<u>Computer Vision Writeup</u>

Rocco Diverdi and Alexander Crease
Computational Robotics Fall 2015

Our final code is in visual_odometry/scripts/optical_flow.py

What was the goal of your project?

The goal of our project was to implement visual odometry on the Neatos. Visual odometry uses the camera to track the motion of the Neato and infer the robot's motion and position. By looking at the differences between each image coming from the Neato's camera, we can identify corresponding keypoints between frames and use the movements of the keypoints to give us information about the movement of the robot. For example, if all of the keypoints in the image are moving to the right, then the robot is turning to the left.

Visual odometry in some cases can be more useful than the odometry determined by the tachometers on the Neato. Visual odometry can succeed where standard odometry techniques fail, because in some cases wheels or actuators can slip or get stuck, and standard odometry does not account for the wheels slipping. As one example, if the Neato runs into something and its motors keep running, visual odometry would be able to detect that the robot was not moving because the image would stay the same, but odometry using a tachometer would think that the robot would still be moving forward, because the robot's wheels would still be turning. Visual odometry also becomes very important when it comes to tracking motion on robots with other means of transportation, like quadcopters and legged robots, because other means of position tracking are impractical.

Over the course of this project, we developed our own optical flow code to track the motion of keypoints in different quadrants of the image. We then used machine learning to develop a transformation between the keypoint motion we were getting in each quadrant and the odometry readings we were getting from the tachometer on the Neato. This transformation matrix we then applied to our code to get an estimate of the robot's motion based on the camera, and then we published the Neato's motion to be displayed by rviz.

Describe how your system works. Make sure to include the basic components and algorithms that comprise your project.

On a high level, our system is fairly straightforward. We identify corresponding keypoints and track their motion, and then use their motion data to give us information about the robot's motion. Going deeper, Our script uses SIFT to identify keypoints in the current frame, and keypoints in the previous frame. Based on the location and surroundings of each keypoint, it matches each point in the current frame to a likely point in the previous frame. Points are filtered out based on proximity to other keypoints and how well they compare to each other. Using two lists of corresponding points and their positions, we calculated the displacement between each of the keypoints in the form of a vector. The image was split up into four quadrants and all of the vectors in those quadrants were averaged to get four "main" vectors. The concept behind this decision was that if the keypoints are all moving toward the center of the image, the robot is driving backward, if they are all moving away from the image, the robot is moving forward, and if there is only horizontal keypoint motion, the robot is turning. With these four vectors we can see the overall trends of the keypoints. An array of each of these vectors were then input into a linear regression machine learning algorithm from the sklearn package that optimized the mapping of the four vector values to the X, Y, and rotation values coming off the actual odometer on the Neato. Sklearn.fit takes the data from the camera and the data from the robot's odometry, and uses linear regression to compute the transformation matrix between the two, and sklearn.predict uses that matrix and the camera data to compute the predicted motion of the robot. The robot motion was fed into rviz to be displayed.

Describe a design decision you had to make when working on your project and what you ultimately did (and why)? These design decisions could be particular choices for how you implemented some part of an algorithm or perhaps a decision regarding which of two external packages to use in your project.

There were two paths that we could have taken to determine the motion of the camera image, the optical flow method or the fundamental matrix method. The fundamental matrix method seemed simpler and more accurate than the optical flow method at first. Using the keypoints determined by the SIFT method, we could calculate the fundamental matrix between the two images. The fundamental matrix could then be used to plot the epipolar lines of the images. Epipolar lines are formed from the intersection of the image and the epipolar plane, the plane connecting the camera and the two matching keypoints. The intersection of multiple epipolar lines corresponds to the center of one image in the other image's reference frame. By getting that information, we could calculate the transformation between the two images and use that vector to determine the Neato's motion. However, this method had a couple of complications. When we ran our code using this method, the image center point seemed to jump all over the place, because the fundamental matrix kept drastically changing. We weren't able to fully track down the reason for this problem because we didn't want to spend too much time chasing down bugs, but we think is has to do with the fact that the fundamental matrix breaks when all of the keypoints lie on the same plane, or when the image is zoomed in or out (corresponding to driving forward or back), but not translated or rotated. About halfway through the project we had to decide whether or not we should continue pursuing the fundamental matrix, or shift gears and go with the optical flow method. The optical flow method seemed conceptually easier, but we would have to start nearly from scratch. We had gotten a fair amount done using the fundamental matrix method, and it seemed like the most direct way to get to the solution. However, we hadn't made much progress working with the fundamental matrix and didn't have time to delve deeper into the problem without jeopardizing the success of our project. We ended up choosing the optical flow method because it involved writing and debugging our own code instead of trying to parse through what others had done to figure out where the code was going wrong.

