CSCI 5561: Assignment #4

FEATURES: HOG and SIFT

1 Submission

- Assignment due: Nov 15th (11:55pm)
- Individual assignment
- Submission through Canvas.
- You will complete hw4.py that contains the following functions:
  - `extract_hog`
  - `get_differential_filter`
  - `filter_image`
  - `get_gradient`
  - `build_histogram`
  - `get_block_descriptor`
  - `face_detection`
  - `find_match`
  - `image_warp_with_sift`

  The code can be found on the canvas.
- The function that does not comply with its specification will not be graded (no credit).
- The code must be run with a Python 3 interpreter.
- Required python packages: `numpy`, `matplotlib`, and `opencv`.
  - numpy & matplotlib: `https://scipy.org/install.html`
  - opencv: `https://pypi.org/project/opencv-python/`
- You are not allowed to use any high-level Python functions of image processing and computer vision, e.g., `cv2.filter2D` unless specified. Please consult the TA if you are unsure about the list of allowed functions.
- We provide a visualization code for HOG. The resulting HOG descriptor must be able to be visualized with the provided code:
  `def visualize_hog(im, hog, cell_size, block_size)`
- We provide a visualization code for face detection. The detected faces must be able to be visualized with the provided code:
  `visualize_face_detection(I_target, bounding_boxes, bb_size)`
- Please follow the Readme file for submission through the autograder.

CSCI 5561: Assignment #4

FEATURES: HOG and SIFT

2 HOG

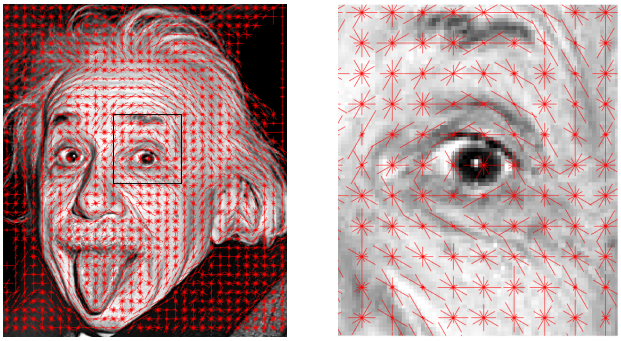


Figure 1: Histogram of oriented gradients. HOG feature is extracted and visualized for (a) the entire image and (b) zoom-in image. The orientation and magnitude of the red lines represents the gradient components in a local cell.

In this assignment, you will implement a variant of HOG (Histogram of Oriented Gradients) in Python proposed by Dalal and Trigg [1] (2015 Longuet-Higgins Prize Winner). It had been long standing top representation (until deep learning) for the object detection task with a deformable part model by combining with a SVM classifier [2]. Given an input image, your algorithm will compute the HOG feature and visualize it as shown in Figure 1 (the line directions are perpendicular to the gradient to show edge alignment). The orientation and magnitude of the red lines represent the gradient components in a local cell.

```
def extract_hog(im):
  ...
  return hog
```

Input: input gray-scale image with `uint8` format.
Output: HOG descriptor.
Description: You will compute the HOG descriptor of input image `im`. The pseudo-code can be found below:

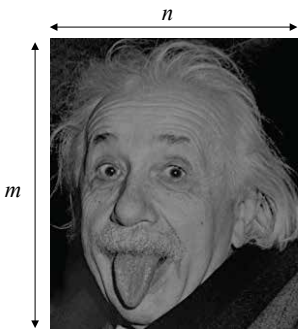
Algorithm 1 HOG

1: Convert the gray-scale image to `float` format and normalize to range [0, 1].
2: Get differential images using `get_differential_filter` and `filter_image`
3: Compute the gradients using `get_gradient`
4: Build the histogram of oriented gradients for all cells using `build_histogram`
5: Build the descriptor of all blocks with normalization using `get_block_descriptor`
6: Return a long vector (`hog`) by concatenating all block descriptors.

