C106B Discussion 9: SLAM

1 Introduction to SLAM

SLAM, which stands for Simultaneous Localization and Mapping, allows a robot to determine its location and detect features in its environment using information from sensors, usually onboard. The algorithm performs repeated updates to correct for noise, improving estimates over time. Our state vector for the problem includes physical information about the robot itself (localization) along with feature positions (mapping).

While vision-based SLAM on slower systems and indoor environments is mostly a solved problem, more complex tasks, such as dealing with evolving conditions or having resource awareness, remain open challenges at the forefront of robotics research.

Problem: What separates SLAM from normal odometry?

SLAM has two major advantages over normal, local odometry:

1. There is a global optimization of poses, rather than simply optimizing over local movements. Despite marginalizing out previous states, the data given by them is still present in the probability distribution we end up solving for.
2. Loop closure is constantly performed. If we think taht the robot has revisited a previous state, we ensure certain constraints are satisfied (i.e. we are consistent with that previous state).

Current approaches toward SLAM can be broken down into 2 components: the **front end** and **back end**. The front end deals with processing the images or other sensor data, extracting the necessary points of interest. The back end uses this data, which is assumed to have some noise, to update estimates on robot and feature locations.

2 Front End

The front end processes data received from sensors, including the robot's camera, to feed to the back end. As a robot moves through a space, it takes multiple photographs. We have information about how the robot has moved thanks to sensors on its wheels, for example. We want to combine this with data on how features in the images taken have shifted in order to create a map of our environment.

Feature matching/correlation between 2 images comes in 3 steps: 1) feature extraction, 2) data association, and 3) outlier rejection. Many conventional computer vision algorithms perform these steps for you and have already been implemented.

2.1 Feature Extraction

Feature extraction identifies points of interest in an image. These points will be used to form connections between 2 images in later parts of the pipeline. On an individual picture, corner detectors usually work the best. They examine the pixels around a particular point and, based on their characteristics, might identify that point as a corner. The Harris Corner Detector is a popular one. These algorithms tend to find too many points; methods like Adaptive Non-Maximal Suppression help reduce the number of pixels we must examine.

2.2 Data Association and Outlier Rejection

Now that we have the points of interest in 2 images of our moving robot, we have to form connections between them. This is done by creating feature vectors describing each point and finding the vector in the other image that best matches the first. The ORB algorithm performs this function in both a scale- and rotation-invariant manner (this is important to make sure it won't matter if our robot turns or moves closer to a particular feature). The matches that ORB suggests aren't always great! That's why we reject outliers.

The two most well-known methods for this are RANSAC and the Mahalanobis distance test. RANSAC repeatedly estimates the fundamental matrix between two randomly chosen subsets of points that have been associated with one another and picks the one that best follows the epipolar constraint. (In other words, it estimates transformation matrices from one set of points to another and eliminates the ones that don't result in points being in the right place following the transformation.) The Mahalanobis distance test uses Gaussian assumptions to check if the new image features match the expected locations.

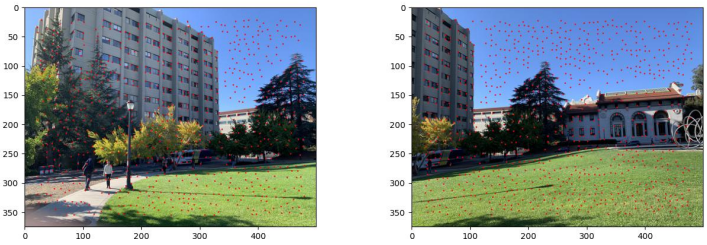

Figure 1: Harris Corners

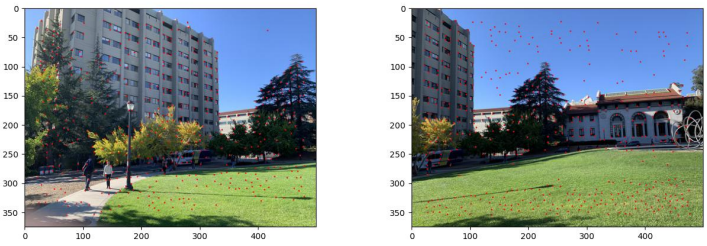

Figure 2: Adaptive Non-Maximal Suppression

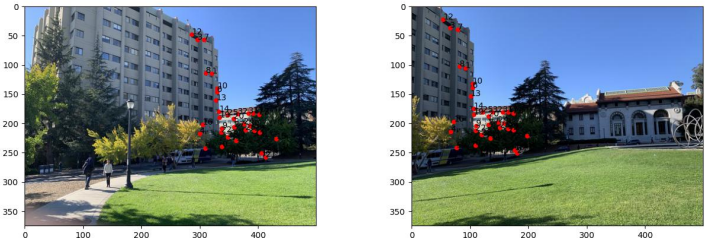

Figure 3: Matched Features with Outlier Rejection

Problem: RANSAC stands for Random Sampling and Consensus. Where does the sampling come from, and why is this better than simply performing a least-squares estimate?

RANSAC samples a few different points at a time to compute a transformation and count inliners. This approach is far better than simple least-squares because least-squares doesn't actually do any outlier rejection; it instead fits to the data as a whole. By sampling, we bias towards the highest number of correct matches rather than minimizing total error.

3 Back End

Now that we have cleaned up our image data and matched the features in our old picture to our new one, it's time to actually perform the updates to the probability distributions of our current state! This is done using an **Extended Kalman Filter** (or a Kalman Filter for linear systems).

