CompRobo Fall 2020 Computer Vision Project Report

VISUAL ODOMETRY USING A MONOCULAR CAMERA SYSTEM

LUKE NONAS-HUNTER
SAMUEL KAPLAN

Project Introduction

We chose to do a deep dive into the math behind visual odometry to better understand how to extract 3D data from 2D images. Typically, visual odometry is used alongside other systems such as wheel odometry and accelerometers to increase the accuracy of the calculated position. Originally we hoped to be able to calculate the translation and rotation matrices that would translate one image to another. However, we quickly realized that this would not be possible under the time constraint and instead we attempted to implement the eight-point algorithm. The algorithm is used to find the fundamental matrix given two images. In this paper we will discuss how we went about implementing the algorithm and the challenges we faced along the way.

What is Visual Odometry?

Much like our Neato robots track their location using wheel encoders, it is also possible to track an entity's (I will refer to it as a robot, although it could be any object with a camera attached to it) location in space through a camera. Nominally, there are two frames (Frame1 and Frame2) taken sequentially. If the robot is moving while these images are recorded, there will inevitably be a translation and rotation [1] occurring between them.

Exploring the Eight-Point Algorithm

The eight-point algorithm is designed to find the fundamental matrix given two images. The fundamental matrix, shown in Equation 1, is able to relate the pixel coordinates of two images using the translation matrix, rotation matrix, and camera calibration matrix.

$$F = K'^{-T}[T_x]RK^{-1} \tag{1}$$

$$F = K'^{-T}EK^{-1} \tag{2}$$

The fundamental matrix can also be written using the essential matrix as shown in Equation 2. As a result, the fundamental matrix and essential matrix share a lot of the same properties. Specifically, given either matrix and a point in one of the images it is possible to find the

[1]While entirely possible that there is neither translation nor rotation, a more plausible edge case is sans-rotation. In that case, the images should be parallel and have epipoles outside the frame, which we will get into later

epiline which describes all the possible positions the associated point in the other image could be. In addition, the fundamental matrix and essential matrix share the property shown in Equation 3. The eight-point algorithm uses this property to solve for F given two matching points, p and p'.

$$p'^{T} F p = 0 \tag{3}$$

To begin, it is helpful to rewrite the constraint as a matrix vector equation as shown in Equation 4 where $p_i = (u_i, v_i, 1)$ and $p'_i = (u'_i, v'_i, 1)$.

$$\begin{bmatrix} u_1 u'_1 & v_1 u'_1 & u'_1 & u_1 v'_1 & v_1 v'_1 & v'_1 & u_1 & v_1 & 1 \\ u_2 u'_2 & v_2 u'_2 & u'_2 & u_2 v'_2 & v_2 v'_2 & v'_2 & u_2 & v_2 & 1 \\ u_3 u'_3 & v_3 u'_3 & u'_3 & u_3 v'_3 & v_3 v'_3 & v'_3 & u_3 & v_3 & 1 \\ u_4 u'_4 & v_4 u'_4 & u'_4 & u_4 v'_4 & v_4 v'_4 & v'_4 & u_4 & v_4 & 1 \\ u_5 u'_5 & v_5 u'_5 & u'_5 & u_5 v'_5 & v_5 v'_5 & v'_5 & u_5 & v_5 & 1 \\ u_6 u'_6 & v_6 u'_6 & u'_6 & u_6 v'_6 & v_6 v'_6 & v'_6 & u_6 & v_6 & 1 \\ u_7 u'_7 & v_7 u'_7 & u'_7 & u_7 v'_7 & v_7 v'_7 & v'_7 & u_7 & v_7 & 1 \\ u_8 u'_8 & v_8 u'_8 & u'_8 & u_8 v'_8 & v_8 v'_8 & v'_8 & u_8 & v_8 & 1 \\ & & & & \ldots & & & & \\ u_i u'_i & v_i u'_1 & u'_i & u_i v'_i & v_i v'_i & v'_i & u_i & v_i & 1 \end{bmatrix} \begin{bmatrix} F_{11} \\ F_{12} \\ F_{13} \\ F_{21} \\ F_{22} \\ F_{23} \\ F_{31} \\ F_{32} \\ F_{33} \end{bmatrix} = 0 \tag{4}$$

