CSE 252A – Computer Vision – Homework 3
Instructor: Ben Ochoa
Revision 0

Instructions:

- This homework should be done in pairs.
- Submit your homework electronically by email to nkinkade@eng.ucsd.edu AND akdave@eng.ucsd.edu with the subject line *CSE 252A Homework 3*. The email should have one file attached. Name this file: `CSE_252A_hw3_lastname1_lastname2.zip`. The contents of the file should be:

  1. A pdf file with your writeup. This should have all code attached in the appendix. Name this file: `CSE_252A_hw3_lastname1_lastname2.pdf`.
  2. All of your source code in a folder called 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 this 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 Verri[1]

To begin, we consider the projection of planes in images. Imagine two cameras C_1 and C_2 looking at a plane π in the world. Consider a point P on the plane π 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) 3x3 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

[1] Available on the course webpage.

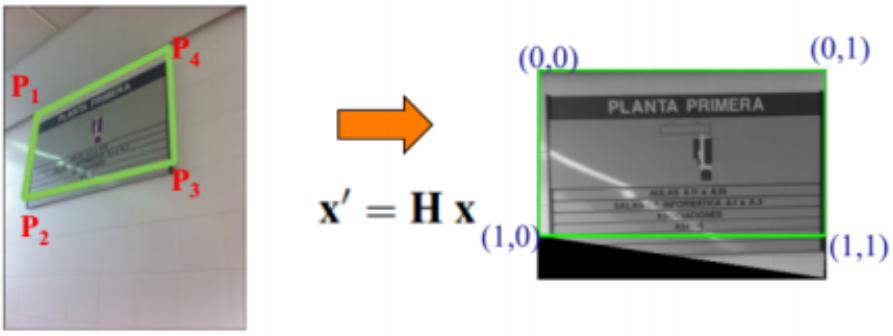


Figure 1: Input image (left) and target (right) for image mapping problem

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.

```
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 [13 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.

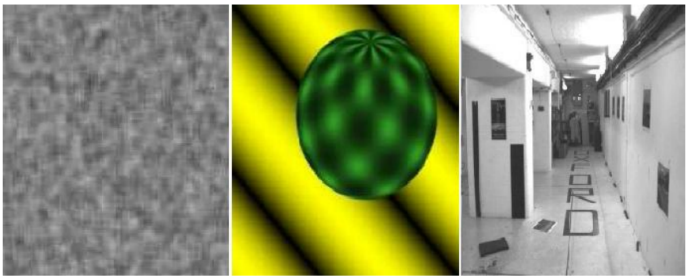


Figure 2: Input images

2.1 Dense Optical Flow [5pts]

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,

