Balancing Robot

Izzy Mones and Heidi Dixon

May 29, 2025

Robot Design

Should have list of all the components. Maybe a picture. Do we need to show circuit stuff?

Measuring the Robot State

The position x and velocity $\dot{x}$ along the x axis are computed from the motor encoders. The encoders return a number of counts which is positive in the clockwise direction and negative in the counter clockwise direction. Using the number of counts per wheel rotation and the circumference of the wheel we can compute the distance in the x direction. The instantaneous velocity can be estimated by computing the change in position over a very small time window times the length of the window. The angle θ is computed from the accelerometer and angular velocity $\dot{\theta}$ is given by the gyroscope.

Model

Need a drawing of model.

Do we want a derive the equations?

We modeled our balancing robot as a pendulum cart system.The wheels and motors are considered to be the cart. All other parts of the robot pivot around the axis created by the wheels and are considered part of the pendulum. Our system is described by the system of differential equations

$$\dot{x} = \dot{x} \tag{1}$$

$$\ddot{x} = \frac{-mg\cos(\theta)\sin(\theta) + mL\dot{\theta}^2\sin(\theta) - \delta\dot{x} + u}{M + m\sin^2\theta} \tag{2}$$

$$\dot{\theta} = \dot{\theta} \tag{3}$$

$$\ddot{\theta} = \frac{(m+M)g\sin(\theta) - \cos(\theta)(mL\dot{\theta}^2\sin(\theta) - \delta\dot{x}) - \cos(\theta)u}{L(M + m\sin^2\theta)} \tag{4}$$

where x is the cart position, $\dot{x}$ is the cart velocity, θ is the pendulum angle, $\dot{\theta}$ is the angular velocity, m is the pendulum mass, M is the cart mass, L is the distance from the pivot point to the center of mass of the pendulum, g is the gravitational acceleration, δ is a friction damping on the cart, and u is the control force applied to the cart. This forms a set of differential equations

$$\frac{d}{dt}\boldsymbol{x} = f(\boldsymbol{x}, \boldsymbol{u}) \tag{5}$$

where $\boldsymbol{x}$ is the state vector $\boldsymbol{x} = [x, \dot{x}, \theta, \dot{\theta}]$ and $\boldsymbol{u}$ is the input vector $\boldsymbol{u} = [u]$.

Modeling Motors

Talk about how we didn't model motor circuit. We assumed that torque is a linear function of duty cycle. We tuned the coefficient experimentally. This model is probably pretty good as long as are not near their max duty cycle. Why?

Linearization

To build a control system for our model we will linearize the non-linear system of equations (1), (2), (3), and (4) around a fixed point $(\boldsymbol{x}_r, \boldsymbol{u}_r)$ where $\boldsymbol{x}_r$ is the position where the robot is vertical, unmoving and positioned at the origin and $\boldsymbol{u}_r$ is the input with motor torque at zero.

The nonlinear system of differential equations (5) can be represented as a Taylor series expansion around the point $(\boldsymbol{x}_r, \boldsymbol{u}_r)$.

$$f(\boldsymbol{x}, \boldsymbol{u}) = f(\boldsymbol{x}_r, \boldsymbol{u}_r) + \left.\frac{\boldsymbol{df}}{\boldsymbol{dx}}\right|_{x_r} (\boldsymbol{x} - \boldsymbol{x}_r) + \left.\frac{\boldsymbol{df}}{\boldsymbol{du}}\right|_{u_r} (\boldsymbol{u} - \boldsymbol{u}_r) + \left.\frac{\boldsymbol{d^2 f}}{\boldsymbol{dx^2}}\right|_{x_r} (\boldsymbol{x} - \boldsymbol{x}_r)^2 + \left.\frac{\boldsymbol{d^2 f}}{\boldsymbol{du^2}}\right|_{u_r} (\boldsymbol{u} - \boldsymbol{u}_r)^2 + \cdots \quad (6)$$

