State Estimation

Tyler Osbey
EECS
College of Engineering
Berkeley, USA
tosbey836@berkeley.edu

Edwin Villalpando
EECS
College of Engineering
Berkeley, USA
edwin23luv@berkeley.edu

Tenzin Norphel
EECS
College of Engineering
Berkeley, USA
tnorphel@berkeley.edu

***Abstract*—In this project, we implemented three state estimation techniques: dead reckoning, the Kalman filter, and the Extended Kalman Filter (EKF). We evaluated these methods in simulation using a TurtleBot and a planar quadrotor. Dead reckoning was applied to both platforms, while the Kalman filter was implemented for the TurtleBot and the EKF for the quadrotor. Our results showed that dead reckoning diverged from the true trajectory over time due to accumulated errors, whereas the Kalman filter and EKF produced estimates closely aligned with ground truth. Since dead reckoning lacks sensor updates, we introduced a time-based stopping criterion for the TurtleBot's trajectory. For the EKF, we fine-tuned the process noise covariance Q and initial state covariance P to small values, while setting the measurement noise covariance R to 100 for better accuracy. Our tuning approach significantly improved state estimation performance.**

I. Methods

A. Dead reckoning

Dead reckoning relies solely on the current state of the robot and current system inputs to calculate the robot's next state, given some timestep:

$$g(\mathbf{x}, \mathbf{u}) = \mathbf{x} + f(\mathbf{x}, \mathbf{u}) \cdot \Delta t, \tag{1}$$

Where g is the estimation of the future state given the current true state at time t and the current system input u, and Δt is the timestep required to execute a movement f in the direction of the estimated state. For the Turtlebot, the dynamics are defined as:

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

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

And the planar quad-rotor's dynamics are similarly defined as:

$$\dot{\boldsymbol{x}} = \begin{bmatrix} \dot{x} \\ \dot{z} \\ \dot{\phi} \\ \ddot{x} \\ \ddot{z} \\ \ddot{\phi} \end{bmatrix} = \begin{bmatrix} \dot{x} \\ \dot{z} \\ \dot{\phi} \\ 0 \\ -g \\ 0 \end{bmatrix} + \begin{bmatrix} 0 & 0 \\ 0 & 0 \\ 0 & 0 \\ 0 & 0 \\ -\frac{\sin(\phi)}{m} & 0 \\ \frac{\cos(\phi)}{m} & 0 \\ 0 & \frac{1}{J} \end{bmatrix} \begin{bmatrix} u_1 \\ u_2 \end{bmatrix} = f(\boldsymbol{x}, \boldsymbol{u})$$

Deriving these is as simple as taking the time derivative of the state variables to derive the motion variables. For both systems this involves approximating the dynamics using Euler integrals and implementing Runge-Kutta approximations fo the state variables. One of the key limitations of dead reckoning, as mentioned above, is that it does not utilize sensor data from the robot. This means that any process noise introduced by the environment and the sensors remains zero throughout the trajectory runtime, which makes the dead reckoning algorithm akin to an open loop/feedforward controller. Errors will continue to accumulate until the dead reckoning trajectory no longer accurately matches the intended trajectory.

B. Kalman Filter

Kalman filtering solves the propagating error problem with dead reckoning by introducing both process and random noise. We define Q to be the covariance of the process noise/system dynamics, R to be the covariance of the measurement noise, and P to be the state noise at timestep t:

$$Q = diag(6) \times 1 \times 10^{-3}$$

Where diag(x) represents an identity matrix of dimension x by x. Here is the how we set up the matrix R:

$$R = diag(2) \times 1 \times 10^{-5}$$

Here is the how we set up the matrix P_0:

$$P = diag(6) \times 1 \times 10^{-3}$$

Where after every time step it updates as:

$$\mathbf{P} = (\mathbf{I} - \mathbf{KC})\,\mathbf{P}$$

Here K represents the Kalman Gain for each time step (this will be explained further in EKF).

