Project 3: State Estimation

Reid Dye, Eugene Ng
Mechanical Engineering
University of California, Berkeley
{reid.dye, eugeneng}@berkeley.edu

Daniel Municio
Undeclared, Rivian Intern
University of California, Berkeley
danielmunicio@berkeley.edu

***Abstract*—This paper describes the implementation for multiple state estimation methods for a unicycle model and quadcopter model in simulation. Furthermore this paper describes the implementation of EKF on a Turtlebot for state estimation.**

***Index Terms*—State Estimation, Dead Reckoning, Kalman Filter, Extended Kalman Filter**

I. METHODS

A. Dead Reckoning

We are given the equation for dead reckoning, a simple Euler discretization of the dynamics.

$$g(x,u) = \hat{x}_{k+1} = \hat{x}_k + f(\hat{x}_k, u_k)\Delta t \tag{1}$$

Where $\hat{x}$ is the estimated robot state and $f(x,u)$ is $\dot{x}$ given a control input u.
In our implementation of dead-reckoning on a unicycle model, we used the given unicycle model dynamics:

$$x := \begin{bmatrix} \phi \\ x \\ y \\ \theta_L \\ \theta_R \end{bmatrix} u := \begin{bmatrix} u_L \\ u_R \end{bmatrix} \tag{2}$$

$$\dot{x} = f(x,u) := \begin{bmatrix} -\frac{r}{2d} & \frac{r}{2d} \\ \frac{r}{2d} & \frac{r}{2}cos\phi \\ \frac{r}{2}sin\phi & \frac{r}{2}\sin\phi \\ 1 & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} u_L \\ u_R \end{bmatrix} \tag{3}$$

The dead reckoning formula (1) is a simple integration technique - it performs a 0-order approximation of the function f over some small time Δt, and computes the solution to the approximation of the full ODE:

$$\dot{x} = f(x,u) \approx c \tag{4}$$

$$\implies x(t) \approx x(0) + ct \tag{5}$$

This is a first-order approximation for $x(t)$, but over small Δt, it is functional. The easiest way to improve dead reckoning is to switch to a higher-order integration method. The fourth order Runge-Kutta method with a fifth order corrector (as implemented in the `ode45` method in Matlab) is a common choice, as it results in a good balance between mathematical accuracy, simplicity, and numerical stability. Its error is $O(\Delta t^6)$, which is significantly better than the $O(\Delta t^2$ of Euler's method. There are also many other options, including multistep methods, symplectic methods, and more; a full review of numerical methods for integration is out of the scope of this work. The main problem with dead reckoning, however, is that it does not account for model inaccuracy. Even with a perfect discretization, there is process noise which goes unaccounted for. Incorporating the information from sensors to create a feedback loop could eliminate the drift problem of dead reckoning. This idea motivates all other state estimators, like Luenberger observers and Kalman filters.

B. Kalman Filter

The Kalman filter builds upon dead-reckoning by correcting the accumulation of error with sensor data, which we assume to be noisy where the noise follows a normal distribution. For the Kalman filter to work, we need a Linear system, however the given unicycle model is not linear, so in this situation we fix our steering angle $\phi = \frac{1}{4}$ to allow the Kalman filter to be linear, giving us a new state and dynamics:

$$x := \begin{bmatrix} x \\ y \\ \theta_L \\ \theta_R \end{bmatrix}, u = \begin{bmatrix} u_L \\ u_R \end{bmatrix} \tag{6}$$

$$x[t+1] \approx Ax[t] + Bu[t] + w[t] \tag{7}$$

$$= I \cdot x[t] + \begin{bmatrix} \frac{r}{2}\cos(\phi) & \frac{r}{2}\cos(\phi) \\ \frac{r}{2}\sin(\phi) & \frac{r}{2}\sin(\phi) \\ 1 & 0 \\ 0 & 1 \end{bmatrix} \Delta t \cdot u[t] + w[t] \tag{8}$$

$$\tag{9}$$

Where we have an A and B matrix in State-Space form. For our sensor data, we use the vehicles position in x and y and assume there is some noise from the sensor giving us an output in state space form as:

$$y[t] = Cx[t] + v[t] = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \end{bmatrix} x[t] + v[t] \tag{10}$$