Because $(\boldsymbol{x}_r, \boldsymbol{u}_r)$ is a fixed point, we know that $f(\boldsymbol{x}_r, \boldsymbol{u}_r) = 0$. Additionally, this approximation is only accurate in a small neighborhood around $(\boldsymbol{x}_r, \boldsymbol{u}_r)$. In this neighborhood, we can assume that the values of both $(\boldsymbol{x} - \boldsymbol{x}_r)$ and $(\boldsymbol{u} - \boldsymbol{u}_r)$ are small, so higher order terms of this series will go to zero. So a fair estimate of our system is

$$\frac{d}{dt}\boldsymbol{x} \simeq \left.\frac{\boldsymbol{df}}{\boldsymbol{dx}}\right|_{x_r} (\boldsymbol{x} - \boldsymbol{x}_r) + \left.\frac{\boldsymbol{df}}{\boldsymbol{du}}\right|_{u_r} (\boldsymbol{u} - \boldsymbol{u}_r) \quad (7)$$

where $\left.\frac{\boldsymbol{df}}{\boldsymbol{dx}}\right|_{x_r}$ is the Jacobian matrix for our system of equations $f(\boldsymbol{x}, \boldsymbol{u})$ with respect to $\boldsymbol{x}$ and $\left.\frac{\boldsymbol{df}}{\boldsymbol{du}}\right|_{u_r}$ is the Jacobian matrix with respect to $\boldsymbol{u}$. Both are then evaluated at the fixed point $(\boldsymbol{x}_r, \boldsymbol{u}_r)$. Our partial differentials with respect to $\boldsymbol{x}$ are

$$\frac{\partial f_1}{\partial x} = \frac{\partial f_1}{\partial \theta} = \frac{\partial f_1}{\partial \dot{\theta}} = \frac{\partial f_3}{\partial x} = \frac{\partial f_3}{\partial \dot{x}} = \frac{\partial f_3}{\partial \theta} = 0$$
$$\frac{\partial f_1}{\partial \dot{x}} = \frac{\partial f_3}{\partial \dot{\theta}} = 1$$

$$\begin{aligned}
\frac{\partial f_2}{\partial x} &= 0 \\
\frac{\partial f_2}{\partial \dot{x}} &= \frac{-\delta}{M + m\sin^2\theta} \\
\frac{\partial f_2}{\partial \theta} &= \frac{(M+m\sin^2\theta)(-mg(\cos^2\theta - \sin^2\theta) + mL\dot{\theta}^2\cos\theta) - (-mg\cos(\theta)\sin(\theta) + mL\dot{\theta}^2\sin(\theta) - \delta\dot{x} + u)(2m\sin\theta\cos\theta)}{(M+m\sin^2\theta)^2} \\
\frac{\partial f_2}{\partial \dot{\theta}} &= \frac{mL\sin\theta}{M+m\sin^2\theta} \cdot 2\dot{\theta}
\end{aligned}$$

$$\begin{aligned}
\frac{\partial f_4}{\partial x} &= 0 \\
\frac{\partial f_4}{\partial \dot{x}} &= \frac{\delta\cos\theta}{L(M+m\sin^2\theta)} \\
\frac{\partial f_4}{\partial \theta} &= \frac{(L(M+m\sin^2\theta))((M+m)g\cos\theta - mL\dot{\theta}^2(\cos^2\theta - \sin^2\theta) - \delta\dot{x}\sin\theta + u\sin\theta)}{L^2(M+m\sin^2\theta)^2} \\
&\quad - \frac{((m+M)g\sin(\theta) - \cos(\theta)(mL\dot{\theta}^2\sin(\theta) - \delta\dot{x}) - \cos(\theta)u)(2mL\sin\theta\cos\theta)}{L^2(M+m\sin^2\theta)^2} \\
\frac{\partial f_4}{\partial \dot{\theta}} &= -\frac{2\dot{\theta}mL\cos\theta\sin\theta}{L(M+m\sin^2\theta)}
\end{aligned}$$

The Jacobian matrix for our system of equations evaluated at $\boldsymbol{x}_r = [0\ 0\ \pi\ 0]$ and $\boldsymbol{u}_r = [0]$ gives the matrix

$$A = \begin{bmatrix} 0 & 1 & 0 & 0 \\ 0 & -\frac{\delta}{M} & -\frac{mg}{M} & 0 \\ 0 & 0 & 0 & 1 \\ 0 & -\frac{\delta}{ML} & -\frac{(m+M)g}{ML} & 0 \end{bmatrix} \tag{8}$$

