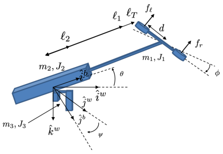
Hummingbird Manual

Introduction to Feedback Control

USING DESIGN STUDIES



RANDAL W. BEARD
TIMOTHY W. MCLAIN
WITH MARK PETERSEN
& DEVON MORRIS

Updated January 15, 2020

TOC

Contents

TOC

1 Introduction

This objective of this document is to provide a detailed introduction to the BYU hummingbird including laboratory assignments that are intended to complement an introductory course in feedback control. The hummingbird is a three degree-of-freedom helicopter with complicated nonlinear dynamics, but the dynamics are amenable to linear analysis and design. The hummingbird has been designed with encoders on each joint that allow full-state feedback, but it is also equipped with an IMU (rate gyros and accelerometers). The sensor configuration allows estimated state information obtained from the IMU and camera to be compared to truth data obtained from the encoders, thereby decoupling the state feedback and state estimation problems, and allowing each to be tuned and analyzed independently. We believe that this decoupling is essential to help students understand the subtleties in feedback and estimator design.

The design process for control system is shown in Figure 1-1. For the hummingbird, the system to be controlled is a physical system with actuators (motors) and sensors (encoders, IMU). The first step in the design process is to model the physical system using nonlinear differential equations. While approximations and simplifications will be necessary at this step, the hope is to capture in mathematics, all of the important characteristics of the physical system. The mathematical model of the system is usually obtained using Euler-Lagrange method, Newton's method, or other methods from physics and chemistry. The resulting model is usually high order and too complex for design. However, this model will be used to test later designs in simulation and is therefore called the Simulation Model, as shown in Figure 1-1. To facilitate design, the simulation model is simplified and usually linearized to create lower-order (and usually linear) design models. For any physical system, there may be multiple design models that capture certain aspects of the design process. For the hummingbird, we will decompose the motion into longitudinal (pitching) motion and lateral (rolling and yawing) motion and we will have different design models for each type of motion. As shown in Figure 1-1, the design models are used to develop control designs. The control

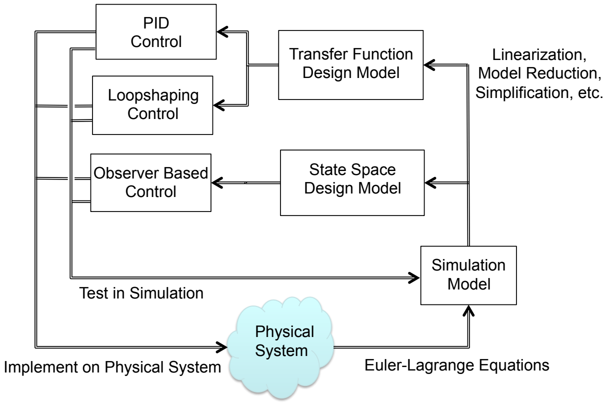


Figure 1-1: The design process. Using physics models the physical system is modeled with nonlinear differential equations resulting in the simulation model. The simulation model is simplified to create design models which are used for the control design. The control design is then tested and debugged in simulation and finally implemented on the physical system.

designs are then tested against the simulation, which sometimes requires that the design models be modified or enhanced if they have not captured the essential features of the system. After the control designs have been thoroughly tested in simulation, they are implemented on the physical system and are again tested and debugged, sometimes requiring that the simulation model be modified to more closely match the physical system.

The physical system or "hardware" interfaces with our models and controllers using the ROS framework. ROS stands for Robot Operating System and is used throughout academia and business and has a large user base. We have chosen to use ROS for our hummingbird hardware since it is modular, fast, and highly flexible. Students will learn the basics of ROS in this course and will prepare themselves to design more complex systems using ROS in the future. The first preliminary lab will cover learning the basics of ROS and setting up the hummingbird workspace. These details are found in c. It will be important for students to follow all the directions found in this section as it will ensure that the hummingbird hardware will run in a safe manner.

The purpose of these notes, and the associated laboratory assignments, is to walk you through the design cycle for the hummingbird. We will consider three different design methodologies: successive loop closure, loopshaping, and observer-based control. We believe that visualizing the physical system is an important part of the simulation model. The simulation model has been implemented as a node in the ROS framework and is found in the Gitlab repository. If students are interested in learning how to animate other control systems they can visit rviz tutorials.

These animations need to behave in a dynamically accurate manner. The equations of motions of the hummingbird are derived using the Euler-Lagrange method in Chapters 2 and 3. Implementing the equations of motion in Python is requires knowledge of numerical integration of odes, which is typically unfamiliar to students. Therefore, a simulator template node has been created for the students. This ROS node will handle the integration of the equations of motion of the hummingbird and students will be required to insert the equations of motion. Understanding the equations of motion and implementing them in the ROS node comprises labs 2 and 3 A full python based approach is described in Appendix a. Support files for this approach are found on the control book website. Once the simulation model is complete, design models are developed in Chapters 4, 5, and 6. Since the design models are in terms of force and torque, the force and torque need to be converted to a left and right pulse width modulation command.

Developing physically realizable design specifications is an important part of the design process and described in Chapter 7. The remainder of the chapters introduce different controllers that control the hummingbird according to the design specifications.

This book is used in conjunction with the Control Book website. The website contains many useful files such as the Design Study solutions for the Single Link Robot Arm, Inverted Pendulum, and Satellite Altitude Control.

W Hummingbird Lab Assignments

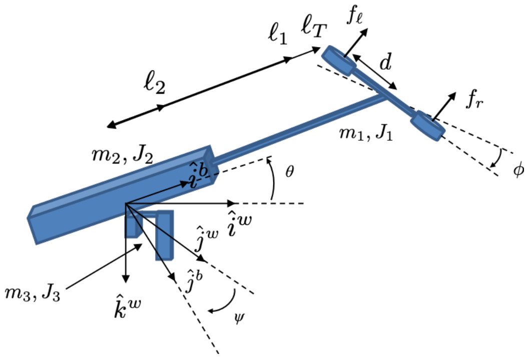


Figure 1-2: The hummingbird system

A descripton of the hummingbird can be found in Appendix d

Laboratory Assignments:

W.1 Intro to hummingbird.
W.2 Kinetic energy.
W.3 Equations of Motion.
W.4 Linearize equations of motion.
W.5 Transfer function model.
W.6 State space model.
W.7 Pole placement using PD.
W.8 Second order design.
W.9 Integrators and system type.
W.10 Digital PID.
W.11 Full state feedback.
W.12 Full state with integrator.
W.13 Observer based control.
W.14 Disturbance observer.
W.15 Frequency Response.
W.16 Loop gain.
W.17 Stability margins.
W.18 Loopshaping design.

Lab Assignment W.1

1. Work through the whirlybird ROS tutorial listed in Appendix c

Part I

Simulation Models

TOC

2 Kinetic Energy of Mechanical Systems

The objective of this chapter is to develop the equations that describe the kinetic energy of the hummingbird. The physical parameters of the hummingbird and a description of each can be found in Appendix d.

2.1 Kinetic Energy

A simplified diagram of the hummingbird is shown in Figure 2-1,

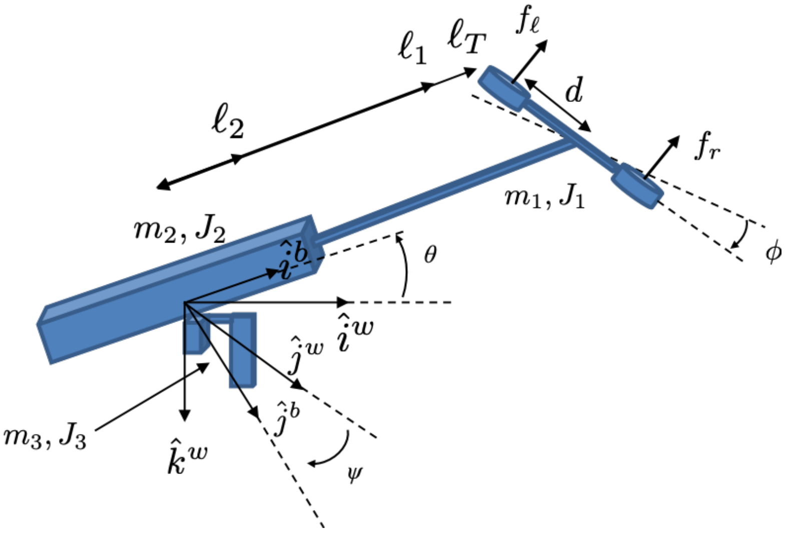


Figure 2-1: A simplified diagram of the hummingbird.

Since there are three different objects that move with independently of each other, we will model the hummingbird as consisting of three masses, as shown in Figure 2-1. The first mass, labeled m_1, J_1 in Figure 2-1 is the final rod and the head of the hummingbird, including the motors, propellers, and shrouds. The second mass, labeled m_2, J_2 in Figure 2-1 is the main body of the hummingbird, the third mass labeled m_3, J_3 in Figure 2-1 consists of the metal block that connects the main body and also the control board.

The world coordinate frame is defined to be the point where the third body spins about the z^w-axis. The $x^w - y^w$ plane is horizontal to the ground, and the world z^w-axis points toward the center of the earth. The body-fixed coordinate frame is defined so that center of the frame corresponds to the center of the world frame, and if an observer were sitting at the center and facing along the main shaft of the hummingbird, the x^b-axis would point forward, along the shaft of the hummingbird, the y^b axis would point to the right, and the z^b axis would point down.

2.1.1 Rotation matrices and transformations between coordinate frames (Bonus Material)

To fully understand the derivation of the equations of motion for the hummingbird requires an understanding of transformations between coordinate frames. One coordinate frame is transformed into another through two basic operations: rotation and translation. In this section we will describe rotation matrices and their use in transforming between coordinate frames, and we will also describes the specific coordinate frames used for the hummingbird.

2.2 Rotation Matrices

We begin by considering the two coordinate frames shown in Figure 2-2. The

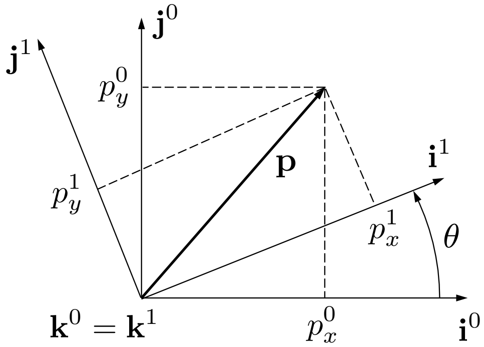


Figure 2-2: Rotation in 2D

vector $\mathbf{p}$ can be expressed in both the $\mathcal{F}^0$ frame (specified by $(\hat{i}^0, \hat{j}^0, \hat{k}^0)$) and in the $\mathcal{F}^1$ frame (specified by $(\hat{\imath}^1, \hat{\jmath}^1, \hat{k}^1)$). In the $\mathcal{F}^0$ frame we have

$$\mathbf{p} = p_x^0 \hat{\imath}^0 + p_y^0 \hat{\jmath}^0 + p_z^0 \hat{k}^0.$$

Alternatively in the $\mathcal{F}^1$ frame we have

$$\mathbf{p} = p_x^1 \hat{\imath}^1 + p_y^1 \hat{\jmath}^1 + p_z^1 \hat{k}^1.$$

The vector sets $(\hat{\imath}^0, \hat{\jmath}^0, \hat{k}^0)$ and $(\hat{\imath}^1, \hat{\jmath}^1, \hat{k}^1)$ are each mutually perpendicular sets of unit basis vectors.

Setting these two expressions equal to each other gives

$$p_x^1 \hat{\imath}^1 + p_y^1 \hat{\jmath}^1 + p_z^1 \hat{k}^1 = p_x^0 \hat{\imath}^0 + p_y^0 \hat{\jmath}^0 + p_z^0 \hat{k}^0.$$

Taking the dot product of both sides with $\hat{\imath}^1$, $\hat{\jmath}^1$, and $\hat{k}^1$ respectively, and stacking the result into matrix form gives

$$\mathbf{p}^1 \triangleq \begin{pmatrix} p_x^1 \\ p_y^1 \\ p_z^1 \end{pmatrix} = \begin{pmatrix} \hat{\imath}^1 \cdot \hat{\imath}^0 & \hat{\imath}^1 \cdot \hat{\jmath}^0 & \hat{\imath}^1 \cdot \hat{k}^0 \\ \hat{\jmath}^1 \cdot \hat{\imath}^0 & \hat{\jmath}^1 \cdot \hat{\jmath}^0 & \hat{\jmath}^1 \cdot \hat{k}^0 \\ \hat{k}^1 \cdot \hat{\imath}^0 & \hat{k}^1 \cdot \hat{\jmath}^0 & \hat{k}^1 \cdot \hat{k}^0 \end{pmatrix} \begin{pmatrix} p_x^0 \\ p_y^0 \\ p_z^0 \end{pmatrix}.$$

From the geometry of Figure 2-2 we get

$$\mathbf{p}^1 = \mathcal{R}_z(\psi)\mathbf{p}^0, \tag{2.1}$$

where

$$\mathcal{R}_z(\psi) \triangleq \begin{pmatrix} \cos\psi & \sin\psi & 0 \\ -\sin\psi & \cos\psi & 0 \\ 0 & 0 & 1 \end{pmatrix}. \tag{2.2}$$

The notation $\mathcal{R}_z$ is used to denote a rotation from coordinate frame $\mathcal{F}^0$ to coordinate frame $\mathcal{F}^1$, where the rotation is about the z axis of $\mathcal{F}^0$.

Proceeding in a similar way, a right-handed rotation of the coordinate system about the y-axis of $\mathcal{F}^0$ by an angle of θ gives

$$\mathcal{R}_y(\theta) \triangleq \begin{pmatrix} \cos\theta & 0 & -\sin\theta \\ 0 & 1 & 0 \\ \sin\theta & 0 & \cos\theta \end{pmatrix}, \tag{2.3}$$

and a right-handed rotation of the coordinate system about the x-axis of $\mathcal{F}^0$ by an angle of ϕ is

$$\mathcal{R}_x(\phi) \triangleq \begin{pmatrix} 1 & 0 & 0 \\ 0 & \cos\phi & \sin\phi \\ 0 & -\sin\phi & \cos\phi \end{pmatrix}. \tag{2.4}$$

The negative sign on the sine term appears above the line with only ones and zeros.

The matrices $\mathcal{R}_*$ in the above equations are examples of a more general class of *orthonormal* rotation matrices that have the following properties:

P.1. $(\mathcal{R}_a^b)^{-1} = (\mathcal{R}_a^b)^\top = \mathcal{R}_b^a$.

P.2. $\mathcal{R}_b^c \mathcal{R}_a^b = \mathcal{R}_a^c$.

P.3. $\det\left(\mathcal{R}_a^b\right) = 1$,

where $\det(\cdot)$ is the determinant of a matrix.

There are several coordinate frames associated with the hummingbird, as explained below. The world frame $\mathcal{F}^w$ is the coordinate frame that is centered at the center of rotation of the hummingbird. When the roll, pitch, and yaw angles are all zero, the $\hat{\imath}^w$-axis is aligned along the body and shaft of the hummingbird, the $\hat{\jmath}^w$-axis is to the right, and the $\hat{k}^w$-axis points down, as shown in Figure 2-3.

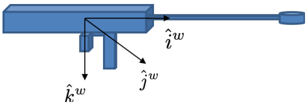


Figure 2-3: The hummingbird world frame. The $\hat{\imath}^w$-axis points along the roll axis, the $\hat{\jmath}^w$-axis points to the right, and the $\hat{k}^w$-axis points into the earth.

The hummingbird 1-frame, denoted $\mathcal{F}^1$, is obtained by rotating the $\mathcal{F}^w$ about the $\hat{k}^w$ axis by an angle of ψ, which is called the yaw angle. A top view of this rotation is shown in Figure 2-4. The rotation matrix that transforms coordinates form $\mathbf{F}^w$ to $\mathbf{F}^1$ is $R_z(\psi)$ as given in Equation (2.2).

