CS 415: Computer Vision I

Homework 2

Prof.: Sathya N. Ravi Assigned: 10/07/2021, Due: 11/04/2021

Few reminders:

- Homework is due at the end of day on the designated date on Blackboard.
- No homework or project is accepted in mailbox of instructor.
- You may discuss homework with classmates and work in groups of up to **three**. However, you may not share any code, carry out the assignment together, or copy solutions from any other groups. Discussions between groups should be minimal, verbal during lecture or if appropriate, on Campuswire only. The submitted version must be worked out, written, and submitted by your group alone.
- **Important:** Each question (or subpart) should have a **lead** who has finalized the submitted solutions after discussions within group. The lead should be *explicitly* indicated in the submissions. There can be two leads for any question. The submitted solution is assumed to be verified by the other person, and so the grade is assigned equally.
- All solutions must be typeset (I recommend Latex, Markdown, Word etc., please do not use nonstandard formats) with code attached in MATLAB or Python format whichever is appropriate.
- Your final submission will be a zip folder with a PDF file containing solutions for each question including text, sample figures, calculation, analysis, and general writeup. Code and more experiments including figures should be provided in a separate folders.
- Submitting someone else's work (outside of the group) as your own is academic misconduct. Such cheating and plagiarism will be dealt with in accordance with University procedures (see the page on Academic Misconduct at this link).

Additional Instructions for Python Questions.

- Install Anaconda. We recommend using Conda to manage your packages.
- The following packages are needed: OpenCV, NumPy, Pillow, Matplotlib, Jupyter, PyTorch. And you will need to figure out how to install them.
- You will not need Cloud Computing or GPU for this assignment.
- Run the notebook using: `jupyter notebook ./code/question4.ipynb`
- You will need to fill in the missing code in: `./code/student_code.py`

- For each Python question, generate the submission once you have finished the project using: `python zip submission.py`

Additional Instructions for MATLAB Questions. If appropriate, you may use the online browser version of MATLAB that you can freely access via the UIC Webstore (login required).

Answer the following questions as completely as possible. Recall that d dimensional vector spaces (for any finite d) over the reals $\mathbb{R}$ can be represented using $\mathbb{R}^d$.

1. **Tracking Points in Plane.** Suppose the origin of the world coordinate system is at the pinhole. Let the z-axis be along the optical axis of the camera (perpendicular to the image plane), and the x and y axes are parallel to the image plane. Let the distance of the image plane from the pinhole be f. Now, consider a point in the scene moving with velocity (u, v, w) from the starting point (x_0, y_0, z_0). Write the equation for the image coordinates (p, q) at time t. Show that the image point moves along a straight line as t increases.

2. **Generalized Corner Detector.** 3D images are often used in medical image analysis in which 2D image slices are stacked to get a three dimensional image of say brain. Describe a generalization of Harris Corner Detector to compute corners of 3D images, including a test to decide when a voxel is a corner.

3. **Harris Corner Detector in Python.** Recall that HCD finds corners by computing the pixels which have high change in intensity over all possible directions. We then choose a window or neighborhood W centered at p and compute the energy E_W of window W at p to be the sum of individual energies of pixels in the window. A natural variant is to take a weighted sum in the window at p which is equivalent to using a kernel/filter w. Assume that the kernel w has size $[W_v, W_h]$, then $E_W = \sum_{p \in W} w(p) E_p$. Now, the change in intensity (which may be also interpreted as energy) along a direction $d = [u, v]^T$ at a pixel $p = [x, y]$ is given by,

$$E_p(d) = E_W = \sum_{p \in W} w(p)(I(p + d) - I(p))^2. \tag{1}$$

   (a) Implement a function `exact_Ep_d.py` to compute $E_p(d)$ in (1) at a pixel p along a direction d given as an angle $\theta \in [0, \pi]$.

   (b) We then find for minimum over all feasible θ for a pixel p. Obviously, there are infinite number of them, so we will compute E_p for $\theta \in \Theta = \{0, \pi/4, \pi/2, 3\pi/4, \pi\}$, and compute the energy at pixel E_p as minimum over all of them:

$$E_p = \min_{\theta \in \Theta} E_p(\theta). \tag{2}$$

   We output the pixels p that have energy in E_W above a given threshold $\tau > 0$ as Morevac corners $\mathcal{C}_M$, that is, we output,

$$\mathcal{C}_M = \{p : E_W > \tau\}. \tag{3}$$

   Implement a function `exact_Ep.py` to compute E_p, and using this output the Moravec corners in `moravec_corners.py`. Plot your results as τ changes from small to large values. What is the optimal τ?