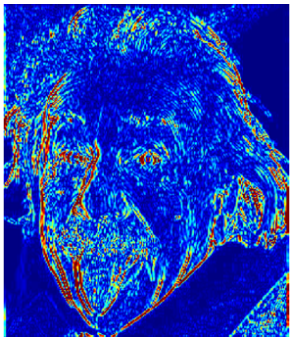
CSCI 5561: Assignment #4

FEATURES: HOG and SIFT

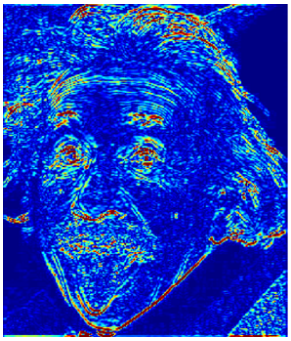
2.1 Image filtering



(a) Input image



(b) Differential along x direction



(c) Differential along y direction

Figure 2: (a) Input image dimension. (b-c) Differential image along x and y directions.

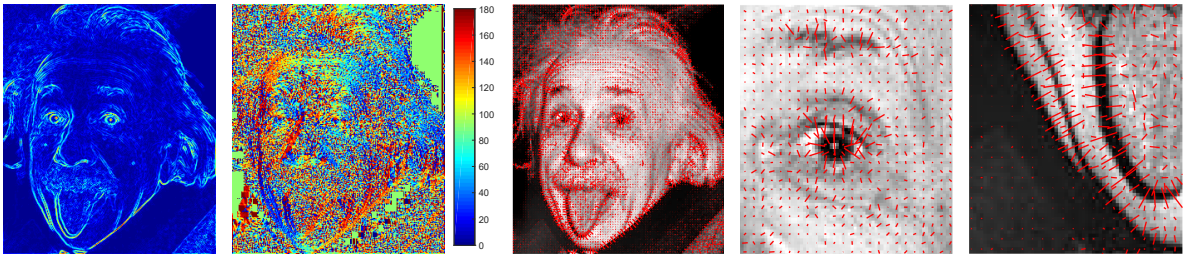
```
def get_differential_filter():
  ...
  return filter_x, filter_y
```

Input: none.
Output: `filter_x` and `filter_y` are 3×3 filters that differentiate along x and y directions, respectively.
Description: You will compute the gradient by differentiating the image along x and y directions. This code will output the differential filters.

```
def filter_image(im, filter):
  ...
  return im_filtered
```

Input: `im` is the gray scale $m \times n$ image (Figure 2(a)) converted to float format and `filter` is a filter ($k \times k$ matrix)
Output: `im_filtered` is $m \times n$ filtered image. You may need to pad zeros on the boundary on the input image to get the same size filtered image.
Description: Given an image and filter, you will compute the filtered image. Given the two functions above, you can generate differential images by visualizing the magnitude of the filter response as shown in Figure 2(b) and 2(c).

2.2 Gradient Computation



(a) Magnitude (b) Angle (c) Gradient (d) Zoomed eye (e) Zoomed neck

Figure 3: Visualization of (a) magnitude and (b) orientation of image gradients. (c-e) Visualization of gradients at every 3rd pixel (the magnitudes are re-scaled for illustrative purpose.).

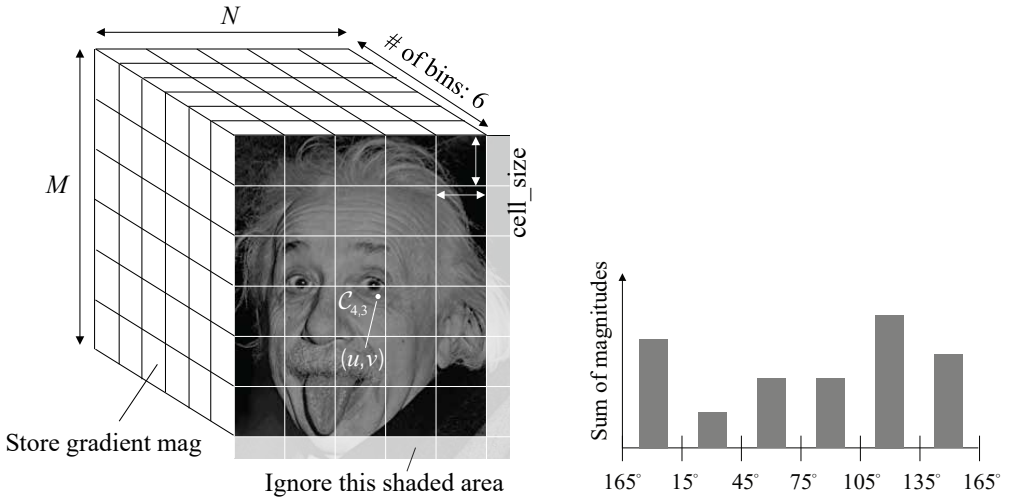
```
def get_gradient(im_dx, im_dy):
  ...
  return grad_mag, grad_angle
```