Our partial differentials with respect to $\boldsymbol{u}$ are

$$\begin{aligned} \frac{\partial f_1}{\partial u} &= 0 \\ \frac{\partial f_2}{\partial u} &= \frac{1}{M + m\sin^2\theta} \\ \frac{\partial f_3}{\partial u} &= 0 \\ \frac{\partial f_4}{\partial u} &= -\frac{\cos\theta}{L(M + m\sin^2\theta)} \end{aligned}$$

Evaluating these equations at $\boldsymbol{x}_r = [0\ 0\ \pi\ 0]$ and $\boldsymbol{u}_r = [0]$ gives the matrix

$$B = \begin{bmatrix} 0 \\ \frac{1}{M} \\ 0 \\ \frac{1}{ML} \end{bmatrix} \tag{9}$$

Now we can approximate our system of equations (1), (2), (3), and (4) with the linear system

$$\frac{d}{dt}\boldsymbol{x} = A(\boldsymbol{x} - \boldsymbol{x}_r) + B(\boldsymbol{u} - \boldsymbol{u}_r). \tag{10}$$

Two Variable Model

We also built a two variable model for the balancing robot that has a state consisting of only the robot angle and angular velocity. This model works if you don't need to control the position of the robot. Working with this model was useful for testing purposes since it is a simpler system. It has the equations of motion

$$\begin{aligned} \dot{\theta} &= \dot{\theta} \\ \ddot{\theta} &= \frac{(m+M)g\sin(\theta) - \cos(\theta)(mL\dot{\theta}^2\sin(\theta)) - \cos(\theta)u}{L(M + m\sin^2\theta)} \end{aligned}$$

and the linearized matrices

$$A = \begin{bmatrix} 0 & 1 \\ -\frac{(m+M)g}{ML} & 0 \end{bmatrix}$$

$$B = \begin{bmatrix} 0 \\ \frac{1}{ML} \end{bmatrix}$$

Algorithms

Linear Quadratic Regulator

A Linear Quadratic Regulator (LQR) is a closed loop feedback system that uses a linearized model of the system based on differential equations to compute the optimal control parameters to stabilize the system

around a fixed point. Additionally, LQR control allows some level of optimization over performance requirements, allowing tradeoffs between reducing the power consumption of our motors and the stability of the system. Our LQR algorithm and accompanying simulations were coded using Python's Control Systems library `control` and the numerical and scientific computing library `numpy`.

We need our linear system (10) to be **controllable**. A system is controllable if the column space of the controllability matrix

$$\mathcal{C} = \begin{bmatrix} B & AB & A^2B & \cdots & A^{n-1}B \end{bmatrix} \tag{11}$$

has n linearly independent columns, where n is the number of variables in our state. At a high-level, controllability means that our control $B\boldsymbol{u}$ has the ability to affect all of our state variables in $\boldsymbol{x}$. This is easily tested in Python with the line

```
print(np.linalg.matrix_rank(ctrb(A, B)))
```

For our system, the controllability matrix has rank 4 confirming that our system is controllable.

LQR works by finding a matrix K that we can use to compute our control input $\boldsymbol{u}$ from the current state $\boldsymbol{x}$

$$\boldsymbol{u} = -K(\boldsymbol{x} - \boldsymbol{x}_r). \tag{12}$$

We know that the bahaviour of our system can be approximated with our linearized model (10). If we substitute (12) into (10) we get

$$A(\boldsymbol{x} - \boldsymbol{x}_r) + B(-K(\boldsymbol{x} - \boldsymbol{x}_r)) = (A - BK)(\boldsymbol{x} - \boldsymbol{x}_r)$$

In a controllable system we can select a matrix K to make the real parts of all eigenvalues of $A - BK$ have negative values. This creates a stable system.