Looking at Equation 4, it may seem like the obvious solution to the equation would be to make F a zero vector. This would solve the equation for all values of p_i and p'_i. However, the fundamental matrix should be non-zero. As a result we now need to find a vector that when multiplied with any set of matching pixels in the image results in the zero vector (or as close to zero vector as possible). To find this vector, we will use SVD to perform PCA on the points matrix to find the singular vector which describes the axis of least change. Using this method we can safely assume the resulting vector is the best non-zero approximation of the F vector we can calculate using this method. However, there are still issues with this solution. For example, the Fundamental matrix must be of rank 2 and the vector found using SVD, when rearranged, will be of rank 3. To fix this issue we can perform SVD on F and reconstruct F using only the vectors associated with the first two singular values as shown in Equation 5.

$$F = U \begin{bmatrix} \Sigma_1 & 0 & 0 \\ 0 & \Sigma_2 & 0 \\ 0 & 0 & 0 \end{bmatrix} V^T \tag{5}$$

Another issue with using SVD to solve for the F vector is that the matrix of points is not optimized for SVD. The result we are looking for consist of most singular values being around the same size except for one, which would be very close to zero. However, the points are written in pixel values and, as a result, are made up of two very large values followed by a 1. When you perform SVD on a matrix representing points like this, you end up getting one very large singular value followed by many very small and almost indistinguishable singular values. To fix this problem we first normalized all the points. This is done by translating all the points so their centroid is located at (0,0) and scaling them so their mean squared distance is roughly two pixels. This transformation can be described by the 3 by 3 matricies T and T' which create the relationship shown in Equation 6 where q_i and q'_i are the transformed points.

$$q_i = T p_i q'_i = T' p'_i \tag{6}$$

Now using q_i and q'_i to populate the points matrix will result in a solution, F_q which does a much better job approximating the behavior of a zero vector for the given points in the q space. Finally, to get the translate the newly calculated fundamental matrix back into the p space, we use Equation 7 to undo the two translations, T and T'.

$$F = T'^T F_q T \tag{7}$$

We've done it! We have officially calculated a close approximation of the fundamental matrix using the eight-point algorithm.

Solution

To begin, we shall assume that we can accurately match features between two frames of the same environment. There are a variety of feature matching algorithms available. We used ORB (Oriented FAST and Rotated BRIEF) because there is good documentation and you don't have to pay for it [2] Feature matching is an extremely interesting area of computer vision, but not what we wanted to delve into for this project. Therefore, something relatively easy to get up and running was a priority.

To start, lets match some features between our two frames: we instantiate a ORB detector object, find points that are easiest to differentiate from each other, and then match them with a Matcher object:

[2]We would have preferred SHIFT, but unfortunately you must pay to use it.

```
# initiate ORB detector
orb = cv.ORB_create()
# find the keypoints and descriptors with ORB
kp1, des1 = orb.detectAndCompute(img1,None)
kp2, des2 = orb.detectAndCompute(img2,None)


# create BFMatcher object
bf = cv.BFMatcher(cv.NORM_HAMMING, crossCheck=True)
# match descriptors
matches = bf.match(des1,des2)
# sort them in the order of their distance
matches = sorted(matches, key = lambda x:x.distance)

# gather points
pts1 = []
pts2 = []
for i, m in enumerate(matches):
    pts2.append(kp2[m.trainIdx].pt)
    pts1.append(kp1[m.queryIdx].pt)
```

This results in something like this:

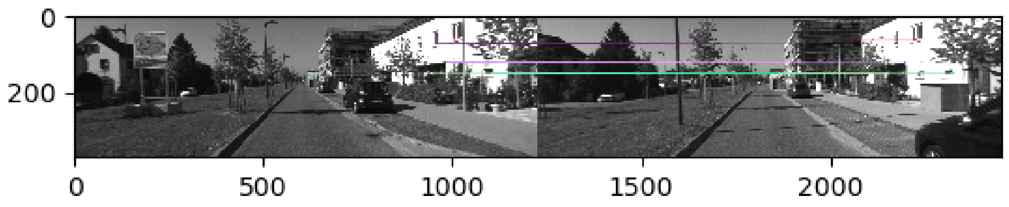

Figure 1: Feature matching

Setting the Stage

For the nominal example mentioned above, you have two Frames, separated by a translation (T) and a rotation (R).

