Modeling and Simulation Lecture Notes - 9

July 2023

Project Sample

Step 1: Understand the Dynamics of a Drone Quadcopter A quadcopter drone has four propellers, and by varying the speed of these propellers, the drone can move up, down, sideways, and rotate about its axis. We can model the drone as a rigid body and derive its equations of motion. These are non-linear differential equations and describe the drone's position and orientation as a function of the propellers' speed.

Step 2: Linearize the Model We are interested in controlling the drone's position and orientation around a set point. Since the drone's dynamics are non-linear, it's difficult to design a control system directly. Therefore, we linearize the model around the set point.

Step 3: Design a Control System We want the drone to stay stable at a specific position and orientation. To achieve this, we can use a PID controller for each of the states we want to control: the x, y, and z positions, and the pitch, roll, and yaw angles.

Step 4: Simulate the System Once we have the controllers, we can simulate the system. We can create a Python script to solve the drone's differential equations using the control inputs from the PID controllers.

We will use Python's built-in functions for differential equation solving. We can also add some random noise to the drone's position and orientation to simulate real-world disturbances.

Step 5: Visualize the Results After simulating the system, we can visualize the drone's position and orientation over time using Python's Matplotlib library. This allows us to see how well our controllers stabilize the drone.

Step 6: Fine-tune the Controllers Based on the simulation results, we can fine-tune the PID controllers' parameters to improve the drone's stability.

This project provides a comprehensive understanding of how to model, simulate, and control a real-world system. It incorporates many of the concepts learned in the course, and provides practical experience in system control and Python programming.

Quadcopter Drone Dynamics

A quadcopter drone has four propellers, each providing an upward thrust force when they spin. The drone can move or rotate by varying these thrust forces. We consider a simplified model, focusing on vertical motion (up and down) and rotation around the vertical axis (yaw).

Vertical motion

The total upward force is the sum of the forces from each propeller. This force counteracts gravity and accelerates the drone up or down. Denote the mass of the drone as m, the gravity as g, the force from each propeller as f_i, and the vertical acceleration as $\ddot{z}$. Newton's second law gives us:

$$m\ddot{z} = \sum_{i=1}^{4} f_i - mg \tag{1}$$

The force from each propeller is proportional to the square of its speed (denoted as ω_i):

$$f_i = k\omega_i^2 \tag{2}$$

where k is a proportionality constant. Substituting this into the previous equation gives:

$$m\ddot{z} = k\sum_{i=1}^{4} \omega_i^2 - mg \tag{3}$$

Yaw rotation

The drone rotates around the vertical axis when there's a net torque about that axis. The torque from each propeller is proportional to its speed, and the direction depends on whether the propeller spins clockwise or counterclockwise. Denote the moment of inertia about the vertical axis as I, the torque from each propeller as τ_i, and the angular acceleration as $\ddot{\psi}$, we can write:

$$I\ddot{\psi} = \sum_{i=1}^{4} \tau_i \tag{4}$$

The torque from each propeller is given by:

$$\tau_i = lk'\omega_i \tag{5}$$

where l is the distance from the propeller to the center of the drone, k' is a proportionality constant, and the sign depends on the direction of the propeller (plus for counterclockwise, minus for clockwise). Substituting this into the previous equation gives:

$$I\ddot{\psi} = lk'\left(\omega_1^2 - \omega_2^2 + \omega_3^2 - \omega_4^2\right) \tag{6}$$

These are the basic equations of motion for our simplified drone model. They are non-linear differential equations that relate the propellers' speeds (ω_i) to the drone's vertical acceleration ($\ddot{z}$) and yaw angular acceleration ($\ddot{\psi}$).

System Linearization

The equations we have derived are nonlinear, and the methods of control design we will apply require a linear model. Therefore, we need to linearize these equations about the equilibrium point. In this case, the equilibrium point is when the drone hovers at a fixed height and orientation.

The equilibrium point of the vertical motion equation is when the upward force equals the gravitational force. Therefore, the sum of all propeller forces f_i equals mg. Similarly, the equilibrium point of the yaw motion equation is when the net torque is zero, which means the total force of the clockwise spinning propellers equals the total force of the counterclockwise spinning propellers.

