Instructions

This lab has four parts. For each part, implement the outlined functions in `lab1.py` and run them in the notebook `lab1.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. You are free to use NumPy functions, but you may not use `numpy.pad()` or other built-in functions of OpenCV aside from those already in the code template.

Upload to LumiNUS your completed `lab1.py` and `lab1.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 27.08 or 31.08.

Objective

This lab allows you to familiarize yourself with basic image pre-processing steps and learn about template matching. We will be exploring template matching for translation symmetry detection. Template matching applies (normalized) cross-correlation to find local maxima at locations of the image that are similar to the template. From the local maxima, we can generate lattices that represent translation-symmetric structures in the image (see Fig. 1).

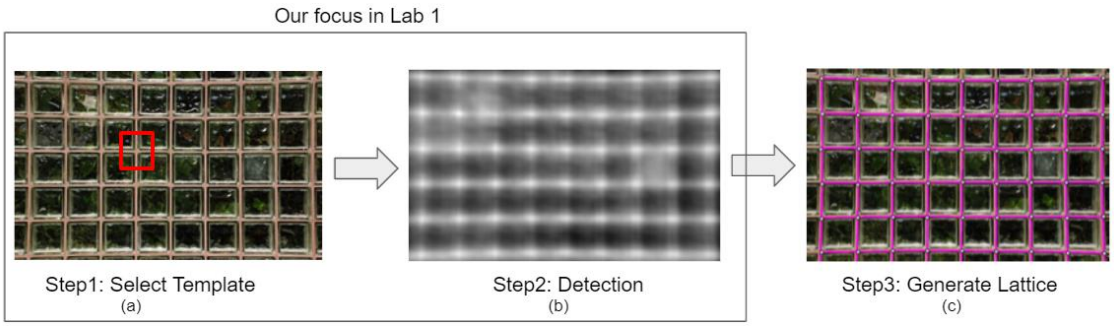

Figure 1: Overview of translation symmetry detection. (a) Original image, with template denoted by red box (b) template matching result, where black and white indicates weak and strong matches respectively. (c) Generated lattice indicated by purple, where vertices correspond to patch centres.

Part 1: Image Preprocessing (15%)

This part implements some basic pre-processing functions required in this lab.

- `pad_zeros()` adds a border of zeros around the input images so that the output size will match the input size after a convolution or cross-correlation operation. Do not to use `numpy.pad()` or other OpenCV functions to implement this function.
- `rgb2gray()` converts a colour image to greyscale. Use `(0.299, 0.587, 0.114)` as the weights for red, green and blue channels respectively.
- `gray2grad()` estimates the gradient map from the grayscale images by convolving with Sobel filters (horizontal and vertical gradients) and Sobel-like filters (gradients oriented at 45° and 135°). For simplicity, you may ignore efficiency issues to be discussed in Part 2.

Part 2: Normalized Cross-Correlation (30%)

This part explores three implementations of various efficiencies for normalized cross-correlation. Starting with a grey-scale-image and a grey-scale template, implement the following three functions.

- `normalized_cross_correlation()` is a naïve version using 4 or 5 nested for-loops iterating over the output and the template. The 4 loops include the height, width of the image, height, width of the template. You can also loop over the channel and there will be 5 for-loops in total.
- `normalized_cross_correlation_fast()` implements the cross correlation with 2 or 3 nested for-loops. The for-loop over the template is replaced with the element-wise multiplication between the kernel and the image regions.
- `normalized_cross_correlation_matrix()` converts cross-correlation into a matrix multiplication operation to leverage optimized matrix operations. To re-formulate as a matrix multiplication, you will need to reshape the template and input image[1].
  - The matrix multiplication to perform is $\mathbf{X}_r = \mathbf{P}_r\mathbf{F}_r$, where $\mathbf{P}_r$ and $\mathbf{F}_r$ are reshaped image and template matrices respectively and $\mathbf{X}_r$ is a reshaped output.
  - $\mathbf{F}_r$ has dimensions $[h_F w_F \times 1]$ where h_F and w_F are the kernel dimensions
  - $\mathbf{P}_r$ has dimensions $[h_X w_X \times h_F w_F]$ where h_X and w_x are the input image dimensions before zero-padding with duplicated elements from the input.
  - $\mathbf{X}_r$ needs another reshape to get the original 2D cross-correlation output $\mathbf{X}$
  - For the normalization term $\frac{1}{|\boldsymbol{F}||\boldsymbol{w}_{ij}|}$, the summation operations for computing the magnitudes can also be solved as "correlation" with a filter or kernel of 1s.
- Generalize your three implementations for colour inputs and templates.
  - Consider the R, G, and B channels as a third dimension of the input image and template matrix. The normalized cross-correlation can be expressed with the same notation as slide 40 of Lecture 2, with c as the channel index

