Instructions

This lab has four parts. For each part, implement the outlined functions in `lab4.py` and run them in the notebook `lab4.ipynb`. For the Q&A in Part 4, write your answers directly into the notebook. Expected outputs are supplied as reference but your results need not match exactly.

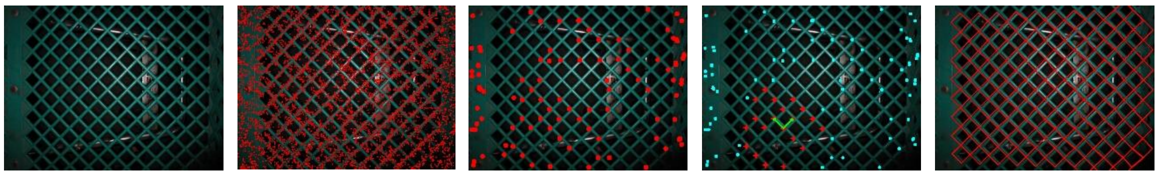
Upload to LumiNUS your completed `lab4.py` and `lab4.ipynb` by zipping them into a file named AXXX1_AXXX2.zip, where AXXX is the student number of the group members. Submit one file per group. Missing files, incorrectly formatted code that does not run, etc. will be penalized.

Ji Bo 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 12.11 or 16.11.

Note: You are free to re-use functions in previous labs and use any NumPy or OpenCV functions, except the built-in functions for mean-shift e.g. `cv2.meanShift`. For k-means clustering, you can use `KMeans` from scikit-learn. Please do not import other high-level computer vision libraries. We provide default parameter values but you are free to adjust these.

Objective

This lab covers keypoint detection, clustering, image transformations, and RANSAC. In Lab 1, we used manually specified templates in template matching for translation symmetry detection. This lab algorithmically "discovers" repeated patterns to propose lattices of translation symmetric structures. Lab instructions are self-contained, but you may read the original paper[1] for your own interest.



(a)RGB Image (b) Detected Keypoints (c) Clustered Points (d) Lattice Model (e) Final Output

Figure 1: Overview of translation symmetry detection. From the original image (a), the detected keypoints (b) are clustered based on similar appearance (c). RANSAC is applied to find vector pairs representing the lattice structure (d) before scoring the proposals and generating the final lattice (e).

Part 1: Shi-Tomasi Keypoint Detection and Clustering (30%)

We cluster keypoints based on the appearance of a local patch of $h_p \times w_p$ around the keypoint to find repeating patterns. Keypoints from the same cluster should be located at the same relative positions for a repeating pattern (see Fig 1c, 1d, the clustered keypoints appear at the fence grid crossing).

- `detect_points()` finds keypoints by the Shi-Tomasi method (see Lecture 11) with the OpenCV method `cv2.goodFeaturesToTrack`. Usually, simply applying a keypoint detector may not find all the required keypoints for finding the repeating patterns in the image to build up a lattice structure. Apply the following modifications to increase the number of detected points N:
    - **Patch-wise keypoint detection:** Given an image of size (h, w), divide the image into approximately 50x50 patches and apply keypoint detection to each patch individually. Your image should have $P \cdot Q$ patches, where $P \approx (h \,/\, 50)$ and $Q \approx (w \,/\, 50)$.

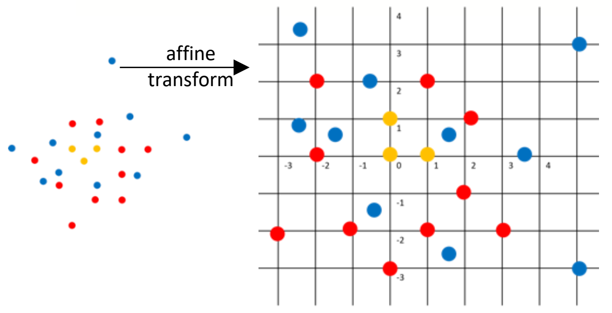
[1] We roughly follow the Phase I algorithm from *Deformed Lattice Detection in Real-World Images Using Mean-Shift Belief Propagation* [pdf].

- **Quality decay:** ρ thresholds the minimum acceptable quality for detected keypoints. For each patch, lower ρ progressively by a factor $\gamma_\rho < 1$. Stop lowering ρ when N_p, a required number of keypoints for each patch, is found or when ρ falls below some threshold τ_ρ.

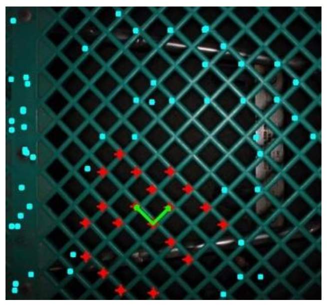
- `cluster()` the patches with mean-shift clustering with bandwidth h to produce a coarse set of clusters $C_i, i = 1, \dots, N_c$. To avoid N_c becoming too large, e.g., $N_c > N / 3$, tune the bandwidth by gradually increasing h by a factor $\gamma_h > 1$. Limiting the number of clusters ensures that resulting clusters are not too small with too few points per cluster.
  - To speed up your mean-shift clustering, consider making use of Speedup 2 as outlined in Lecture 5, slide 39. In this case, you need not loop over every single point, instead you can assign all points within some radius c in your search path to the same mode. In the extreme case, assign c to be equal to your bandwidth h. This feature is optional, and no points will be deducted even if you do not implement it.
- Refine the mean shift clusters with the following conditions:
  - Discard small clusters, i.e. those with less than τ_1 points.
  - Partition large clusters with more than τ_2 points by applying k-means, $K = N_i / \tau_2$, where N_i is the number of points in the i-th cluster.