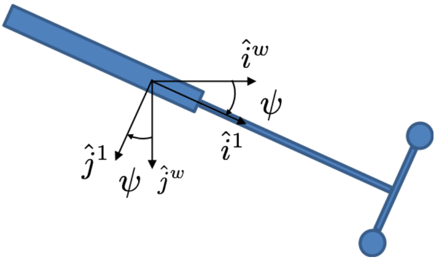


Figure 2-4: The hummingbird 1-frame is obtained from the world frame by rotating ψ about $\hat{k}^w$.

The hummingbird 2-frame, denoted $\mathcal{F}^2$, is obtained by rotating $\mathcal{F}^1$ about the $\hat{\jmath}^1$-axis by an angle of θ, which is called the pitch angle. A side view of this rotation is shown in Figure **??**. The rotation matrix that transforms coordinates form $\mathbf{F}^1$ to $\mathbf{F}^2$ is $R_y(\theta)$ as given in Equation (2.3).

The hummingbird body frame, denoted $\mathcal{F}^b$, is obtained by rotating $\mathcal{F}^2$ about the $\hat{\imath}^2$-axis by an angle of ϕ, which is called the roll angle. A view of the head looking from the rotation center along the hummingbird shown in Figure **??**. The rotation matrix that transforms coordinates form $\mathbf{F}^2$ to $\mathbf{F}^b$ is $R_x(\phi)$ as given in Equation (2.4).

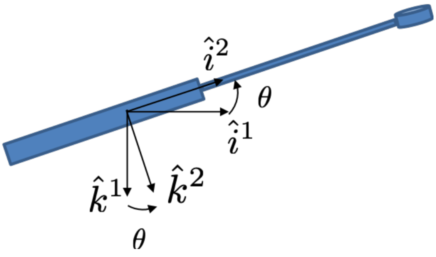


Figure 2-5: The hummingbird 2-frame is obtained from the hummingbird 1-frame by rotating θ about $\hat{\jmath}^1$.

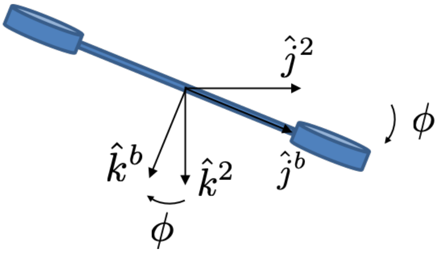


Figure 2-6: The hummingbird body frame is obtained from the hummingbird 2-frame by rotating ϕ about $\hat{\imath}^2$.

2.2.1 Kinetic energy of the hummingbird

The hummingbird spins about the world z^w axis by the yaw angle ψ as shown in Figure 2-1. After rotating about z^w by ψ, the hummingbird spins about the body y^b-axis by the pitch angle θ. After yawing and pitching, the hummingbird spins about the body x^b-axis by the roll angle ϕ, as shown in Figure 2-1. Therefore, the configuration of the hummingbird is given by the three generalized variables

$$\mathbf{q} = \begin{pmatrix} \phi \\ \theta \\ \psi \end{pmatrix}.$$

The position of the center of mass of m_3 is a function of the yaw angle ψ. The position of the center of mass of m_2 is a function of both the yaw angle ψ and the pitch angle θ, and the position of the center of mass of m_1 is a function of all three angles ψ, θ, and ϕ. Therefore, the total kinetic energy of the system is given by

$$K = \sum_{i=1}^{3} \left(\frac{1}{2} m_i \mathbf{v}_i^T \mathbf{v}_i + \frac{1}{2} \boldsymbol{\omega}_i^T \hat{\jmath}_i \boldsymbol{\omega}_i \right).$$

If the position of the center of mass of m_1 in the hummingbird 2-frame is given by $\mathbf{p}_1^2 = \begin{pmatrix}\ell_1 & 0 & 0\end{pmatrix}^\top$, then in the world frame, the position is given by

$$\mathbf{p}_1^w = R_z(\psi)R_y(\theta)\begin{pmatrix}\ell_1\\0\\0\end{pmatrix} = \begin{pmatrix}c_\psi & -s_\psi & 0\\ s_\psi & c_\psi & 0\\ 0 & 0 & 1\end{pmatrix}\begin{pmatrix}c_\theta & 0 & s_\theta\\ 0 & 1 & 0\\ -s_\theta & 0 & c_\theta\end{pmatrix}\begin{pmatrix}\ell_1\\0\\0\end{pmatrix}$$
$$= \begin{pmatrix}\ell_1 c_\theta c_\psi\\ \ell_1 c_\theta s_\psi\\ -\ell_1 s_\theta\end{pmatrix},$$

where to make the expressions easier to read, we have used the notation $c_\phi \triangleq c_\phi$ and $s_\phi \triangleq s_\phi$. Therefore, the velocity of the first mass is given by

$$\mathbf{v}_1 = \dot{\mathbf{p}}_1 = \begin{pmatrix}-\ell_1\dot\theta s_\theta c_\psi - \ell_1\dot\psi c_\theta s_\psi\\ -\ell_1\dot\theta s_\theta s_\psi + \ell_1\dot\psi c_\theta c_\psi\\ -\ell_1\dot\theta c_\theta\end{pmatrix} = \begin{pmatrix}0 & -\ell_1 s_\theta c_\psi & -\ell_1 c_\theta s_\psi\\ 0 & -\ell_1 s_\theta s_\psi & \ell_1 c_\theta c_\psi\\ 0 & -\ell_1 c_\theta & 0\end{pmatrix}\begin{pmatrix}\dot\phi\\ \dot\theta\\ \dot\psi\end{pmatrix} \tag{2.5}$$

$$= V_1(\mathbf{q})\dot{\mathbf{q}}, \tag{2.6}$$

where

$$V_1(\mathbf{q}) \triangleq \begin{pmatrix}0 & -\ell_1 s_\theta c_\psi & -\ell_1 c_\theta s_\psi\\ 0 & -\ell_1 s_\theta s_\psi & \ell_1 c_\theta c_\psi\\ 0 & -\ell_1 c_\theta & 0\end{pmatrix}. \tag{2.7}$$

Calculating the angular velocity is a bit tricky because the angular rates $\dot\phi$, $\dot\theta$ and $\dot\psi$ are all expressed in different frames. The roll rate $(\dot\phi, 0, 0)^\top$ is expressed in the hummingbird body frame. The pitch rate $(0, \dot\theta, 0)^\top$ is expressed in the hummingbird 2-frame, and the yaw rate $(0, 0, \dot\psi)^\top$ is expressed in the hummingbird 1-frame. Since the inertia matrix of J_1 is relative to the body frame, we also need to express $\boldsymbol{\omega}_1$ in the body frame. Therefore

$$\boldsymbol{\omega}_1 = \begin{pmatrix}\dot\phi\\0\\0\end{pmatrix} + \begin{pmatrix}1 & 0 & 0\\ 0 & c_\phi & s_\phi\\ 0 & -s_\phi & c_\phi\end{pmatrix}\begin{pmatrix}0\\ \dot\theta\\ 0\end{pmatrix} \tag{2.8}$$

$$+ \begin{pmatrix}1 & 0 & 0\\ 0 & c_\phi & s_\phi\\ 0 & -s_\phi & c_\phi\end{pmatrix}\begin{pmatrix}c_\theta & 0 & -s_\theta\\ 0 & 1 & 0\\ s_\theta & 0 & c_\theta\end{pmatrix}\begin{pmatrix}0\\0\\ \dot\psi\end{pmatrix} \tag{2.9}$$

$$= \begin{pmatrix}1 & 0 & -s_\theta\\ 0 & c_\phi & s_\phi c_\theta\\ 0 & -s_\phi & c_\phi c_\theta\end{pmatrix}\begin{pmatrix}\dot\phi\\ \dot\theta\\ \dot\psi\end{pmatrix} \tag{2.10}$$

$$= W_1(\mathbf{q})\dot{\mathbf{q}}, \tag{2.11}$$

where

$$W_1(\mathbf{q}) \triangleq \begin{pmatrix}1 & 0 & -s_\theta\\ 0 & c_\phi & s_\phi c_\theta\\ 0 & -s_\phi & c_\phi c_\theta\end{pmatrix}, \tag{2.12}$$

and where we have expressed the roll, pitch, and yaw rates in the body frame.

If the position of the center of mass of m_2 in the body frame is given by $\mathbf{p}_2^b = \begin{pmatrix}\ell_2 & 0 & 0\end{pmatrix}^\top$, then in the inertial frame, the position is given by

$$\begin{aligned}\mathbf{p}_2^b = R_z(\psi)R_y(\theta)\begin{pmatrix}\ell_2\\0\\0\end{pmatrix} &= \begin{pmatrix}c_\psi & -s_\psi & 0\\ s_\psi & c_\psi & 0\\ 0 & 0 & 1\end{pmatrix}\begin{pmatrix}c_\theta & 0 & s_\theta\\ 0 & 1 & 0\\ -s_\theta & 0 & c_\theta\end{pmatrix}\begin{pmatrix}\ell_2\\0\\0\end{pmatrix}\\ &= \begin{pmatrix}\ell_2 c_\theta c_\psi\\ \ell_2 c_\theta s_\psi\\ -\ell_2 s_\theta\end{pmatrix},\end{aligned}$$

where to make the expressions easier to read, we have used the notation $c_\phi \triangleq \cos\phi$ and $s_\phi \triangleq \sin\phi$. Therefore, the velocity of the second mass is given by

$$\mathbf{v}_2 = \dot{\mathbf{p}}_3 = \begin{pmatrix}-\ell_2\dot\theta s_\theta c_\psi - \ell_2\dot\psi c_\theta s_\psi\\ -\ell_2\dot\theta s_\theta s_\psi + \ell_2\dot\psi c_\theta c_\psi\\ -\ell_2\dot\theta c_\theta\end{pmatrix} \tag{2.13}$$

$$= \begin{pmatrix}0 & -\ell_2 s_\theta c_\psi & -\ell_2 c_\theta s_\psi\\ 0 & -\ell_2 s_\theta s_\psi & \ell_2 c_\theta c_\psi\\ 0 & -\ell_2 c_\theta & 0\end{pmatrix}\begin{pmatrix}\dot\phi\\ \dot\theta\\ \dot\psi\end{pmatrix} \tag{2.14}$$

$$= V_2(\mathbf{q})\dot{\mathbf{q}}, \tag{2.15}$$

where

$$V_2(\mathbf{q}) \triangleq \begin{pmatrix}0 & -\ell_2 s_\theta c_\psi & -\ell_2 c_\theta s_\psi\\ 0 & -\ell_2 s_\theta s_\psi & \ell_2 c_\theta c_\psi\\ 0 & -\ell_2 c_\theta & 0\end{pmatrix}. \tag{2.16}$$

The angular velocity of the second mass will be a function $\dot\theta$, and $\dot\psi$ and since J_2 is defined relative to the hummingbird 2-frame, $\boldsymbol{\omega}_2$ is also expressed in the hummingbird 2-frame as

$$\begin{aligned}\boldsymbol{\omega}_2 &= \begin{pmatrix}0\\ \dot\theta\\ 0\end{pmatrix} + \begin{pmatrix}c_\theta & 0 & -s_\theta\\ 0 & 1 & 0\\ s_\theta & 0 & c_\theta\end{pmatrix}\begin{pmatrix}0\\0\\ \dot\psi\end{pmatrix}\\ &= \begin{pmatrix}0 & 0 & -s_\theta\\ 0 & 1 & 0\\ 0 & 0 & c_\theta\end{pmatrix}\begin{pmatrix}\dot\phi\\ \dot\theta\\ \dot\psi\end{pmatrix}\\ &= W_2(\mathbf{q})\dot{\mathbf{q}},\end{aligned}$$

where

$$W_2(\mathbf{q}) \triangleq \begin{pmatrix}0 & 0 & -s_\theta\\ 0 & 1 & 0\\ 0 & 0 & c_\theta\end{pmatrix}. \tag{2.17}$$

If the position of the center of mass of m_3 in the body frame is given by $\mathbf{p}_3^b = \begin{pmatrix}\ell_{3x} & \ell_{3y} & \ell_{3z}\end{pmatrix}^\top$, then in the inertial frame, the position is given by

$$\mathbf{p}_3^b = R_z(\psi)\begin{pmatrix}\ell_{3x}\\ \ell_{3y}\\ \ell_{3z}\end{pmatrix} = \begin{pmatrix}c_\psi & -s_\psi & 0\\ s_\psi & c_\psi & 0\\ 0 & 0 & 1\end{pmatrix}\begin{pmatrix}\ell_{3x}\\ \ell_{3y}\\ \ell_{3z}\end{pmatrix} = \begin{pmatrix}\ell_{3x}c_\psi - \ell_{3y}s_\psi\\ \ell_{3x}s_\psi + \ell_{3y}c_\psi\\ \ell_{3z}\end{pmatrix}.$$

Therefore, the velocity of the third mass is given by

$$\mathbf{v}_3 = \dot{\mathbf{p}}_3 = \dot{\psi}\begin{pmatrix}-\ell_{3x}s_\psi - \ell_{3y}c_\psi\\ \ell_{3x}c_\psi - \ell_{3y}s_\psi\\ 0\end{pmatrix} \tag{2.18}$$

$$= \begin{pmatrix}0 & 0 & -(\ell_{3x}s_\psi + \ell_{3y}c_\psi)\\ 0 & 0 & (\ell_{3x}c_\psi - \ell_{3y}s_\psi)\\ 0 & 0 & 0\end{pmatrix}\begin{pmatrix}\dot{\phi}\\ \dot{\theta}\\ \dot{\psi}\end{pmatrix} \tag{2.19}$$

$$= V_3(\mathbf{q})\dot{\mathbf{q}}, \tag{2.20}$$

where

$$V_3(\mathbf{q}) \triangleq \begin{pmatrix}0 & 0 & -(\ell_{3x}s_\psi + \ell_{3y}c_\psi)\\ 0 & 0 & (\ell_{3x}c_\psi - \ell_{3y}s_\psi)\\ 0 & 0 & 0\end{pmatrix}. \tag{2.21}$$

The angular velocity of the third mass will be a function $\dot{\psi}$ and is given by

$$\boldsymbol{\omega}_3 = \begin{pmatrix}0\\ 0\\ \dot{\psi}\end{pmatrix} = \begin{pmatrix}0 & 0 & 0\\ 0 & 0 & 0\\ 0 & 0 & 1\end{pmatrix}\begin{pmatrix}\dot{\phi}\\ \dot{\theta}\\ \dot{\psi}\end{pmatrix} \tag{2.22}$$

$$= W_3(\mathbf{q})\dot{\mathbf{q}}, \tag{2.23}$$

where

$$W_3(\mathbf{q}) \triangleq \begin{pmatrix}0 & 0 & 0\\ 0 & 0 & 0\\ 0 & 0 & 1\end{pmatrix}. \tag{2.24}$$

Therefore, the total kinetic energy is given by

$$K = \sum_{i=1}^{3}\left(\frac{1}{2}m_i\mathbf{v}_i^T\mathbf{v}_i + \frac{1}{2}\boldsymbol{\omega}_i^T\hat{j}_i\boldsymbol{\omega}_i\right) \tag{2.25}$$

$$= \sum_{i=1}^{3}\left(\frac{1}{2}m_i\dot{\mathbf{q}}^\top V_i^\top V_i\dot{\mathbf{q}} + \frac{1}{2}\dot{\mathbf{q}}^\top W_i^\top J_i W_i\dot{\mathbf{q}}\right) \tag{2.26}$$

$$= \frac{1}{2}\dot{\mathbf{q}}^\top\left(\sum_{i=1}^{3} m_i V_i^\top V_i + W_i^\top J_i W_i\right)\dot{\mathbf{q}} \tag{2.27}$$

$$\triangleq \frac{1}{2}\dot{\mathbf{q}}^\top M(\mathbf{q})\dot{\mathbf{q}}, \tag{2.28}$$

where

$$M(\mathbf{q}) = \sum_{i=1}^{3} m_i V_i^\top(\mathbf{q}) V_i(\mathbf{q}) + W_i^\top(\mathbf{q}) J_i W_i(\mathbf{q}). \tag{2.29}$$

