CSC 589 Introduction to Computer Vision

Project 4: Local Image feature matching

Brief

Due date: April 24, Friday, end of day
Team work: encouraged, maximum number of members, 3
Totals points: 100pts + 20pts

Reminder: This project is tedious and quite hard. Start early!

The good news is that if you really finish this project with success, you are really getting on the hard-core of computer vision!!

Overview

The goal of this assignment is to create a local feature-matching algorithm using techniques described in Szeliski chapter 4.1. The pipeline we suggest is a simplified version of the famous **SIFT** pipeline. The matching pipeline is intended to work for *instance-level* matching -- multiple views of the same physical scene.

Details

For this project, you need to implement the three major steps of a local feature matching algorithm:

- Interest point detection in get_interest_points.py (see Szeliski 4.1.1)
- Local feature description in get_features.py (see Szeliski 4.1.2)
- Feature Matching in match_features.py (see Szeliski 4.1.3)

There are numerous papers in the computer vision literature addressing each stage. For this project, we will suggest specific, relatively simple algorithms for each stage. You are encouraged to experiment with more sophisticated algorithms!

Interest point detection (get_interest_points.py)

You will implement the Harris corner detector as described in the lecture materials and Szeliski 4.1.1. See Algorithm 4.1 in the textbook for pseudocode. The Solem book provided decent code as well. The starter code gives some additional suggestions. You do not need to worry about scale invariance or keypoint orientation estimation for your baseline Harris corner detector.

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

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

$$= \sum_p \begin{pmatrix} w_p I_{x_p}^2 & w_p I_{x_p} I_{y_p} \\ w_p I_{x_p} I_{y_p} & w_p I_{y_p}^2 \end{pmatrix} = \begin{pmatrix} \sum_p w_p I_{x_p}^2 & \sum_p w_p I_{x_p} I_{y_p} \\ \sum_p w_p I_{x_p} I_{y_p} & \sum_p w_p I_{y_p}^2 \end{pmatrix}$$

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 Sobel operator to compute the x, y derivatives. The weights w_p should be chosen to be circularly symmetric (for rotation invariance). You should use a 5x5 Gaussian mask with 0.5 sigma.

Note that H is a 2x2 matrix. To find interest points, first compute the corner strength function:

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

For filtering the image and computing the gradients, you can either use the following functions or implement you own filtering code as you did in the second assignment:

- *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

Local feature description (get_features.py)

You will implement a SIFT-like local feature as described in the lecture materials and Szeliski 4.1.2. See the placeholder get_features.py for more details. If you want to get your matching pipeline working quickly (and maybe to help debug the other algorithm stages), you might want to start with normalized patches (MOSP, 4.10, chapter 4.1) as your local feature. Again, you can refer to any online materials for computing SIFT features. But you are not allowed to use SIFT from OpenCV.

Bells & whistles for local feature description. The simplest thing to do is to experiment with the numerous SIFT parameters: how big should each feature be? How many local cells should it have? How many orientations should each histogram

have? Different normalization schemes can have a significant effect, as well. Don't get lost in parameter tuning, though. If your keypoint detector can estimate orientation, your local feature descriptor should be built accordingly so that your pipeline is rotation invariant. Likewise, if you are detecting keypoints at multiple scales, you should build the features at the corresponding scales. You can also try different spatial layouts for your feature (e.g. GLOH) or entirely different features (e.g. local self-similarity).

Feature matching (match_features.py)

You will implement the "ratio test" or "nearest neighbor distance ratio test" method of matching local features as described in the lecture materials and Szeliski 4.1.3. See equation 4.18 in particular. Simply compute all pairs of distances between all features.

1. The SSD (Sum of Squared Distances) distance. This basically means computing the Euclidean distance between the two feature vectors. You must implement this distance.
2. The Ratio test. Compute (SSD distance of the best feature match)/(SSD distance of the second best feature match) and store this value as the distance between the two feature vectors. This is called the "ratio test". You must implement this distance.

Bells & whistles: Report the performance (i.e., the ROC curve and AUC) on the provided benchmark image sets. Refer to lecture slides for ROC curve.

Using the starter code (proj2.py)

The top-level proj2.py script provided in the starter code includes file handling, visualization, and evaluation functions for you as well as calls to placeholder versions of the three functions listed above. Running the starter code without modification will visualize random interest points matched randomly on the particular Notre Dame images shown at the top of this page. For these two images there is a ground truth evaluation in the starter code, as well. evaluate_correspondence.py will classify each match as correct or incorrect based on similarity to hand-provided matches (run show_ground_truth_corr.py to see the ground truth annotations).

As you implement your feature matching pipeline, you should see your performance according to evaluate_correspondence.py increase. Hopefully you find this useful, but don't *overfit* to these particular (and relatively easy) images. The baseline algorithm suggested here and in the starter code will give you full credit and work fairly well on these Notre Dame images. If you want to use additional images, you might have to

built your own ground truth or find them on the Internet. Let me know if you are interested in pursuing this.

Useful functions

Here is a summary of potentially useful functions (you do not have to use any of these):

- `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`

Write up

For this project, and all other projects, you must do a project report in HTML. We provide you with a placeholder .html document which you can edit. In the report you will describe your algorithm and any decisions you made to write your algorithm a particular way. Then you will show and discuss the results of your algorithm.

In the case of this project, show how well your matching method works not just on the Notre Dame image pair, but on additional test cases. For the Notre Dame images, you can show eval.jpg which the starter code generates. For other image pairs, there is no ground truth evaluation (you can make it!) so you can show vis.jpg instead. A good writeup will assess how important various design decisions were. E.g. by using SIFT-like features instead of normalized patches, I went from 70% good matches to 90% good matches. This is especially important if you did some of the More Bells & Whistles and want extra credit. You should clearly demonstrate how your additions changed the behavior on particular test class.

Handing in

This is very important, as you will lose points if you do not follow instructions. Every time after the first that you do not follow instructions, you will lose 5 points. The folder you hand in must contain the following:

- README - text file containing anything about the project that you want to tell the TAs
- code/ - directory containing all your code for this assignment
- html/ - directory containing all your html report for this assignment (including images). Only this folder will be published to the course web page, so your webpage cannot contain pointers to images in other folders of your hand-in.
- html/index.html - home page for your results
- Zip the folder and submit to blackboard with your names.
- Please do not change the structure of the folder, especially the style folder.
- You can upload your folder onto your own website. If you do so, you are still required to submit the folder and the address of the website via blackboard.

Rubric

- +40 pts: Implementation of Harris corner detector in get_interest_points.py
- +30 pts: Implementation of SIFT-like local feature in get_features.py
- +10 pts: Implementation of "Ratio Test" matching in match_features.py
- +20 pts: Writeup with several examples of local feature matching.
- +10 pts: Extra credit (up to ten points)
- -5*n pts: Lose 5 points for every time (after the first) you do not follow the instructions for the hand in format

Credits

Assignment originally developed by James Hays (Brown University) with MATLAB. Bei Xiao re-wrote the stencil codes in Python.