For simplicity's sake, there is one point (P) in 3D space. Its projection in frame 1 is p and its projection in frame 2 is $p^{'}$.

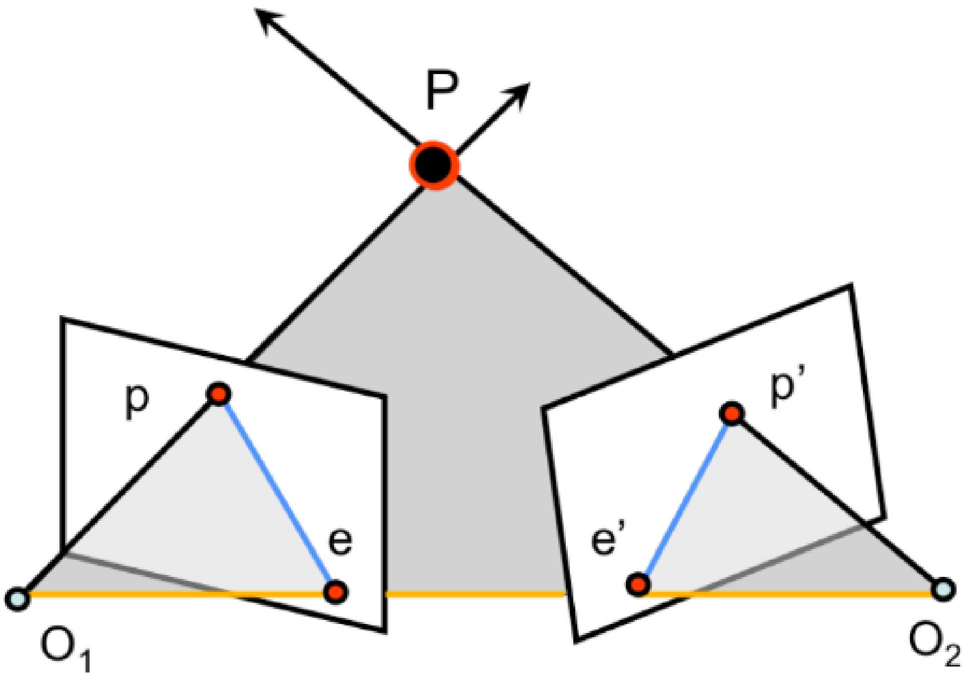

Figure 2: Img. Source: Stanford CS231A Course Notes 3: Epipolar Geometry

Clearly demonstrated here is the epipolar plane created by O_1P, O_1O_2 and O_2P, P's projection in each image, and the epipolar lines e and $e^{'}$, which are simply all possible places that p could be projected in Frame2 (given we can't get depth from a single image) and vice-versa.

Once we have some points matched, we can start to construct the Fundamental Matrix to hopefully get translation and rotation data.

To get something that hopefully works for practical applications, we have to normalize our points:

```
def normalize_points(arr):
    centroid = np.mean(arr, axis=0)
    scaling_factor = 2 / np.mean(np.square(arr), axis=0)
    transformation_matrix = np.matrix([
            [scaling_factor[0],0,-centroid[0]*scaling_factor[0]],
            [0,scaling_factor[1],-centroid[1]*scaling_factor[1]],
            [0,0,1]])
    array = np.array([np.append(row, 1) for row in arr])
    return ((arr - centroid) * scaling_factor, transformation_matrix)
```

```
pts1_norm, T = normalize_points(pts1)
pts2_norm, T_prime = normalize_points(pts2)
```

As per the 8 Point Algorithm, we must make a matrix with the titular 8 points, which is just different combinations of points from *pts1* and *pts2*.

```
w = []
for p1, p2 in zip(pts1_norm, pts2_norm):
    u, v = p1
    u_prime, v_prime = p2
     # collating necessary values to perform 8-point algorithm
    temp = [u*u_prime, v*u_prime, u_prime,
    u*v_prime, v*v_prime, v_prime, u, v, 1]
    w.append(temp)
```