We represent the uncertainty/error with Q, R, and P and use these matrices to estimate the state at time t. Additionally, we define new matrices for the current system state transitions, the control inputs, and the measurements/sensor data:

$$A = diag(6) \times 1$$

$$\boldsymbol{B} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & \frac{r}{2}\cos\phi\Delta t & \frac{r}{2}\cos\phi\Delta t \\ 0 & \frac{r}{2}\sin\phi\Delta t & \frac{r}{2}\sin\phi\Delta t \\ 0 & \Delta t & 0 \\ 0 & 0 & \Delta t \end{bmatrix}$$

$$\boldsymbol{C} = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \end{bmatrix}$$

For the Turtlebot, we lock the value of ϕ in order to linearize the system, as Kalman Filtering is not intended for use on nonlinear system such as the unicycle model. Even given this linearization, however, the Kalman Filter algorithm is limited for nonlinear systems.

C. Extended Kalman Filter - EKF

In our EKF implementation, we used three key matrices and used the Jacobian to linearize the system around the current state estimate. A limitation of using the Jacobian for linearization is that if the initial state estimate is poor or far from the true value, the nonlinear system can diverge and become unstable. This results in inaccurate predictions and poor state estimation. Similarly, if the system takes on a more drastic, nonlinear behavior at the given estimation state, the linear representation at the estimated state will be more prone to error.

Another challenge arises from EKF's assumption that noise follows a Gaussian distribution, as seen with a regular Kalman Filter. If the initial guess is significantly off, the filter's performance can degrade over time, as errors vary through following estimations.

To address these limitations, one of the best approaches is to improve the initial state estimate, ensuring it is as close as possible to the true state. Reducing the initial error between the estimated and true state helps maintain stability and accuracy. If improving the initial estimate is not feasible, another approach is to decrease the discrete time step Δt. A smaller time step minimizes the impact of nonlinearities by making discrete time steps smaller, thereby reducing the error found within the nonlinear system.

In order to implement EKF, we tuned values for Q, R, and P where Q is 6 x 6 dimension matrix. R is 2 x 2 matrix. And P is 6 x 6 matrix. Here is the how we set up the matrix Q:

$$Q = diag(6) \times 1 \times 10^{-8}$$

Here is the how we set up the matrix R:

$$R = diag(2) \times 100$$

Here is the how we set up the matrix P:

$$P = diag(6) \times 1 \times 10^{-4}$$

We then implemented the update function, which predicts the next state using the current state n via the function$g(x, u)$, where x represents the current state estimate and u is the control input. We require the function g and h for state motion and output the predicted state $n + 1$.

The Jacobian of state transition matrix A is listed below, taking the current state and most updated control input into consideration:

$$A = \begin{bmatrix} 1 & 0 & 0 & dt & 0 & 0 \\ 0 & 1 & 0 & 0 & dt & 0 \\ 0 & 0 & 1 & 0 & 0 & dt \\ 0 & 0 & -\frac{dt \cdot u_1 \cos(\phi)}{m} & 1 & 0 & 0 \\ 0 & 0 & -\frac{dt \cdot u_1 \sin(\phi)}{m} & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix}$$

The matrix $P \in \mathbb{R}^{6\times6}$ represents the **covariance prediction** step, which highlights the change in estimation error between the previous and current time steps. It reflects how the system's uncertainty influences the current state estimate, while also taking into account the noise.

Here is how the matrix $\mathbf{P}$ (covariance prediction) is expressed in mathematical form:

$$\mathbf{P} = \mathbf{APA}^\top + \mathbf{Q}$$

And the Euclidean distance to the landmark for approximating matrix C is given by:

$$d = \sqrt{(l_x - x_{\text{pos}})^2 + l_y^2 + (l_z - x_2)^2}$$

Kalman Gain

$$\mathbf{K} = \mathbf{PC}^\top \left(\mathbf{CPC}^\top + \mathbf{R}\right)^{-1}$$

The logic behind the kalman gain is to approximate how much ratio is needed between prediction value and measurement value. If the Kalman gain K is high, we focus more onto measurement value. If the Kalman gain K is low, we focus more onto prediction value

State Update

$$\hat{x}_k = \hat{x}_k^- + \mathbf{K}\left(y_k - h(\hat{x}_k^-, y_k)\right)$$

Once we figure out from Kalman gain, whether prediction or measurement value, we then update the state matrix, and perform correction based on prior(placeholder value/latest data).

