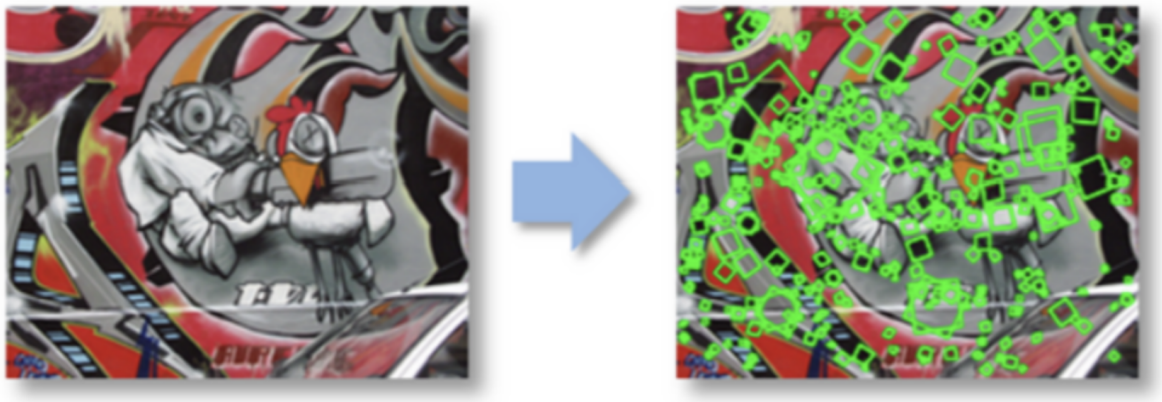
Project 2: Feature Detection Matching

CS4670/5670: Computer Vision, Spring 2024



Assigned: March 27, 2024
Teams: This assignment can be done in groups of 2 students or individually
Due: April 13, 2024 at 11:59 PM
Late Due April 23, 2024 at 11:59 PM
Submission (CMS) feature_detection.py

SETUP

Skeleton code: Available through CMS.
Environment: Please use Python 3.6 or higher and install the following required packages:

```
pip install numpy scipy matplotlib Pillow opencv-python
```

SYNOPSIS

The goal of feature detection and matching is to identify a pairing between a point in one image and a corresponding point in another image. These correspondences can then be used to stitch multiple images together into a panorama.

In this project, you will write code to detect discriminating features (which are reasonably invariant to translation, rotation, and illumination) in an image and find the best matching features in another image.

To help you visualize the results and debug your program, we provide a jupyter notebook that displays detected features and best matches in another image.

The code you need to write will be for your feature detection methods, your feature descriptor methods and your feature matching methods. All your required edits will be in **feature_detection.py**.

1. FEATURE DETECTION

In this step, you will identify points of interest in the image using the Harris corner detection method. The steps are as follows (see the lecture slides/readings for more details). For each point in the image, consider a window of pixels around that point. Compute the Harris matrix H for (the window around) that point, defined as

$$\begin{aligned} H &= \sum_p w_p \nabla I_p (\nabla I_p)^T \\ &= \sum_p w_p \begin{bmatrix} I_{x_p}^2 & I_{x_p} I_{y_p} \\ I_{x_p} I_{y_p} & I_{y_p}^2 \end{bmatrix} \\ &= \begin{bmatrix} w_p I_{x_p}^2 & w_p I_{x_p} I_{y_p} \\ w_p I_{x_p} I_{y_p} & w_p I_{y_p}^2 \end{bmatrix} \\ &= \begin{bmatrix} \sum_p w_p I_{x_p}^2 & \sum_p w_p I_{x_p} I_{y_p} \\ \sum_p w_p I_{x_p} I_{y_p} & \sum_p w_p I_{y_p}^2 \end{bmatrix} \end{aligned}$$

where the summation is over all pixels p in the window. I_{x_p} is the x derivative of the image at point p, the notation is similar for the y derivative. You should use the 3×3 Sobel operator to compute the x, y derivatives. Extrapolate pixels outside the image by repeating border values. (Note that NumPy calls this "edge" while SciPy calls this "nearest"). The weights w_p should be circularly symmetric (for rotation invariance) - use a 5×5 Guassian mask with 0.5 sigma. Note that H is a 2×2 matrix. Then use H to compute the corner strength function, $c(H)$, at every pixel. Note

$$c(H) = \det(H) - 0.1(\mathrm{Tr}(H))^2$$

.
We will also need the orientation in degrees at every pixel. Compute the approximate orientation as the angle of the gradient. The zero angle points to the right and positive angles are counter-clockwise. Note: do not compute the orientation by eigen analysis of the structure tensor.

We will select the strongest keypoints (according to $c(H)$) which are local maxima in a 7×7 neighborhood.

ToDo

The function **detectCorners** is one of the main ones you will complete, along with the helper functions **computeHarrisValues** (computes Harris scores and orientation for each

pixel in the image) and **computeLocalMaximaHelper** (computes a Boolean array which tells for each pixel if it is a local maximum). These implement Harris corner detection. You may find the following functions helpful:

- *scipy.ndimage.sobel*: Filters the input image with Sobel filter
- *scipy.ndimage.gaussian_filter*: Filters the input with a Gaussian filter
- *scipy.ndimage.filters.maximum_filter*: Filters the input image with a maximum filter
- *scipy.ndimage.convolve*: Filters the input image with the selected filter

You can look up these functions on the Scipy documentation page.