Note that

$$\begin{aligned}
V_1^\top V_1 &= \ell_1^2 \begin{pmatrix} 0 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & c_\theta^2 \end{pmatrix} \\
V_2^\top V_2 &= \ell_2^2 \begin{pmatrix} 0 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & c_\theta^2 \end{pmatrix} \\
V_3^\top V_3 &= \begin{pmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & \ell_{3x}^2 + \ell_{3y}^2 \end{pmatrix}.
\end{aligned}$$

Also note that if J_i are diagonal matrices, then

$$\begin{aligned}
W_1^\top J_1 W_1 &= \begin{pmatrix} J_{1x} & 0 & -J_{1x} s_\theta \\ 0 & (J_{1y} c_\phi^2 + J_{1z} s_\phi^2) & (J_{1y} - J_{1z}) s_\phi c_\phi c_\theta \\ -J_{1x} s_\theta & (J_{1y} - J_{1z}) s_\phi c_\phi c_\theta & (J_{1x} s_\theta^2 + J_{1y} s_\phi^2 c_\theta^2 + J_{1z} c_\phi^2 c_\theta^2) \end{pmatrix} \\
W_2^\top J_2 W_2 &= \begin{pmatrix} 0 & 0 & 0 \\ 0 & J_{2y} & 0 \\ 0 & 0 & (J_{2x} s_\theta^2 + J_{2z} c_\theta^2) \end{pmatrix} \\
W_3^\top J_3 W_3 &= \begin{pmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & J_{3z} \end{pmatrix}.
\end{aligned}$$

Therefore

$$M(\mathbf{q}) = \begin{pmatrix} J_{1x} & 0 & -J_{1x} s_\theta \\ 0 & M_{22} & M_{23} \\ -J_{1x} s_\theta & M_{23} & M_{33} \end{pmatrix} \tag{2.30}$$

where

$$M_{22} = m_1 \ell_1^2 + m_2 \ell_2^2 + J_{2y} + J_{1y} c_\phi^2 + J_{1z} s_\phi^2 \tag{2.31}$$

$$M_{23} = (J_{1y} - J_{1z}) s_\phi c_\phi c_\theta \tag{2.32}$$

$$M_{33} = (m_1 \ell_1^2 + m_2 \ell_2^2 + J_{2z} + J_{1y} s_\phi^2 + J_{1z} c_\phi^2) c_\theta^2 \tag{2.33}$$

$$\quad + (J_{1x} + J_{2x}) s_\theta^2 + m_3 (\ell_{3x}^2 + \ell_{3y}^2) + J_{3z}. \tag{2.34}$$

Lab Assignment W.2

1. Carefully work the derivation of the kinetic energy for the whirlybird by hand to verify that they are correct.

Hints

1. Since the solution is provided in the manual, there are no hints. Good Luck!

3 The Euler Lagrange Equations

The objective of this Chapter is to define the potential energy for the hummingbird. And show how to use the hummingbird's definitions of kinetic and potential energy to derive the equations of motion using the Euler-Lagrange equation.

3.1 Potential Energy

In this section, we will define the hummingbird's potential energy.

Since there are no springs in the system, the only source of potential energy is due to gravity. Therefore the potential energy is given by

$$P = m_1 g h_1 + m_2 g h_2 + P_0$$

where h_1 is the height of the center of mass of m_1 and h_2 is the height of the center of m_2. The height of the third mass does not change with $\mathbf{q}$ and so it is included in P_0. From the geometry of the hummingbird we we have

$$P(q) = m_1 g \ell_1 s_\theta + m_2 g \ell_2 s_\theta + P_0.$$

3.2 Euler-Lagrange Equations

The objective of this section is to develop the simulation model for the hummingbird using the Euler-Lagrange equation

$$\frac{d}{dt}\left(\frac{\partial L}{\partial \dot{\mathbf{q}}}\right) - \frac{\partial L}{\partial \mathbf{q}} = \boldsymbol{\tau} - B\dot{\mathbf{q}}, \tag{3.1}$$

where L is the Lagrangian, $\mathbf{q}$ is the vector of generalized coordinates, $\boldsymbol{\tau}$ are the generalized forces, and B is the damping matrix.

The Lagrangian is defined as

$$\begin{aligned} L &\stackrel{\triangle}{=} K(\mathbf{q},\dot{\mathbf{q}}) - P(\mathbf{q}) \\ &= \frac{1}{2}\dot{\mathbf{q}}^T M(\mathbf{q})\dot{\mathbf{q}} - P(\mathbf{q}). \end{aligned}$$

The generalized forces are

$$\boldsymbol{\tau} = \begin{pmatrix} d(f_\ell - f_r) \\ \ell_T(f_\ell + f_r)c_\phi \\ \ell_T(f_\ell + f_r)c_\theta s_\phi - d(f_\ell - f_r)s_\theta \end{pmatrix},$$

where ℓ_T is the length from the pivot point to the head of the hummingbird. Taking the partial of L with respect to $\mathbf{q}$ gives

$$\frac{\partial L}{\partial \mathbf{q}} = \frac{1}{2}\begin{pmatrix} \dot{\mathbf{q}}^T\frac{\partial M}{\partial q_1} \\ \dot{\mathbf{q}}^T\frac{\partial M}{\partial q_2} \\ \dot{\mathbf{q}}^T\frac{\partial M}{\partial q_3} \end{pmatrix}\dot{\mathbf{q}} - \frac{\partial P}{\partial \mathbf{q}}.$$

Also

$$\frac{d}{dt}\frac{\partial L}{\partial \dot{q}} = \frac{d}{dt}M(q)\dot{q} = M(q)\ddot{q} + \dot{M}\dot{q}.$$

Therefore, the Euler-Lagrange equation gives

$$M(\mathbf{q})\ddot{\mathbf{q}} + \left[\dot{M} - \frac{1}{2}\begin{pmatrix} \dot{\mathbf{q}}^T\frac{\partial M}{\partial q_1} \\ \dot{\mathbf{q}}^T\frac{\partial M}{\partial q_2} \\ \dot{\mathbf{q}}^T\frac{\partial M}{\partial q_3} \end{pmatrix}\right]\dot{\mathbf{q}} + \frac{\partial P}{\partial \mathbf{q}} = \boldsymbol{\tau} - B\dot{\mathbf{q}}.$$

If we define

$$c(\mathbf{q},\dot{\mathbf{q}}) \stackrel{\triangle}{=} \dot{M}\dot{\mathbf{q}} - \frac{1}{2}\begin{pmatrix} \dot{\mathbf{q}}^T\frac{\partial M}{\partial q_1} \\ \dot{\mathbf{q}}^T\frac{\partial M}{\partial q_2} \\ \dot{\mathbf{q}}^T\frac{\partial M}{\partial q_3} \end{pmatrix}\dot{\mathbf{q}},$$

then the equations of motion are given by

$$M(\mathbf{q})\ddot{\mathbf{q}} + c(\mathbf{q},\dot{\mathbf{q}}) + \frac{\partial P}{\partial \mathbf{q}} = \boldsymbol{\tau} - B\dot{\mathbf{q}}.$$

We have that

$$\dot{M} = \begin{pmatrix} 0 & 0 & -J_{1x}c_\theta\dot{\theta} \\ 0 & \dot{M}_{22} & \dot{M}_{23} \\ -J_{1x}c_\theta\dot{\theta} & \dot{M}_{23} & \dot{M}_{33} \end{pmatrix}, \tag{3.2}$$

where

$$\dot{M}_{22} = 2(J_{1z} - J_{1y})s_\phi c_\phi \dot{\phi} \tag{3.3}$$

$$\dot{M}_{23} = (J_{1y} - J_{1z})[(c_\phi^2 - s_\phi^2)c_\theta\dot{\phi} - s_\phi c_\phi s_\theta \dot{\theta}] \tag{3.4}$$

$$\dot{M}_{33} = 2(-m_1\ell_1^2 - m_2\ell_2^2 - J_{2z} + J_{1x} + J_{2x} + J_{1y}s_\phi^2 + J_{1z}c_\phi^2)s_\theta c_\theta\dot{\theta} \tag{3.5}$$

$$+ 2(J_{1y} - J_{1z})s_\phi c_\phi \dot{\phi}. \tag{3.6}$$

In addition, we have

$$\frac{\partial M}{\partial \phi} = \begin{pmatrix} 0 & 0 & 0 \\ 0 & 2(J_{1z}-J_{1y})s_\phi c_\phi & (J_{1y}-J_{1z})(c_\phi^2 - s_\phi^2)c_\theta \\ 0 & (J_{1y}-J_{1z})(c_\phi^2 - s_\phi^2)c_\theta & 2(J_{1y}-J_{1z})s_\phi c_\phi c_\theta^2 \end{pmatrix} \tag{3.7}$$

$$\frac{\partial M}{\partial \theta} = \begin{pmatrix} 0 & 0 & -J_{1x}c_\theta \\ 0 & 0 & -(J_{1y}-J_{1z})s_\phi c_\phi s_\theta \\ -J_{1x}c_\theta & -(J_{1y}-J_{1z})s_\phi c_\phi s_\theta & N_{33} \end{pmatrix} \tag{3.8}$$

$$\frac{\partial M}{\partial \psi} = \begin{pmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix}, \tag{3.9}$$

where

$$N_{33} = 2(J_{1x} + J_{2x} - m_1\ell_1^2 - m_2\ell_2^2 - J_{2z} - J_{1y}s_\phi^2 - J_{1z}c_\phi^2)s_\theta c_\theta.$$

Working through the math, we get

$$c(\mathbf{q}, \dot{\mathbf{q}}) = \begin{pmatrix} (J_{1y}-J_{1z})s_\phi c_\phi(\dot\theta^2 - c_\theta^2\dot\psi^2) + \Big[(J_{1y}-J_{1z})(c_\phi^2 - s_\phi^2) - J_{1x}\Big] c_\theta\dot\theta\dot\psi \\ --- \\ 2(J_{1z}-J_{1y})s_\phi c_\phi\dot\phi\dot\theta + \Big[(J_{1y}-J_{1z})(c_\phi^2 - s_\phi^2) + J_{1x}\Big] c_\theta\dot\phi\dot\psi - \frac{1}{2}N_{33}\dot\psi^2 \\ --- \\ \dot\theta^2(J_{z1} - J_{y1})s_\phi c_\phi s_\theta \\ + \Big[(J_{1y}-J_{1z})(c_\phi^2 - s_\phi^2) - J_{1x}\Big] c_\theta\dot\phi\dot\theta \\ +(J_{1z}-J_{1y})s_\phi c_\phi s_\theta\dot\theta^2 + 2(J_{1y}-J_{1z})s_\phi c_\phi\dot\phi\dot\psi \\ +2\Big[-m_1\ell_1^2 - m_2\ell_2^2 - J_{2z} + J_{1x} + J_{2x} + J_{1y}s_\phi^2 + J_{1z}s_\phi^2\Big] s_\theta c_\theta\dot\theta\dot\psi \end{pmatrix},$$

and

$$\frac{\partial P}{\partial \mathbf{q}} = \begin{pmatrix} 0 \\ (m_1\ell_1 + m_2\ell_2)gc_\theta \\ 0 \end{pmatrix}.$$

We will model damping in each axis using

$$B = \begin{pmatrix} b_\phi & 0 & 0 \\ 0 & b_\theta & 0 \\ 0 & 0 & b_\psi \end{pmatrix}. \tag{3.10}$$

3.2.1 Summary

The equations of motion are

$$M(\mathbf{q})\ddot{\mathbf{q}} + c(\mathbf{q}, \dot{\mathbf{q}}) + \frac{\partial P}{\partial \mathbf{q}} = \boldsymbol{\tau} - B\dot{\mathbf{q}}. \tag{3.11}$$

Lab Assignment W.3

1. Verify that the equations of motion derived in this section are correct.

2. Implement the dynamics of the whirlybird in the `wb_st_dynamics` package. This is done by editing the file `wb_dynamics.py` found in

   `wb_st_dynamics/scripts/`.

   You only need to edit the part that says "Implement Dynamics for Accelerations Here." You will need to implement this using numpy. For the creation of numpy arrays, these links will be helpful. You may find the function numpy.linalg.solve to be helpful. The ODE is integrated using Runge-Kutta numerical integration, but that is all handled for you in the `propagate` method.

3. Once you have implemented the dynamics, you will need to test them. We have given you a pre-compiled controller node that was installed in your workspace upon building it. The controller node subscribes to the `/wb_reference_states` topic to receive the desired pitch and yaw angles. It then uses an algorithm to compute the pwm commands and publishes these commands to the `/wb_command` topic. The dynamics node that you complete in this lab will subscribe to the `/wb_command` topic, update the states, and publish them to the `/wb_states` topic.

   With this setup, we can use the pre-compiled controller to test your dynamics. If the pre-compiled controller can move the whirlybird to the desired orientation, then your dynamics are most likely correct, but there is no absolute guarantee with this.

   To run the pre-compiled controller, open up the sim.launch launch file in your `wb_st_control/launch` folder which should look like

```
<launch>

  <include file="$(find wb_st_dynamics)/launch/dynamics_viz.launch"/>

  <node name="wb_st_control" pkg="wb_st_control" type="controller.py" />
  <!— <node name="wb_st_control" pkg="wb_st_control" type="pre_compiled_controller" /> —>

</launch>
```

   Note that there are two controller nodes shown in this launch file, and that one of them in commented out. You will need to un-comment the `pre_compiled_controller` and comment out all other controller nodes. For example, your launch file should look like

```
<launch>

  <include file="$(find wb_st_dynamics)/launch/dynamics_viz.launch"/>

  <!— <node name="wb_st_control" pkg="wb_st_control" type="controller.py" /> —>
  <node name="wb_st_control" pkg="wb_st_control" type="pre_compiled_controller" />

</launch>
```

   Now run the launch file using the command in a terminal.

```
roslaunch wb_st_control sim.launch
```

This should pull up an RVIZ window. The panel should not be enabled, and set to publish to the `/wb_states` topic. Change the topic to `/wb_reference_states` and click the enabled button. Play around with the sliders and see if the simulation moves to the desired pitch and yaw angles. This is a good indication that your dynamics are correct. Once you think your dynamics are correct, have a TA pass you off for the lab.

Hints

TOC

Part II

Hummingbird Design Models

TOC

The equations of motion developed in the previous chapter are too complex to be useful for control design. Therefore the objective of this chapter is to develop several different design models for the hummingbird that capture certain behaviors of the system. The essential idea is to notice that since the total force on the head of the hummingbird is $F = f_l + f_r$, the pitching motion can be controlled by manipulating F. Also, since the torque on the head of the hummingbird is $\tau = (f_l - f_r)d$, the rolling motion can be controlled by manipulating τ. We also note that the yawing motion is driven primarily by the roll angle ϕ.

Accordingly, we define the *longitudinal* motion of the hummingbird, to be the motion involving the state variables $(\theta, \dot{\theta})$, and the *lateral* motion of the hummingbird to be the motion involving the state variables $(\phi, \psi, \dot{\phi}, \dot{\psi})$. While these motions are coupled, the coupling is weak and will be neglected in the design models.

TOC

4

Equilibria and Linearization

In this chapter we will discuss the equilibria and linearization of the longitudinal and lateral dynamics of the hummingbird.