We have implemented the Moravec detector. HCD builds on this by approximating the energy function $E_p(d)$ in (1).

(c) Using Taylor's series we will approximate the energy in the window $E_{W(p)}$ (or just E_W) along $d = [u, v]^T$. Harris energy $\hat{E}_W$ given by,

$$\hat{E}_W := \sum_{p \in W} w(p) \left(d^T \nabla I_p \right)^2, \tag{4}$$

where $\nabla I_p = [I_x, I_y]^T$ is the image gradient at pixel p. (Note that $\hat{E}_W$ implicitly depends on p but we have suppressed it here.) Simplifying terms in (4), there is a matrix M such that $\hat{E}_W = d^T M d$. Implement a function `harris_energy.py` that computes the matrix M_p.

(d) Implement a function `harris_corners.py` that outputs Harris corners as,

$$\mathcal{C}_{HC} = \{p : R(p, \lambda) > \tau\}, \tag{5}$$

where $\tau > 0$ and $\lambda > 0$ are tuning parameters, and the Harris Score $R(p, \lambda) := \det(M_p) - \lambda \text{trace}(M_p)$. Plot your results as τ and λ are varied. What are the optimal values of τ and λ? (I suggest to analyze the case where we fix $\lambda = 1$ and vary τ.) Your file should also return $R(p, \lambda)$ as a heatmap (so it must be of the same size as the image I).

(d) **[Extra Credit.]** How can you speed up the computation if w is a separable filter? Implement `harris_energy_separable.py` and compare with `harris_energy.py`. Your file should also return $R(p, \lambda)$ as a heatmap (so it must be of the same size as the image I).

What to turn in?

(a) You will turn in the following **five** files/functions with the input output described on their rows.

File	**Input**	**Output**
`exact_Ep_d.py`	I, p, θ	$E_p(d) \in \mathbb{R}_+$
`exact_Ep.py`	I, p, Θ	E_p
`moravec_corners.py`	I, τ, w	$\mathcal{C}_M$
`harris_energy.py`	I, w	M_p
`harris_corners.py`	I, w, λ, τ	$\mathcal{C}_{HC}, R(p, \lambda)$

(b) A pdf file describing your tuning results and plots.

4. **Feature Descriptors in Matlab.** This is a conceptually a continuation of problem 2 but we will do this in Matlab to remove dependency problems within the homework. Moreover, setting up high level code is often nontrivial in Python version of Opencv whereas Matlab

has in-built functions that are globally valid, so you can easily check your implementation with what Matlab outputs easily. We will implement three different feature descriptors around each keypoint detected. You can use any of Matlab in-built interest point detectors, such as, `detectHarrisFeatures, detectSURFFeatures` etc.. Each of the three descriptors takes a grayscale image and a list of keypoints as input, and outputs a descriptor (vector) for each keypoint.

(a) **Basic Neighborhood.** The first descriptor that you will implement is a simple neighborhood descriptor which extracts the pixel intensity values in the 5×5 neighborhood around each keypoint. This should be easy to implement and should work well when the images you are comparing are related by a translation.

As discussed in class, these naive patch based descriptors do not carry any semantic information about the keypoint. To do that, we will use other semantic information such as orientation, and scale. In short, patches with these points as centers do not serve as good features. The local appearance of features will change in orientation and scale, and sometimes even undergo affine transformations. Extracting a local scale, orientation, or affine frame estimate and then using this to resample the patch before forming the feature descriptor is thus usually preferable. Even after compensating for these changes, the local appearance of image patches will usually still vary from image to image. How can we make image descriptors more invariant to such changes, while still preserving discriminability between different (non-corresponding) patches?