$$x_{ij} = \frac{1}{|\boldsymbol{F}||\boldsymbol{w}_{ij}|}\sum_{u=-k}^{k}\sum_{v=-k}^{k}\sum_{c=1}^{3} f_{uvc}\cdot p_{i+u,j+v,c}$$

  - Re-arranging the matrices for matrix version for coloured images and templates is a bit trickier. Refer to Fig. 2 to help visualize the reshaping operation. The new $\mathbf{F}_r$ will have dimensions $[3h_F w_F \times 1]$ while the new $\mathbf{P}_r$ will have dimensions $[h_X w_X \times 3h_F w_F]$

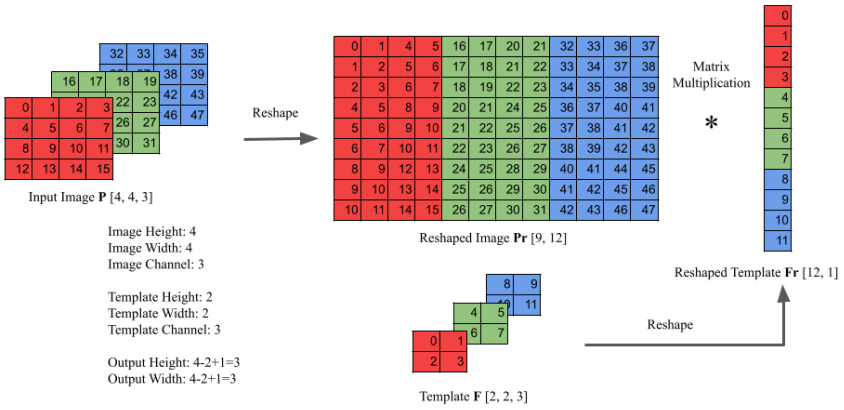

Figure 2: Reshaping operation in `normalized_cross_correlation_matrix()`.

Your actual run-times will differ from our examples depending your hardware, but each version should be successively faster. Note the fastest matrix-multiplication version results in a data array of size is

[1] This specialized type of reshaping is sometimes called image to column, or im2col.

$[3h_P w_P \times 3h_F w_F]$. If your input image and templates get large, this may cause out-of-memory errors. We will consider only the fast version as the default for subsequent parts of the lab.

Part 3: Non-Maximum Suppression (10%)

The matched template locations occur at local maximum in the output $\mathbf{X}$. These maxima can be found via a non-maximum suppression procedure. We will use a greedy form by iteratively finding the global maximum and zeroing the neighbouring region.

- `non maximum suppression()`
  1. Set a threshold τ; values in $\mathbf{X}<\tau$ will not be considered. Set $\mathbf{X}<\tau$ to 0.
  2. While there are non-zero values in $\mathbf{X}$
     a. Find the global maximum in $\mathbf{X}$ and record the coordinates as a local maximum.
     b. Set a small window of size $w \times w$ points centered on the found maximum to 0.
  3. Return all recorded coordinates as the local maximum.

Default values of the parameters τ and w are provided in the Python notebook but you may tune them if you wish to adjust your non-maximum suppression outputs. To help with visualization, the provided `show_img_with_rec()` draws small green squares centered on the found maxima.