4.1 Equilibria and Linearization for the Longitudinal Dynamics

In this section, we will derive an expression for the equilibrium force using the equations of motion derived in the previous chapter. Since we are interested only in the equilibrium force of the longitudinal dynamics, we will use the governing equation for θ stated in the second line of Equation (3.11) which is

$$\begin{aligned}(m_1\ell_1^2 + m_2\ell_2^2 + J_{2y} + J_{1y}c_\phi^2 + J_{1z}s_\phi^2)\ddot{\theta} + (J_{1y} - J_{1z})s_\phi c_\phi c_\theta \ddot{\psi} \\ - \dot{\psi}^2 s_\theta c_\theta \left[J_{1x} + J_{2x} - m_1\ell_1^2 - m_2\ell_2^2 - J_{2z} - J_{1y}s_\phi^2 + J_{1z}c_\phi^2\right] \\ + 2\dot{\phi}\dot{\theta}(J_{1z} - J_{1y})s_\phi c_\phi + \dot{\phi}\dot{\psi}c_\theta\left[J_{1x} + (J_{1y} - J_{1z})(c_\phi^2 - s_\phi^2)\right] \\ + (m_1\ell_1 + m_2\ell_2)gc_\theta = \ell_T(f_l + f_r)c_\phi - b_\theta\dot{\theta}. \quad (4.1)\end{aligned}$$

We are interested in expressing the longitudinal dynamics assuming that $\phi = \dot{\phi} = \ddot{\psi} = \dot{\psi} = 0$ in order to decouple the logitudinal and lateral dynamics. This is a reasonable assumption since the longitudinal and lateral dynamics do not strongly influence each other. Setting $\phi = \dot{\phi} = \ddot{\psi} = \dot{\psi} = 0$ gives

$$(m_1\ell_1^2 + m_2\ell_2^2 + J_{1y} + J_{2y})\ddot{\theta} + (m_1\ell_1 + m_2\ell_2)g\cos\theta = \ell_T(f_l + f_r)c_\phi - b_\theta\dot{\theta}.$$

Defining $F \overset{\triangle}{=} f_l + f_r$ gives

$$(m_1\ell_1^2 + m_2\ell_2^2 + J_{1y} + J_{2y})\ddot{\theta} + b_\theta\dot{\theta} + (m_1\ell_1 + m_2\ell_2)g\cos\theta = \ell_T F. \quad (4.2)$$

Setting $\dot{\theta} = \ddot{\theta} = 0$ gives the expression for the equilibrium force

$$F_e = \frac{(m_1\ell_1 + m_2\ell_2)g\cos\theta_e}{\ell_T}.$$

Alternatively, we could feedback linearize the system by setting

$$F = \frac{(m_1\ell_1 + m_2\ell_2)g\cos\theta}{\ell_T} + \tilde{F}$$

to get the feedback linearized system

$$\ddot{\theta} + \left(\frac{b}{m_1\ell_1^2 + m_2\ell_2^2 + J_{1y} + J_{2y}}\right)\dot{\theta} = \left(\frac{\ell_T}{m_1\ell_1^2 + m_2\ell_2^2 + J_{1y} + J_{2y}}\right)\tilde{F}. \tag{4.3}$$

The damping coefficient is difficult to measure and should be relatively small. Therefore, assuming $b = 0$ gives

$$\ddot{\theta} = \left(\frac{\ell_T}{m_1\ell_1^2 + m_2\ell_2^2 + J_{1y} + J_{2y}}\right)\tilde{F} = b_\theta\tilde{F}, \tag{4.4}$$

where

$$b_\theta = \left(\frac{\ell_T}{m_1\ell_1^2 + m_2\ell_2^2 + J_{1y} + J_{2y}}\right). \tag{4.5}$$

4.2 Equilibria and Linearization for the Lateral Dynamics

The objective of this section is to define the equilibrium orientation and to linearize the lateral dynamics. The governing equations for ϕ and ψ are the first and third lines of Equation (3.11) which, after setting $\theta = \dot{\theta} = \ddot{\theta} = 0$ are

$$\begin{aligned} J_{1x}\ddot{\phi} - \dot{\psi}^2(J_{1y} - J_{1z})s_\phi c_\phi &= d(f_l - f_r) \\ (J_T + J_{1y}s_\phi^2 + J_{1z}c_\phi^2)\ddot{\psi} + 2\dot{\psi}\dot{\phi}(J_{1y} - J_{1z})s_\phi c_\phi &= \ell_T(f_l + f_r)s_\phi, \end{aligned}$$

where

$$J_T \overset{\triangle}{=} m_1\ell_1^2 + m_2\ell_2^2 + J_{2z} + m_3(\ell_{3x}^2 + \ell_{3y}^2).$$

To decouple the motion from the longitudinal states, we will assume that $f_l + f_r = F_e$, the equilibrium force required to maintain $\theta = \theta_e$.

Defining the torque $\tau \overset{\triangle}{=} d(f_l - f_r)$ gives

$$\begin{aligned} J_{1x}\ddot{\phi} - \dot{\psi}^2(J_{1y} - J_{1z})s_\phi c_\phi &= d\tau \\ (J_T + J_{1y}s_\phi^2 + J_{1z}c_\phi^2)\ddot{\psi} + 2\dot{\psi}\dot{\phi}(J_{1y} - J_{1z})s_\phi c_\phi &= \ell_T F_e s_\phi. \end{aligned} \tag{4.6}$$

The equilibrium values are found by setting $\dot{\phi} = \dot{\psi} = \ddot{\phi} = \ddot{\psi} = 0$ in Equation (4.6) to obtain

$$\begin{aligned} \tau_e &= 0 \\ F_e \sin\phi_e &= 0 \end{aligned}$$

Since F_e is not zero, we get that $\phi_e = 0$. Linearizing the nonlinear terms of equation (4.6) around $\theta_e = \phi_e = 0$, we find that

$$\begin{aligned} \dot{\psi}^2 \sin\phi\cos\phi \approx \dot{\psi_e}^2 \sin\phi_e \cos\phi_e &+ \frac{\partial}{\partial\dot{\psi}} \dot{\psi}^2 \sin\phi\cos\phi\Big|_{(\dot{\psi},\phi)} (\dot{\psi} - \dot{\psi}_e) \\ &+ \frac{\partial}{\partial\phi} \dot{\psi}^2 \sin\phi\cos\phi\Big|_{(\dot{\psi},\phi)} (\phi - \phi_e) \\ &= \dot{\psi_e}^2 \sin\phi_e\cos\phi_e + 2\dot{\psi}_e \sin\phi_e \cos\phi_e \dot{\tilde{\psi}} + \dot{\psi_e}^2 \cos(2\phi_e)\tilde{\phi} \\ &= 0 \end{aligned}$$

$$\begin{aligned} \sin^2(\phi)\ddot{\psi} \approx \sin^2(\phi_e)\ddot{\psi}_e &+ \frac{\partial}{\partial\phi}\sin^2(\phi)\ddot{\psi}\Big|_{(\phi_e,\ddot{\psi}_e)} (\phi - \phi_e) + \frac{\partial}{\partial\ddot{\psi}}\sin^2(\phi)\ddot{\psi}\Big|_{(\phi_e,\ddot{\psi}_e)} (\ddot{\psi} - \ddot{\psi}_e) \\ &= \sin^2(\phi_e)\ddot{\psi}_e + \sin(2\phi_e)\ddot{\psi}_e\tilde{\phi} + \sin^2\phi_e \ddot{\tilde{\psi}} \\ &= 0 \end{aligned}$$

$$\begin{aligned} \cos^2(\phi)\ddot{\psi} \approx \cos^2(\phi_e)\ddot{\psi}_e &+ \frac{\partial}{\partial\phi}\cos^2(\phi)\ddot{\psi}\Big|_{(\phi_e,\ddot{\psi}_e)} (\phi - \phi_e) + \frac{\partial}{\partial\ddot{\psi}}\cos^2(\phi)\ddot{\psi}\Big|_{(\phi_e,\ddot{\psi}_e)} (\ddot{\psi} - \ddot{\psi}_e) \\ &= \cos^2(\phi_e)\ddot{\psi}_e - \sin(2\phi_e)\ddot{\psi}_e\tilde{\phi} + \cos^2\phi_e \ddot{\tilde{\psi}} \\ &= \ddot{\tilde{\psi}} \end{aligned}$$

$$\begin{aligned} \dot{\psi}\dot{\phi}\sin\phi\cos\phi \approx \dot{\psi}_e\dot{\phi}_e \sin\phi_e\cos\phi_e &+ \frac{\partial}{\partial\phi}\dot{\psi}\dot{\phi}\sin\phi\cos\phi\Big|_{(\phi_e,\dot{\phi}_e,\dot{\psi}_e)} (\phi - \phi_e) + \frac{\partial}{\partial\dot{\phi}}\dot{\psi}\dot{\phi}\sin\phi\cos\phi\Big|_{(\phi_e,\dot{\phi}_e} \\ &+ \frac{\partial}{\partial\dot{\psi}}\dot{\psi}\dot{\phi}\sin\phi\cos\phi\Big|_{(\phi_e,\dot{\phi}_e,\dot{\psi}_e)} (\dot{\psi} - \dot{\psi}_e) \\ &= \dot{\psi}_e\dot{\phi}_e \sin\phi_e\cos\phi_e + \dot{\psi}_e\dot{\phi}_e\cos(2\phi_e)\tilde{\phi} + \dot{\psi}_e \sin\phi_e\cos\phi_e\dot{\tilde{\phi}} + \dot{\phi}_e\sin\phi_e\cos\phi_e\dot{\tilde{\psi}} \\ &= 0 \end{aligned}$$

$$\begin{aligned} \sin\phi \approx \sin\phi_e &+ \frac{\partial}{\partial\phi}\sin\phi\Big|_{\phi_e} (\phi - \phi_e) \\ &= \sin\phi_e + \cos\phi_e\tilde{\phi} \\ &= \tilde{\phi} \end{aligned}$$

where we have defined $\tilde{\phi} \triangleq \phi - \phi_e$, $\tilde{\psi} \triangleq \psi - \psi_e$, and $\tilde{\tau} \triangleq \tau - \tau_e$. Since $\phi_e = \psi_e = \tau_e = 0$, we have $\tilde{\phi} = \phi$, $\tilde{\psi} = \psi$, and $\tilde{\tau} = \tau$. We can now write Equation 4.6 in its linearized form as

$$J_{1x}\ddot{\tilde{\phi}} = \tilde{\tau} \tag{4.7}$$

$$(J_T + J_{1z})\ddot{\tilde{\psi}} = \ell_T F_e \tilde{\phi}. \tag{4.8}$$

4.3 Model of the Motor-Propeller

In the previous sections, we modeled the motors as producing a force perpendicular to the hummingbird x-y plane. The input to the motors is actually a pulse-width-modulation (PWM) command that regulates the duty cycle of the current supplied to the motor. The motor stops when the PWM is below $1060\ \mu s$ (0% duty cycle) and is at full power when the PWM is at $1860\ \mu s$ (100% duty cycle). To make things easier, code has been added to the hummingbird's firmware to convert the duty cycle to PWM. That means that the output command to the hummingbird needs to be converted from force to duty cycle, and the duty cycle is constrained in the range $[0, 1]$, where 0 represents zero percent duty cycle, or zero current supplied to the motor, and 1 represents 100 percent duty cycle, or full current supplied to the motor.

While there are dynamics in the internal workings of the motor, we will neglect these dynamics and model the relationship between PWM command u and force as

$$F = f(u).$$

F is not linearly proportional to u, but rather a function of u. However, for the purposes of this class we will assume it to be linearly porportional to u by the scalar k_m.

$$F = k_m u.$$

We can make this assumption since F and u are almost linearly proportional to each other in the region of operation. The region of operation is centered around the force needed to maintain the hummingbird in equilibrium when $\psi = \theta = \phi = 0$.

To experimentally find the value of k_m we apply an equal PWM signal to each motor and increase this signal until the force balances the hummingbird at constant pitch angle. In the equilibrium position the forces cancel and we have

$$\ell_T F_e = m_1 \ell_1 g + m_2 \ell_2 g.$$

Setting $F_e = f_{l_e} + f_{r_e} = k_m(u_{l_e} + u_{r_e})$, and solving for k_m we get

$$k_m = \frac{m_1 \ell_1 g + m_2 \ell_2 g}{\ell_T (u_{l_e} + u_{r_e})}.$$

While PWM is the input to the system, it is easier to think in terms of torque and force applied to the hummingbird. The relationship between PWM commands and force and torque are given by

$$\begin{pmatrix} \tau \\ F \end{pmatrix} = k_m \begin{pmatrix} d & -d \\ 1 & 1 \end{pmatrix} \begin{pmatrix} u_l \\ u_r \end{pmatrix}. \tag{4.9}$$

Inverting the matrix and solving for PWM command gives

$$u_l = \frac{1}{2k_m}\left(F + \frac{\tau}{d}\right) \tag{4.10}$$

$$u_r = \frac{1}{2k_m}\left(F - \frac{\tau}{d}\right). \tag{4.11}$$

Lab Assignment W.4

1. Verify that the linearized equations of motion for the lateral and longitudinal dynamics are correct.

2. Calculate the value of k_m as described in Section 4.3.

3. Replace the entry in `whirlybird_sim/param/whirlybird.yaml` with the correct value of k_m.

Hints

To calculate the value of k_m follow the steps below.

1. Fix the rolling and yawing motion of the whirlybird using the pin and screw. tch upwards.

2. Connect to the closest whirlybird and launch the (whirlybird_description visualize_pwm.launch).

3. We are interested in finding the value of k_m at equilibrium such that $F_e = k_m(u_{l_e} + u_{r_e})$ where F_e is the equilibrium force calculated from the equations of motion, and u_{l_e}, u_{r_e} is the pulse width modulation command being sent to the left and right whirlybird motors, which range from 0 to 1 in value. Slowly increase the value of the PWM slider, until the whirlybird balances at 0 degrees pitch . This is the equilibrium PWM value PWM_e and is the value of u_{l_e} and u_{r_e} such that $\text{PWM}_e = u_{l_e} = u_{r_e}$ and $F_e = k_m(2\text{PWM}_e)$. Solve for k_m.

5

Transfer Function Models

The the response of a system to an input is described by its transfer function. By modeling the linearized lateral and logitudinal dynamics with a transfer function, we can gain intuition on how the hummingbird will respond to a given force or torque input. These transfer functions will be used in a later chapter to design the control system for the hummingbird.

5.1 Linear Transfer Function Model for Longitudinal Dynamics

The longitudinal equations of motion have been decoupled from the lateral generalized coordinates ϕ and ψ, and linearized. The linearized equation (4.4) is a second-order differential equation relating $\tilde{\theta}$ and $\tilde{F}$. The longitudinal transfer function is obtained by taking the Laplace transform of (4.4) and solving for $\theta(s)$.

$$\tilde{\theta}(s) = \frac{b_\theta}{s^2}\tilde{F}(s). \tag{5.1}$$

5.2 Linear Transfer Function Model for Lateral Dynamics

The lateral equations of motion have been decoupled from the longitudinal generalized coordinate θ and linearized. The linearized equations are shown in Equation (4.8). The first is a second-order differential equation relating $\tilde{\tau}$ and $\ddot{\tilde{\phi}}$, and the second is a second-order differential equation relating $\tilde{\phi}$ and $\ddot{\tilde{\psi}}$.

The Laplace transform of the first equation gives

$$\tilde{\phi}(s) = \frac{(1/J_{1x})}{s^2}\tilde{\tau}(s), \tag{5.2}$$

TOC

and the Laplace transform of the second gives

$$\tilde{\psi}(s) = \frac{\left(\frac{\ell_T F_e}{J_T + J_{1z}}\right)}{s^2}\phi(s) = \frac{b_\psi}{s^2}\phi(s), \tag{5.3}$$

where

$$b_\psi = \left(\frac{\ell_T F_e}{J_T + J_{1z}}\right) \tag{5.4}$$

Equations (5.2) and (5.3) show that the transfer functions for the lateral dynamics form a cascade structure as shown in Figure 5-1. As shown in Figure 5-1 the torque τ can be thought of as the input to the roll dynamics given by Equation (5.2). The roll angle ϕ which is the output of the roll dynamics can be thought of as the input to the yaw dynamics given by Equation (5.3). This cascade structure will be exploited in later labs where we will design controllers for the roll dynamics and yaw dynamics separately and enforce suitable bandwidth separation to ensure a good design.