To select our matrix K, we define two diagonal matrices Q and R. Q contains weights for the cost of deviating from the goal state and R weights the cost of our control inputs. We use the Python `control.matlab.lqr` command to pick a K that minimizes a cost function built from Q and R. If the weight in R is low relative to the weights in Q, our choice of K will favor sticking close to our goal state with minimal limits on the size of motor torque. This can cause the robot to respond agressively to changes in state and make it overly sensitive to noise. Typically, an LQR implementation will require some tuning of the weights in Q and R to meet the requirements of the system.

```
from control.matlab import lqr

# equations of motion linearized about vertical pendulum position
A = np.array([[0, 1, 0, 0],\
              [0, -d/M, -m*g/M, 0],\
              [0, 0, 0, 1],\
              [0, -d/(M*L), -(m+M)*g/(M*L), 0]])

# linearization of control matrix
B = np.array([0,1/M,0,1/(M*L)]).reshape((4,1))

Q = np.array([[1, 0, 0, 0],\
              [0, 0.01, 0, 0],\
              [0, 0, 10, 0],\
              [0, 0, 0, 0.1]])
R = 1
K = lqr(A,B,Q,R)[0]
```

A properly tuned K matrix can be used in the control loop to compute the motor torque required to respond to the current state. This value is then used to power the motors.

```
def loop_iteration():
    theta, theta_dot = imu_sensor.angle_data()
    x, x_dot = motors.position_data()
    current_state = np.array([x, x_dot, theta, theta_dot])
    u = -K@(current_state - x_r)
    motors.run(u * duty_coeff)
```

Linear Quadratic Gaussian

Need a discussion here of why we switched to LQG. What were the drawbacks of LQR approach? Like we don't have a way to measure velocity directly, we can only estimate it from the change in position over a short time window. It's better to let the model estimate it because the model knows more about the behaviour of the system.

A linear quadratic gaussian algortihm (LQG) is an extension of the LQR method with two advantages. It doesn't require sensor readings for all of the state variables and it can filter out noisy sensor data. Instead of reading the current state directly from the sensors, the algorithm estimates the current state using readings from the sensors together with predictions about the state from the linear model. State variables without sensors readings are estimated entirely.

Because we may only have measurements for some of our state values we will need a measurement model described by a matrix.

$$C = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{13}$$

The matrix C for our implentation indicates that state variables x, θ and $\dot{\theta}$ will be measured. We now need to show that our system is **observable**. Observability is a dual concept to controllability. At a high-level, a system is observable if the measured variables have the ability to predict all the state variables in $\boldsymbol{x}$. For example, the measurement model $C = [1, 0, 0, 0]$ that measures only the position x together with our A matrix (8) is observable but the measurement model $C = [0, 0, 1, 0]$ that measures only the angle θ is not. The angle θ can be constructed from the position x, but the reverse is not true. The x position cannot be constructed fully from the angle θ. I need to study this a little more until I understand it.

A system is observable with measurement model C if the rows of the observability matrix

$$\mathcal{O} = \begin{bmatrix} C \\ CA \\ CA^2 \\ \vdots \\ CA^{n-1} \end{bmatrix} \tag{14}$$

span all of $\mathbb{R}^n$. We used the control library command `control.matlab.obsv` to construct the observability matrix for our measurement model (13). Observability can be demonstrated by taking the transpose of the observability matrix and testing that it's columns are full rank.

We derive the Kalman filter matrix K_f using the `lqr` function from python control library. Our cost matrices Q and R will now represent the cost of deviating from the state, or disturbances, relative to deviations from the sensors, or noise. On a high-level this tells the model how much to trust the predictions of the model and how much to trust the readings from the sensors.

```
# C is our measurement model
C = np.array([[1, 0, 0, 0], \
              [0, 0, 1, 0], \
              [0, 0, 0, 1]])

# This is our state disturbance matrix
Q = np.eye(4)

# This is our sensor noise matrix
R = np.eye(3)

# Generate our Kalman Filter
Kf = lqr(A.transpose(), C.transpose(), Q, R)[0].transpose()
```

