Instructions:

- Homework 3 has to be submitted in groups of 3.
- Review the academic integrity and collaboration policies on the course website.
- Submit your PDF report on gradescope. Make sure you include the source code under Appendix listing at the end of your report.
- Submit your zipped code folder electronically by email to atripath@eng.ucsd.edu, mpatanka@eng.ucsd.edu & lmelvix@eng.ucsd.edu, with the subject line CSE 252A Homework 3. Please include the PIDs of your group members in the email. The email should have one file attached. Name this file: CSE_252A_hw3_lastname1_lastname2_lastname3.zip. The contents of the file should be:
  1. All your source code should be in a folder named code.
  2. The structure should be CSE_252A_hw3_lastname1_lastname2_lastname3.zip containing the code folder which contains all source code.

The code is thus attached both as text in the writeup appendix and as source-files in the compressed archive.
- No physical hand-in for the assignment.
- Coding is to be done only in MATLAB.
- In General MATLAB code does not have to be efficient. Focus on clarity, correctness and function here, and we can worry about speed in another course.

1 Image warping and merging [10 pts]

All data necessary for this assignment is available on the course web page (plotsquare.m, stadium.jpg).

Introduction

In this problem, we consider a vision application in which components of the scene are replaced by components from another image scene.

Optical character recognition, OCR, is one computer visions more successful applications. However OCR can struggle with text that is distorted by imaging conditions. In order to help improve OCR, some people will ‘rectify’ the image. An example is shown in Fig. 1. Reading signs from Google street view images can also benefit from techniques such as this.

This kind of rectification can be accomplished by finding a mapping from points on one plane to points another plane. In Fig 1, $P_1$, $P_2$, $P_3$, $P_4$ are mapped to $(0,0), (1,0), (1,1), (0,1)$. To solve this section of the homework, you will begin by deriving the transformation that maps one image onto another in the planar scene case. Then you will write a program that implements this transformation and uses it to rectify ads from a stadium. As a reference, see pages 316-318 in Introductory Techniques for 3-D Computer Vision by Trucco and Vern[1].

To begin, we consider the projection of planes in images. Imagine two cameras $C_1$ and $C_2$ looking at a plane $\pi$ in the world. Consider a point $P$ on the plane $\pi$ and its projections $p = (u_1, v_1, 1)^\top$ in image 1 and $q = (u_2, v_2, 1)^\top$ in image 2.

Fact 1 There exists a unique (up to scale) $3 \times 3$ matrix $H$ such that, for any point $P$:

$$q \equiv Hp$$

(Here $\equiv$ denotes equality in homogeneous coordinates, where the homogeneous coordinates $q$ and $p$ are equal up to scale) Note that $H$ only depends on the plane and the projection matrices of the two cameras.

The interesting thing about this result is that by using $H$ we can compute the image of $P$ that would be seen in camera $C_2$ from the image of the point in camera $C_1$ without knowing its three-dimensional location. Such an $H$ is a projective transformation of the plane, also referred to as a homography.

Problem definition

Write files `computeH.m` and `warp.m` that can be used in the following skeleton code. `warp` takes as inputs the original image, corners of an ad in the image, and the homography $H$. Note that the homography should map points from the destination image to the original image, that way you will avoid problems with aliasing and sub-sampling effects when you warp. You may find the following MATLAB files useful: meshgrid, inpolygon, fix, interp2.

```matlab
I1 = imread('stadium.jpg');
% get points from the image
figure(10)
imshow(I1)
% select points on the image, preferably the corners of an ad.
points = ginput(4);
figure(1)
subplot(1,2,1);
imshow(I1);

new_points = [...] % choose your own set of points to warp your ad too
H = computeH(points, new_points);

% warp will return just the ad rectified
warped_img = warp(I1, new_points, H);
subplot(1,2,2);
imshow(warped_img);
```
Report

For three of the ads in stadium.jpg, run the skeleton code and include the output images in your report.