The linearization process involves taking the first-order Taylor series approximation of the nonlinear equations about the equilibrium point. The process yields two linear differential equations that describe the small perturbations of the vertical position and yaw angle about the equilibrium point.

For the purposes of this demonstration, I'll leave the linearization process as an exercise. It involves a bit of calculus, and you can find the detailed steps in many textbooks and online resources. The linearized system equations will be in the form of $\dot{x} = Ax + Bu$ where x is the state vector, u is the control input, A is the system matrix, and B is the input matrix.

Once the system is linearized, we can use the methods of linear control theory to design a controller that stabilizes the drone in the hover position and allows it to follow a desired height and yaw angle. In the next step, we will perform the controller design.

Consider a general nonlinear system defined by the differential equations:

$$\frac{dx}{dt} = f(x, u)$$

where $x \in \mathbf{R}^n$ is the state vector, $u \in \mathbf{R}^m$ is the control input vector, and $f : \mathbf{R}^n \times \mathbf{R}^m \to \mathbf{R}^n$ is a nonlinear function.

To linearize the system about an equilibrium point (x_e, u_e), we take the first-order Taylor series approximation of f about (x_e, u_e):

$$f(x, u) \approx f(x_e, u_e) + \frac{\partial f}{\partial x}(x_e, u_e)(x - x_e) + \frac{\partial f}{\partial u}(x_e, u_e)(u - u_e)$$

At the equilibrium point, $f(x_e, u_e) = 0$. Defining $x' = x - x_e$ and $u' = u - u_e$ as the small perturbations about the equilibrium, the above equation simplifies to:

$$\frac{dx'}{dt} = \frac{\partial f}{\partial x}(x_e, u_e)x' + \frac{\partial f}{\partial u}(x_e, u_e)u'$$

This is a linear system of the form $\dot{x} = Ax + Bu$, where $A = \frac{\partial f}{\partial x}(x_e, u_e)$ and $B = \frac{\partial f}{\partial u}(x_e, u_e)$.

The matrices A and B can be obtained by taking the partial derivatives of f with respect to x and u and evaluating them at the equilibrium point. The specific procedure will depend on the form of the nonlinear system equations.

This linearized system can then be used for control design, as we will see in the next step.

Step 2: Linearization

A nonlinear system can often be approximated by a linear system for small deviations from an equilibrium point. This process is called linearization. For the cart-pole system, the equilibrium points are when the pole is in its upright position (either pointing straight up or straight down), and the cart is stationary.

Let's denote the state vector as $x = [\text{position, velocity, angle, angular rate}]^T \in \mathbf{R}^4$ and the control input as $u \in \mathbf{R}$ (the force on the cart). The system dynamics can then be written as $\dot{x} = f(x, u)$ where f is a nonlinear function.

Around an equilibrium point, we can approximate the system dynamics as a linear time-invariant (LTI) system of the form $\dot{x} = Ax + Bu$, where $A \in \mathbf{R}^{4\times4}$ is the state matrix, and $B \in \mathbf{R}^4$ is the input matrix. The matrices A and B are obtained by taking the Jacobian of f with respect to x and u, respectively, evaluated at the equilibrium point.

Step 3: Control Design

After obtaining the linearized system, we can now proceed to design the control law. For this example, let's use a full-state feedback control law. The full-state feedback control law is given by:

$$u' = -Kx'$$

where K is the gain matrix to be determined, x' is the state vector, and u' is the control input. The purpose of the control law is to drive the state x' to zero, i.e., to bring the system to the desired equilibrium point. The gain matrix K is chosen such that the closed-loop system (the system under the control law) has desirable characteristics, such as stability and fast response.

The selection of K can be achieved by several methods, such as pole placement, Linear Quadratic Regulator (LQR), or others. In the pole placement method, the gain matrix K is selected such that the eigenvalues (or poles) of the closed-loop system matrix $(A - BK)$ are at desired locations in the complex plane. In the LQR method, K is selected to minimize a cost function that trades off between control effort and state deviation.