Input: `im_dx` and `im_dy` are the x and y differential images (size: $m \times n$).
Output: `grad_mag` and `grad_angle` are the magnitude and orientation of the gradient images (size: $m \times n$). Note that the range of the angle should be $[0, \pi)$, i.e., unsigned angle ($\theta == \theta + \pi$).
Description: Given the differential images, you will compute the magnitude and angle of the gradient. Using the gradients, you can visualize and have some sense with the image, i.e., the magnitude of the gradient is proportional to the contrast (edge) of the local patch and the orientation is perpendicular to the edge direction as shown in Figure 3.

CSCI 5561: Assignment #4

FEATURES: HOG and SIFT

2.3 Orientation Binning



(a) ori_histo (b) Histogram per cell

Figure 4: (a) Histogram of oriented gradients can be built by (b) binning the gradients to corresponding bin.

```
def build_histogram(grad_mag, grad_angle, cell_size):
  ...
  return ori_histo
```

Input: `grad_mag` and `grad_angle` are the magnitude and orientation of the gradient images (size: $m \times n$); `cell_size` is the size of each cell, which is a positive integer.
Output: `ori_histo` is a 3D tensor with size $M \times N \times 6$ where M and N are the number of cells along y and x axes, respectively, i.e., $M = \lfloor m/\texttt{cell\_size} \rfloor$ and $N = \lfloor n/\texttt{cell\_size} \rfloor$ where $\lfloor \cdot \rfloor$ is the round-off operation as shown in Figure 4(a).
Description: Given the magnitude and orientation of the gradients per pixel, you can build the histogram of oriented gradients for each cell.

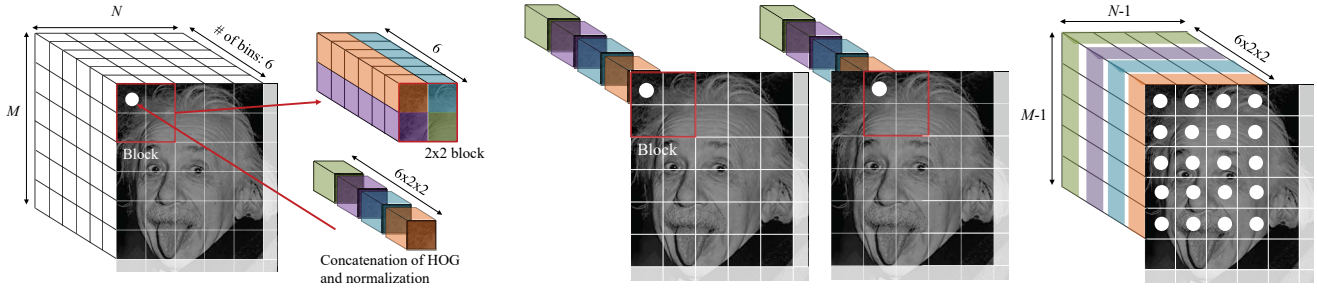
$$\texttt{ori\_histo}(i,j,k) = \sum_{(u,v) \in \texttt{C}_{i,j}} \texttt{grad\_mag}(u,v) \quad \text{if } \texttt{grad\_angle}(u,v) \in \theta_k \tag{1}$$

where $\mathcal{C}_{i,j}$ is a set of x and y coordinates within the (i,j) cell, and θ_k is the angle range of each bin, e.g., $\theta_1 = [165°, 180°) \cup [0°, 15°)$, $\theta_2 = [15°, 45°)$, $\theta_3 = [45°, 75°)$, $\theta_4 = [75°, 105°)$, $\theta_5 = [105°, 135°)$, and $\theta_6 = [135°, 165°)$. Therefore, `ori_histo(i,j,:)` returns the histogram of the oriented gradients at (i,j) cell as shown in Figure 4(b). Using the `ori_histo`, you can visualize HOG per cell where the magnitude of the line proportional to the histogram as shown in Figure 1. Typical `cell_size` is 8.