(b) **Histogram Descriptor.** You will implement the SIFT feature descriptor that we discussed in class. The SIFT algorithm is as follows:

i. Compute the gradient at each pixel in a 16×16 window around each detected keypoint.

ii. In each 4×4 quadrant of the window, a gradient orientation histogram is generated.

iii. The resulting *normalized* 128 non-negative values form a raw version of the SIFT descriptor vector. Optionally, the descriptor can be clipped to remove noise, and then renormalized.

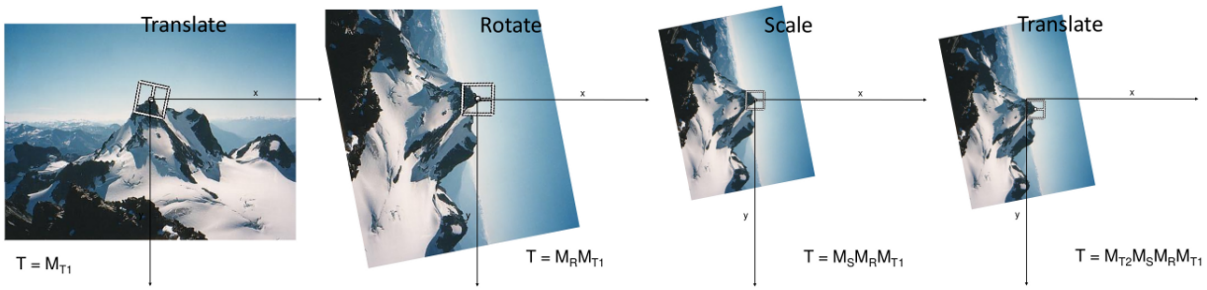
(c) **MOPS for Oriented Descriptors.** You will implement a simplified version of the MOPS descriptor. Given keypoints in the form of (x, y, θ), the MOPS descriptor can be computed as follows:

i. Form an image patch 8×8 sub-sampled from a 40×40 pixel region around the feature by picking every fifth pixel.

ii. Form 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. In 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. Optionally, you can figure the extrema in scale and use it.

iii. You should also normalize the descriptor to have zero mean and unit variance.

Hint. You will use `imwarp` with `affine2d` to perform the transformation. The easiest way to generate the 2×3 matrix is by *composing* multiple primitive transformations. A sequence of translation T_1, rotation R, scaling S and translation T_2 will work. Left-multiplied transformations are combined right-to-left so the transformation matrix is the matrix product $T_2 \cdot S \cdot R \cdot T_1$. The figures below illustrate the compositions:



4. **Robust Image stitching in Python.**

   Image stitching or photo stitching combines multiple photographic images that have overlapping fields of view to produce a segmented panorama or high-resolution image.

   (a) Load both images: `left1.jpg` and `right1.jpg`.

   (b) Use HCD (from Problem I) or any corner detector – you can use packages in Python to detect corners.

   (c) Extract local neighborhoods around every keypoint in both images, and form descriptors simply by "flattening" the pixel values in each neighborhood to a vector. You may instead using descriptors in OpenCV, describe how you get those descriptors and **explain** the meaning of the descriptors.

   (d) Compute distances between every descriptor in one image and every descriptor in the other image. Experiment with computing normalized correlation, or Euclidean distance after normalizing all descriptors to have zero mean and unit standard deviation. Report your choices. **Note:** You are not allowed to use built-in functions to match features for you, including but not limit to `cv2.BFMatcher`. However, you can use them to debug your code and compare to your implementation.

   (e) Using the distance matrix, match the pairs of points in the two images that are at a distance below a specific threshold τ or select a top $k-$hundred pairs with the least distance. Report your choices.

   (f) Run RANSAC to estimate homography mapping one image onto the other. For the best fit, **report** the number of inliers and the average residual for the inliers, and **display** the locations of inlier matches in both the images. You can use `cv2.drawKeypoints` to draw matches.

   Note: You need to implement RANSAC and calculate the transform matrix. You are not allowed to use functions that do RANSAC in one line, including but not limit to

`cv2.findHomography` or `cv2.getPerspectiveTransform`. However, you can use them to debug your code and compare to your implementation.

(g) Warp one image onto the other using the estimated transformation. To do this, you will need to learn about `cv2.warpPerspective`. Please read the documentation.

(h) Create a new image big enough to hold the panorama and composite the two images into it. You can composite by simply averaging the pixel values where the two images overlap. **Here is a sample output:**