These methods are available in control system libraries in Python. For example, the control library provides the function place for pole placement and lqr for the LQR method.

In the next step, we will simulate the system under the control law to evaluate its performance.

Step 4: Design of the LQR Controller

Linear quadratic regulator (LQR) is a type of optimal control method that minimizes the cost function, which is a combination of the state and the control input. The basic idea behind the LQR controller is to find a control law that minimizes this cost function.

The control law in state-space form is $u = -Kx$ where K is the gain matrix that we want to find. To find K, we need to solve the Algebraic Riccati Equation (ARE):

$$0 = A^T P + PA - PBR^{-1}B^T P + Q$$

where $P \in \mathbf{R}^{4\times4}$, $Q \in \mathbf{R}^{4\times4}$ is the state cost matrix, and $R \in \mathbf{R}$ is the control cost scalar.

```
# Define the state cost matrix Q and the control cost scalar R
Q = np.diag([10, 10, 1, 1])
R = 1
```

```
# Compute the LQR gain
K, S, E = control.lqr(A, B, Q, R)

print("The LQR gain K is ", K)
```

This code sets the state cost matrix Q and the control cost scalar R, and then uses the control.lqr function from the Python control library to compute the LQR gain K. The function also returns the solution S of the Algebraic Riccati Equation and the closed-loop poles E. The computed LQR gain K is then printed out.

Step 5: Simulation of the Closed-loop System

With the LQR controller designed, the next step is to simulate the closed-loop system and visualize the response to different initial conditions. The closed-loop system can be formed by subtracting the product of the gain matrix K and the state vector x from the system dynamics.

The state-space model of the closed-loop system is represented as:

$$\dot{x} = (A - BK)x$$

We will use the step response to visualize the behavior of the system. It is also important to compare the system response with different initial conditions to validate the robustness of the controller.

```
# Form the closed-loop system
sys_cl = control.StateSpace(A - B @ K, B, C, D)

# Simulate the step response
T, yout = control.step_response(sys_cl)

# Plot the response
plt.figure()
plt.plot(T.T, yout.T)
plt.xlabel('Time (s)')
plt.ylabel('Angle (rad)')
plt.title('Step Response with LQR Control')
plt.grid(True)
plt.show()
```

In the above Python code, the closed-loop system is formed by subtracting the product of the gain matrix K and the state vector x from the system dynamics A and B. The step response of the closed-loop system is then simulated and plotted over time. The step response represents the output of the system when the input changes from zero to one at time zero. This response is often used to analyze the stability and performance of control systems.

Step 6: Analysis and Discussion of the Results

After simulating the closed-loop system, the final step is to analyze and discuss the results. This should include:

- Observations about the behavior of the system with the LQR controller applied.
- Comparisons of the controlled system's performance with different initial conditions.
- Discussion of how changes in the weighting matrices Q and R affect the system's performance.
- Discussion of the limitations of the LQR controller and potential improvements.

These discussions should be grounded in the theoretical understanding of the LQR control strategy and the specific application at hand.

The overall goal of the project is not only to design a controller but also to understand the trade-offs involved in control design, such as stability versus performance, and robustness versus accuracy.

Implementing in Python

Here's a simple code that models a drone flying in 2D space using basic physics principles:

```
import numpy as np
import matplotlib.pyplot as plt

# Drone parameters
mass = 1.0  # kg
g = 9.81  # m/s^2

# Time parameters
T = 10.0  # total time to simulate
dt = 0.01  # time step
N = int(T / dt) + 1  # number of time-steps
t = np.linspace(0.0, T, num=N)  # time grid

# Initial conditions
z0 = 0.0  # altitude
b0 = 0.0  # upward velocity resulting from thrust
zt = 100.0  # target altitude
b_t = 10.0  # initial upward velocity
```

```
# Initial velocity and state
u = np.array([z0, b0])

# Initialize an array to hold the changing elevation values
z = np.zeros(N)
z[0] = z0

# Simulation parameters
alpha = 0.02
gamma = g

# Simulation
for i in range(1, N):
    rhs = np.array([u[1], g - 1/mass * (u[1] + alpha * (u[1] - b_t))])
    u = u + dt * rhs
    z[i] = u[0]  # store the elevation at this time-step

# Visualization
plt.figure(figsize=(9, 4))
plt.grid(True)
plt.title('2D Drone flight simulation')
plt.xlabel('Time [s]')
plt.ylabel('Altitude [m]')
plt.plot(t, z, 'b-', linewidth=2, label='Elevation')
plt.legend()
plt.show()
```