With these matrices, we apply the general form for the Kalman filter as given in the project document. Where it involves matrices Q, R, P_0 where we can treat Q as the process noise covariance, as a measurement of how much we want to trust the prediction $g(x, \mathbf{u})$ which we can believe as having unaccounted forces in the dynamics or pertubations to the system. We can treat R as as our guess for the covariance of our measurement or how uncertain we are in our measurements/sensors as in the real world, sensors aren't perfect and may be noisey so we try to account for this in the Kalman Filter. Lastly we have matrix P_0 as how much we trust our initial state x_0, as we need to start from an initial state to perform state estimation, the sensors may not know where the robot actually starts, so a high covariance means we don't have a good estimate on where the initial state of the robot is.

Given the physical intuition of our covariance matrices, to tune these hyperparameters, we utilized RMSE error to evaluate our model, and performed a grid search over a range of values. Where we found our optimal hyperparameters

TABLE I
KALMAN FILTER RMSE WITH DIFFERENT Q, P, AND R SCALING FACTORS

Q Scale	P Scale	R Scale	X and Y RMSE
$\times 10$	-	-	0.05190070911023071
$\times 1$	-	-	0.0606944433892956
$\times 0.1$	-	-	0.08510808533780556
$\times 1.5$	-	-	0.051599359689252354
$\times 0.5$	-	-	0.05392368012380627
$\times 0.85$	-	-	0.051174547223242155
$\times 0.75$	-	-	0.05150846183349716
$\times 0.85$	$\times 2$	-	0.05105270627197948
$\times 0.85$	$\times 5$	-	0.0509224883119383
$\times 0.85$	$\times 10$	-	0.050861403729942516
$\times 0.85$	$\times 15$	-	0.05083837233400024
$\times 0.85$	$\times 20$	-	0.05082630699294504
$\times 0.85$	$\times 50$	-	0.050803629124811664
$\times 0.85$	$\times 100$	-	0.050603364686163124
$\times 0.85$	$\times 250$	-	0.05079101794950357
$\times 0.85$	$\times 150$	-	0.050793145028018384
$\times 0.85$	$\times 100$	$\times 2$	0.05479819269750544
$\times 0.85$	$\times 100$	$\times 0.5$	0.051854752037295115
$\times 0.85$	$\times 100$	$\times 0.8$	0.050703773755173015
$\times 0.85$	$\times 100$	$\times 0.9$	**0.050690785051053865**

As we are in a simulation environment, we have access to the ground truth for our measurement model, leading to our C matrix being able to extract the x and y positions of the robot from our sensor. However if we unable to get the ground truth and state of our robot as a measurement, we would need to derive a measurement model that relates a sensor to the robot state, for example a LiDAR or depth camera can give us the depth of a known landmark, and we can derive the distance of our robot to that landmark using the distance formula. With that, we can develop a measurement model and linearize it to use for a Kalman filter. With this we can do a similar tuning method to get a Q, R, and P_0 matrix for this measurement model.

C. Extended Kalman Filter

To account for the nonlinearity of the system, we implemented an Extended Kalman Filter (EKF), which differs from the standard Kalman filter in two ways:

- The prediction step is performed using the full nonlinear discrete dynamics.
- The update step uses a linearization of the dynamics about the current state estimate.