2 Optical Flow [21 pts]

In this problem you will implement the Lucas-Kanade algorithm for computing a dense optical flow field at every pixel. You will then implement a corner detector and combine the two algorithms to compute a flow field only at reliable corner points. Your input will be pairs or sequences of images and your algorithm will output an optical flow field \((u,v)\). Three sets of test images are available from the course website. The first contains a synthetic (random) texture, the second a rotating sphere\(^2\) and the third a corridor at Oxford university\(^3\). Before running your code on the images, you should first convert your images to grayscale and map intensity values to the range \([0,1]\). I use the synthetic dataset in the instructions below. Please include results on all three datasets in your report. For reference, your optical flow algorithm should run in seconds if you vectorize properly (for example, the eigenvalues of a 2x2 matrix can be computed directly). Again, no points will be taken off for slow code, but it will make the experiments more pleasant to run.

![Figure 2: Input images](image)

2.1 Dense Optical Flow [8pts]

Implement the single-scale Lucas-Kanade optical flow algorithm. This involves finding the motion \((u,v)\) that minimizes the sum-squared error of the brightness constancy equations for each pixel in a window. As a reference, read pages 191-198 in Introductory Techniques for 3-D Computer Vision by Trucco and Verri\(^4\). Your algorithm will be implemented as a function with the following inputs,

\[
\text{function } [u, v, \text{hitMap}] = \text{opticalFlow}(I1, I2, \text{WindowSize}, \tau)
\]

Here, \(u\) and \(v\) are the \(x\) and \(y\) components of the optical flow, \(\text{hitMap}\) a binary image indicating where the corners are valid (see below), \(I1\) and \(I2\) are two images taken at times \(t = 1\) and \(t = 2\) respectively, \(\text{WindowSize}\) is the width of the window used during flow computation, and \(\tau\) is the threshold such that if the smallest eigenvalue of \(A^T A\) is smaller than \(\tau\), then the optical flow at that position should not be computed. Recall that the optical flow is only valid in regions where

\[
A^T A = \begin{pmatrix}
\sum I_x^2 & \sum I_x I_y \\
\sum I_x I_y & \sum I_y^2
\end{pmatrix}
\]

\(^3\)Courtesy of the Oxford visual geometry group
\(^4\)Available on the course webpage.
has rank 2 (why?), which is what the threshold is checking. A typical value for $\tau$ is 0.01. Using this value of $\tau$, run your algorithm on all three image sets (the first two images of each set), for three different window sizes of your choice, to produce an image similar to Fig. 3. Also provide some comments on performance, impact of windowsize etc.

![Figure 3: Result for the dense optical flow problem on the corridor image.](image)

2.2 Corner Detection [5pts]

Use your corner detector from Assignment 2 to detect 50 corners in the provided images. Use a smoothing kernel with standard deviation 1, and windowsize of 7 by 7 pixels for your corner detection throughout this assignment. Include a image similar to Fig. 4a in your report. If you were unable to create a corner detection algorithm in the previous assignment, please email the TA for code.

2.3 Sparse Optical Flow [8pts]

Combine Parts A and B to output an optical flow field at the 50 detected corner points. Include result plots as in Fig. 4b. Select appropriate values for windowsize and $\tau$ that gives you the best results. Provide a discussion about the focus of expansion (FOE) and mark manually in your images where it is located. Is it possible to mark the FOE in all image pairs? Why / why not?
3 Iterative Coarse to Fine Optical Flow [15 pts]

Implement the iterative coarse to fine optical flow algorithm described in the class lecture notes (pages 8 and 9 in lecture 13). Show how the coarse to fine algorithm works better on the first two frames inside of flower.zip than dense optical flow. You can do this by creating a quiver plot using your code from problem 2 and a quiver plot for the coarse to fine algorithm. Try 3 different window sizes: one of your choice, 5, and 15 pixels. Where does the dense optical flow algorithm struggle that this algorithm does better with? Can you explain this in terms of depth or movement distance of pixels? Comment on how window size affects the coarse to fine algorithm? Do you think that the coarse to fine algorithm is strictly better than the standard optical flow algorithm? Example output shown in Fig. 5a. Note: Like in problem 2, convert the image to intensity gray scale images.