This code models a drone flying in 2D space, starting from an initial altitude of 0m. The drone's thrust results in an initial upward velocity. The force of gravity opposes this thrust and pulls the drone back towards the ground.

This is a simplified model and does not include factors like air resistance, rotational dynamics, stability control, etc. Also, it assumes the drone is a point mass, which is an approximation.

This script is a starting point and you can build on it by adding more complexity to the model like wind disturbances, rotational dynamics, etc.

For control, you can implement a simple PID controller to control the altitude based on the error between the desired and current altitude. However, for more realistic drone control, you would need to look into more advanced control methods like Linear Quadratic Regulator (LQR) or Model Predictive Control (MPC).

Let's add a PID controller to our drone simulation.

```
class PIDController:
```

```
    def __init__(self, Kp=0.0, Ki=0.0, Kd=0.0, dt=0.01):
        self.Kp = Kp
        self.Ki = Ki
        self.Kd = Kd
        self.dt = dt
        self.integral = 0.0
        self.prev_error = 0.0

    def update(self, error):
        derivative = (error - self.prev_error) / self.dt
        self.integral += error * self.dt
        self.prev_error = error
        return self.Kp * error + self.Ki * self.integral + self.Kd * derivative

# PID controller parameters
Kp = 10.0
Ki = 0.0
Kd = 20.0

# Create the PID controller
pid = PIDController(Kp, Ki, Kd, dt)

# Simulation with PID control
for i in range(1, N):
    error = zt - u[0]
    u[1] = pid.update(error)
    rhs = np.array([u[1], g - 1/mass * u[1]])
    u = u + dt * rhs
    z[i] = u[0]

# Visualization
plt.figure(figsize=(9, 4))
plt.grid(True)
plt.title('2D Drone flight simulation with PID control')
plt.xlabel('Time [s]')
plt.ylabel('Altitude [m]')
plt.plot(t, z, 'b-', linewidth=2, label='Elevation with PID control')
plt.legend()
plt.show()
```

In the code above, we first define a simple PID controller class. The update method calculates the PID output using the current error (difference between target and actual output). We then update our simulation loop to use the PID controller's output to control the drone's thrust. We plot the result, and you'll see the drone is now able to maintain the desired altitude.

Please note that the gains (Kp, Ki, Kd) used here are arbitrary and may not provide the best performance. Tuning the gains of a PID controller to get the desired performance is a bit of an art and can be a challenging task. There are methods to tune these gains, such as Ziegler–Nichols method, but those are beyond the scope of this conversation.

Remember, this is a simple 2D simulation and does not capture all the complexities of drone flight, which is a 3D problem and involves rotational dynamics as well. But it should give you a good starting point for understanding how control systems work for drones.

* * *

We can implement a **Linear Quadratic Regulator** (LQR) instead of a PID controller. For our 2D drone flight, we will need to linearize our equations of motion and then solve the LQR problem.