Mathematically, we use the same euler discretization as (1) to perform the prediction step, and we let the state jacobian A be re-evaluated at each timestep, making the covariance extrapolation step as follows:

$$P_{k+1} = \left(\frac{\partial g}{\partial x}\bigg|_{\hat{x}_k, u_k} \right) P_k \left(\frac{\partial g}{\partial x}\bigg|_{\hat{x}_k, u_k} \right)^\top + Q \tag{11}$$

This still assumes that the noise is gaussian and zero-mean, but it allows the observer to account for nonlinearities in the system model, like the turning. We do not account for nonlinearities in the measurement model because it is linear; in the general case, it should be repeatedly linearized about the current state estimate just like the system dynamics. In addition to still assuming zero-mean gaussian noise, the EKF has two other main limitations:

- Since the system is nonlinear, the marginalization is no longer accurate (even if the current state estimate and system model are perfect) and the linear approximation to the problem means that we effectively only perform one step of an SQP solve per timestep. This can still be stable if the optimization converges faster than the true optimal value changes, but this is not guaranteed. We also therefore lose all sense of theoretical optimality.
- When the system is linearized in the update step, there is no guarantee that the point of linearization is anywhere near the true system state, and therefore there is no guarantee that the linearization is at all valid. Thus, among other things, the stability of the EKF may depend on the initial guess being within some finite region around the true state.

In general, however, these limitations are expected. When transitioning from a linear to nonlinear system, we lose the knowledge that stability is always global, so it makes sense for this to apply to observers as well.

When we implemented the EKF on the real turtlebot, we defined our process model to be the same as in the sim - the same form as (1) with f defined by (7). We defined the process noise to be zero and the sensor noise to be gaussian, with the following (somewhat arbitrary) definition:

$$y = h(x) + \mathcal{N}\left(h(x),\ 10^{-2} \cdot \begin{bmatrix} 1 & 0.5 & 0.1 \\ 0.5 & 0.1 & 0.2 \\ 0.1 & 0.2 & 0.2 \end{bmatrix} \right) \tag{12}$$

This noise was added to the output as shown above, and the EKF's measurement covariance parameter R was set accordingly. We did not add any process noise, but instead tuned the state covariance matrix manually. We defined our measurement model to be the same as (10) because that is the data accessible in the `tf2` lookup. We treat that as the ground truth and choose to add fabricated noise because that is easier than coming up with our own external localization method. However, we feel that this is mostly still valid, since we do not know how the transform is computed. Furthermore, if we (for example) set up a camera above the turtlebot to read its AR tag, the sensor noise would likely be roughly gaussian, assuming the camera is close enough to directly overhead. We also choose to add no process noise, since the dynamics model was (qualitatively from experiment) very good at predicting the turtlebot's true position - the simulated ground truth was always very very close to the true physical location of the turtlebot. We decided that, as this error would be orders of magnitude less significant than that of a rudimentary external vision based localization system, we would only focus on sensor noise.

II. Experimental Results

A. Dead Reckoning

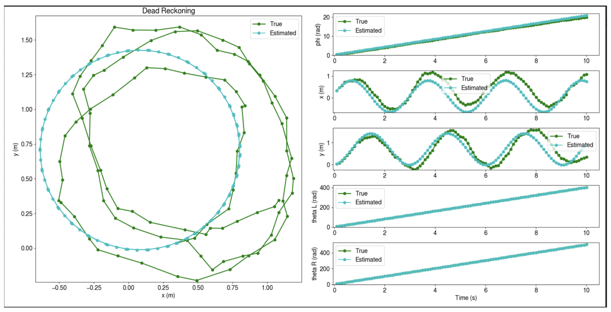

Fig. 1. Dead Reckoning on Unicycle Simulation

In figure 1 we can see the results of Dead-Reckoning for state estimation, where because we only use the vehicle dynamics and the control input, we assume a perfect environment with no perturbations or noise which is represented in the estimated trajectory. Furthermore there is no correction based off of sensor feedback causing the estimated trajectory and ground truth to have the same shape because of the control inputs, but differ greatly.

