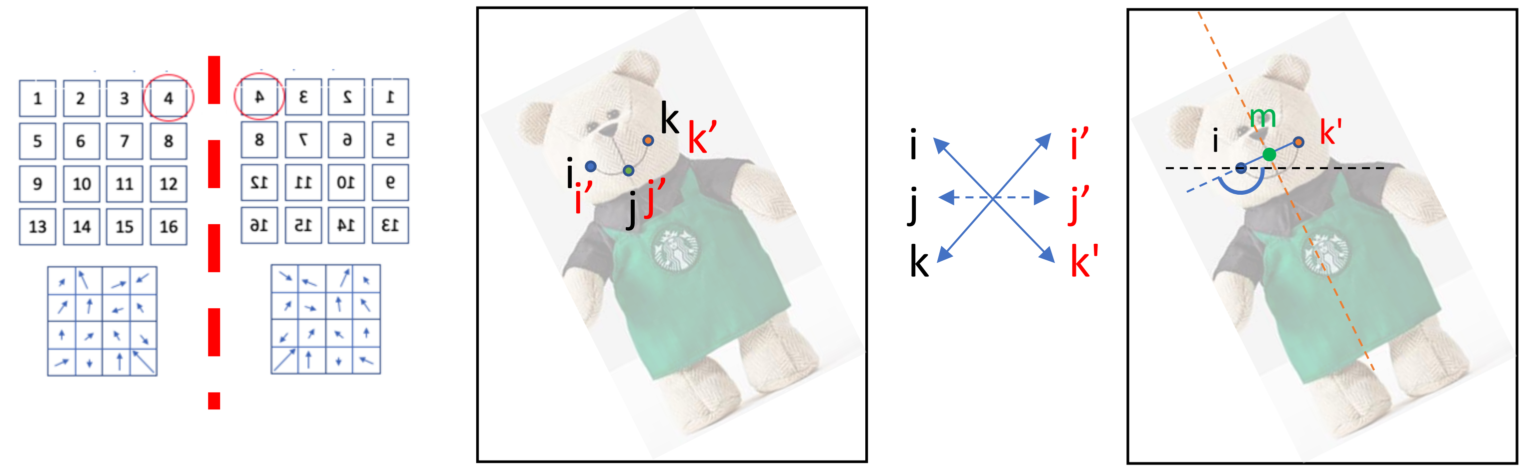
This part uses the matched descriptor pairs from **Part 1** and solves the homography between the two images. The images are then warped and stitched together. For this part, use your SIFT descriptor implementation (simple_sift) to match the keypoints for image `fp1.png` and `fp2.png`.

- `compute_homography()` solves for the homography matrix between two images using the normalized DLT algorithm based on matched keypoint coordinates. Refer to Lecture 9, slide 17. If you need more details to help you with the implementation, refer to this StackOverflow post.
- `ransac_homography()` solves for a more robust homography by using RANSAC. Give as input arguments: s, the number of matches to compute the homography, and δ, the inlier threshold. Matched keypoints are considered inliers if the projected point from one image lies within distance δ to the matched point on the other point. Run your algorithm for N iterations; store the largest set of inliers and return a final homography based on this inlier set. Note that you may need to tune the parameters s, δ and N to get more accurate results.
- For implementing the above two functions, use the provided `transform_homography` function that applies a given homography to a set of points.

<u>Part 3: Mirror Symmetry Detection (25%)</u>

For Parts 3 and 4, use the provided function `compute_cv2_descriptor`, which uses OpenCV's SIFT implementation since we require rotation-invariant descriptors. Mirror symmetry is defined by a line of reflection or symmetry line (dashed red line in Fig. 3b). This part performs SIFT keypoint matching to solve for candidate points on the symmetry line. SIFT descriptors are rotationally invariant, but not mirror-invariant, so how can they be matched? We create a virtual set of "mirror" descriptors by re-assigning the cell indices and histogram bin indices.

- `create_mirror_descriptors()` using the provided `compute_cv2_descriptor`, detect keypoints and compute descriptors from the original image I_o denoted as $F_o = \{F_{O1}, F_{O2} \dots F_{Oi} \dots F_{OM}\}$.
- `shift_sift_descriptor()` creates a virtual set of mirror descriptors (see Fig. 3b). Treat the image as if it has been mirrored over a virtual vertical axis (see Figure 3a for conceptual visualization). Shift the cell indices and bin indices of each 8-dimensional vector accordingly. For each 8-dim. vector, as SIFT already shifts the indices to keep the dominant orientation first, the first index (0) will stay in the same position. Remaining bins are reversed, i.e. [0, 1, 2, ..7] remaps to [0, 7, 6, .., 2, 1]. See `lab3.py` for an example of a SIFT histogram vs. its mirrored version.
- `match_mirror_descriptors()` matches keypoints between F_o and mirrored descriptors $F_{O'} = \{F_{O'1}, F_{O'2} \dots F_{O'i} \dots F_{O'M}\}$ (see Fig. 3c)



(a) Mirror the descriptor by remapping the cell indices and reassigning the histogram indices within each cell. Note the numbers do not correspond to actual indices.

(b) Detected keypoints shown in black and their virtual mirror descriptors shown in red.

(c) Matches of original descriptor set to virtual mirror descriptor set.

(d) A line passing through matched keypoints j and k' has angle and will intersect with the symmetry line half-way at virtual point m.