Let's start by linearizing the system dynamics around the hover state:

$$\dot{z} = v \quad \dot{v} = g - \frac{1}{m}u \tag{7}$$

The linearized system dynamics can be written in the form $\dot{x} = Ax + Bu$, where $x = [z, v]^T$ is the state vector and u is the control input. The matrices A and B are given by

$$A = \begin{bmatrix} 0 & 1 \\ 0 & 0 \end{bmatrix}, \quad B = \begin{bmatrix} 0 \\ -\frac{1}{m} \end{bmatrix} \tag{8}$$

To design an LQR controller, we need to define the cost matrices Q and R. For simplicity, we can choose $Q = I$ and $R = 1$. The LQR problem then becomes to find the gain matrix K that minimizes the cost function

$$J = \int (x^T Q x + u^T R u) dt \tag{9}$$

Now let's implement this in Python:

```
import control
import numpy as np
from scipy import signal
import matplotlib.pyplot as plt

# System parameters
```

```
g = 9.81    # gravity constant, m/s^2
m = 1.0     # drone mass, kg

# Time discretization
dt = 0.01   # time step, s
N = 3000    # number of time steps
t = np.linspace(0, dt*(N-1), N)

# Initial conditions
z0 = 0.0    # initial altitude, m
v0 = 0.0    # initial velocity, m/s

# Desired altitude
zt = 1.0    # target altitude, m

# System dynamics matrices
A = np.array([[0, 1], [0, 0]])
B = np.array([[0], [-1/m]])

# Cost matrices
Q = np.eye(2)
R = np.array([[1]])

# LQR control gain
K, _, _ = control.lqr(A, B, Q, R)

# State vector
x = np.array([z0 - zt, v0])

# Simulation
z = np.zeros(N)
v = np.zeros(N)
z[0] = z0
v[0] = v0
for i in range(1, N):
    u = -K @ x
    dx = A @ x + B @ u
    x = x + dt * dx
    z[i] = x[0] + zt
    v[i] = x[1]

# Visualization
plt.figure(figsize=(9, 4))
plt.grid(True)
plt.title('2D Drone flight simulation with LQR control')
plt.xlabel('Time [s]')
```

```
plt.ylabel('Altitude [m]')
plt.plot(t, z, 'b-', linewidth=2, label='Elevation with LQR control')
plt.legend()
plt.show()
```

This code implements the LQR controller for the 2D drone flight. Note that the control.lqr function computes the optimal control gain K. The rest of the code is the same as before, but now we use the LQR controller instead of the PID controller. You should see that the drone reaches the desired altitude and stays there, just like with the PID controller.

Inverted Pendulum Example:

We will follow the steps previously outlined to implement this project. In the first step, we formulated the system dynamics in state-space representation. Now, let's move to step 2 and choose our state variables.

We will take the following variables:

The angle of the pendulum from the vertical (θ) The rate of change of the angle ($d\theta/dt$) The horizontal position of the cart (x) The rate of change of the position (dx/dt) Now let's implement this in Python:

```
import numpy as np
import control
import matplotlib.pyplot as plt

# Define system parameters
m = 0.5    # mass of pendulum
M = 1.0    # mass of cart
L = 0.5    # length of pendulum
g = 9.81   # acceleration due to gravity

# Formulate the system dynamics
A = np.array([[0, 1, 0, 0],
              [0, 0, -m*g/M, 0],
              [0, 0, 0, 1],
              [0, 0, g*(M+m)/(M*L), 0]])

B = np.array([[0], [1/M], [0], [-1/(M*L)]])

C = np.eye(4)
```

```
D = np.zeros((4,1))

# Define the state-space system
sys = control.StateSpace(A, B, C, D)
```

This code sets up our state-space system for the inverted pendulum. We have defined the system parameters, formulated the system dynamics, and finally defined the state-space system using the control library in Python. The next step would be designing the LQR controller and simulating the system, but for now, let's check that the state-space system is correctly defined.

We have successfully defined our system, let's move on to step 3: designing the Linear Quadratic Regulator (LQR) controller. The LQR is an optimal control in which we try to minimize the cost function defined as $J = \int_0^\infty (x^T Q x + u^T R u) dt$, where Q and R are the cost matrices. Q determines the weight of the state error in the cost function, and R determines the weight of the control effort.

In Python, we can use the control.lqr() function to calculate the optimal control gain K. Let's implement this:

```
# Define the cost matrices
Q = np.array([[1, 0, 0, 0],
              [0, 1, 0, 0],
              [0, 0, 10, 0],
              [0, 0, 0, 100]])

R = np.array([[1]])

# Compute the optimal control gain K
K, S, E = control.lqr(sys, Q, R)
```

Here, we have defined the cost matrices Q and R, and computed the optimal control gain K using the control.lqr() function. The control.lqr() function also returns the solution S of the associated Algebraic Riccati equation, and the closed-loop eigenvalues of the system E, but we won't need these for our simulation.