B. Kalman Filter

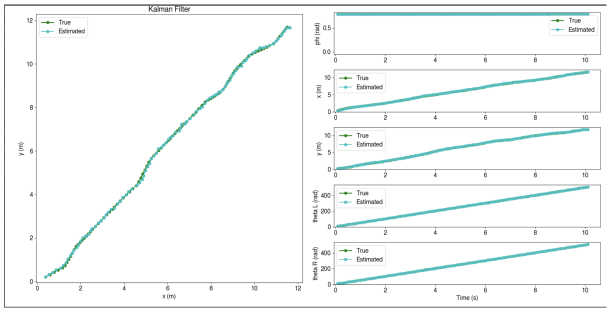

Fig. 2. Kalman Filter with identity covariance

In figure 2 we can see the results of implementing a Kalman filter with all covariance matrices being the identity matrix, and we also plot the error in figure 3 for each item in the state to visualize error over time. With these plots and the RMSE value we are able to tune our hyperparameters with the method described previously.

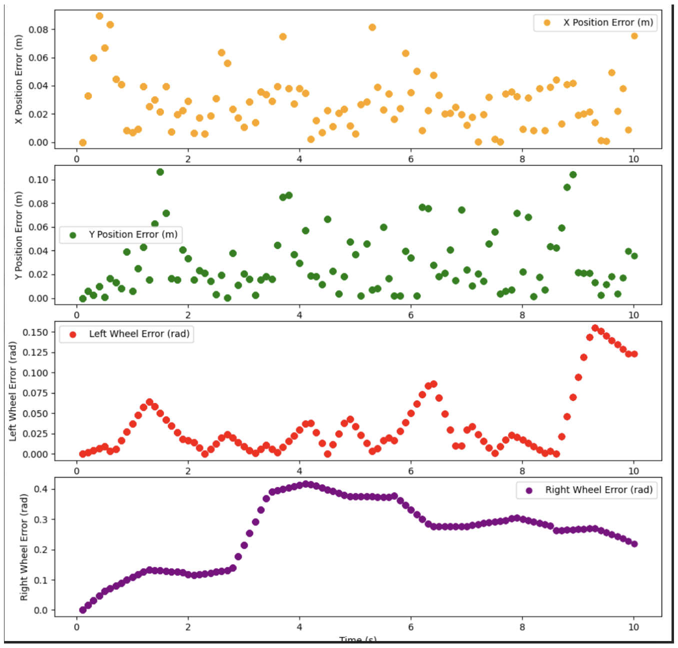

Fig. 3. Kalman Filter Identity covariance error

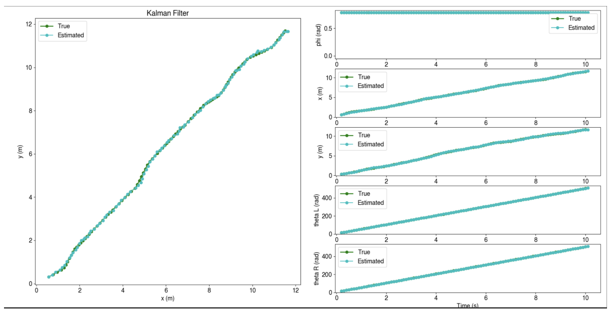

Fig. 4. Tuned Kalman Filter

In figure 4 we can see the final tuned Kalman Filter Where we get an RMSE of 0.050690785051053865.

C. Extended Kalman Filter

With the implementation of the extended Kalman filter, we are able to have nonlinear dynamics as well as nonlinear measurement models that allow us to have a more diverse range of tracking. Even without tuning, it's able to yield solid results, and then near-perfect results after tuning.