CSCI 5561: Assignment #4

FEATURES: HOG and SIFT

2.4 Block Normalization



(a) Block descriptor

(b) Block overlap with stride 1

Figure 5: HOG is normalized to account illumination and contrast to form a descriptor for a block. (a) HOG within (1,1) block is concatenated and normalized to form a long vector of size 24. (b) This applies to the rest block with overlap and stride 1 to form the normalized HOG.

```
def get_block_descriptor(ori_histo, block_size):
  ...
  return ori_histo_normalized
```

Input: `ori_histo` is the histogram of oriented gradients without normalization. `block_size` is the size of each block (e.g., the number of cells in each row/column), which is a positive integer.

Output: `ori_histo_normalized` is the normalized histogram (size: $(M-(\texttt{block\_size}-1)) \times (N-(\texttt{block\_size}-1)) \times (6 \times \texttt{block\_size}^2)$.

Description: To account for changes in illumination and contrast, the gradient strengths must be locally normalized, which requires grouping the cells together into larger, spatially connected blocks (adjacent cells). Given the histogram of oriented gradients, you apply L_2 normalization as follow:

1. Build a descriptor of the first block by concatenating the HOG within the block. You can use `block_size`=2, i.e., 2×2 block will contain $2 \times 2 \times 6$ entries that will be concatenated to form one long vector as shown in Figure 5(a).
2. Normalize the descriptor as follow:

$$\hat{h}_i = \frac{h_i}{\sqrt{\sum_i h_i^2 + e^2}} \tag{2}$$

   where h_i is the i^{th} element of the histogram and $\hat{h}_i$ is the normalized histogram. e is the normalization constant to prevent division by zero (e.g., $e = 0.001$).
3. Assign the normalized histogram to `ori_histo_normalized`(1,1) (white dot location in Figure 5(a)).
4. Move to the next block `ori_histo_normalized`(1,2) with the stride 1 and iterate 1-3 steps above.

The resulting `ori_histo_normalized` will have the size of $(M-1) \times (N-1) \times 24$.

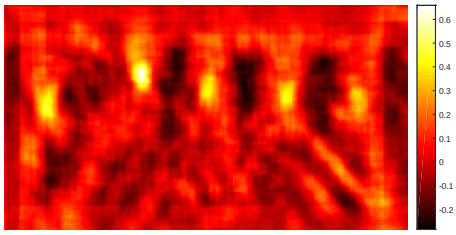
CSCI 5561: Assignment #4

FEATURES: HOG and SIFT

2.5 Application: Face Detection

(a) Template image

(b) Target image



(c) Response map

(d) Thresholding

(e) Non-maximum suppression

Figure 6: You will use (a) a single template image to detect faces in (b) the target image using HOG descriptors. (c) HOG descriptors from the template and target image patches can be compared by using the measure of normalized cross-correlation (NCC). (d) Thresholding on NCC score will produce many overlapping bounding boxes. (e) Correct bounding boxes for faces can be obtained by using non-maximum suppression.

Using the HOG descriptor, you will design a face detection algorithm. You can download the template and target images from here:
`https://www-users.cs.umn.edu/~hspark/csci5561_S2020/hw1.zip`.

```
def face_recognition(I_target, I_template):
  ...
  return bounding_boxes
```

Input: `I_target` is the image that contains multiple faces. `I_template` is the template face image that will be matched to the image to detect faces.
Output: `bounding_boxes` is $n \times 3$ array that describes the n detected bounding boxes. Each row of the array is $[x_i, y_i, s_i]$ where (x_i, y_i) is the left-top corner coordinate of the i^{th} bounding box, and s_i is the normalized cross-correlation (NCC) score between the bounding box patch and the template:

$$s = \frac{\overline{\mathbf{a}} \cdot \overline{\mathbf{b}}}{\|\overline{\mathbf{a}}\| \|\overline{\mathbf{b}}\|} \tag{3}$$

where $\overline{\mathbf{a}}$ and $\overline{\mathbf{b}}$ are two normalized descriptors, i.e., zero mean:

$$\overline{a}_i = a_i - \widetilde{\mathbf{a}} \tag{4}$$

where $\overline{a}_i$ is the i^{th} element of $\mathbf{a}$, and a_i is the i^{th} element of the HOG descriptor. $\widetilde{\mathbf{a}}$ is the mean of the HOG descriptor.