Figure 3. Symmetric feature matching.

- `find_symmetry_lines()` takes in pairs of matched descriptors and votes for a candidate symmetry line via the Hough transform. For a matched pair of points i and k' (see Fig 3d), form a line that intersects the two points; this line is perpendicular to the symmetry line. Solve for the midpoint $m = (x_m, y_m)$ between i and k' on the intersecting line and compute the angle θ_m that the line makes with the x-axis.
- `hough_vote_mirror()` performs Hough voting with parameter space (ρ,θ). Each intersecting point m has one vote defined by θ_m and $\rho_m = x_m \times cos\ cos\ (\theta_m) + y_m \times sin\ sin\ (\theta_m)$. The line(s) of symmetry in the image will be represented by local maxima in the Hough vote space. Use the function `find_peak_params` (from Lab 2); `num_lines` is a parameter which limits the number of local maxima that are returned.

Part 4: Rotation Symmetry Detection (15%)

Keypoint matching and Hough voting can also be used to detect rotational symmetry. For this part, we use the orientation and scale attributes for the `cv2.Keypoint` class named `angle` and `size`.

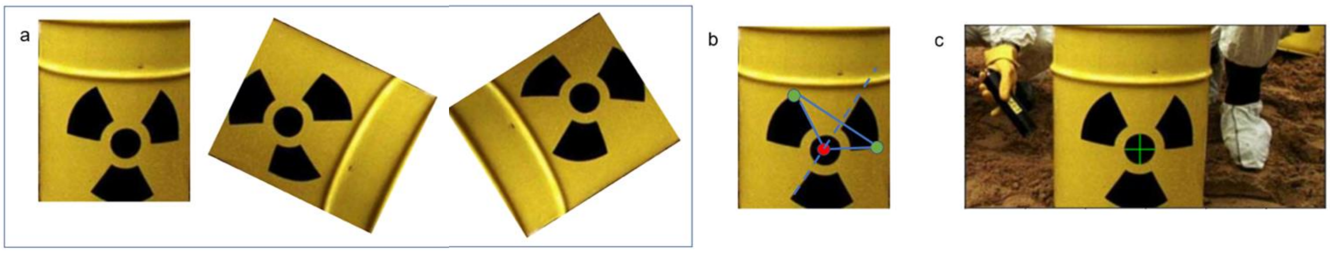


Figure 5. Rotational symmetry detection: a) A shape looks the same after rotations; b) A pair of matched keypoints (green) vote for one candidate center of rotation (red); c) Hough voting is used to select a maxima point from the candidate centers as the predicted center of rotation.

- Using `compute_cv2_descriptor`, detect keypoints and compute descriptors from a given image denoted as $F = \{F_1, F_2 \dots F_i \dots F_M\}$.
- Then implement the function `match_with_self` which uses `top_k_matches`, giving F as both F_1 and F_2 since we are matching descriptors on an image to itself to find the best three matches. Eliminate the trivial match (a keypoint is matched with itself) and perform the ratio test on the other two matches. If no match is eliminated, pick the best two.
- `find_rotation_centers()` estimates, for a matched pair of keypoints $p_i = (x_i, y_i)$ and $p_j = (x_j, y_j)$, the coordinates for a candidate centre of rotation:
  a. Remove "parallel" matches based on the keypoint orientation i.e. the `angle`. Wrap around the orientations Φ_i and Φ_j for p_i and p_j respectively to ensure they are within $[0, 2\pi)$. If Φ_i and Φ_j are within 1 degree of each other, discard p_i and p_j. The similar orientations suggest "parallel" keypoints, i.e. there exists no centre of rotation about which p_i can be rotated to coincide with p_j.
  b. Form a line of intersection (see red line in Fig. 4) between p_i and p_j and solve for this line the length d and the angle γ that it forms with the x-axis.
  c. A center of rotation $c_{ij} = (x_c, y_c)$ between p_i and p_j can be defined[1] by angle $\beta = \frac{(\Phi_i - \Phi_j + \pi)}{2}$ and radius $r = \frac{d \times \sqrt{1+(\beta)}}{2}$ with coordinates $x_c = x_i + r \times cos(\beta + \gamma)$ and $y_c = y_i + r \times sin(\beta + \gamma)$. Discard if the center is out of image bounds.

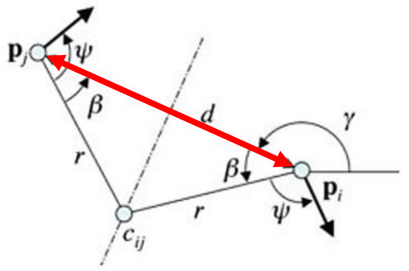


Figure 5. Illustration for computing centre

- `hough_vote_rotation()` does Hough voting on the rotation coordinates with parameter space (x, y). Matched keypoints p_i and p_j vote for the associated center location (x_c, y_c) with the vote weighted by the keypoint scales s_i and s_j respectively. The vote weight is defined as $w = e^{q^2}$ where $q = \frac{-|s_i - s_j|}{(s_i + s_j)}$. The global maxima should correspond to the center of rotation.

- Note: pay attention to our coordinate choice because to get the point (x_i, y_i) you will need to call `I[yi][xi]` in your code.

[1] If you are interested, see the precise derivation on page 7 and 8 of the original paper [link].