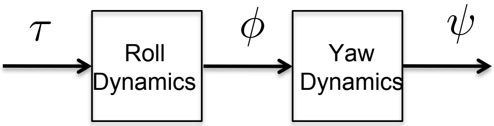


Figure 5-1: The lateral dynamics for the hummingbird can be thought of as a cascade system, where the torque drives the roll dynamics and the roll angle drives the yaw dynamics.

Lab Assignment W.5

1. Verify the transfer functions derived in this section.
2. Why does the cascade structure shown in Figure 5-1 make sense physically?

6 State Space Models

The objective of this chapter is to develop the linear transfer function models for the longitudinal and lateral dynamics of the hummingbird.

6.1 Linear State Space Model for Longitudinal Dynamics

The linear equations of motion for the longitudinal dynamics are given by

$$\ddot{\theta} = b_\theta \tilde{F},$$

where we assume that $\tilde{F} = F_{fl} + F$ is the feedback linearized force.

Defining the state as

$$x_{lon} \stackrel{\triangle}{=} (\theta, \dot{\theta})^T$$

and the deviated control input as

$$\tilde{u}_{lon} = \tilde{F},$$

then the linear state model is given by

$$\dot{x}_{lon} = A_{lon} x_{lon} + B_{lon} \tilde{u}_{lon},$$

where

$$A_{lon} = \begin{pmatrix} 0 & 1 \\ 0 & 0 \end{pmatrix}, \tag{6.1}$$

$$B_{lon} = \begin{pmatrix} 0 \\ b_\theta \end{pmatrix}. \tag{6.2}$$

If the output is $y_{lon} = \theta$, then the output equation is

$$y_{lon} = C_{lon} x_{lon},$$

where $C_{lon} = (1, 0)$.

Note that if we design a controller law $\tilde{u}_{lon}(x_{lon})$, then the commanded force will be

$$F = F_{fl} + \tilde{u}_{lon}.$$

6.2 Linear State Space Model for Lateral Dynamics

From Equation (4.8), the linearized equations for the lateral dynamics are given by

$$\ddot{\tilde{\phi}} = \frac{1}{J_{1x}} \tilde{\tau} \tag{6.3}$$

$$\ddot{\tilde{\psi}} = \frac{1}{J_T + J_{1z}} \ell_T F_e \tilde{\phi}. \tag{6.4}$$

Define the lateral state as $x_{lat} = (\tilde{\phi}, \tilde{\psi}, \dot{\tilde{\phi}}, \dot{\tilde{\psi}})^\top$, and the lateral input $u_{lat} = \tilde{\tau}$. The lateral output is $y_{lat} = (\tilde{\phi}, \tilde{\psi})^\top$. Then the linearized state equations are given by

$$\dot{x}_{lat} = A_{lat} x_{lat} + B_{lat} u_{lat} \tag{6.5}$$

$$y_{lat} = C_{lat} x_{lat}, \tag{6.6}$$

where

$$A_{lat} = \begin{pmatrix} 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \\ \frac{\ell_T F_e}{J_T + J_{1z}} & 0 & 0 & 0 \end{pmatrix} \tag{6.7}$$

$$B_{lat} = \begin{pmatrix} 0 \\ 0 \\ \frac{1}{J_{1x}} \\ 0 \end{pmatrix} \tag{6.8}$$

$$c_{lat} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \end{pmatrix}. \tag{6.9}$$

Lab Assignment W.6

1. Verify the state space equations derived in this section.

Part III

PID Control Design

TOC

7 Pole Placement for Second Order Systems

The objective of this lab is to implement PD control for the longitudinal motion of the hummingbird in Python/Simulink. The block diagram for the longitudinal controller is shown in Figure 7-1.

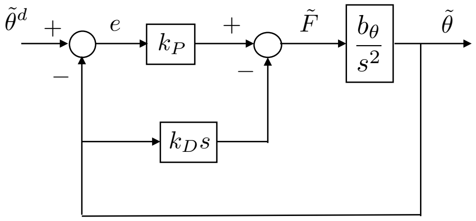


Figure 7-1: PD control design example.

A feedback linearizing force that cancels the gravity term can be calculated as

$$F_{fl} = (m_1\ell_1 + m_2\ell_2)\frac{g}{\ell_T}\cos\theta.$$

Keep in mind that the control force sent to the propeller motors is given by $F = F_{fl} + \tilde{F}$, where $\tilde{F}$ is calculated as shown in Figure 7-1.

Lab Assignment W.7

1. Find the transfer function from $\tilde{\Theta}^d$ to $\tilde{\Theta}$ of the closed loop system shown in Figure 7-1.

2. Find the characteristic polynomial $\Delta_{cl}(s)$ of the closed loop system.

3. Find k_P and k_D so that the rise time is $t_r = 1.5\ s$ and the damping ratio is $\zeta = 0.707$. Make the rise time a tuning parameter that you can adjust to get good performance.

4. For this lab you will need to constrain the lattitude dynamics of the whirlybird. In hardware this is done by placing the pin to constrain the roll and yaw movements. In simulation, this is done by ensuring that the torque command is always zero.

5. In the folder `wb_st_control\scripts`, copy the template controller, `controller.py` and give it a new name but keep the python extension. Ex: `controller_7.py`. Edit the new controller file to implement the longitudinal PD controller. You may assume that pitch and its derivative are available to you. Your PD control can be implemented as $\tilde{F} = k_P(\tilde{\Theta}^d - \tilde{\Theta}) - k_D\dot{\tilde{\Theta}}$. Recall that $\tilde{F} = F - F_{fl}$ and therefore $F = F_{fl} + \tilde{F}$ which implies that the feedback linearizing force needs to be added to the output of the PD controller. Note: Your force outputs are then scaled using the k_m value you found in a previous lab.

6. Implement saturation, by not not allowing your PWM values to go above 0.7. Note: Your PWM values also should not be allowed to go below 0, since it doesn't make sense for PWM to be negative.

7. To use your new controller, you will need to edit the launch file `wb_st_control\launch\sim(hw).launch` so that your new controller runs. Edit the tag

   ```
   <node name="wb_st_control" pkg="wb_st_control" type="
   pre_compiled_controller" />
   ```

   so that they **type** parameter is set to the name of your new controller. Ex.

   ```
   <node name="wb_st_control" pkg="wb_st_control" type="
   controller_7.py" />
   ```

8. Using the formulas you derived for k_P and k_D, tune your controller design using the rise time t_r and the damping ratio ζ parameters so that the step response of the system is well-damped and responsive. To test this, drive your controller using the sliders from -15 degrees to 15 degrees. Tune your design so that saturation does not occur for this step size (30 degrees).

Hints

- If yaw and roll are constrained what does this imply about $\dot{\phi}$ and $\dot{\psi}$?

- Observe that the controller is implemented using a python class structure. Your gains are stored as member variables of this class. These are referenced using self.<variable>. You can modify these member variables in any method, but it makes most sense to tune your gains in the `__init__` method. Here is a tutorial on classes and object oriented programming in python.
- Saturation has been mostly implemented at the end of the `whirlybirdCallback` method. You should only have to edit a handful of values.
- You can test if your system is saturating by using `rqt_plot` on the `/command` topic. If you see clipping then your system is saturating and your gains will need to be retuned.

8

Design Strategies for Second-Order Systems

The objective of this lab assignment is to use the desired transient response characteristics, rise time and damping ratio, to determine PD control gains that acheive the desired response. The longitudinal control design will utilize a single PD control loop, while the lateral control design will use two PD loops designed using principles of successive loop closure.

Longitudinal Control Design

In the previous chapter we showed that the transfer function from $\tilde{F}$ to $\tilde{\theta}$ is

$$\tilde{\theta}(s) = \frac{b_\theta}{s^2}\tilde{F}(s).$$

The closed-loop transfer function for the PD implementation shown in Figure 7-1 is

$$\frac{\tilde{\theta}(s)}{\tilde{\theta}^d(s)} = \frac{k_p b_\theta}{s^2 + k_d b_\theta s + k_p b_\theta}$$

The transfer function for a canonical second-order system can be expressed in terms of the damping ratio ζ_θ and the natural frequency ω_{n_θ} as

$$\frac{\omega_{n_\theta}^2}{s^2 + 2\zeta_\theta \omega_{n_\theta} s + \omega_{n_\theta}^2}.$$

Values for ζ_θ and ω_{n_θ} are determined from the desired transient response characteristics of the system. The natural frequency and desired rise time are approximately related by the expression

$$t_r \approx \frac{2.2}{\omega_{n_\theta}},$$

while ζ_θ can be selected to give the desired overshoot and settling time behavior. Control gains are calculated by equating the coefficients of the desired characteristic equation

$$s^2 + 2\zeta_\theta \omega_{n_\theta} s + \omega_{n_\theta}^2 = 0$$

with those of the characteristic equation of the closed-loop system, in this case

$$s^2 + k_d b_\theta s + k_p b_\theta = 0.$$

Lateral Control Design

The lateral motion of the hummingbird is described by its rolling and yawing motion. Changes in the roll angle ϕ are caused by torques produced when the left and right thrust values are different. When ϕ is non-zero, a lateral force is produced on the hummingbird causing the to yaw, which is described by the yaw angle ψ. These lateral motions are modeled by the transfer functions derived in Section 5.2. The control strategy we will use for the control of the lateral dynamics has the transfer functions for the rolling and yawing dynamics cascaded together in a successive loop closure configuration as shown in Figure 8-1

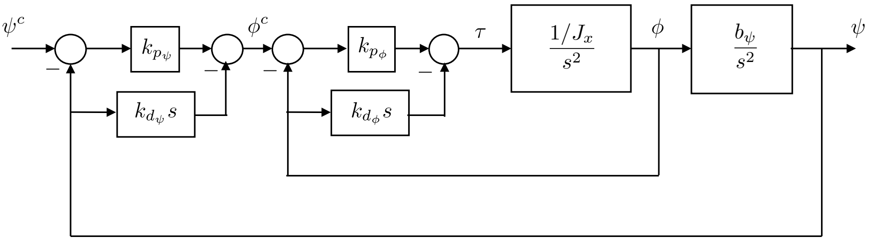


Figure 8-1: Successive loop closure block diagram for lateral dynamics.

Roll Loop Design

To design the lateral dynamics control system for the hummingbird using successive loop closure, we begin with the inner roll control loop. By design, the inner roll loop response will be at least ten times faster than the outer yaw loop dynamic response. This will allow us to approximate the inner loop by its closed-loop DC gain when designing the outer loop.

The open loop transfer function of the inner roll loop is

$$\phi(s) = \frac{(1/J_{1x})}{s^2} \tau(s)$$

The closed-loop transfer function of the roll loop is

$$\frac{\phi(s)}{\phi^d(s)} = \frac{k_{p_\phi}/J_{1x}}{s^2 + \left(k_{d_\phi}/J_{1x}\right) s + \left(k_{p_\phi}/J_{1x}\right)} \tag{8.1}$$

with the corresponding characteristic equation

$$s^2 + \left(k_{d_\phi}/J_{1x}\right) s + \left(k_{p_\phi}/J_{1x}\right) = 0.$$

Based on the desired rise time t_{r_ϕ} and damping characteristics ζ_ϕ of the inner loop, we can determine the natural frequency of the roll dynamics ω_{n_ϕ} and the desired characteristic equation

$$s^2 + 2\zeta_\phi \omega_{n_\phi} s + \omega_{n_\phi}^2 = 0.$$

Proportional and derivative gains for the roll loop, k_{p_ϕ} and k_{d_ϕ} are determined by matching coefficients of the desired and actual characteristic equations.

Yaw Loop Design

Successive loop closure requires that there be adequate bandwidth separation between successive loops in the control design. The inner loop is designed to be the fastest (highest natural frequency), with each successive loop designed to be a factor of 10 to 20 slower in frequency. Higher natural frequencies correspond to shorter rise times. In the case of the yaw loop design, if we define M as the bandwith separation factor (a constant between 10 and 20), and if the roll loop has a designed rise time of t_{r_ϕ}, then we would calculate the rise time of the slower outer yaw loop to be $t_{r_\psi} = M t_{r_\phi}$. Similarly, if the inner roll loop natural frequency is given by ω_{n_ϕ}, the outer yaw loop natural frequency would be calculated to be $\omega_{n_\psi} = \omega_{n_\phi}/M$. Bandwidth separation allows the inner loop dynamics to be replaced by the DC gain of the inner loop when designing the outer loop control, thus simplifying the outer-loop design significantly.

From the closed-loop transfer function for the roll loop given by equation 8.1, we can see that the DC gain of the inner loop is 1. Approximating the inner-loop dynamics with a transfer function of 1, the yaw closed-loop transfer function can be written as

$$\frac{\psi(s)}{\psi^d(s)} = \frac{k_{p_\psi} b_\psi}{s^2 + k_{d_\psi} b_\psi s + k_{p_\psi} b_\psi}$$

with the corresponding characteristic equation

$$s^2 + k_{d_\psi} b_\psi s + k_{p_\psi} b_\psi = 0.$$

As discussed above, the desired rise time t_{r_ψ} of the outer yaw loop is chosen to be a factor of 10 to 20 slower than the rise time of the inner roll loop. The yaw loop natural frequency ω_{n_ψ} is calculated from the desired yaw loop rise time. The damping ratio of the yaw loop ζ_ψ is chosen to give the desired overshoot and settling characteristics. With values for ω_{n_ψ} and ζ_ψ determined, the desired characteristic equation for the yaw loop can be expressed as

$$s^2 + 2\zeta_\psi \omega_{n_\psi} s + \omega_{n_\psi}^2 = 0.$$

Proportional and derivative gains for the yaw loop, k_{p_ψ} and k_{d_ψ} are determined by matching coefficients of the desired and actual characteristic equations. With the gains for both the roll and yaw loops calculated, the lateral successive loop closure design is complete.

Lab Assignment W.8

1. Suppose that the longitudinal controller design requirements are that the time rise is $t_{r_\theta} \approx 1.4$ sec and a damping ratio of $\zeta_\theta = 0.7$. Find the proportional and derivative gains k_p and k_d for the longitudinal controller to achieve these requirements.

2. Suppose that the inner roll loop of the lateral controller design requirements are that the time rise is $t_{r_\phi} \approx 0.3$ sec and a damping ratio of $\zeta_\phi = 0.7$. Find the proportional and derivative gains, k_{p_ϕ} and k_{d_ϕ}, for the roll loop of the lateral controller to achieve these requirements.

3. Find the DC-gain of the roll loop and draw the block diagram of the outer loop for controlling the yaw angle ψ, where the roll loop has been replaced by its DC gain.

4. Suppose that the outer yaw loop of the lateral controller has design requirements for a damping ratio of $\zeta_\psi = 0.7$. Specify the yaw loop rise time requirement t_{r_ψ} based on bandwidth separation recommendations. Find the proportional and derivative gains k_{p_ψ} and k_{d_ψ} for the yaw loop of the lateral controller to achieve these requirements.

5. Implement the longitudinal and lateral controllers in your `wb_st_controller` `controller.py` and use the rise time and damping parameters to tune the controllers as needed so that the performance specifications are met.

Hints

- Don't forget to remove the lateral constraints in `whirlybird_sim` if you haven't already.

9

System Type and Integrators

Lab Assignment W.9

1. When the inner loop controller for the whirlybird is PD control, what is the system type with respect to tracking of the inner loop? Characterize the steady state error when $\tilde{\phi}^r$ is a step, a ramp, and a parabola. What is the system type with respect to an input disturbance?

2. When the outer loop controller for the whirlybird is PD control, what is the system type with respect to tracking of the outer loop? Characterize the steady state error when $\tilde{\psi}^r$ is a step, a ramp, and a parabola. What is the system type with respect to an input disturbance?

3. When the longitudinal controller for the whirlybird is PD control, what is the system type with respect to tracking of the longitudinal loop? Characterize the steady state error when $\tilde{\phi}^r$ is a step, a ramp, and a parabola. What is the system type with respect to an input disturbance?

TOC

10

Digital Implementation of PID Controllers

The purpose of this lab is to implement the controller designed in the last lab on the hummingbird hardware.