How did you structure your code?

We set up our code in a VisualOdometry class with a bunch of functions that we could swap in and out of our main loop if we wanted to test different implementations. Our code was essentially written with step-by-step functions that would run sequentially or build off of each other. To implement the machine learning algorithm, we had a finite-state controller that switched between the "Train" state and the "Trained" state. The code would initially start out in the "Train" state, in which it would collect data from the Neato's odometry and data from the camera. Once that state is exited (by pressing the "q" key), the state switches, and the robot only uses its camera and the matrix calculated by the machine learning algorithm to determine its odometry. Our functions followed the following progression:

Starts in "Train" mode:

- Find and match keypoints
- filter out bad keypoint guesses
- track the movement of each keypoint
- average all of the movements for each quadrant of the image
- develop a matrix of all of the average vector values in each quadrant

- develop a matrix of odometry coming from the actual tachometers

Ends training loop:

- Run a machine learning function to determine the transformation matrix from our vectors to the odometry of the robot

Switches to “Trained” mode:

- Find and match keypoints
- filter out bad keypoint guesses
- track the movement of each keypoint
- average all of the movements for each quadrant of the image
- develop a matrix of all of the average vector values in each quadrant
- multiply the vector array by the previously calculated transformation matrix
- publish the transformation to rviz

What if any challenges did you face along the way?

The biggest challenge that we faced was having to go through and understand where our code was messing up. This was why it was so hard for us to make a decision between the optical flow method and the fundamental matrix method. In order to figure out where the bugs were, we started from our final output of the epilines and worked our way backward, checking that each section of our code performs the way we expect it to. We were finally able to track the problem back to the fundamental matrix, as it was changing drastically between images. A lot of our debugging processes followed a similar pattern. For example, our averaged vectors in each of the four quadrants were not really making sense, and we couldn’t tell if our computation was wrong or if not enough information could be gained from the vectors. We were able to track the bug all the way back to the keypoint matching threshold, and to the way that our vectors were being initiated.

What would you do to improve your project if you had more time?

The machine learning function was sort of a last minute addition to our code because we were running out of time and we needed to finish our project. If we had more time, it would be very interesting to go more into the math surrounding the motion of the camera image and how it translates to the motion of the robot. We would like to find the horizon line in the image, and determine the motion of the robot based upon how fast and how far away keypoints are moving with respect to that horizon line. All the points moving toward the horizon line imply a robot moving backward, and all the points moving away from it imply forward motion, while points moving from side to side imply rotation. This is what our code does via the machine learning algorithm, so our code has the capability to do this because of the way we set up the keypoint movement calculations, but we did not have the time to go through the math and actually determine the transform we are applying.

Did you learn any interesting lessons for future robotic programming projects? These could relate to working on robotics projects in teams, working on more open-ended (and longer term) problems, or any other relevant topic.

We learned not to trust anything we find on the internet, including code. This lesson primarily came out of the fact that all the things we tried to implement (mostly just the fundamental matrix calculations) kept running into issues that we weren’t sure how to solve because we were fairly confident that our code worked. What we really did learn from this was to do more research and understanding before throwing in functions that we find on the internet into our code. For parts of this project, we just assumed that functions we were given worked, and didn’t really look into where they could go wrong. Also, machine learning sounds really cool and potentially really useful in a variety of situations!