Now we have:

$$W\mathbf{f} = 0$$

a where W is our collated eight points and f is the Fundamental Matrix we want to find. To accomplish this (excluding the case were $\mathbf{f} = 0$, we must perform SVD on it, get the eigenvalues corresponding to the smallest eigenvector (because we want the shortest axis of spread), and do SVD on that to reduce it to rank 2:

```
# SVD first pass
u, s, vh = np.linalg.svd(w_np)

fund_matrix = vh[:,8].reshape((3,3))

# SVD to reduce rank
u, s, vh = np.linalg.svd(fund_matrix)
identity = np.identity(3)
sigma = identity*s
sigma[2,2] = 0
F1 = np.matmul(np.matmul(u, sigma), vh) # Fundamental Matrix!
F1 = np.matmul(T_prime.T, np.matmul(F1, T)) # Undoing our transformation
# from before so we get a normal pixel coordinate system
```

This ultimately yielded an image set with the following epilines:

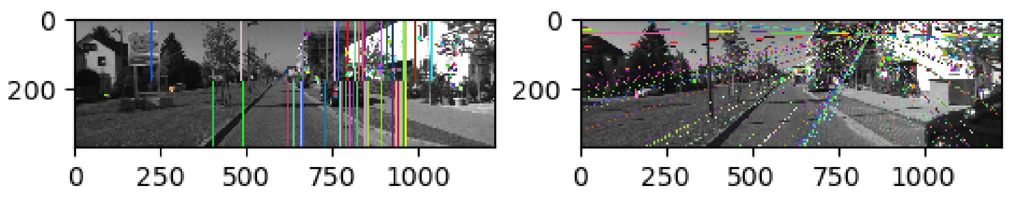

Figure 3: 8 Point Algorithm, homestyle

As you can see, there's some funky stuff going on. The right image is pretty comparable to the one OpenCV yields, but the left one definitely shoudl not have completely vertical epilines. For comparison, here is the black-boxed OpenCV Fundamental Matrix drawn epilines:

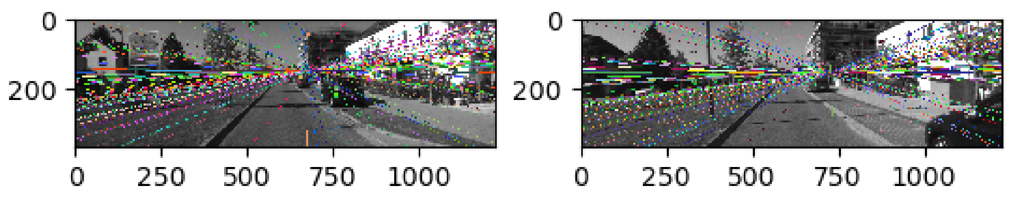

Figure 4: OpenCV epilines: much better!

So were did we end up? We have our own Fundamental Matrix, whose values is pretty close to 0 (meaning we did something right), but whose epilines aren't quite good enough to derive a translation and rotation from. Unfortunately, this is where we have to wrap it up for now.

Reflection

Here are some general thoughts about the project, written very late at night.

Improvements

Due to the fact our program is currently unable to replicate the results calculated by OpenCV we know there must be ways to improve our algorithm to get better results. Our first step towards achieving better results would be to triple check our code one more time to ensure it is doing what we expect it to be doing. Secondly, it may be smart to try a different approach.

While the eight-point algorithm was a fairly intuitive algorithm, we found there are much more complicated algorithms used to solve this problem which we decided not to investigate due to their complexity.

Design Decisions

To be honest, there weren't many decisions to make in this category as we were simply trying to implement an established algorithm with minimal OpenCV functions.

Lessons

Computer vision is hard, and there's a reason OpenCV is so popular: it allows lay people like us to blackbox functionality like "visual odometry" so we can immediately focusing on more practical applications.

However there is use in knowing how these work, and we definitely think this was a worthwhile project. Out most notable lessons include: learning to stare at papers until they make sense and transferring more academic material to code that *does* something.

Bibliography

[1] Hata, Kenji, and Silvio Savarese. “CS231A Course Notes 3: Epipolar Geometry.” Web.standford.ed, CS231A Course Notes 3: Epipolar Geometry.