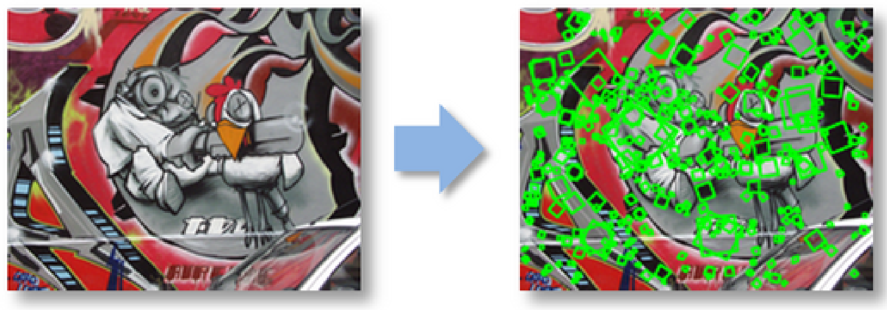
CS4670/5670: Computer Vision, Spring 2020

Project 2: Feature Detection and Matching



Assigned: Friday, March 13, 2020
Teams: This assignment can be done in groups of 2 students or individually.
Deadline: Friday, March 27, 2020 at 11:59 PM
Submission (CMS): features.py
[Do not submit extracredit.py extracredit_readme.txt, and/or ROC.jpg unless you have completed the extra credit.]