To build the linear state space for our Kalman filter

$$\frac{d}{dt}\boldsymbol{x} = A_{kf}(\boldsymbol{x} - \boldsymbol{x}_r) + B_{kf}(\boldsymbol{u} - \boldsymbol{u}_r) \tag{15}$$

we build the new matrices

$$A_{kf} = A - K_f C$$
$$B_{kf} = \begin{bmatrix} B & K_f \end{bmatrix}$$

Our input vector $\boldsymbol{u} = [u, x_s, \theta_s, \dot{\theta}_s]$ is our motor torque u and our sensor readings, position x_s, angle θ_s, and angular velocity $\dot{\theta}_s$

```
def loop_iteration():
    # estimate the state
    dx = A_kf@(x - x_r) + B_kf@(u - u_r)
    x = x + dx*dt

    # read sensors and compute torque
    theta, theta_dot = imu_sensor.angle_data()
    x = motors.position_data()
    torque = -md.K@(x - x_r)
    u = np.array([torque, x, theta, theta_dot])
    motors.run(u[0] * duty_coeff)
```

0.1 Simulations with Python

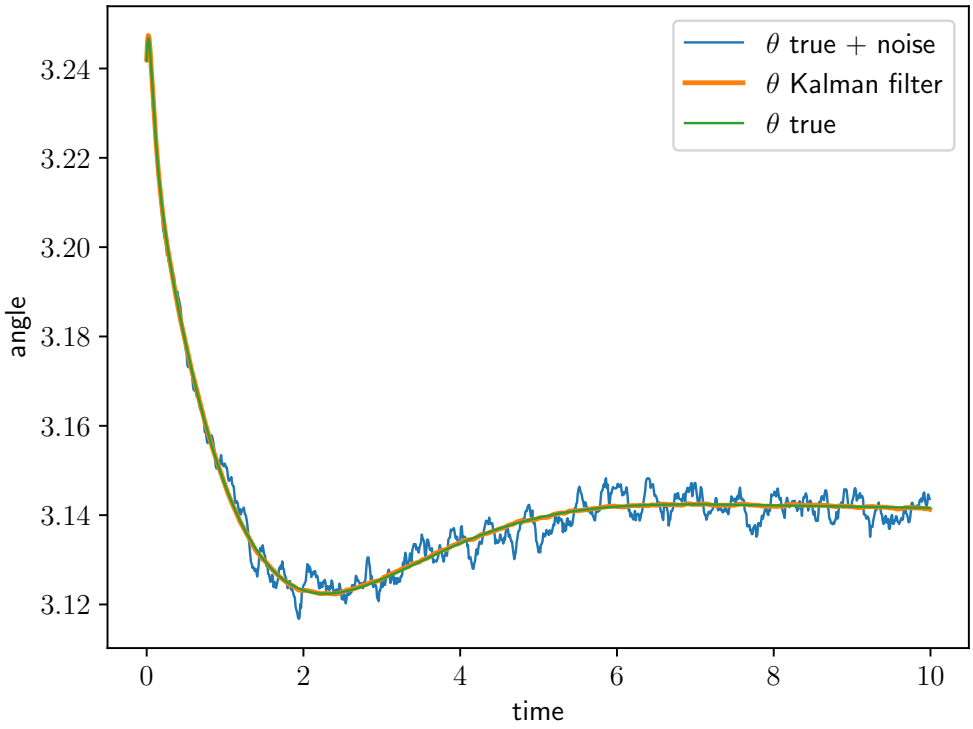

Figure 1: Simulation comparison of Kalman filter state estimation of angle with the true angle and noisy data.