Part 2: Lattice Model Proposals and Evaluations (20%)

For a cluster of similar keypoints, we can solve for a lattice model, i.e. after applying some global transformation, inlier keypoint should fall onto a uniformly spaced grid (see Figure 2a). The grid of the lattice model is defined by a starting point p_o and its associated basis vector pair (t_1, t_2) (see yellow points in Figure 2a). Depending on the number of clusters present, many proposals will be generated. Use an appearance score[2] (A-score) to evaluate the proposals. The A-score is the average per-pixel standard deviation among the aligned texels (see Figure 2c); it gives low scores to texels that are similar pixel-wise. We aim to find proposals with the lowest A-scores.



(a) An affine transformation projects inlier points (red) onto a grid. Keypoints forming the basis vectors are shown as yellow dots; p_o is at (0,0); (t_1, t_2) are defined by unit vectors from (0,0) to (0,1) and (1,0).


(b) Result with inliers marked as red dots, outliers as blue dots and basis vectors as green arrows.

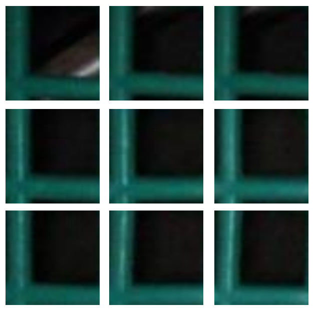

(c) Texels are patches defined by keypoints from 3 or 4 corners, with a perspective transformation. 9 texels shown.

Figure 2: Overview of lattice model proposal. And proposal scoring.

- `get_proposal()` applies RANSAC to find the most suitable basis vector pair for each cluster. For each cluster $C_i, i = 1, \dots, N_c$:
  1. "Randomly" sample a triplet (denoted as yellow points in Figure 2a), with priority given to points close to each other. Denote the three points as $\{a, b, c\}$. Define as point a the point opposite the longest edge of the triangle formed by the three points.

[2] Eq (4) of *Discovering Texture Regularity as a Higher-Order Correspondence Problem* [link]

Estimate an affine transformation matrix M that maps $\{a, b, c\}$ respectively onto the integer lattice basis as $\{(0,0), (1,0), (0,1)\}$. An affine transformation is a limited case of homography (see Lecture 9, slide 9). Following the notation of Lecture 9, $P' = M \cdot P$, where P and P' denote the original and transformed (homogeneous) coordinates locations respectively and $M = \begin{bmatrix} m_1 & m_2 & m_3 \\ m_4 & m_5 & m_6 \\ 0 & 0 & 1 \end{bmatrix}$. To solve for M, use `cv2.getAffineTransform`.

2. Apply M to transform the X points in the cluster closest to a. Count the number of inliers (Figure 2a, red points) whose transformed coordinate falls within a threshold τ_a of an integer position (i, j) on the lattice grid. Remaining points are labelled as outliers (Figure 2b, blue points), i.e. those which are either too far or do not sit on the grid.
3. Repeat Step 1-3 for N_a times and return the triplet $\{a, b, c\}$ with the largest number of inliers.

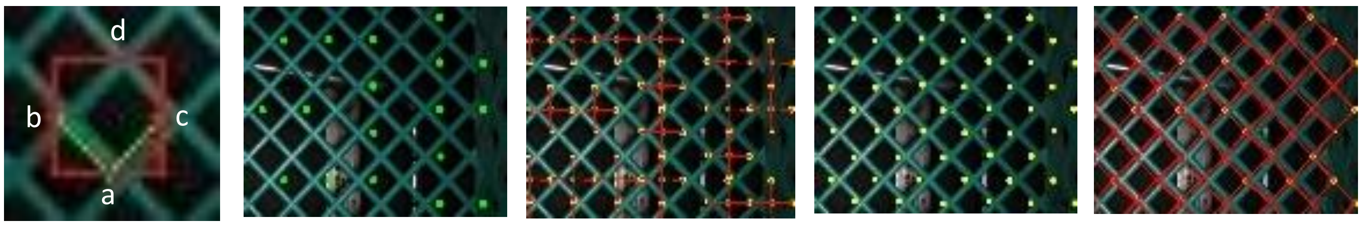
- `find_texels()` forms a set of texels $\{T_k, k = 1, \dots, n\}$ of shape (U, V) based on the inlier keypoint set and their associated integer positions. Each texel is defined by 3 or 4 inlier keypoints on the corners. You are provided code in `lab4.ipynb` to project the texels from the image space into uniform square patches. For each patch, we compute a perspective transformation and warp it with `cv2.getPerspectiveTransform` and `cv2.warpPerspective` (please be careful about the coordinate ordering of these functions). Normalize the intensity of texels so that they have a mean 0.0 and a standard deviation of 1.0 before the calculation of A-score.
- `score_proposal()` computes an appearance score for the texels of a proposal by measuring the standard deviation of the aligned pixels. For a set of n texels, the score is defined as:

$$\text{A-score} = \frac{\sum_{u,v=1,1}^{U,V} \text{std}\big(T_1(u,v), \dots, T_k(u,v), \dots, T_K(u,v)\big)}{U \cdot V \cdot \sqrt{K}},$$

  where $T_k, k = 1, \dots, K$ are the texels and (u, v) are the local pixel indices of the texel. The denominator weighs the score based on $U \cdot V$, the number of pixels in the warped texel; K is the number of texels and std() is standard deviation function. In case of RGB texels, compute the A-score for each channel and then compute an average on the three channels. Keep the 3 proposals with the lowest A-scores for further processing to generate the lattice.

Part 3: Generate Lattice (20%)

The proposals from Part 2 can be applied to capture the repetitive patterns to generate the lattice:



(a) Template marked by red rectangle based on proposal points {a,b,c,d} (b) Local maxima found by template matching (c) Extrapolated lattice grid points from local maxima (d) Merged latticed grid points (e) Visualized lattice

Figure 3: Overview of lattice generation.

- `template_match()` estimates a template and performs template matching. The template is defined by a patch T which is the smallest rectangle that encloses the corner coordinates $\{a, b, c, d\}$ of the proposal, where the point d can be predicted from $\{a, b, c\}$ (see Figure 3a). Perform template matching over the entire image and perform non-maximum suppression to isolate the local peaks similar to the template T of Lab 1 (see Figure 3b).

- `maxima2grid()` estimates 4 lattice grid points from each local maxima. The maxima coincides with the template center; based on the locations of $\{a, b, c, d\}$ on the original template, estimate the analogous grid points $\{a', b', c', d'\}$ for each found maxima (see Figure 3c).
- `refine_grid()` Merge any "duplicate" grid points from neighbouring maxima, e.g. b' with the c' of a maxima to the left, and a' with d' of a neighbour below, by taking the mid-point between the two to recover a set of unique lattice points. Interpolate missing grid points e.g. based on weak maxima (see Figure 3d).
- `grid2latticeunit()` converts each found lattice grid point in image coordinates to the integer lattice grid shown in Figure 2a. For example, the points $\{a, b, c\}$ from the proposal corresponds respectively to $\{(0,0), (1,0), (0,1)\}$. You can implement this any way you want, but clearly describe your intuition in the notebook. We show sample results in the notebook.
  - *Hint: Use the affine transformation to build the mapping. Consider multiple transformations to be applied locally in one region and then extend progressively to cover the entire image.*
  - Note that the lattice grid point location depends highly on the keypoint clusters. Grid points that coincide with any repeated patterns are all acceptable.
- `draw_lattice()` visualizes the grid points and their associated integer grids found by `refine_grid()` and `grid2latticeunit()` (see Figure 3e). This is provided for you.

Part 4: Exploratory Questions (30%)

- **Patch-wise keypoint detection**: Instead of detecting keypoints in a patch-wise manner, apply `cv2.goodFeaturesToTrack` to the entire image `fence.jpg` directly. Compare the difference when you take $N = (P \cdot Q) \cdot N_p$ points for the entire image vs. considering N_p points for each $(P \cdot Q)$ patch. Briefly describe the difference in outcome and explain why differences arise. What are the benefits of patch-wise keypoint detection for translation symmetry detection?
- **Clustering**: Part 1 groups the points first with mean-shift clustering and then refines with a K-means clustering. For the first stage, explain the benefits of applying mean-shift instead of K-means. For the refinement stage, why do we discard the small clusters? Please briefly describe the relationship between the radius c and running time in the efficient implementation of mean-shift clustering.
- **Affine transformation**: In Part 2, affine transformation and RANSAC are used to find proposals. In the function of `get_proposal()`, only the points closest to a are transformed. Why don't we transform all points in the cluster to count the number of inliers?
- **Different samples**: Please apply your algorithm to at least 3 other images in the `inputs` folder. For each image, select two results you think are the best and append them to the notebook. In this question, you do not need to select the two results with the lowest A-score. Instead, you should notice that the A-score does not really give us the best results. Please briefly explain the disadvantages of A-score.
- **Hard samples**: Generate lattices for `wallpaper.jpg` and `house.jpg` and append your results to the notebook. Explain your outcome based on your method for generating the lattices. For those methods which succeed, explain which aspect of the lattice generating algorithm allows it to work on these samples. For those methods which fail, explain why your algorithm fails and possibilities for improvement. Possibilities for failure include not detecting the full lattice structure (some repetitive patterns are not detected), or a distorted structure mis-aligned to the underlying image.