Covariance Update

$$\mathbf{P} = (\mathbf{I} - \mathbf{KC})\,\mathbf{P}$$

In this step, we assess the deviation between the predicted and actual state and evaluate how much uncertainty is present. We incorporate the most updated data from the prior prediction. Here, the I term refers to the identity matrix, and the KC term reflects the correction applied to the uncertainty based on the Kalman gain and observation matrix. Therefore, the larger the KC term, the greater the reduction in uncertainty, and the smaller the KC term, the less reduction in uncertainty. The P term refers to the prior covariance matrix, which represents the amount of uncertainty before incorporating the new measurement data.

Here is the summary of algorithm in pseudocode:

Inputs: $\mathbf{x}_0, (\mathbf{u}[0], \ldots, \mathbf{u}[T-1]), (\mathbf{y}[0], \ldots, \mathbf{y}[T]), \Delta t, T, \boldsymbol{A}, \boldsymbol{B}, \boldsymbol{C}, \boldsymbol{Q}, \boldsymbol{R}, \boldsymbol{P}_0$
Outputs: $\hat{\mathbf{x}}[0], \ldots, \hat{\mathbf{x}}[T]$

$t \leftarrow 0$
$\hat{\mathbf{x}}[t] \leftarrow \mathbf{x}_0$
$\boldsymbol{P}[t] \leftarrow \boldsymbol{P}_0$
while $t \leq T-1$ **do**
  $\hat{\mathbf{x}}[t+1 \mid t] \leftarrow g(\hat{\mathbf{x}}[t], \mathbf{u}[t])$ (state extrapolation)
  $\boldsymbol{A}[t+1] \leftarrow \left.\frac{\partial g}{\partial \mathbf{x}}\right|_{(\hat{\mathbf{x}}[t], \mathbf{u}[t])}$ (dynamics linearization)
  $\boldsymbol{P}[t+1 \mid t] \leftarrow \boldsymbol{A}[t+1]\boldsymbol{P}[t]\boldsymbol{A}^\top[t+1] + \boldsymbol{Q}$ (covariance extrapolation)
  $\boldsymbol{C}[t+1] \leftarrow \left.\frac{\partial h}{\partial \mathbf{x}}\right|_{\hat{\mathbf{x}}[t+1|t]}$ (measurement linearization)
  $\boldsymbol{K}[t+1] \leftarrow \boldsymbol{P}[t+1 \mid t]\boldsymbol{C}[t+1]^\top \left(\boldsymbol{C}[t+1]\boldsymbol{P}[t+1 \mid t]\boldsymbol{C}[t+1]^\top + \boldsymbol{R}\right)^{-1}$ (Kalman gain)
  $\hat{\mathbf{x}}[t+1] \leftarrow \hat{\mathbf{x}}[t+1 \mid t] + \boldsymbol{K}[t+1]\left(\mathbf{y}[t+1] - h(\hat{\mathbf{x}}[t+1 \mid t])\right)$ (state update)
  $\boldsymbol{P}[t+1] \leftarrow (\boldsymbol{I} - \boldsymbol{K}[t+1]\boldsymbol{C}[t+1])\boldsymbol{P}[t+1 \mid t]$ (covariance update)
  $t \leftarrow t+1$
end while
return $\hat{\mathbf{x}}[t]$ for $t = 0, \ldots, T$

Fig. 1. EKF Algorithm in pseudocode

II. Experimental Results

After implementing Dead Reckoning on the turtlebot and quadrotor, KF to the turtlebot, and EKF to the quadrotor, we were able to see the benefits and drawbacks of each model. Qualitatively, we were able to see perfect trajectories being performed by both models using Dead Reckoning, which was to be expected. From KF and EKF, we can see corrections being made to follow closer to the actual trajectory of either model. Additionally, we saw some variation in the per-step computational run-time for each form of state estimation. All in all, this was to be expected. Our results are as follows:

Algorithm	**Mean Squared Error (MSE)**	**Per-Step Time (s)**
Quadrotor [MSE x , MSE z , MSE ϕ]		
Dead Reckoning	$[7.25 \times 10^4,\ 2.13 \times 10^3,\ 51.09]$	4.87×10^{-5}
EKF	[3509.26, 681.44, 19.52]	1.00×10^{-4}
TurtleBot [MSE ϕ, MSE x, MSE y, MSE θ_L, MSE θ_R]		
Dead Reckoning	[26.35, 9.55, 7.93, 190.84, 302.48]	0.09899
Kalman Filter	[0.00, 0.182, 0.241, 0.283, 7.76]	0.09899