You can adjust the values in Q and R to change the performance of the controller. In this case, we are placing a higher weight on the position of the cart (third state variable) and the rate of change of the angle of the pendulum (fourth state variable), and a lower weight on the control effort.

Now, let's move on to step 4: simulating the system.

```
import scipy.integrate as spi
# Define the physical parameters
m_c = 0.5  # mass of the cart
m_p = 0.2  # mass of the pendulum
l = 0.6    # length of the pendulum
g = 9.81   # gravitational acceleration
# Define the system dynamics
def system_dynamics(x, t, K):
    # State feedback control law
    u = -K @ x
    # System dynamics
    dx = np.zeros_like(x)
    dx[0] = x[1]
    dx[1] = (u - m_p*l*np.sin(x[2])*x[3]**2 -
    m_p*g*np.sin(x[2])*np.cos(x[2])) / (m_c + m_p*(1-np.cos(x[2])**2))
    dx[2] = x[3]
    dx[3] = (u*np.cos(x[2]) + m_p*l*x[3]**2*np.cos(x[2])*np.sin(x[2]) +
    (m_c+m_p)*g*np.sin(x[2])) / (l*(m_c + m_p*(1-np.cos(x[2])**2)))
    return dx

# Initial condition
X0 = np.array([0, 0, 0.1, 0])

# Simulate the system response using odeint
xout = spi.odeint(system_dynamics, X0, t, args=(K,))

# Plot the system response
plt.figure(figsize=(12,8))
plt.plot(t, xout[:, 0])
plt.plot(t, xout[:, 2])
plt.legend(["Cart position (m)", "Pendulum angle (rad)"])
plt.xlabel("Time (s)")
plt.title("System Response with LQR Control")
plt.grid()
plt.show()
```

Here, system_dynamics is a function that represents the differential equations of the system. spi.odeint is a function from the scipy.integrate module that integrates a system of ordinary differential equations. We use odeint to simulate the system response given the initial condition X0 and the time vector t. The function odeint automatically handles the computation of the state x at each time step. We then plot the cart position and pendulum angle as functions of time.

In this code, the parameters m_c, m_p, l, and g represent the mass of the cart,

the mass of the pendulum, the length of the pendulum, and the gravitational acceleration, respectively.

Step 7: Analyzing and Discussing the Results

In this step, we take a close look at the plots we generated and discuss the behavior of the system. Let's analyze the results from the simulation:

1. **Cart position:** From the plot of the cart position over time, we can observe how the cart moves as a result of the controller's actions. The cart should initially move to balance the pendulum and then come to a stop at the desired position.
2. **Pendulum angle:** The plot of the pendulum angle over time shows how the angle changes. The pendulum should initially swing up from its initial angle, then oscillate around the upright position before eventually stabilizing due to the controller's actions.

After running the simulation, your results will give you the details to write these observations. Remember to discuss whether the system behavior aligns with the expected behavior and the design requirements. If the system isn't performing as expected, discuss what factors could be contributing to the discrepancy and suggest improvements or further work.

For this project, you can also discuss:

- The challenges of balancing an inverted pendulum and why it's a benchmark problem in control theory.
- The advantages and limitations of the LQR method used for controller design.
- How the choice of weighting matrices Q and R in the LQR design affects the system behavior.

And there you go! That's a full process of modeling, simulating, and controlling a system, from literature review to controller design to Python implementation and results analysis. You can apply this process to many other systems and control problems, adapting the specific steps to suit the system and the control objectives.

This comprehensive project covers many aspects of the course and provides a practical, hands-on experience with control system design and Python programming. The skills you've learned and practiced here - understanding system dynamics, designing controllers, programming in Python, analyzing results - are fundamental skills in control engineering and will be valuable in your further studies or career.