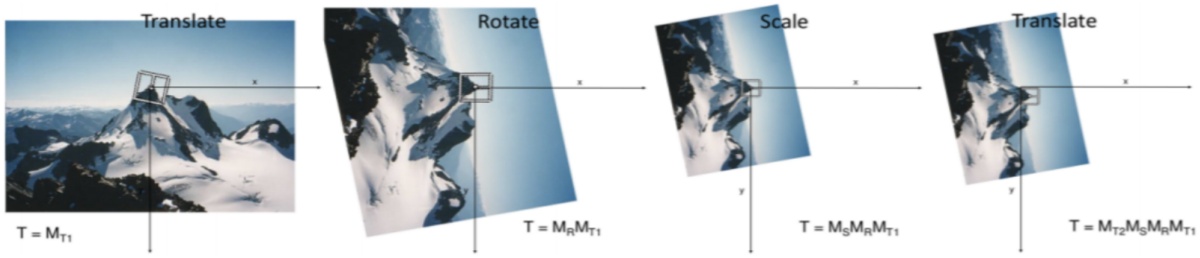
2. FEATURE DESCRIPTION

Now that you have identified points of interest, the next step is to come up with a descriptor for the feature centered at each interest point. This descriptor will be the representation you use to compare features in different images to see if they match.

You will implement two descriptors for this project. For starters, you will implement a simple descriptor which is the pixel intensity values in the 5×5 neighborhood. This should be easy to implement and should work well when the images you are comparing are related by a translation.

Second, you will implement a simplified version of the MOPS descriptor. You will compute an 8×8 oriented patch sub-sampled from a 40×40 pixel region around the feature. You have to come up with a transformation matrix which transforms the 40×40 rotated window around the feature to an 8×8 patch rotated so that its keypoint orientation points to the right. You should also normalize the patch to have zero mean and unit variance.

You will use **cv2.warpAffine** to perform the transformation. **warpAffine** takes a 2×3 forward warping afffine matrix, which is multiplied from the left so that transformed coordinates are column vectors. The easiest way to generate the 2×3 matrix is by combining multiple transformations. A sequence of translation (T1), rotation (R), scaling (S) and translation (T2) will work. Left-multiplied transformations are combined right-to-left so the transformation matrix is the matrix product T2 S R T1. The figures below illustrate the sequence.



ToDo

You will need to implement a MOPS feature descriptor within the **computeMOPSDescriptors** function. This function takes the location and orientation information already stored in a set of key points (e.g., Harris corners), and compute descriptors for these key points, then store these descriptors in a NumPy array which has the same number of rows as the computed key points and columns as the dimension of the feature.

For the MOPS descriptor implementation, you should create a matrix which transforms the 40×40 patch centered at the key point to a canonical orientation and scales it down by 5, as described in the lecture slides. You may find the functions in **transformations.py** helpful. Reading the opencv documentation for **cv2.warpAffine** is recommended.

3. FEATURE MATCHING

Now that you have detected and described your features, the next step is to write code to match them (i.e., given a feature in one image, find the best matching feature in another image).

The simplest approach is the following: compare the two features and calculate a scalar distance between them. The best match is the feature with the smallest distance. You will implement the following distance function:

1. **Ratio test**: Find the closest and second closest features by the sum of squared distances (SSD):

   $$d(x, y) = \|x, y\|^2$$

   .

   The ratio test distance is their ratio (i.e. SSD distance of the closest feature match divided by SSD distance of the second closest feature match).

ToDo

Finally, you will implement a function for matching features. You will implement the **produceMatches** function which will return a tuple of all matches in the following form of (idx1, idx2, score):

- **idx1:** the index in the first image that is matched
- **idx2:** the index in the corresponding second image that is matched
- **score:** the scalar difference between the points defined by the ratio test

You may find **scipy.spatial.distance.cdist** and **numpy.argmin** helpful when implementing these functions.

NOTES

1. TESTING

You can test your TODO code by running "python tests.py". This will load a small image, run your code and compare against the correct outputs. This will let you test your code incrementally without needing all the TODO blocks to be complete.

We have implemented tests for TODO's 1-5. You should design your own tests for TODO 6. Lastly, be aware that the supplied tests are very simple - they are meant as an aid to help you get started. Passing the supplied test cases does not mean the graded test cases will be passed.

2. RUNTIME

Your code should (ideally) run through test.py in less than 20 seconds. We will be grading your code using a separate, more comprehensive test suite. Most solutions should take 40 seconds to finish, but we will cap runtime at 40 minutes. So even if your tests are running slow, we have a very lenient runtime constraint. You should not be worried unless you notice a significant lag in your code.

3. CODING RULES

You may use NumPy, SciPy and OpenCV2 functions to implement mathematical, filtering and transformation operations. Do not use functions which implement keypoint detection or feature matching.

Here is a list of potentially useful functions (you are not required to use them):

- scipy.ndimage.sobel
- scipy.ndimage.gaussian_filter
- scipy.ndimage.filters.convolve
- scipy.ndimage.filters.maximum_filter
- scipy.spatial.distance.cdist

- cv2.warpAffine
- np.max, np.min, np.std, np.mean, np.argmin, np.argpartition
- np.degrees, np.radians, np.arctan2,
- transformations.get_rot_mx (in **transformations.py**)
- transformations.get_trans_mx (in **transformations.py**)
- transformations.get_scale_mx (in **transformations.py**)

4. VISUALIZATION

Now you are ready to go! Using the jupyter notebook, you can view detected features and feature matches that your algorithm computes for a set of images that we provide. This jupyter notebook is a tool to help you visualize the keypoint detections and feature matching results. Keep in mind that your code will be evaluated numerically, not visually.