Figure 4: Corner detection and sparse optical flow

(a) Result of the corner detection problem on the corridor image. (b) Result of sparse optical flow algorithm on the corridor image.

Figure 5: Example dense optical flow vs coarse to fine optical flow

(a) Result of dense optical flow on the flower sequence (b) Result of coarse to fine optical flow algorithm on flower sequence
4 Background Subtraction and Motion Segmentation

4.1 Background subtraction [3pt]

In this problem you will remove the dynamic portions of an image from a static background. In this case, the camera is not moving and objects within the scene or moving. For each consecutive pair of frames, background subtraction will calculate:

\[ |I(x, y, t) - I(x, y, t-1)| > \tau \]

where \( I(x, y, t) \) is the pixel intensity of the \( t^{th} \) frame of the image I, at position \( (x, y) \). Also \( \tau \) is the threshold parameter. \( |I(x, y, t) - I(x, y, t-1)| > \tau \) is the foreground mask for the frame at time \( t \).

By masking out the pixels that are greater than \( \tau \), you can create an estimate background for the frame. By calculating the mean of the estimated backgrounds you can create a global background for the sequence, as shown in the figure. Note: Convert the image to a gray scale image. Your function prototype should look like:

```matlab
function [background] = backgroundSubtract(framesequence, tau)
```

Figure 6: backgroundSubtract

The variable framesequence can be a string representing a directory of frame images or cell array of frames or any other reasonable input. Run your code on the highway and truck sequence and include the backgrounds for each sequence as a figure.

![Figure 7: (a) is a image from a frame t, (b) is the background image](image)

4.2 Motion Segmentation [5pt]

The previous algorithm only works on static cameras and a stable background. However, it is also possible to do a similar segmentation using motion cues, however it is much harder. An example motion segmentation is shown in fig 8.

Using the outputs of your iterative coarse to ne optical ow algorithm, can you segment out the tree from the rest of the first frame of the flower sequence? **Hint: the magnitudes of the motion vectors at different depths can be quite different.**

Provide a figure of the segmented tree (an image with just the tree in it) and explain how you were able to do this. Note: Like in each problem convert the image to gray scale image.
5 Hough Lines 10 Pts

In this problem, you will implement a Hough Transform method for finding lines. For the algorithm, refer lecture 12. You may use the inbuilt matlab functions for detecting the edges. However, keep in mind that you may need to experiment with other parameters till you get the expected edges from the given image (Refer matlab documentation of edge function for more details). You should not use any of the inbuilt Hough methods.

- Produce a simple $11 \times 11$ test image made up of zeros with 5 ones in it, arranged like the 5 points as shown in figure 9. Compute and display its Hough Transform; the result should look like figure 9. Threshold the HT by looking for any $(\rho, \theta)$ cells that contains more than 2 votes then plot the corresponding lines in $(x,y)$-space on top of the original image.

- Load in the image ‘lane.png’. Compute and display its edges using the Sobel operator with your threshold settings similar to that in HW2. Now compute and display the HT of the binary edge image $E$. As before, threshold the Hough Transform and plot the corresponding lines atop the original image; this time, use a threshold of 75% maximum accumulator count over the entire HT, i.e. \(0.75 \times \max(HT(;))\).

- Now that you have mastered Hough transform, Repeat the procedure for MS Commons room images (common1.jpg & common2.jpg). Set the appropriate threshold parameters to plot corresponding lines atop the original image. You will be graded on the accuracy of the result.
Figure 9: (a) Original Image and (b) is the image with hough lines drawn.