3.1 Kalman Filtering

Remember how the observations of a Hidden Markov Model are used to update the estimates of the next state? A Kalman Filter works much the same way! We use the observations from our front end (where are the features we had identified in a previous step now located?) to figure out where our robot is now. We can predict where those features *should* be based on our movement and compare them to what we actually see.

A Kalman Filter relies heavily on the idea of Gaussian noise. The error in our system measurements is predicted to be normally distributed with 0 mean and some estimated covariance matrix. Because of the 0-mean property, averaging out our predictions and observations over time should make estimates more reliable.

Problem: Write out the update equations for our system and observations assuming a linear model.

The state is updated according to the dynamics of the system with some added noise. The state update is $A_t x_t$, and the noise is Gaussian with $w_t \sim \mathcal{N}(0, \Sigma_w)$.

$$x_{t+1} = A_t x_t + w_t$$

The observation is updated in a similar way with the camera model $C_t x_t$ and Gaussian noise $v_t \sim \mathcal{N}(0, \Sigma_v)$.

$$y_t = C_t x_t + v_t$$

The propagation step predicts the next state given all of the observations up until the current time step. We also have access to the dynamics model.

Problem: Write out the equations for the propagation step.

For the propagation step, we are given

1. An estimate of our current state conditioned on observations so far along with its covariance: $\mu_{t|1:t}$ and $\Sigma_{t|1:t}$
2. The system $x_{t+1} = A_t x_t + w_t$

To propagate, we want to find the expected value and covariance of our next time step: $\mu_{t+1|1:t}$ and $\Sigma_{t+1|1:t}$.

$$\begin{aligned}
\mu_{t+1|1:t} &= \mathbb{E}[x_{t+1}|y_{1:t}] \\
&= \mathbb{E}[A_t x_t + w_t|y_{1:t}] \\
&= A_t \mathbb{E}[x_t|y_{1:t}] + 0 \\
&= A_t \mu_{t|1:t}
\end{aligned}$$

$$\begin{aligned}\Sigma_{t+1|1:t} &= \mathbb{E}[(x_{t+1} - \mu_{t+1})(x_{t+1} - \mu_{t+1})^T | y_{1:t}] \\ &= \mathbb{E}[(A_t(x_t - \mu_t) + w_t)(A_t(x_t - \mu_t) + w_t)^T | y_{1:t}] \\ &= A_t \mathbb{E}[(x_t - \mu_t)(x_t - \mu_t)^T] A_t^T + 0 + \mathbb{E}[w_t w_t^T] \\ &= A_t \Sigma_{t|1:t} A_t^T + \Sigma_w\end{aligned}$$

The update step adds the new observation to the set we have conditioned on. This allows the algorithm to push time forward.

Problem: What is computed in the update step?

We are given

1. Estimate of $x_{t+1}|y_{1:t}$ from the propagation step
2. Our new observation y_{t+1}
3. We know our observation equation $y_{t+1} = C_{t+1}x_{t+1} + w_{t+1}$

We want to find an estimate of our new state conditioned on all of our observations so far (up to and including this time step): $x_{t+1}|y_{1:t+1}$

3.2 Extended Kalman Filtering

The extended Kalman Filter allows us to work with nonlinear systems, like a bicycle model car. The means are updated using the nonlinear model, whereas the covariances use the Jacobian lineariation of this model.

Problem: Write out the nonlinear system along with its Jacobian linearization.

We now have a nonlinear system! It will now be a function, rather than a direct matrix multiplication. We can linearize it using the Jacobian G_t:

$$x_{t+1} = g_t(x_t) + w_t \approx g_t(\mu_{t|t}) + G_t(x_t - \mu_{t|t}) + w_t$$

An analogous set of equations can be written for y_t:

$$y_t = h(x_t) + v(t) \approx h_t(\mu_{t|t}) + H_t(x_t - \mu_{t|t}) + v_t$$

Problem: What does the propagation step look like?

$$\begin{aligned}\mu_{t+1|t} &= g_t(\mu_{t|t} \\ \Sigma_{t+1|t} &= G_t \Sigma_{t|t} G_t^T + \Sigma_w\end{aligned}$$

3.3 EKF SLAM

When EKF SLAM is actually implemented, it happens in 3 steps to account for new features that may be added. These fit into the general framework for SLAM.

The first is the *cost construction* step. Any new features are added into the x vector.

The second is the *Gauss-Newton update* for efficient cost minimization. Based on our history of observations, the distribution for the location of existing features is updated.

Finally, in the *state propagation step*, we update our robot state to the next one. We marginalize out any information that becomes unnecessary to the problem. In EKF SLAM, this is all past poses (since we use only our estimate for the current pose to compute the next one).

4 Frontiers of SLAM

Problem: What is **active perception**?

Often, we have situations where we understand most of the map but are unsure of certain regions or don't have enough information about some areas to plan around. Active perception involves the robot itself understanding the limitations of its data and moving toward these regions to improve its internal representations of the world.

Problem: What is **semantic SLAM**?

We might be moving through our space and updating a map of the features we have. We might want to label these features in order to do something with them. (For example, we might want to label a chair and have the robot move the chair around). This involves a two-step pipeline, where a neural network will label the features, while a normal optimization backend calculates where to move the robot and how to accomplish a given task.

Problem: What is **dynamic SLAM**?

In applications like autonomous cars, we might want to keep track of moving objects and label their new locations. This might include people, other cars, or debris on the road. Dynamic SLAM involves performing these calculations, which may include other attributes like intent inference, in order to increase the safety and efficiency of trajectories.

In addition to all of the above, developments in 3D vision, new forms of trajectory planning, integration of both vision and planning steps, among other ideas, are in development! Localization and mapping, along with trajectory generation, are very interesting problems, and it's exciting to see where this will go.