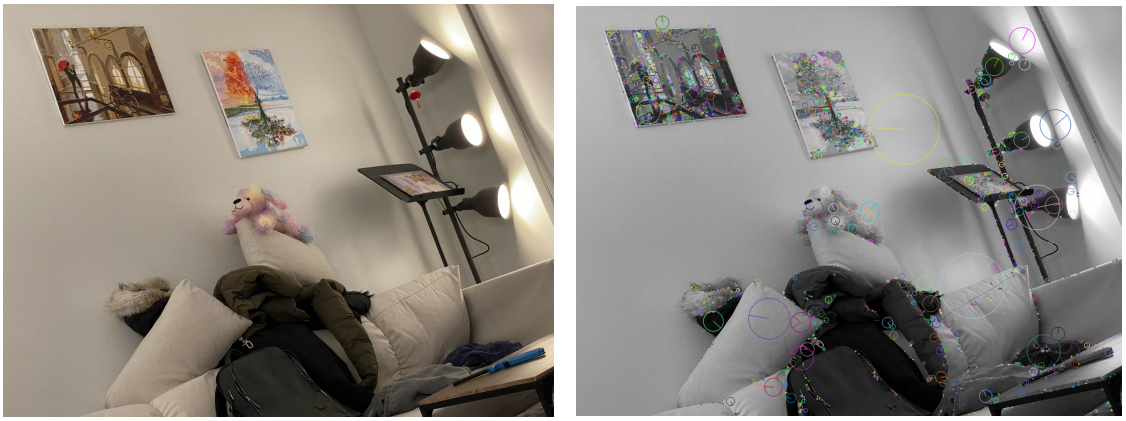
Description: You will use thresholding and **non-maximum suppression** with IoU 50% to localize the faces.

You may use `visualize_face_detection(I_target, bounding_boxes, bb_size)` to visualize your detection.

CSCI 5561: Assignment #4

FEATURES: HOG and SIFT

3 Image Warping with SIFT Features



(a) Image (b) SIFT

Figure 7: Given an image (a), you will extract SIFT features using OpenCV.

The goal of this question is to give you some experience with a complete computer vision pipeline using image features and projective geometry based methods you have learned so far. In the previous section, you worked on the HOG features for an image. In this part, you'll extract and match SIFT features between images to obtain correspondences. Then, you'll use them to warp the images, as you've done in the previous homework.