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
```

Extra images: Look inside the resources subdirectory.

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 user interface that displays detected features and best matches in another image. We also provide an example ORB feature detector, a current best of breed technique in the vision community, for comparison.

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 **features.py**.

The feature computation and matching methods are called from the UI functions in **featuresUI.py**. Feel free to extend the functionality of the UI code but remember that only the code in features.py will be graded.

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

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

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 3x3 Sobel operator to compute the x, y derivatives. Extrapolate pixels outside the image by repeating the 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 5x5 Gaussian mask with 0.5 sigma. Note that H is a 2x2 matrix.

Then use H to compute the corner strength function, c(H), at every pixel.

$$c(H) = det(H) - 0.1(trace(H))^2$$

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 7x7 neighborhood.

Todo

The function **detectKeypoints** in **HarrisKeypointDetector** 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 **computeLocalMaxima** (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 image with a Gaussian filter.
- *scipy.ndimage.filters.maximum_filter*: Filters the input image with a maximum filter.
- *scipy.ndimage.filters.convolve*: Filters the input image with the selected filter.

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

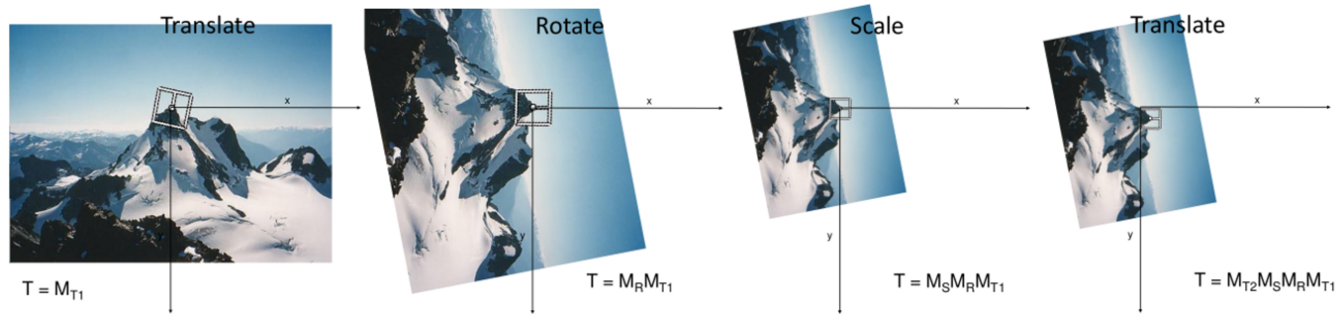
2. Feature description

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 5x5 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 8x8 oriented patch sub-sampled from a 40x40 pixel region around the feature. You have to come up with a transformation matrix which transforms the 40x40 rotated window around the feature to an 8x8 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 2x3 forward warping afffine matrix, which is multiplied from the left so that transformed coordinates are column vectors. The easiest way to generate the 2x3 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 two feature descriptors in **SimpleFeatureDescriptor** and **MOPSFeaturesDescriptor** classes. The **describeFeatures** function of these classes take 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 (e.g., 25 for the 5x5 simple feature descriptor).

For the MOPS implementation, you should create a matrix which transforms the 40x40 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 two features and calculate a scalar *distance* between them. The best match is the feature with the smallest distance. You will implement two distance functions:

1. Sum of squared differences (SSD):

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

2. The ratio test: Find the closest and second closest features by SSD distance. 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 **matchFeatures** function of **SSDFeatureMatcher** and **RatioFeatureMatcher**. These functions return a list of *cv2.DMatch* objects. You should set the *queryIdx* attribute to the index of the feature in the first image, the *trainIdx* attribute to the index of the feature in the second image and the *distance* attribute to the distance between the two features as defined by the particular distance metric (e.g., SSD or ratio). 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 1-6. You should design your own tests for TODO 7 and 8. 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, and match features using the Yosemite benchmark images in less than 5 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 20 minutes. So even if your tests/UI 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`
- `transformations.get_scale_mx`

4. Visualization

Now you are ready to go! Using the UI and skeleton code that we provide, you can load in a set of images, view the detected features, and visualize the feature matches that your algorithm computes. Watch this video to see the UI in action.

By running *featuresUI.py*, you will see a UI where you have the following choices:

- ***Keypoint Detection***
  You can load an image and compute the points of interest with their orientation.
- ***Feature Matching***
  Here you can load two images and view the computed best matches using the specified algorithms.
- ***Benchmark***
  After specifying the path for the directory containing the dataset, the program will run the specified algorithms on all images and compute *ROC curves* for each image. (You can learn more about ROC curves on the web here and here).

The UI 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.

We are providing a set of benchmark images to be used to test the performance of your algorithm as a function of different types of controlled variation (i.e., rotation, scale, illumination, perspective, blurring).

You should also go out and take some photos of your own to see how well your approach works on more interesting data sets. For example, you could take images of a few different objects (e.g., books, offices, buildings, etc.) and see how well it works!

5. Extra Credit

Implement **CustomKeypointDetector** and/or **CustomFeatureDescriptor** to improve mean AUC for all reference directories by at least 5%, using either the ratio test or SSD . There are no runtime constraints on the extra credit. Here are two suggestions and you are encouraged to come up with your own ideas as well!

- Implement adaptive non-maximum suppression (MOPS paper)
- Make your feature detector scale invariant.

You must submit extra credit solutions to CMS **in addition** to features.py (i.e. after you have completed the assignment, duplicate features.py as extracredit.py and modify this instead). The first solution will be graded by the project rubric so it must implement the algorithm described in this document. Extra credit solutions should include **extracredit_readme.txt** that describes your changes to the algorithm and the thresholds for each benchmark image. The submitted code will be checked against this description. Extra credit submissions which do not improve mean AUC by at least 15% will not be graded. You can measure your mean AUC by running the UI benchmark on the 5 datasets (bikes, graf, leuven, wall, yosemite).

If you are worried about runtime for your extra credit solution, also submit a screenshot of one of your resulting ROC curves. However, this submission is optional, and is only helpful if course staff has difficulty running your code in a reasonable amount of time.

It is important that you solve the main assignment first before attempting the extra credit. The main assignment will be worth significantly more points than the extra credit.

Acknowledgements

The instructor is extremely thankful to Prof. Steve Seitz for allowing us to use their project as a basis for our assignment.