Lab Assignment W.10

1. Make sure you pull the changes using `git pull` before starting this lab. If you cannot pull use `git stash` and `git stash pop` to update. These changes allow the parameters dynamic model to be perturbed from their nominal values, thus introducing modeling uncertainty in your dynamic model of the whirlybird. Try running your PD controller from the prior lab on this perturbed system. You should see some steady-state error in pitch due to the dynamic model now having mass parameters that are different from those used to calculate your feedforward force command.

2. In the prior lab, you were able to access the angular velocity states of the whirlybird to use for derivative feedback in your PD controller. In this lab, you will only be able to use the angular position states and you will have to calculate the derivative of your angular position states using a difference equation implementation. Implement PID control by adding derivative and integrator terms to `wb_st_controller controller.py`. Use the same rise time and damping ratio values you found while tuning in from lab assignment **??.??**. If saturation occurs, tune your rise time parameters to reduce the control gains so that saturation is eliminated. With your integrator implementation, implement an anti-windup scheme as described in Appendix e. *Pass off your simulation with the TA before proceding to the next step.*

3. Implement PID control on the whirlybird hardware. This can be done by changing the `ROS_MASTER_URI` and using the `hw.launch` file in the

`wb_st_controller` package. This results in swapping out your simulator node for the actual hardware. *Please ensure that you have a TA with you the first time you operate one of the whirlybirds. They can be dangerous!* To avoid injuries and damage to the hardware, please be cautious and careful as you use the hardware. The hardware is fragile, so please do not allow the whirlybird to go out of control. Have your lab partner stand behind the whirlybird to grab the counter-weight if necessary. Pass off your hardware implementation to the TA showing your ability to control both the pitch and the yaw of the whirlybird.

Part IV

Observer-based Control Design

11

Full-state Feedback

The purpose of this lab is to design and implement full-state feedback control strategies.

11.1 Lateral Control using Linear State Feedback

The objective of lateral control is to design a linear state feedback of the form

$$\tau = -K_{lat} x_{lat} + k^r_{lat} \psi^c$$

that places the poles of the lateral dynamics at the desired pole locations. Gain values can be calculated using the `place` command in Matlab using the A_{lat} and B_{lat} matrices and the desired pole locations. Suppose that we desire to tune the controller using the two natural frequencies ω_{n_ϕ} and ω_{n_ψ}, and the damping ratios ζ_ϕ and ζ_ψ, then the desired closed-loop poles are given by

$$\begin{aligned} p_{lat} &= [p_1, p_2, p_3, p_4] \\ &= \left[-\zeta_\phi \omega_{n,\phi} \pm j\omega_{n,\phi}\sqrt{1-\zeta_\phi^2}, -\zeta_\psi \omega_{n,\psi} \pm j\omega_{n,\psi}\sqrt{1-\zeta_\psi^2} \right]. \end{aligned}$$

In Matlab, the command `K_lat = place(A_lat, B_lat, p_lat);` can be used to calculate the lateral state feedback gains. The feedforward gain k^r_{lat} is given by the formula

$$k^r_{lat} = -\frac{1}{C_{yaw}(A_{lat} - B_{lat}K_{lat})^{-1}B_{lat}},$$

where $C_{yaw} = (0, 1, 0, 0)$.

11.2 Longitudinal Control using Linear State Feedback

The form of the controller for the longitudinal portion of the system is

$$F = F_e - K_{lon}(x_{lon} - x_{lon,e}) + k^r_{lon}(\theta^c - \theta_e).$$

The first term is the feedforward equilibrium force found in Chapter 4. The second term is the state feedback control based on the linearized state. The third term is a feedforward term that ensures that the DC-gain of the closed-loop system is one. If the equilibrium states are zero, $x_e = (\theta_e, \dot{\theta}_e)^T = (0, 0)^T$, then the controller simplifies to

$$F = F_e - K_{lon}x_{lon} + k^r_{lon}\theta^c.$$

The control design requires the placement of two poles similar to what was done in Chapter 7. Assuming that the desired closed-loop poles are characterized by a natural frequency ω_{n_θ} and damping ratio ζ_θ, they are given by

$$\begin{aligned} p_{lon} &= [p_1, p_2] \\ &= \left[-\zeta_\theta \omega_{n,\theta} \pm j\omega_{n,\theta}\sqrt{1-\zeta_\theta^2} \right]. \end{aligned}$$

In Matlab, the command `K_lon = place(A_lon, B_lon, p_lon);` can be used to calculate the longitudinal state feedback gains. The feedforward gain k^r_{lon} is given by the formula

$$k^r_{lon} = -\frac{1}{C_{lon}(A_{lon} - B_{lon}K_{lon})^{-1}B_{lon}}.$$

Lab Assignment W.11

1. Implement a method to compute your lateral gains K_{lat} and k^r_{lat} where the desired closed-loop pole locations are found by solving characteristic polynomials defined by ω_{n_ϕ}, $\omega_{n,\psi}$, ζ_ϕ and ζ_ψ. If you do this in Matlab you will need to create a Matlab script that generates the gains and then copy the gains over to your controller. To do this in Python, first familiarize yourself with the python control library. You may need to install this on your machine to get it to run. This is done with `pip install -user control`. This toolbox provides many powerful functions associated with controller design and attempts to follow matlab's own control toolbox in syntax. Once you understand how to use this toolbox, you may implement your gain calculations in your controller.

2. Wherever you calculate your gains, make sure to also have a check programmed in to verify that the systems (A_{lat}, B_{lat}) and (A_{lon}, B_{lon}) are controllable. If you're doing this in Python, this is controllability checking function..

3. Implement a method to compute the longitudinal gains K_{lon} and k^r_{lon} where the desired closed-loop pole locations are found by solving characteristic polynomials defined by $\omega_{n,\theta}$ and ζ_θ.

4. Implement the longitudinal and lateral controllers for your simulation of the whirlybird. Note: Since SS controllers are implemented differently than PID controllers, it may be a good idea to copy `controller.py` to a new file and use it as a template moving forward. Verify that you satisfy your design specifications in terms of rise time. Tune the controllers by adjusting values for damping ratio and natural frequency until you are satisfied with their performance.

Hints

12

Integrator with Full State Feedback

Lab Assignment W.12

1. Augment the lateral and longitudinal state-space representations to include a state that is the integral of the output error. For the lateral dynamics, this is the integral of the yaw error and for the longitudinal dynamics, this is the integral of the pitch error.

2. Compute the lateral gains K_{lat} and $k_{lat,I}$ using the `place` command, where the desired closed-loop pole locations are defined by ω_{n_ϕ}, $\omega_{n,\psi}$, ζ_ϕ, ζ_ψ, and the integral pole location. For starters, select the integral pole location to be $-\omega_{n,\psi}/2$.

3. Compute the longitudinal gains K_{lon}, $k_{lon,I}$ using the `place`, where the desired closed-loop pole locations are defined by $\omega_{n,\theta}$, ζ_θ, and the integral pole location. For starters, select the integral pole location to be $-\omega_{n,\theta}/2$.

4. Implement the longitudinal and lateral controllers with integral feeedback in your controller and verify that the controller performs well on the simulation. Introduce a constant disturbance to the longitudinal control (e.g., an error in the equilibrium force) and a small constant disturbance torque to the yaw control and verify that they are mitigated by the integral control implemented. This can be done by adding some constant value to your calculated force and torque. Tune the controllers by adjusting values for damping ratio, natural frequency, and integral pole until you are satisfied with their performance.

5. Implement the longitudinal and lateral controllers on the whirlybird hardware. Tune the controllers until you and the TA are satisfied with their performance.

TOC

TOC

13 Observers

Lab Assignment W.13.a

In this lab, we will be designing and implementing a full-state estimator, sometimes called a Luenberger observer, using pole placement methods. Your design will assume that only the angle states (roll, pitch, and yaw) are measured and available for feedback. Our full-state feedback requires both the angle and rate states, so we will use an estimator (or observer) to estimate the state vector for the Whirlybird.

1. Compute the lateral observer gains L_{lat} using the `place` command (in MATLAB or the python control library), where the desired observer pole locations are five times faster than the lateral controller pole locations.

2. Compute the longitudinal observer gains L_{lon} using the `place` command, where the desired observer pole locations are five times faster than the controller pole locations.

3. Using your `controller.py` file from the full-state feedback lab, implement a full-state estimator to estimate the state vector for the longitudinal and lateral dynamics the Whirlybird.

4. Implement full-state feedback controllers for longitudinal and lateral dynamics using estimated states for feedback.

5. Test your controller-observer system in simulation.

6. Implement the longitudinal and lateral observers on the whirlybird hardware in conjunction with the full-state feedback designs from a prior lab. Demonstrate their correct function. Tune estimator pole locations if needed.

TOC

Hints

- When you place your observer poles, remember to take the transpose of A and C. This is due to the mathematical duality between observability of (A, C) and controllability of (A^T, C^T).

- As with the other Whirlybird labs, when debugging, it may be beneficial to consider the longitudinal and lateral systems separately. This can be done by zeroing the state derivatives in the `dynamics.py` file for those states that you want to ignore. You can then focus on the control and estimation for each subsystem separately before combining them together.

Lab Assignment W.13.b

This lab is designed to stretch your skills in ROS and controls. So, don't be afraid to ask for help from a TA. In this lab, we will be designing a Luenberger observer and you will be writing the ROS Node to implement it.

1. By hand, draw the ROS network for the combined controller-observer system, carefully identifying the topics and message types each node will publish and subscribe two. Before, proceeding, have a TA check off your network.

2. Using `Controller.py` (state space with an integrator) as a baseline, create a new file called `Observer.py`.

3. Make necessary changes to the node name so it doesn't conflict with `Controller.py`.

4. Add in your publishers and subscribers to `Observer.py`.

5. Set the publishers and subscribers in `Observer.py`.

6. In `Observer.py`, verify that the systems (A_{lat}, C_{lat}) and (A_{lon}, C_{lon}) are observable.

7. Make callbacks for the subscribers in `Observer.py`. There should be 2 callbacks, one for each subscriber.

8. Compute the lateral observer gains L_{lat} using the `place` command, where the desired observer pole locations are five times faster than the lateral controller pole locations.

9. Compute the longitudinal observer gains L_{lon} using the `place` command, where the desired observer pole locations are five times faster than the controller pole locations.

10. Test your observer to make sure that it localizes correctly. Note: at this point if you run `rqt_graph` the system will not be closed loop, we are only testing to make sure we get a good estimate.

11. Have the TA check your `rqt_graph` and observer estimate to make sure your observer is working properly.

12. Modify `Controller.py` to subscribe to the estimate coming out of your observer, and use that estimate for your control. Note: this means you should no longer be differentiating states to get velocity estimates.

13. Test your controller-observer system in simulation.

14. Implement the longitudinal and lateral observers on the whirlybird hardware in conjunction with the full-state plus integral feedback designs from the last lab. Demonstrate their correct function. Tune estimator pole locations if needed.

Hints

- When drawing your ROS network, you should only concern yourself with 3 nodes in your system. Don't worry about the nodes that publish to Rviz.

- For the publishers/subscribers associated with your state estimate $\hat{x}$, you can use `geometry_msgs/Twist`. Basically it contains 2 Vector3 attributes. You can store the roll, pitch and yaw in linear and the derivatives in angular.

- You do not need to make a separate node for the longitudinal and latitudinal observers. These should reside in the same node.

- It's best to set up `Observer.py` with a class structure and store your state estimates as member variables.

- For help with setting the publishers and subscribers refer to `Controller.py` and rospy pub sub tutorial.

- When you place your observer poles, remember to take the transpose of A and C. This is due to the duality between observability of (A, C) and controllability of (A^T, C^T).

- `Observer.py` should have 2 subscribers and 1 publisher. See Figure 13.1 in the controlbook.

- When making the callbacks, have the callback associated with u update some member variable and have the callback associated with y_m propogate the dynamics of your observer.

- Use RK4 to propogate the dynamics of your observer. You can modify the RK4 method in your simulator to accomplish this.

- You can test your observer localization by adding in another publisher for the whirlybird estimate with message type `Whirlybird.msg`. You can view this message through Rviz to see your state estimate.

- You can streamline testing your controller-observer system by adding an additional node tag to `controller_sim.launch` and `controller_hw.launch`.

14

Disturbance Observers

Lab Assignment W.14

1. Verify that the lateral and longitudinal systems are observable using the $control.obsv()$ command.

2. Compute the estimator gains L for the lateral and longitudinal systems using the $scipy.signalimportplace_poles$ command.

3. Implement the lateral and longitudinal controllers using the estimated states $xhat$. Verify that the controller meets the time rise requirements. Tune the gains if necessary.

14.0.1 Python*

4. Modify your main.py and sim_plot.py files to plot the estimated states along with the actual states. Tune the estimator gain as necessary so that the estimated states closely approximate the real states.

14.0.2 Matlab*

4. Modify your simulink to plot the estimated states along with the actual states. Tune the estimator gain as necessary so that the estimated states closely approximate the real states.

Hints

Part V

Loopshaping Control Design

TOC

15

Frequency Response of LTI Systems

Lab Assignment W.15

Hints

TOC

16 Frequency Domain Specifications

Lab Assignment W.16

Hints

TOC

17

Stability and Robustness Margins

Lab Assignment W.17

Hints

TOC

18

Compensator Design

Lab Assignment W.18

Hints

TOC

a Modeling Dynamic Systems Using Python

The equations of motion are a set of differential equations that describe the behavior of a physical system in terms of its motion and exterior forces. They are used to calculate the current derivatives of the system which are integrated as a function of time to obtain the next states of the physical system.

Example: Inverted Pendulum

In this section we will discuss how to implement the equations of motion for the Inverted Pendulum in python. This will require

- Create a dynamics.py file to implement the equations of motion for the Inverted Pendulum.
- Modify the main.py file to include the dynamics.py file.
- Modify the slider_input.py file so that the only input is the force acting on the cart.

The equations of motion for the inverted pendulum are

$$\begin{pmatrix} (m_1 + m_2) & m_1 \ell \cos\theta \\ m_1 \ell \cos\theta & m_1 \ell^2 \end{pmatrix} \begin{pmatrix} \ddot{z} \\ \ddot{\theta} \end{pmatrix} = \begin{pmatrix} m_1 \ell \dot{\theta}^2 \sin\theta - b\dot{z} + F \\ m_1 g \ell \sin\theta \end{pmatrix} \tag{a.1}$$

They can be rearranged to solve for $\ddot{z}$ and $\ddot{\theta}$ given that z, θ, $\dot{z}$, $\dot{\theta}$, and F are known.

Let

$$M = \begin{pmatrix} (m_1 + m_2) & m_1 \ell \cos\theta \\ m_1 \ell \cos\theta & m_1 \ell^2 \end{pmatrix} \tag{a.2}$$

$$c = \begin{pmatrix} m_1 \ell \dot{\theta}^2 \sin\theta - b\dot{z} + F \\ m_1 g \ell \sin\theta \end{pmatrix} \tag{a.3}$$

TOC

then

$$\begin{pmatrix} \ddot{z} \\ \ddot{\theta} \end{pmatrix} = inv(M)c \tag{a.4}$$

Once the derivatives are known, they can be integrated with respect to time to solve for the next states of the system. This integration is done using an approximate integral.

$$\begin{pmatrix} z(t+\Delta t) \\ \theta(t+\Delta t) \\ \dot{z}(t+\Delta t) \\ \dot{\theta}(t+\Delta t) \end{pmatrix} = \begin{pmatrix} \dot{z}(t) \\ \dot{\theta}(t) \\ \ddot{z}(t) \\ \ddot{\theta}(t) \end{pmatrix} \Delta t + \begin{pmatrix} z(t) \\ \theta(t) \\ \dot{z}(t) \\ \dot{\theta}(t) \end{pmatrix}$$

The Euler's first order approximation shown is not what will be implemented, but is used to give a general idea of what they dynamics file will be doing.