Algorithm 2 Image Warping with SIFT

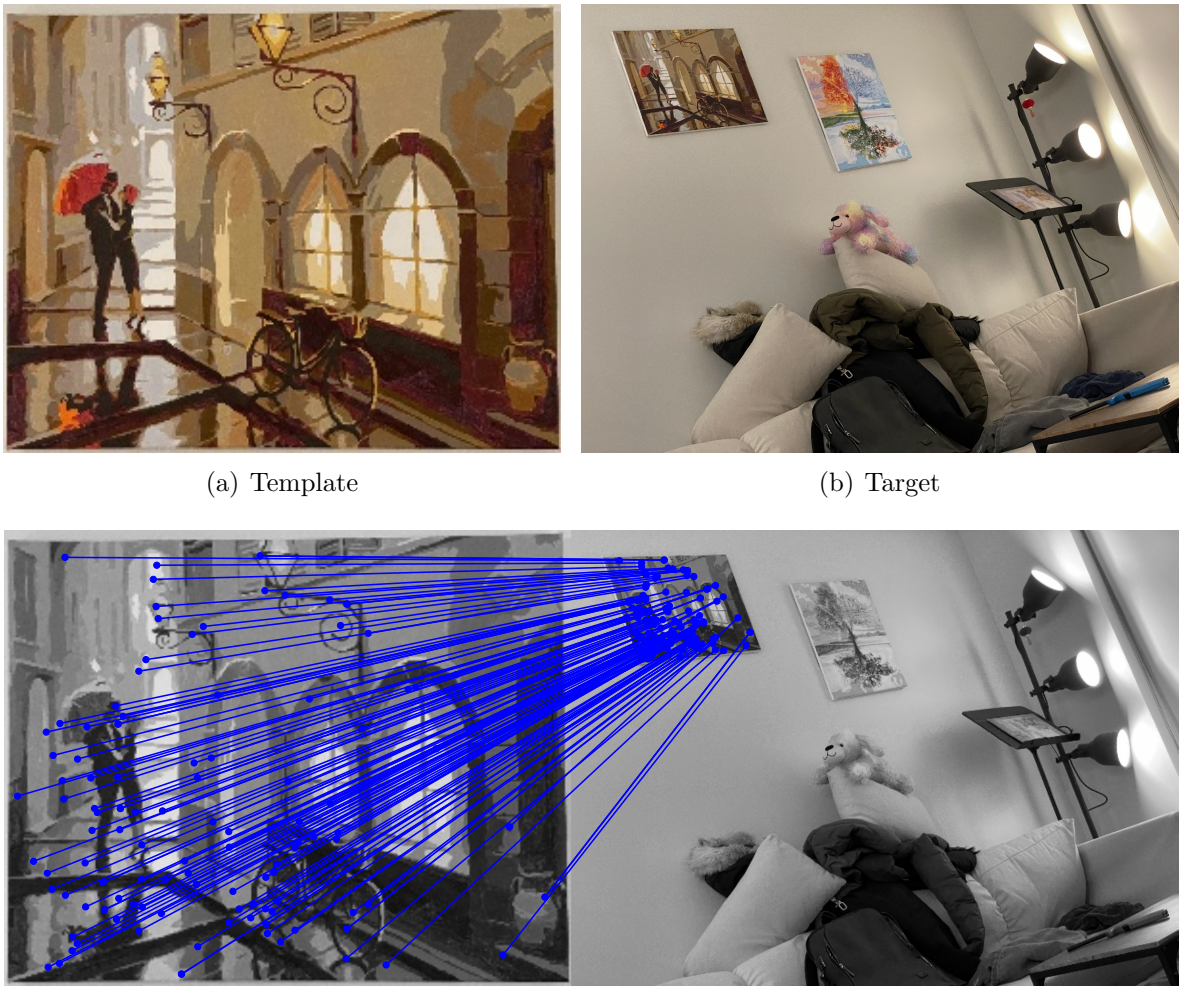
1: Extract SIFT features from both images.
2: Find match keypoints in `find_match`
3: Calculate affine transformation between the template and the target image.
4: Warp the **target** image to the **template** coordination frame and compare.

3.1 Sift Feature Extraction and Matching

We will use OpenCV library for SIFT features extraction given your images.

(Note) Using the PROVIDED function `visualize_sift`, you can visualize SIFT features with scale and orientation as shown in Figure 7. SIFT is now included with the *core* OpenCV library, as the patent for the algorithm expired in 2020. But the older version of OpenCV will not have it in the core library, so please make sure you have a recent version of OpenCV.

The SIFT is composed of scale, orientation, and 128-dimensional local feature descriptor (integer), $\mathbf{f} \in \mathbb{Z}^{128}$. You will use the SIFT features to match between two images, I_1



(a) Template (b) Target

(c) SIFT matches with ratio test

Figure 8: You will match points between the template and target image using SIFT features.

and I_2. Use two sets of descriptors from the template and target, and find the matches using the nearest neighbor with the ratio test. You may use `NearestNeighbors` function imported from `sklearn.neighbors` (You can install `sklearn` package easily by "pip3 install -U scikit-learn").

```
def find_match(img1, img2, dist_thr=.7):
  ...
  return x1, x2
```

Input: two input gray-scale images with `uint8` format.
Output: `x1` and `x2` are $n \times 2$ matrices that specify the correspondence.
Description: Each row of `x1` and `x2` contains the (x, y) coordinate of the point correspondence in I_1 ad I_2, respectively, i.e., `x1(i,:)` $\leftrightarrow$ `x2(i,:)`.