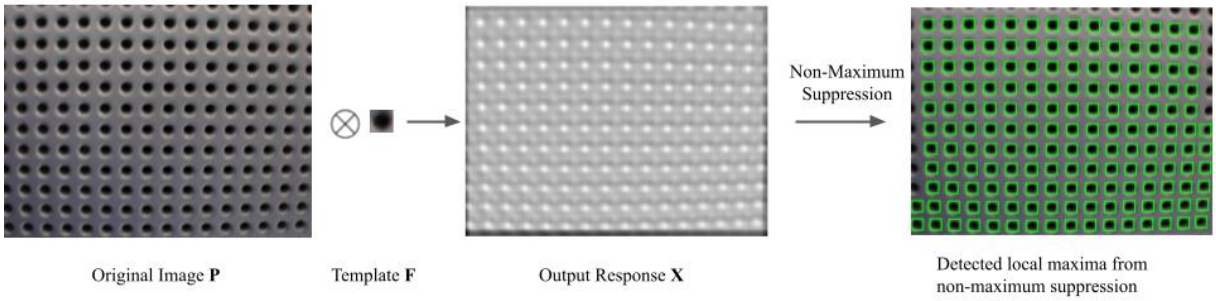

Figure 3: Sample output for normalized cross-correlation with non-maximum suppression, with each found maximum visualized by a green square.

Part 4: Study on Template Matching (45%)

This part explores the various parameters that affects our template matching results. You are also asked to implement a more "practical" version of normalized cross-correlation.

- **Image Inputs (5%).** Apply template matching to the same image in greyscale, RGB or as a set of gradient images. You can consider the gradient images analogous to an RGB image but with 4 channels instead. Verify that you get similar maxima for all three types of inputs.
- **Template Inputs (6%).** Consider the `building.jpg` image. Apply the supplied templates of different patterns and compare the normalized cross-correlation outputs with `show_img_with_rec()`.
  - Explain why their response positions are different.
- **Template Size (6%).** Consider the `holes.jpg` image and compare the normalized cross-correlation outputs with `show_img_with_rec()` using four different templates with a grid of 1x1, 1x3, 3x1 vs. 3x3 holes.
  - How can each of these templates to detect every hole present in the input image?
  - Analyze and explain the subtle differences in the output borders.
  - Describe the correlation output with templates with a 1x1.5 or a 1x2 array of holes.

- **Mean-subtracted cross correlation (16%).** It is more common in practice to subtract the mean from the image window and template[2] before applying normalized cross-correlation. For colour images, the means are calculated separately for each channel, i.e.

$$x_{ij} = \frac{1}{|\boldsymbol{F}'||\boldsymbol{w}'_{ij}|} \sum_{u=-k}^{k} \sum_{v=-k}^{k} \sum_{c=1}^{3} f'_{uvc} \cdot p'_{i+u,j+v,c}$$

  where $f'_{uvc} = f_{uvc} - \frac{1}{(2k+1)^2} \sum_{u'=-k}^{k} \sum_{v'=-k}^{k} f_{u'v'c}$ are elements of the mean-subtracted template $\boldsymbol{F}'$ and $p'_{ijc} = p_{ijc} - \frac{1}{(2k+1)^2} \sum_{u'=-k}^{k} \sum_{v'=-k}^{k} p_{i-u',j-v',c}$, are elements of the mean-subtracted input window $\boldsymbol{w}'_{ij}$.
  - `normalized_cross_correlation_ms()`. Implement this function based on the equations above; for simplicity, use the "fast" version.
  - Apply the mean-subtracted cross correlation with the provided templates to `holes.jpg`; compare the outputs with the version which does not subtract the mean. What are the benefits of subtracting the mean?
- **Auto-correlation (6%).** cross-correlates and input image with itself (i.e. uses the entire image as a template. Apply the provided auto-correlation implementation to `holes.jpg`, and observe the response map. Explain why the correlation output responses decrease as one gets further away from the center of the output.
- **Limitations (6%).** Apply template matching with any version of the cross-correlations to `fence.jpg`. Observe the outputs and explain why template matching fails in certain regions.

[2] See more details at [this link] The version studied in lecture corresponds to `TM_CCORR_NORMED`. The. mean-subtracted version corresponds to `TM_CCOEFF_NORMED`.