TABLE I
Mean Squared Error (MSE) and Per-Step Computational Time for Each Estimation Algorithm

*NOTE: For the Quadrotor, the top image on the right side is the ϕ position (rad), the middle one is the x position (m), and the bottom one is x position (m). For the Turtlebot, the top one is the ϕ position (rad), then the x position (m), then the y position (m), θ_L position (rad), and finally the θ_R position (rad).

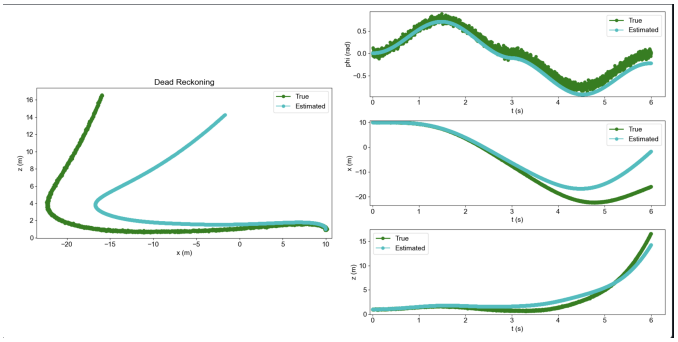

Fig. 2. Planar Quadrotor simulation using dead reckoning

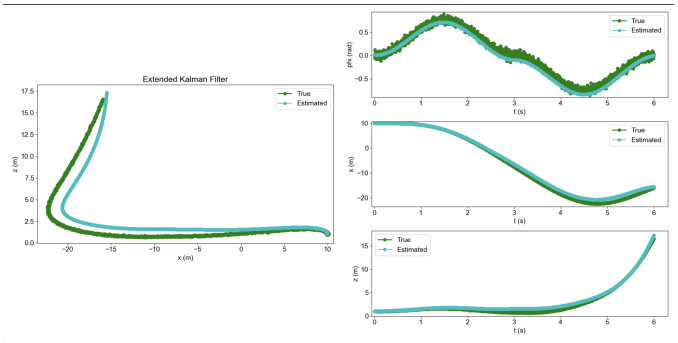

Fig. 3. Planar Quadrotor simulation using Extended Kalman Filter

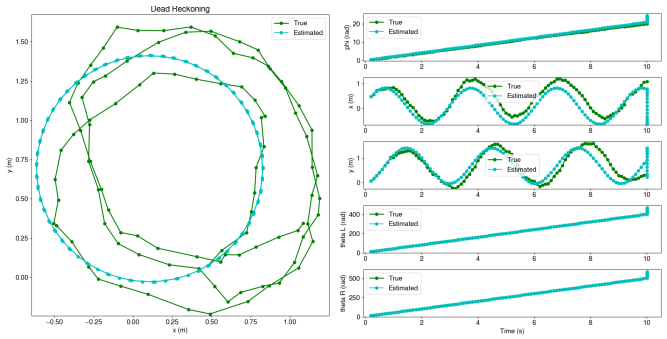

Fig. 4. Turtlebot simulation using dead reckoning

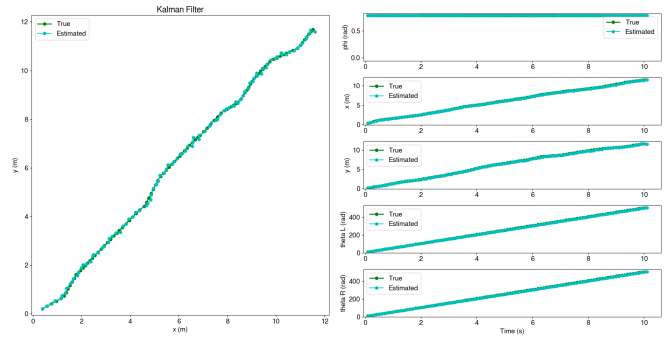

Fig. 5. Turtlebot simulation using Kalman Filter

III. Discussions

Since dead reckoning was implemented twice, it was the easiest to see its flaws and benefits compared to those of KF/EKF. Without comparison, we can see that dead reckoning performs poorly regardless of the model it is being used on. Dead reckoning on the quadrotor, as seen with **Fig. 2**, does a decent job at following phi, however, it struggled to follow the x and z positions correctly. This is because of its inability to deal with noise from the environment, such as drag and other erroneous components of the quadrotor and model. Other than that, its benefit resides in its simplicity as dead reckoning follows simple dynamic models. Thus, dead reckoning can be viewed as a simple feedback loop where the outputs of one state are inserted directly into the next state without modification.