(Note) You can only use the SIFT module of `OpenCV` for the SIFT descriptor extraction. You may want to follow the `https://docs.opencv.org/4.x/da/df5/tutorial_py_sift_intro.html`

You need to implement the matching with Lowe's ratio test yourself. You may follow this article `https://datahacker.rs/feature-matching-methods-comparison-in-opencv/`. However, you will need to perform the ratio test of a match in both directions and use cv2 functions only for computing sift features.

(Note) You will use a ratio of 70%.

3.2 Image alignment and warping

Now that you have correspondence between the two images. You've used them in both homework 2 and homework 3. Since you've already done the homography calculation and image warping in homework 2. You are allowed to use `OpenCV` functions directly.

3.2.1 Homography

In the 3rd step of algorithm 3, you'll calculate a 3×3 affine transformation. The affine transform will transform x_1 to x_2, i.e., $x_2 = Ax_1$. You may use `cv2.estimateAffine2D`. To validate your implementation, you may visualize the transformed template's inliers and boundaries.

3.2.2 Image warping

In the 4th step of the algorithm. 3, given an affine transform A, you will write a code to warp an image $I(x) \rightarrow I(Ax)$. You'll use `cv2.warpPerspective` to do the image warping.

(Validation) Using the warped image, the error map $|I_{\text{tpl}} - I_{\text{wrp}}|$ can be computed to validate the correctness of the transformation, where I_{tpl} and I_{wrp} are the template and warped images.

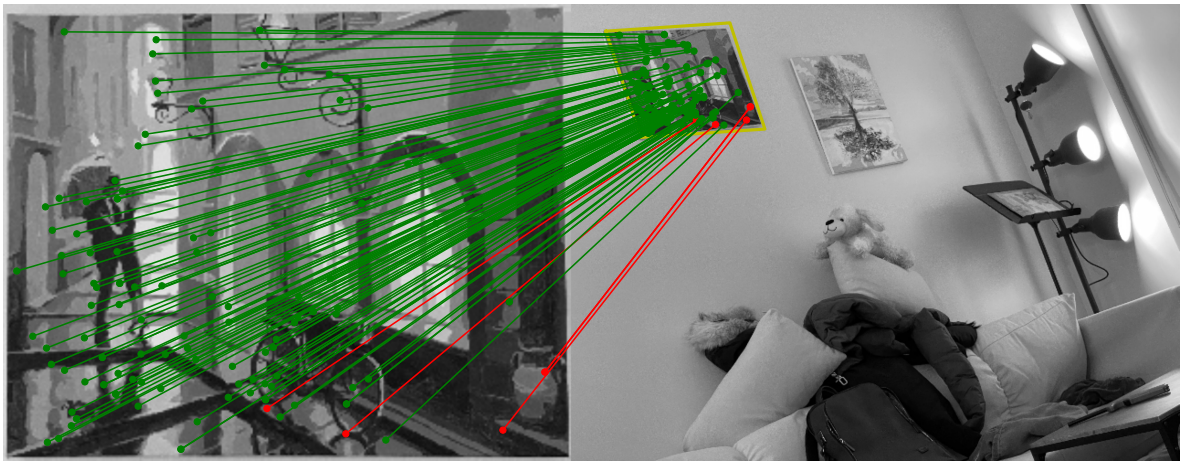


Figure 9: You will compute an affine transform using SIFT matches filtered by RANSAC. Red: outliers; Green: inliers; Yellow: the boundary of the transformed template.

References

[1] N. Dalal and B. Triggs, "Histograms of oriented gradients for human detection," in *CVPR*, 2005.

[2] P. F. Felzenszwalb, R. B. Girshick, D. McAllester, and D. Ramanan, "Object detection with discriminatively trained part based models," *TPAMI*, 2010.

CSCI 5561: Assignment #4

FEATURES: HOG and SIFT

(a) Warped image (b) Template (c) Error map

Figure 10: You will use the affine transform to warp the target image to the template using the inverse mapping. Using the warped image, the error map $|I_{\text{tpl}} - I_{\text{wrp}}|$ can be computed to validate the correctness of the transformation where I_{tpl} and I_{wrp} are the template and warped images.