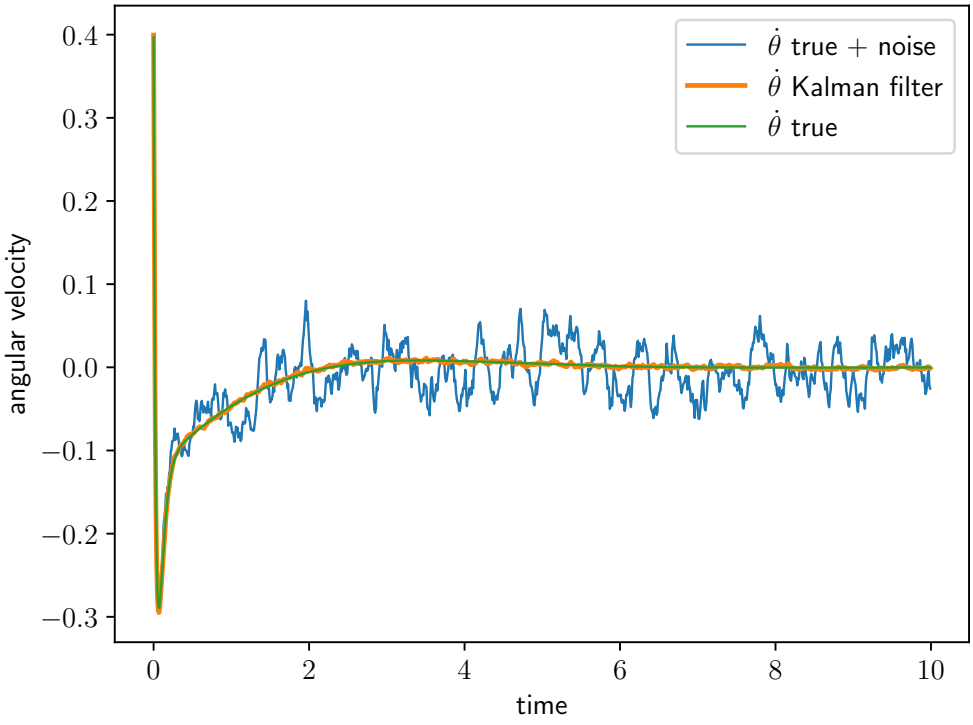

Figure 2: Simulation comparison of Kalman filter state estimation of angular velocity with the true angular velocity and noisy data.

Continuous vs. Discrete Models

Linear systems can be model as continuous or discrete. A continuous system (10) tells us how the system is changing. The state is estimated by multiplying this change in state `dx` by a time delta `dt` and then adding that to the previous state.

```
# estimate the state
dx = A@(x - x_r) + B@(u - u_r)
x = x + dx*dt
```

A discrete system estimates the current state $\boldsymbol{x}_{k+1}$ from the previous state $(\boldsymbol{x}_k, \boldsymbol{u}_k)$

$$\boldsymbol{x}_{k+1} = A_d\boldsymbol{x}_k + B_d\boldsymbol{u}_k \tag{16}$$

where $\boldsymbol{x}_{k+1} = \boldsymbol{x}_k + \Delta t$,

$$A_d = e^{A\Delta t}$$
$$B_d = \int_0^{\Delta t} e^{A\Delta t} B d\tau$$

and $e^{A\Delta t}$ is the matrix exponential. Python's `control` library can create a discrete system from a continuous one.

```
# construct discrete system from a continuous system
sys_c = ss(A, B, C, np.zeros_like(B))
sys_d = c2d(sys_c, dt, 'zoh')
```

Then the current state can be estimated from the previous state.

```
x = sys_d.A@(x - x_r) + sys_d.B@(u - u_r) + x_r
```

We simulated both methods and found they produced very similar results.