This first order approximation integral is very accurate if the physical system is linear, and it can approximate non-linear physical systems with some error. This error can be reduced to zero as $\lim_{\Delta t \to 0}$. However, as the limit of Δt approaches zero, the computational demand increases drastically, making it not very efficient. Higher order approximation integrals can be used to increase accuracy while sacrificing little computational efficiency. The method that we will use is the Runge-Kutta method.

The Runge-Kutta method will not be described in detail in this appendix. However, it is encouraged that you research it to gain a better understanding of what is happening.

The file that implements the Runge-Kutta method is called *dynamics.py*. The *dynamics.py* file of the inverted pendulum has been included in Listing

```
import numpy as np
import param as P



class PendulumDynamics:

    def __init__(self):

        # Initial state conditions
        self.state = np.matrix([[P.z0],          # z initial position
                                [P.theta0],      # Theta initial orientation
                                [P.zdot0],       # zdot initial velocity
                                [P.thetadot0]]) # Thetadot initial velocity

    def propagateDynamics(self,u):
        # P.Ts is the time step between function calls.
        # u contains the force and/or torque input(s).

        # RK4 integration
        k1 = self.Derivatives(self.state, u)
        k2 = self.Derivatives(self.state + P.Ts/2*k1, u)
        k3 = self.Derivatives(self.state + P.Ts/2*k2, u)
        k4 = self.Derivatives(self.state + P.Ts*k3, u)
        self.state += P.Ts/6 * (k1 + 2*k2 + 2*k3 + k4)


    # Return the derivatives of the continuous states
    def Derivatives(self,state,u):

        # States and forces
        z = state.item(0)
        theta = state.item(1)
        zdot = state.item(2)
        thetadot = state.item(3)
        F = u[0]

        # ctheta and stheta are used multiple times. They are
        # precomputed and stored in another variable to increase
        # efficiency.
        ctheta = np.cos(theta);
        stheta = np.sin(theta);

        # The equations of motion.
        M = np.matrix([[P.m1+P.m2,         P.m1*P.ell*ctheta],
                       [P.m1*P.ell*ctheta, P.m1*P.ell**2    ]])

        C = np.matrix([[P.m1*P.ell*thetadot**2*stheta + F-P.b*zdot],
                       [P.m1*P.g*P.ell*stheta                     ]])

        tmp = np.linalg.inv(M)*C


        zddot = tmp.item(0)
        thetaddot = tmp.item(1)

        xdot = np.matrix([[zdot],[thetadot],[zddot],[thetaddot]])

        return xdot


    # Returns the observable states
    def Outputs(self):
        # Return them in a list and not a matrix
        return self.state[0:2].T.tolist()[0]

    # Returns all current states
    def States(self):
        # Return them in a list and not a matrix
        return self.state.T.tolist()[0]
```

Listing a.1: Inverted Pendulum Dynamics.py file

When the class is instantiated, a 4 X 1 matrix called *state* is created that holds the initial conditions of the states imported from the *param.py* file.

The function propagateDynamics(self,u) implements the Runge-Kutta method. The parameter u that is passed into the function is a list containing the force and/or torque inputs. In this case, u is a list of size 1 containing the current force being applied to the cart of the inverted pendulum. The parameter u is passed into the function Derivatives(self,state,u that is used to implement the Runge-Kutta method.

The function propagateDynamics(self,u) should never need to be modified.

The function Derivatives(self,state,u) begins by unpacking the current states and forces. It then implements equations a.2, a.3, and a.4. It finishes by packing up the derivatives in a numpy.matrix and returns the matrix.

The last two functions Outputs(self) and States(self) are used to return states (notice that they are converted from a numpy.matrix to list type) to the *main.py* file.

The function Outputs(self) returns only the observable states (the states that are being measured). In the case of the Inverted Pendulum, only z, and θ are observable.

The function States(self) returns all of the states. This function can be used for debugging or other purposes that will be made known later.

In order for us to use the *dynamics.py* file, the *main.py* file needs to be modified to include it. Listing shows the modifications of the *main.py* file.

```
import time
import sys
import numpy as np
import matplotlib.pyplot as plt
import param as P
from slider_input import Sliders

# The Animation.py file is kept in the parent directory,
# so the parent directory path needs to be added.
sys.path.append('..')
from dynamics import PendulumDynamics
from animation import PendulumAnimation

t_start = 0.0   # Start time of simulation
t_end = 50.0    # End time of simulation
t_Ts = P.Ts     # Simulation time step
t_elapse = 0.1  # Simulation time elapsed between each iteration
t_pause = 0.01  # Pause between each iteration

user_input = Sliders()              # Instantiate Sliders class
simAnimation = PendulumAnimation()  # Instantiate Animate class
dynam = PendulumDynamics()          # Instantiate Dynamics class

t = t_start               # Declare time variable to keep track of simulation time elap

while t < t_end:

  plt.ion()                           # Make plots interactive
  plt.figure(user_input.fig.number)   # Switch current figure to user_input figure
  plt.pause(0.001)                    # Pause the simulation to detect user input

  # The dynamics of the model will be propagated in time by t_elapse
  # at intervals of t_Ts.
  t_temp = t +t_elapse
  while t < t_temp:
    dynam.propagateDynamics(        # Propagate the dynamics of the model in time
37        user_input.getInputValues())
38      t += t_Ts                       # Update time elapsed
39
40    plt.figure(simAnimation.fig.number) # Switch current figure to animation figure
41    simAnimation.drawPendulum(          # Update animation with current user input
42      dynam.Outputs())
43
44
45    # time.sleep(t_pause)
```

Listing a.2: Inverted Pendulum main.py file

First, notice in line 11 that the *dynamics.py* file is kept in the parent directory. This is because the same *dynamics.py* file will be used for all future assignments. On line 22, the PendulumDynamics class is instantiated.

The file now contains two while-loops. The outer while-loop runs the interior code until the simulation time runs out. The first lines in the outer while-loop, lines 28-30, makes the figure that displays the sliders interactive. On line 34 the variable t_temp is created. This variable is used by the inner while-loop.

Each iteration of the inner while-loop represents t_Ts seconds in simulation time. During each iteration, the user input values (F in the case of the Inverted Pendulum) are passed into the function propagateDynamics(). This is when the dynamics of the physical system are updated. Once the inner while-loop ends, the animation is updated and displayed.

The reason why there are two while-loops is to allow the user to have the dynamics update at a faster rate than the animation. This will drastically improve simulation speed, since rendering an image every t_Ts simulation seconds is computationally heavy. Basically, t_elapse controls how frequently user input in sampled and the animation is update in simulation time, and t_Ts controls the rate at which the dynamics of they system are update in simulation time.

We will not discuss how to modify the *slider_input.py* file since it is a simple modification.

The complete collection of file for the Inverted Pendulum can be found on the Control Book website.

Lab Assignment W.a

The equations of motion for the Whirlybird are

$$M(q)\ddot{q} + c(q,\dot{q}) + \frac{\partial P}{\partial q} = Q, \qquad \text{(a.5)}$$

where

$$M(q) = \begin{pmatrix} J_x & 0 & -J_x s_\theta \\ 0 & m_1\ell_1^2 + m_2\ell_2^2 + J_y c_\phi^2 + J_z s_\phi^2 & (J_y - J_z)s_\phi c_\phi c_\theta \\ -J_x s_\theta & (J_y - J_z)s_\phi c_\phi c_\theta & (m_1\ell_1^2 + m_2\ell_2^2 + J_y s_\phi^2 + J_z c_\phi^2)c_\theta^2 + J_x s_\theta^2 \end{pmatrix},$$

$$c(q,\dot q) = \begin{pmatrix} -\dot\theta^2(J_z-J_y)s_\phi c_\phi + \dot\psi^2(J_z-J_y)s_\phi c_\phi c_\theta^2 \\ -\dot\theta\dot\psi c_\theta\left[J_x-(J_z-J_y)(c_\phi^2-s_\phi^2)\right] \\ --- \\ \dot\psi^2 s_\theta c_\theta\left[-J_x+m_1\ell_1^2+m_2\ell_2^2+J_y s_\phi^2+J_z c_\phi^2\right] \\ -2\dot\phi\dot\theta(J_z-J_y)s_\phi c_\phi - \dot\phi\dot\psi c_\theta\left[-J_x+(J_z-J_y)(c_\phi^2-s_\phi^2)\right] \\ --- \\ \dot\theta^2(J_z-J_y)s_\phi c_\phi s_\theta - \dot\phi\dot\theta c_\theta\left[J_x+(J_z-J_y)(c_\phi^2-s_\phi^2)\right] \\ -2\dot\phi\dot\psi(J_z-J_y)c_\theta^2 s_\phi c_\phi + 2\dot\theta\dot\psi s_\theta c_\theta\left[J_x-m_1\ell_1^2-m_2\ell_2^2-J_y s_\phi^2-J_z c_\phi^2\right] \end{pmatrix},$$

$$\frac{\partial P}{\partial q} = \begin{pmatrix} 0 \\ (m_1\ell_1-m_2\ell_2)gc_\theta \\ 0 \end{pmatrix},$$

and

$$Q = \begin{pmatrix} d(f_l-f_r) \\ \ell_1(f_l+f_r)c_\phi \\ \ell_1(f_l+f_r)c_\theta s_\phi + d(f_r-f_l)s_\theta \end{pmatrix}.$$

Thus,

$$\ddot q = \begin{pmatrix} \ddot\phi \\ \ddot\theta \\ \ddot\psi \end{pmatrix} = M(q)^{-1}\left(Q - c(q,\dot q) - \frac{\partial P}{\partial q}\right).$$

PYTHON

1. The simulation is difficult to maneuver using sliders for f_l and f_r. We can improve things slightly by converting to input sliders for force F and torque τ applied to the head of the whirlybird. Note that the applied torque about the roll angle is $\tau = d(f_l - f_r)$ and the applied force is $F \stackrel{\triangle}{=} f_l + f_r$. In matrix form you can write

$$\begin{pmatrix} F \\ \tau \end{pmatrix} = \begin{pmatrix} 1 & 1 \\ d & -d \end{pmatrix}\begin{pmatrix} f_l \\ f_r \end{pmatrix}.$$

Inverting the matrix we get

$$\begin{pmatrix} f_l \\ f_r \end{pmatrix} = \begin{pmatrix} \frac{1}{2} & \frac{1}{2d} \\ \frac{1}{2} & -\frac{1}{2d} \end{pmatrix}\begin{pmatrix} F \\ \tau \end{pmatrix}.$$

Therefore, given a desired F and τ, the required right and left forces are

$$f_l = \frac{1}{2}F + \frac{1}{2d}\tau$$
$$f_r = \frac{1}{2}F - \frac{1}{2d}\tau$$

Create a function called convertForces in *main.py* with the parameter u. The parameter u is the output from the sliders (F and τ in a list). The function will convert F and τ to fl and fr, and return fl and fr in a list. The output of this function will be passed into the function propagateDerivative().

2. In your *param.py* set all of the initial conditions to 0. $\phi_0 = \theta_0 = \psi_0 = \dot{\phi}_0 = \dot{\theta}_0 = \dot{\psi}_0 = 0$

3. Modify the *slider_input.py* file to have two sliders. Let slider 1 be force and slider 2 be torque. Set the initial value of slider 1 (force) to $F = 5.22$. This is the equilibrium force when all of the initial conditions are set to zero. When the simulation is initiated, the whirlybird should not move from its equilibrium state until the force is changed or torque is added.

4. Create a *dynamics.py* file for the whirlybird. Add the equation of motions to the function Derivatives(), and make sure that the input, u, to the function is a list containing fl and fr.

5. Test the simulation by verifying that the whirlybird does not move from equilibrium when a force of 5.22 N is applied. If it begins to move, something is wrong. Go back and carefully check the equations of motion for the whirlybird.

MATLAB

1. Rename your Simulation animation file whirlysim.slx, and create a new matlab file called param.m

2. Enter all of the parameters from Appendix d into the parameter file. A trick that can be used to enable all future parameters to be passed into the simulink model using only one argument, is to put all parameters in a structure. For example, I use the structure "P" to denote parameters. So $P.m1$ is the first mass, etc. The feedback gain defined above can be labeled $P.K$.

3. Using the equations of motion defined above, modify the s-function to create a model of the whirlybird system using f_l and f_r as the inputs and $x = (\phi, \dot{\phi}, \theta, \dot{\theta}, \psi, \dot{\psi})^T$ as the state and the output. Set the initial conditions to 0. $\phi_0 = \theta_0 = \psi_0 = \dot{\phi}_0 = \dot{\theta}_0 = \dot{\psi}_0 = 0$

4. Connect the s-function to the animation block developed in last week's lab.

5. The simulation is difficult to maneuver using sliders for f_l and f_r. We can improve things slightly by converting to input sliders for force F and torque τ applied to the head of the whirlybird. Note that the applied torque about the roll angle is $\tau = d(f_l - f_r)$ and the applied force is $F \overset{\triangle}{=} f_l + f_r$. In matrix form you can write

$$\begin{pmatrix} F \\ \tau \end{pmatrix} = \begin{pmatrix} 1 & 1 \\ d & -d \end{pmatrix} \begin{pmatrix} f_l \\ f_r \end{pmatrix}.$$

Inverting the matrix we get

$$\begin{pmatrix} f_l \\ f_r \end{pmatrix} = \begin{pmatrix} \frac{1}{2} & \frac{1}{2d} \\ \frac{1}{2} & -\frac{1}{2d} \end{pmatrix} \begin{pmatrix} F \\ \tau \end{pmatrix}.$$

Therefore, given a desired F and τ, the required right and left forces are

$$f_l = \frac{1}{2}F + \frac{1}{2d}\tau$$
$$f_r = \frac{1}{2}F - \frac{1}{2d}\tau$$

Use the simulink matrix gain block to convert input sliders in F and τ into the system inputs f_1 and f_r. The equilibrium force for the system will be $F = 5.22N$.

6. Test the simulation by verifying that the whirlybird does not move from equilibrium when a force of 5.22 N is applied. If it begins to move, something is wrong. Go back and carefully check the equations of motion for the whirlybird.

Hints

PYTHON

1. The hummingbird has three degrees of motion ϕ, θ, and ψ. These degrees of motion along with their derivatives ($\dot{\phi}$, $\dot{\theta}$, and $\dot{\psi}$ are the states that we are interested in tracking, and make up the self.state variable that is in the *dynamics.py* file.

2. To invert a matrix use numpy.linalg.inv(MATRIX)

3. The observable states are ϕ, θ, and ψ.

MATLAB

1. Your file called param.m should be a Matlab script. Matlab scripts do not have input or output arguments. Run the script before starting your simulink simulation. Type help script at the matlab prompt or view the Matlab online documentation for more information about scripts. If the script changes, then you will need to re-run param.m. Another option is to click on the small wheel in the simulink diagram and select "Model Properties" as shown in Figure 1-1, and then select "Callbacks" and "InitFcn" as shown in Figure 1-2. Typing param.m in the window will cause the script to be executed each time the simulation is initialized, i.e., at the beginning of each simulation.

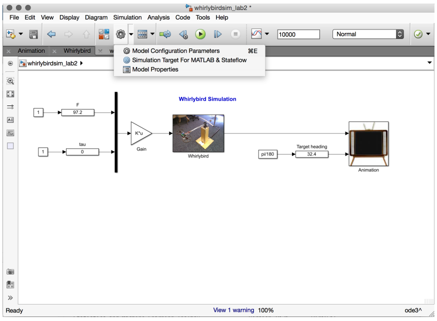
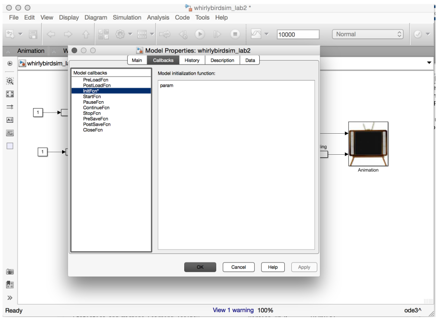


Figure 1-1: Select model properties, to auto run param.m at the beginning of each simulation.