D. Extended Kalman Filter on Turtlebot

E. Comparison between methods

Algorithm	**RMSE**	**Computation Time (ms)**
Dead-Reckoning	0.4278	0.07904
Kalman Filter	0.05069	0.24195
EKF	0.09418	0.04066

III. Discussion

Dead Reckoning performed characteristically poorly, but was obviously the fastest, as it uses the least amount of

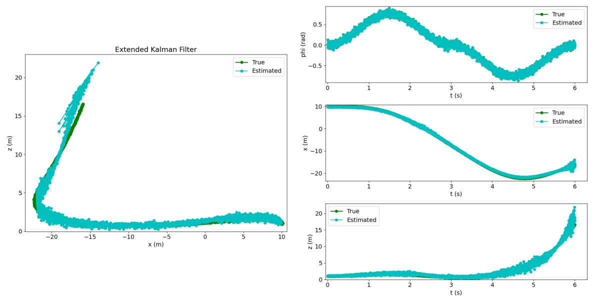
Fig. 5. Untuned Extended Kalman Filter on Planar Drone

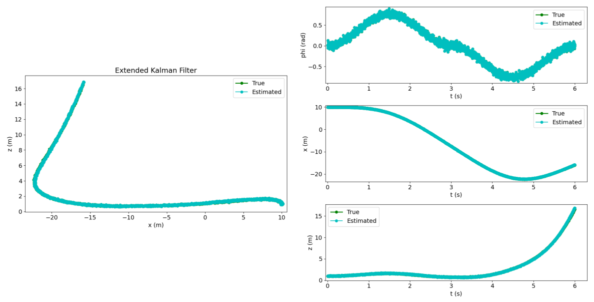
Fig. 6. Well-Tuned Extended Kalman Filter on Planar Drone

information. This is not necessarily a bad thing, as it's not realistic to expect good results from such a simple method of state estimation, but it definitely leaves more to be desired. Dead Reckoning would likely only be used when there is no other measurement information available. It is the fastest estimator, but the extra compute from a Kalman Filter or EKF is incredibly small, and is well worth it in the majority of cases.

The Kalman Filter performed incredibly well on the linear system compared to dead reckoning. The main downside however, is that we had to fix the angle of the turtlebot to make the system linear. While it does have the benefit of being the mathematically most optimal estimator, there are very few interesting systems in this world which are linear. It also requires that other measurements are given in the system, but this is usually not a tall ask, as most robots come with a variety of sensors that give measurements that can be related to the state of the robot. For a turtlebot, this can be a GPS, IMU, camera, LiDar, wheel encoder, etc. It's also still computationally very light, as the state of a system usually contains only a handful of variables, and matrix multiplication of nxn matrices is almost instant, assuming n is a small number. The Kalman Filter did take the longest time out of all algoirhtms, but in an order of magnitude so small, there is no real difference. Lastly, the kalman filter gave incredibly accurate results, even while using the identity matrix, which shows that even without fine-tuned covariances, or specific information about the sensors, you can achieve much better results using a kalman filter than with dead reckoning.

Implementing the EKF on a real turtlebot was tricky. A few different issues arose which do not regularly occur in simulation. For starters, we have variable `dt`, which is simple enough to implement, but came with the headache of floating point precision loss where `dt` would get rounded to 0, since time was a very large number with many significant figures. Additionally, there is no real perfect ground truth state. Instead, we used the turtlebot's odom transform, and treated that as ground truth state. We then took measurements from the ϕ, x, y of the odom transform. But this form of treating our measurements as ground truth, filtering them with our control inputs, and then comparing the result to our "ground truth measurements", is similar to convincing yourself you don't have schizophrenia, because all your friends told you so. To confirm that the implementation could handle outside measurement noise, we added noise to the odom transformation before sending it as a measurement. The EKF still held up fine with this addition of noise The main takeaway here being, there is no 'ground truth' in the real world, but there are measurements you can trust more than others, which is the whole point of the EKF and KF.

One notable promising result was that the EKF output had about the same amount of noise in x and y, despite σ_{xx} being half as big as σ_{yy}. This means that the EKF was able to correct for the very asymmetric noise (after we told it the shape of the noise).