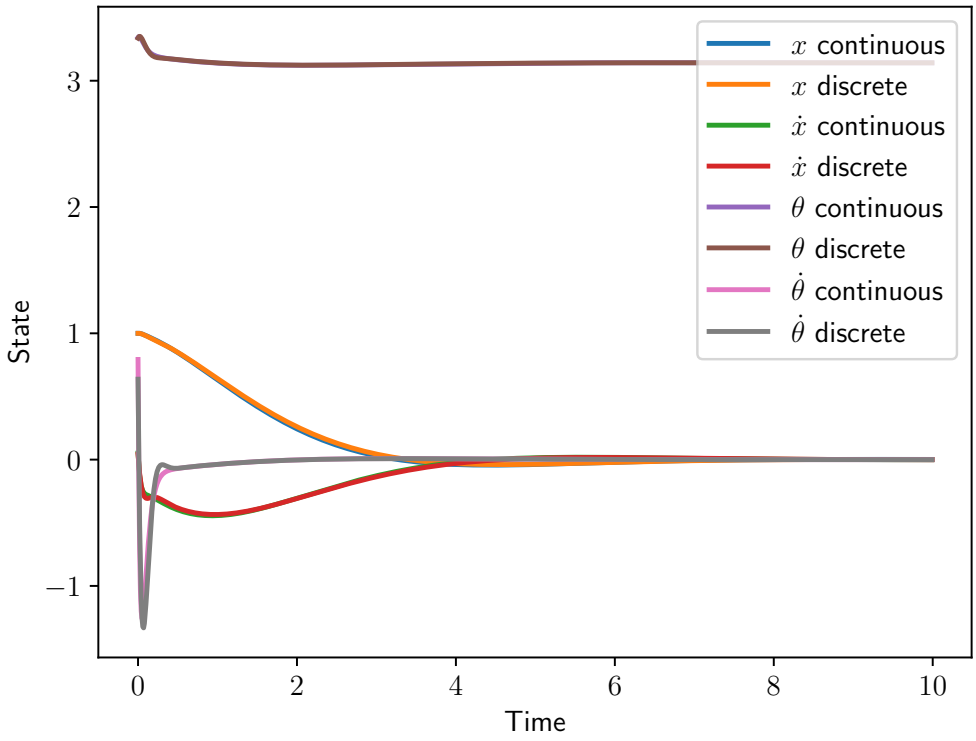

Figure 3: Simulation comparing continuous and discrete methods for state estimation.

The discrete method was slightly faster at 4.1×10^{-6} seconds per iteration with the continuous method taking 4.7×10^{-6} seconds per iteration.

Experiments

Conclusions

References

[AGH+18] Joel A E Andersson, Joris Gillis, Greg Horn, James B Rawlings, and Moritz Diehl. CasADi – A software framework for nonlinear optimization and optimal control. *Mathematical Programming Computation*, 2018.

[BK19] Steven Brunton and J. Nathan Kutx. *Data Driven Science & Engineering: Machine Learning, Dynamical Systems, and Control*. Cambridge University Press, 2019.

> A **pdf** for this textbook is available online. It has a chapter on Linear Control Theory and their primary example is the pendulum cart problem. There is accompanying code in both MatLab and Python Jupiter notebooks. Their emphasis in on modeling and simulation rather than real-world implementations. Steve Brunton also has a great series of videos that closely follow the Linear Control chapter on YouTube.

[FH20] Feriyonika Feriyonika and Asep Hidayat. Balancing control of two-wheeled robot by using linear quadratic gaussian (lqg). *Journal of Telecommunication, Electronic and Computer Engineering (JTEC)*, 12(3):55–59, Aug. 2020.

> This paper gives good real-world implementation details for an LQG implementation of a balancing robot. It uses a two state variable model that controls the robot's angle and angular velocity.

[Ker20] Justin Kerr. Self-balancing robot. https://github.com/kerrj/segway, 2020.

> This is an series of balancing robot implementations including, LQR and MPC. It is implemented in Python and the linear MPC is implementation uses the osqp solver. It also uses the ROS operating system.

[KSS24] Nils Keller, Jan Schilliger, and Yannick Schnider. Predictive control of a two-wheeled balancing robot: Lab practice control systems experiment script. https://ethz.ch/content/dam/ethz/special-interest/mavt/dynamic-systems-n-control/idsc-dam/Lectures/Control-Lab/SigiStudent.pdf, 2024.

> This is a set of lecture notes for a control theory lab experiment. It has a really well written explanation of model predictive control and it's relation to LQR and control theory in general. The experiment is to implement linear MPC control for a balancing robot.

[Rao20] Akshay S Rao. Lqr-balancebot. https://github.com/iamAkshayrao/LQR-BalanceBot/tree/master, 2020.

> This appears to be a school project for an LQR balancing robot. It provides C++ Arduino code and a very detailed write up of their model and algorithm. Their model is more detailed than the simple pendulum cart model. They do a more sophisticated analysis of moments of inertia and they model their motor circuit behaviour in their equations of motion.

[RK25] Monika Rani and Sushma S. Kamlu. Optimal lqg controller design for inverted pendulum systems using a comprehensive approach. *Scientific Reports*, 15(1):4692, Feb 2025.

This provides a comparison of PID, LQR, LQG and Model Predictive Control methods for implementing a balancing robot. It is well written and easy to follow. It uses a simple pendulum cart model. The paper focuses on simulations and provides no real-world implementation details.

[RMD17] J. Rawlings, D.Q. Mayne, and Moritz Diehl. *Model Predictive Control: Theory, Computation, and Design*. 01 2017.