Kalman Filter was also simple, but effective. As explained previously, KF was implemented for the turtlebot, however, since it can only be used for a linear system, the problem was simplified further by having the quadrotor fly in a constant direction. As seen in **Fig. 5**, KF worked flawlessly, with no visible error in any of the measured positions. It was a little difficult to tune, however, making it the biggest struggle for an incredibly successful estimation method.//

In Extended Kalman Filter(EKF), we were able to linearize the state estimation using Jacobians, which efficiently deal with nonlinearity. **Fig. 3** depicts the characteristics found when implementing (EKF). For the most part, the quadrotor was able to follow the ϕ, x, and z positions with very little error. However, we can see a slight deviation in the estimated trajectory. After tuning for multiple hours, this is the best trajectory achieved, however, I believe there must be some ideal value to get even better results. AS a result, the downside to EKF is tuning, as it is quite difficult to get right.

When it comes to comparing these state estimation methods to one another, both quantitative and qualitative results point to dead reckoning being the worst. As seen with the turtlebot graphs, KF outperforms dead reckoning when it comes to generating an accurate trajectory. The cost function shown in **TABLE 1** is also magnitudes better than the one for dead reckoning. When it comes to how computationally expensive KF is compared to dead reckoning, I was expecting some faster times for dead reckoning, however, to my surprise and multiple checks, they both take the same amount of time per-step. Thus, for a linear system, KF should be chosen as the only benefit found for dead reckoning is its simplicity to implement. Choosing between EKF and dead reckoning is not as easy. We can see that the runtime for dead reckoning is much faster compared to that of EKF. However, the trajectory produced by EKF is much better, as we can see the estimated final state to be very close to the true final state. The cost function for EKF is also much better than dead reckoning. When it comes to choosing between dead reckoning and EKF, EKF should be chosen most of the time, unless noise is negligible within that given system. EKF and KF will not be compared as they are the same. Use KF if you have a linear system, otherwise, for a nonlinear system, use EKF.

To improve dead reckoning, without changing the reasoning behind its creation too much, we can include some correcting factor by incorporating a sensor for minor feedback corrections as seen with basic controllers (i.e. PID, Lyapunov, etc). Possible improvements for KF and EKF are generally the same. To get better results, we can incorporate finer tuning of the covariance matrices. A potential improvements for EKF can stem from Newtons Method when it comes to training. Instead of calculating the gradient/Jacobian at a given state, we can get more accurate results by calculating the hessian instead.

Acknowledgment

The authors would like to express their sincere gratitude to EE106B Distinguished Professor S. Shankar Sastry for his guidance and support throughout the project. Special thanks to our lab TAs, K. El-Refai and Yarden Goraly, for their invaluable assistance in the development and conceptual understanding of the optimization planner, RRT Planner (Rapid Exploring Random Tree), and sinusoid planner with their implementations on TurtleBot. Additionally, we extend our appreciation to Jaeyun Stella for her support in logistics and policy-related aspects of the project.

APPENDIX

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

IMPROVEMENTS

The project was effective at teaching the learning goals and helping us learn about state estimation. I think the pedagogical success could be improved if it was clearer how to set up the project and for each of the state estimation methods, their algorithm is pretty clear. But it could have been much better if the matrix P is clearly defined and P_0 and go indepth onto process and measurement noise and how it impacts different state estimation methods. Also, explaining how kalman gain impacts on uncertainty, and prior(state of the system before the latest update). In terms of starter code, it looks fine and everything has its own component and proper variables being used. For instance, the functions g and h are properly set up from the mathematical form. Overall, the lab documentation and starter code is well defined, and well provided instructions for each of the state estimation methods.