The EKF should actually take measurements from a sensor, some popular choices would be a GPS or camera, the GPS giving the x and y position of the turtlebot, and the camera could give the turtlebot's position relative to a landmark. This of course requires some information about the landmarks position before hand, to be able to derive the position of the turtlebot relative to the landmark. If no previous information about the landmark is given, then some form of SLAM will have to be used.

In the vast majority of physical systems, an EKF would be my implementation of choice. While we cannot mathematically prove it is the most optimal estimator, the first order linearization about an equilibrium point is good enough in most cases, and the EKF doesn't add a noticeable amount of computational tax onto the system. It also doesn't require a significant amount of tuning to be better than dead-reckoning. Even without exact covariances, an experienced engineer will generally know which sensors he can trust more than others.

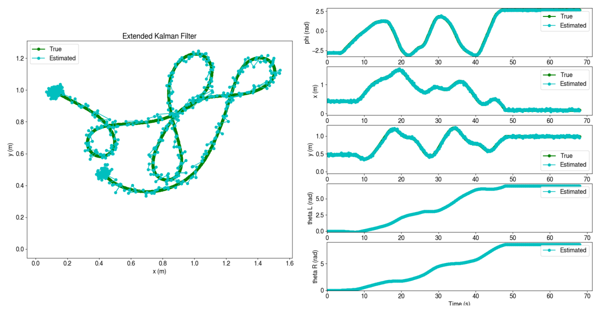
Fig. 7. EKF on turtlebot with added noise

For example, I generally have more faith in a wheel speed sensor or encoder, than an IMU, for getting the velocity of a vehicle (IMU relies on integrating acceleration, and integrating twice to get the position, which will magnify noise). So even with no prior knowledge of the covariances of the sensors, I can have much better results than dead-reckoning, without noticeably slowing down my estimator.

References

[1] Timothy D Barfoot. State Estimation for Robotics. Cambridge University Press, 2017.

[2] Kevin M Lynch and Frank C Park. Modern robotics: Mechanics, Planning, and Control. Cambridge University Press, 2017.

IV. Appendix

Github: https://github.com/ucb-ee106-classrooms/project-3-autonomotrons

Videos of Turtlebot EKF: https://tinyurl.com/ekfturtlebot

Backup Video in case of url not working:
$https : //drive.google.com/drive/folders/1OGWCY_5Ebxo_oJ7faZeuSlj8r9pgZU0E$

A. Learning Goals and Outcomes:

I think this project did a great job of showing how to implement a Kalman Filter and EKF, and why they are both significantly better than dead-reckoning, which I assume was the goal of the project. The algorithm steps which were included in the project doc were phenomenal at explaining how to integrate the algorithms effectively. I also enjoyed the amount of self work that was required, including manually calculating the jacobians. I do think introducing a symbolic libary like sympy to show how you can calculate jacobians that way would've also been cool, and a helpful tool for the future.

Lastly, making projects which you can do solely in python (without ROS) is very much appreciated!

B. Random Issues and Starter Code

There were a few random issues we ran into, some which I think should be added to a project common mistakes doc for future classes, and some which should be fixed entirely.

- We were unable to pip install the requirements.txt immediately
  - I imagine this is more to do with the lab computers, considering the requirements was just numpy, matplotlib, and scipy.
  - We had to manually pip install –upgrade matplotlib to be able to get the proper animations
- We still cannot catkin make and source install/setup.bash directly after cloning the repo
  - The turtlebot directory has to be put in a src directory
  - This is not the first project we've had the issue, but it's become a known and simple fix by now
- The turtlebot odom footprint gives orientation in quaternions
  - This set us back 2 hours. Quaternions are fine, and we probably should've looked closer when topic echo'ing odom, but should be something to remind students before the project.
- Rospy.time gives a really really long time, and when we try to calculate dt, it rounds it off to 0 because of some floating point issues.
  - We switched to time's perf counter and got much better results
  - Shoutout ME136 where Professor Mueller warned the entire class about this causing issues in the first lecture, and it has always stuck with me