```
function [u, v, hitMap] = opticalFlow(I1, I2, windowSize, tau)
```

Here, u and v are the x and y components of the optical flow, 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, windowSize is the width of the window used during flow computation, and τ is the threshold such that if the smallest eigenvalue of $A^T A$ is smaller than τ, then the optical flow at that position should not be computed. Recall that the optical flow is only valid in regions where

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

has rank 2 (why?), which is what the threshold is checking. A typical value for τ is 0.01. Using this value of τ, run your algorithm on all three image sets (the first two images of each set), for three different windowsizes of your choice, to produce an image similar to Fig. 3. Also provide some comments on performance, impact of windowsize etc.

[2]Courtesy of http://www.cs.otago.ac.nz/research/vision/Research/OpticalFlow/opticalflow.html
[3]Courtesy of the Oxford visual geometry group
[4]Available on the course webpage.

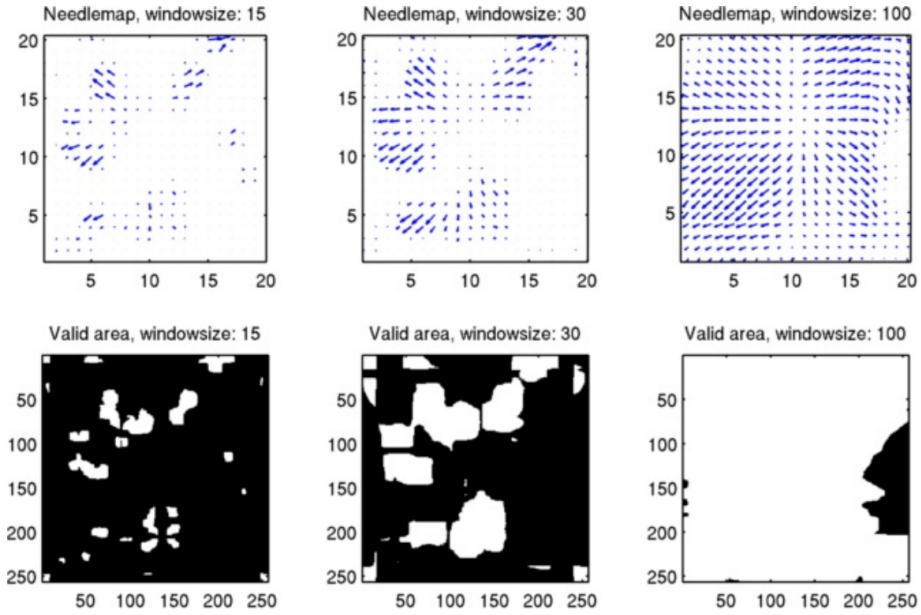


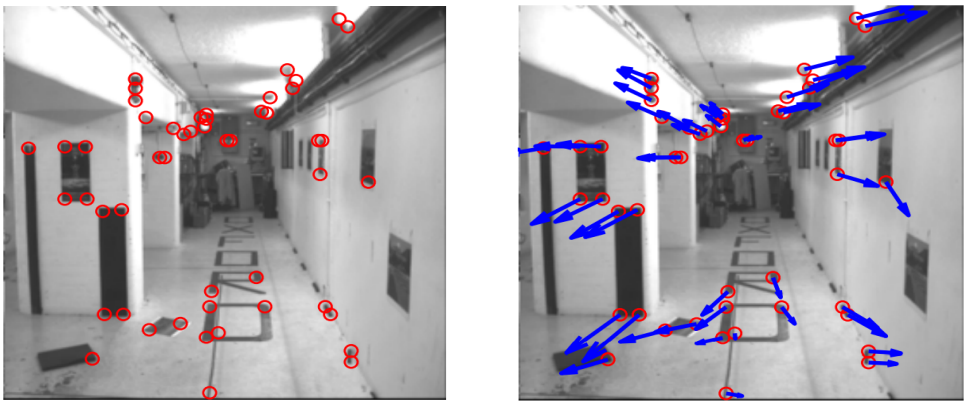
Figure 3: Result for the dense optical flow problem on the corridor image.

2.2 Corner Detection [2pts]

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 algorihtm in the previous assignment, please email the TA for code.

2.3 Sparse Optical Flow [3pts]

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 τ 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?



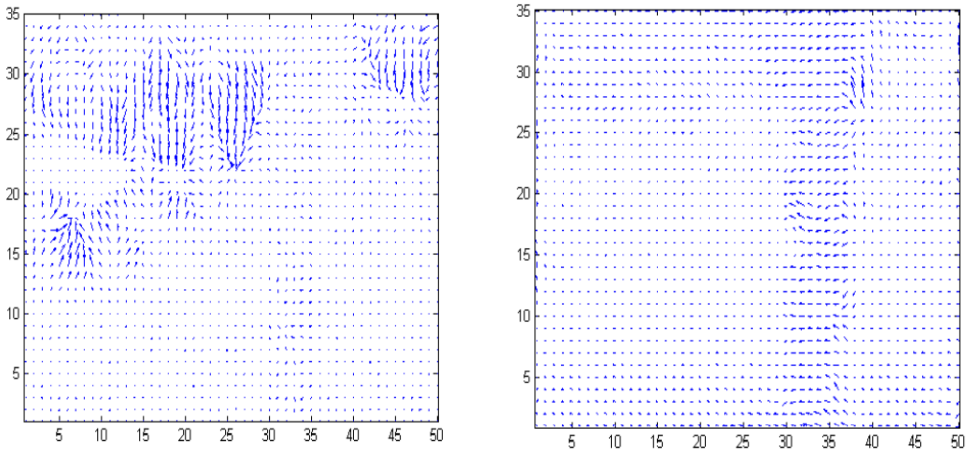
(a) Result of the corner detection problem on the corridor image.

(b) Result of sparse optical flow algorithm on the corridor image.

Figure 4: Corner detection and sparse optical flow

3 Iterative Coarse to Fine Optical Flow [10 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 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.



(a) Result of dense optical flow on the flower sequence

(b) Result of coarse to fine optical flow algorithm on flower sequence

Figure 5: Example dense optical flow vs coarse to fine