Model initialization function:

param

Figure 1-2: In the InitFcn callback, type param. This will cause simulink to execute the script param.m at the beginning of each simulation run.

2. Remember to check the inputs of your function that plots the hummingbird. The input vector u was 3×1 for lab 1 and now it will be 6×1: ϕ, θ, ψ, $\dot{\phi}$, $\dot{\theta}$, $\dot{\psi}$.

3. Inside your Simulink S-Function code you should be making changes to mdlInitializeSizes, mdlDerivatives, and mdlOutputs.

6

supportFiles

This appendix discusses the support files: slider_input.py, signal_generator.py, and sim_plot.py which are used to simulate in Python.

slider_input.py

The *slider_input.py* file that will be looked at is from the example discussed in the section Animation Example: Inverted Pendulum **??** . This file has been provided to make organizing multiple sliders on one axes simple. The file contains three classes: Sliders, mySlider and Slider. The Sliders class is the parent class that inherits mySlider. It organizes one or more instances of mySlider class based on the parameters set by the user, and is the only class the user needs to interact with. The mySlider class uses the parameters set in Sliders to organize multiple Slider classes in one axes. An inheritance diagram has been included to help visualize the structure.

(Silders → mySlider → Slider)

The file might look long and complex, but the user only needs to be familiar with the three sections of the file that are surrounded by two rows of the # sign.

Section 1

The first section only contains the variable self.num_of_sliders. This variable indicates the number of sliders that will be created and displayed in the same axes. The Inverted Pendulum has two inputs: θ and z. This means that there needs to be a slider for each input with a total number of two sliders.

Section 2

In the second section, the objects of mySlider class are created for each input. The mySlider class takes in 7 parameters. These parameters have default values which the user can customize. The default values allow the users to not have to pass in every parameter. If a parameter is not passed in, the class assumes the default value. Also, the parameters can be passed in by order or by name. Both ways are demonstrated in section 2 of Listing b.1.

mySlider(self,num_of_sliders=1, slider_number=1, maxV=1,minV =-1,intV=0,gain=1,name='Slider')

num_of_sliders - The total numbers of sliders that will be rendered.

slider_number - Each slider object will have a unique slider number. This determines the relative position of a specific slider to other sliders. Lower number sliders will be placed above higher number sliders.

maxV - The maximum value the slider can have.

minV - The minimum value the slider can have.

intV - The initial value of the slider.

gain - The value of the slider is multiplied by gain before it is passed to the parent class. This allows for unit conversion or scaling. For example, the slider values may be in degrees, but the gain can convert it from degrees to radians as done in this example.

name - The name of the slider.

In this example, an object of mySlider class needs to be created for z and θ. Slider z will be placed above slider θ since its slider_number is lower than $\theta's$ slider_number. Both of them will have a maxV of 90, minV of -90, initV of 0, and a gain of $\frac{\pi}{180}$ to convert the value from degrees to radians.

Section 3

Section 3 resides in the function getInputValues which returns a list of the sliders' values to the file *main.py*. In Section 3, the user can organize the order in which the values are returned.

In this example, since there are two sliders (z and θ), only the two values from them will be returned. The value from slider z will be element 0 and the value from slider θ will be element 1 in the list object that is returned.

```
import numpy as np
import param as P
import matplotlib.pyplot as plt
from matplotlib.widgets import Slider

class Sliders:
    ''' This class inherits the Slider class. Its purpose is to
        help organize one or more sliders in an axes.'''

    # You only need to modify the code inside '#'s as shown below.
    ##################################################################
    #                       CODE TO BE MODIFIED
    ##################################################################

    def __init__(self):
        # Creates a figure and axes for the Sliders.
        self.fig,self.ax = plt.subplots()
        plt.axis("off")               # Suppress the axis from showing.

        # SECTION 1
        ###############################################################
        self.num_of_sliders = 2 # The number of Slider objects that
                                # will be used. The minimum value is 1
        ###############################################################

        ''' Down below you will create Slider objects for each slider
            you need. The format for each object is as follows:
            self.Object1 = Slider(self.num_of_sliders, slider_number,
                maxV = 1,minV = -1,intV = 0, gain = 1, name = 'Slider')
            self.Object2 = Slider(self.num_of_sliders, slider_number,
                maxV = 1,minV = -1,intV = 0, gain = 1, name = 'Slider')
                ...
            self.ObjectN = Slider(self.num_of_sliders, slider_number,
                maxV = 1, minV = -1,intV = 0, gain = 1, name = 'Slider')

            For a description of each parameter, see the Slider Class
            below.'''


        # SECTION 2
        ################################################################
        # Instantiate objects here
        self.Sz = mySlider(self.num_of_sliders,1,
            2*P.ell,-2*P.ell,P.z0,1,'z')
        self.Stheta = mySlider(num_of_sliders = self.num_of_sliders,
            slider_number = 2, maxV = 90,minV = -90,intV =0,
            gain = np.pi/180, name ='theta')

        ################################################################

        plt.draw()


    def getInputValues(self):
        ''' This function is called to return the sliders' values.
            The variable values is a list that contains the values of
            all of the sliders. You will need to insure that this
            function returns the values in the following format.
            values = [self.object1.getValue(),
                self.object2.getValue(),...,
                self.ObjectN.getValue] '''

        # SECTION 3
        ################################################################
        values = [self.Sz.getValue(),self.Stheta.getValue()]
        ################################################################
        return values

```

```


# You do not need to modify the class below. However, it would be good
# to become familiar with it.

class mySlider:

    def __init__(self,num_of_sliders=1, slider_number=1, maxV = 1,
        minV = -1, intV = 0, gain = 1, name = 'Slider'):
        # num_of_sliders - The number of Sliders that you will use
        # slider_number - This number starts at 1 and its max value
        #                 should be num_of_sliders . This variable
        #                 determines the location of the Slider in
        #                 the axes; 1 corresponds to the top Slider
        #                 in the axes and largest number corresponds
        #                 to the bottom of the axes. Each Slider
        #                 object needs a unique slider_number.

        self.data = 0            # Current value of Slider
        self.name = name         # Name of Slider
        self.maxValue = maxV     # Max value of the Slider
        self.minValue = minV     # Min value of the Slider
        self.initValue = intV    # Initial value of Slider
        self.gain = gain         # The gain of the Slider can be used for
                                 # conversion purposes. EX, If you want the
                                 # Slider to be in terms of degrees but the
                                 # value passed in radians. The max and min
                                 # values will in terms of degrees and the
                                 # gain would be pi/180 to convert
                                 # the Slider's value from degrees to radians.
        self.Slider_length = 0.5 # This is the length of the Slider in terms
                                 # of the percentage of the figure's length
        self.Slider_width = 0.03 # This is the width of the Slider in terms
                                 # of the percentage of the figure's width


        # Sets the position and color of the Slider

        # The horizontal position of the Slider in the figure.
        hpos = 0.8 - 0.6/(num_of_sliders+1)*slider_number

        # Specify a subsection of the current axes where the slider will go.
        self.axSlider = plt.axes([0.5-self.Slider_length/2,hpos, self.Slider_length,
            self.Slider_width], axisbg = 'orange')

        # Instantiate a slider, and create a handle to it
        self.SliderHandle = Slider(self.axSlider, self.name, self.minValue,
            self.maxValue, valinit = self.initValue)
        self.data = self.initValue

        # When a change occurs on the slider, the function self.update
        # will be called.
        self.SliderHandle.on_changed(self.update)



    # Updates the value of the Slider when the Slider is changed
    def update(self,val):
        self.data = self.SliderHandle.val

    # Returns the current value of the Slider(s)
    def getValue(self):
        return self.data*self.gain
```

Listing b.1: Slider Input File

step.py

In this section we will discuss the step.py file that can be used to generate step functions with a specified amplitude and time delay. Many of the assignments will ask you to implement a step funciton, you can use this file, modify it, or create your own.

```
def step_function(t,T,A):
  # t is the sim time.
  # T is the sim time when the step
  # function assumes the value of A
  # A is the amplitude of the step function
  u = 0;
  if t > T:
    u = A
  return u
```

Listing b.2: step.py

As you can see from listing b.2, the function *step_function* accepts three parameters: t, T, and A. t is the current simulation time, T is the time delay in simulation time, and A is the amplitude of the step function. This file is fairly straight forward. For an example of this file, look at the robot arm design study assignment 7 that is on the Control Book Website.

Appendix b.1 sim_plot.py

In this section we will discuss the sim_plot.py file that can be used to plot key data. This section uses an example based off of the Robot Arm assignment 7 that is discussed in Appendix e Section e.

An example of the sim_plot.py file is shown in Listing b.3.

```
import matplotlib.pyplot as plt
from matplotlib.lines import Line2D
import numpy as np


class plotGenerator:
    ''' The purpose of this class is to organize multiple plots in one
        figure using subplots.The plots will be organized in n x m
        dimensions in the figure; where n represents the number of rows,
        and m represents the number of columns. Ex, a subplot with
        dimensions 3 x 2 will hold 6 plots. 3 plots per row, and 2 plots
        per column.

        Not every plot needs to be used. Ex, if you were to plot only
        5 plots, you would still need dimensions 3 x 2 and one of the
        plots would never be populated.'''

    # You only need to modify the code inside '#'s as shown below.
    ##################################################################
    #                       CODE TO BE MODIFIED
    ##################################################################
    def __init__(self):
        #SECTION 1
        ##############################################################
        # Number of subplots = num_of_rows*num_of_cols
        self.num_of_rows = 2    # Number of subplot rows
        self.num_of_cols = 1    # Number of subplot columns
        ##############################################################

        # Crete figure and axes handles
        self.fig, self.ax = plt.subplots(self.num_of_rows,
            self.num_of_cols, sharex=True)

        # A list to hold the time history
        self.time_history = []

        # The list type variable will store your plot objects
        self.handle = []

        ''' Create the handles below.
            The class plotGenerator inherits the class myPlot, and
            organizes one or more instances of myPlot in a figure. The
            myPlot class organizes one or more line objects in a single
            axes. The syntax for myPlot is

            myPlot(axes,gain,xlabel,ylable,etc)

            To see a full description of myPlot, read the comments located
            in the class myPlot. However, note that all arguments have
            default values except the axes argument.
            (i.e., A myPlot object could easily be created by

            myPlot(self.ax[0])

            The class myPlot also supports pass-by-reference and
            pass-by-name. Examples are given below.

            self.handle.append(myPlot(self.ax[0],2,'xlabel','ylabel',
                                   'title'))
            self.handle.append(myPlot(self.ax[1],180.0/np.pi,
                                   title="y_r/y")
            self.handle.append(myPlot(self.ax[2],1,title="Force",
                                   legend=("Force","Torque"))

            The myPlot class also comes with default line and color
            styles. You can either specify the line or color styles
            that you want when creating the myPlot object or you can
            modify the default values that are contained within the
            myPlot class.
            ’’’
        # Section 2
        ####################################################################
        self.handle.append(myPlot(self.ax[0],180.0/np.pi,’t(s)’, ’deg’,
                                 ’theta_r/theta’))
        self.handle.append(myPlot(self.ax[1],1,’t(s)’,’Nm’,’torque’))
        ####################################################################


    # Update the history
    def updateDataHistory(self,new_t, new_data):

        """
            This function updates the data history of all the plots.
            - new_t: The current simulation time.
            - new_data: Is a list of data lists for each plot. The
                        order of the data must be the same order in which
                        the classes myPlot were created.

            Ex: There are two subplots. Instantiated as shown below

                self.handle.append(myPlot(self.ax[0],180.0/np.pi,
                                  ’t(s)’, ’deg’,’theta_r/theta’))

                self.handle.append(myPlot(self.ax[1],1,’t(s)’,
                                  ’Nm’,’torque’))

            The first subplot plots theta_r and theta and the second subplot
            plots Torque. Since this is the order in which the myPlot
            classes were appended to the handle list, this is the order
            in which the data must be passed.

            Continuing the example:

            new_data = [[theta_r,theta],[torque]]

            Notice that new_data is a list of lists. The first inner list
            contains the data meant for the first plot, and the second
            inner list is meant for the second plot.


            """

        # Add the new time data
        self.time_history.append(new_t)

        # Update all other data
        for i in range(len(self.handle)):
            self.handle[i].updateHistory(new_data[i])

    # Renders the data to the plots
    def update_plots(self):
        for i in range(len(self.handle)):
            self.handle[i].update_plot(self.time_history)
```

```
class myPlot:
    ’’’ This class organizes one or more line objects on one axes’’’

    def __init__(self, ax, gain = 1,xlabel = ’x-axis’,
                 ylabel = ’y-axis’,title = ’title’,
                 colors = [’b’,’r’,’g’,’c’,’m’,’y’,’b’],
                 line_styles = [’-’,’--’,’-.’,’:’],
                 legend = None, grid = True):

        ’’’ ax - This is a handle to the an axes of the figure
            gain - a scalar variable used on the data. This can be used
                        to convert between units. Ex. to convert from
                        radians to degrees, the gain should be 180/np.pi
            colors - Indicates the line color. If there are multiple lines,
                     colors can be a list of different colors. Below is a
                     list of options. Note that they are strings.

                    ’b’ - blue
                    ’g’ - green
                    ’r’ - red
                    ’c’ - cyan
                    ’m’ - magenta
                    ’y’ - yellow
                    ’k’ - black

            line_style - Indicates the line style. If there are multiple
                     lines, line_style can be a list of different line
                     styles. Below is a list of options. Note that they
                     are strings.

                     ’-’  - solid line
                     ’--’ - dashed line
                     ’-.’ - dash_dot
                     ’:’  - dotted line

            legend - A tuple of strings that identify the data.
                     EX: ("data1","data2", ... , "dataN")

            xlable - Label of the x-axis
            ylable - Label of the y-axis
            title - Plot title
            gird - Indicates if a grid is to be overlapped on the plot
        ’’’


        self.legend = legend
        self.data_history = []        # Will contain the data history
        self.ax = ax                  # Axes handle
        self.gain = gain              # The scales the data
        self.colors = colors          # A list of colors.
                                      # The first color in the list
                                      # corresponds to the first line
                                      # object.
        self.line_styles=line_styles  # A list of line styles.
                                      # The first line style in the list
                                      # corresponds to the first line
                                      # object.
        self.line = []

        # Configure the axes
        self.ax.set_ylabel(ylabel)
        self.ax.set_xlabel(xlabel)
        self.ax.set_title(title)
        self.ax.grid(grid)


        # Keeps track of initialization
        self.init = True


    # Adds the new data to the data history list after
    # scaling it by the gain.
    def updateHistory(self,new_data):
        # new_data: a list containing the new data.
        # Ex: new_data = [theta_r, theta]
        self.data_history.append([t*self.gain for t in new_data])

    def update_plot(self,time_history):

        # If it is being initialized
        if self.init == True:

            # size contains the number of line objects to create
            size = len(self.data_history[0])

            for i in range(size):
                # zip rearranges the list
                data = zip(*self.data_history)

                # Instantiate line object and add it to the axes
                self.line.append(Line2D(time_history,data[i],
                    color = self.colors[np.mod(i,len(self.colors)-1)],
                    ls = self.line_styles[np.mod(i,len(self.line_styles)-1)],
                    label = self.legend[i] if self.legend != None else None))

                self.ax.add_line(self.line[i])

            self.init = False
            if self.legend != None:
                plt.legend(handles=self.line)
        else: # Add new data to the plot

            # zip rearranges the list
            data = zip(*self.data_history)

            # Updates the x and y data of each line.
            for i in range(len(self.line)):
                self.line[i].set_xdata(time_history)
                self.line[i].set_ydata(data[i])

```

```
238        # Adjusts the axis to fit all of the data.
239        self.ax.relim()
240        self.ax.autoscale()
```

Listing b.3: robotArm/hw_7/sim_plot.py

The sim_plot.py file contains the parent class plotGenerator which organizes multiple child classes called myPlot. The class plotGenerator organizes multiple plots in one figure using subplots. The plots are organized in n x m dimensions; n represents the number of rows and m represents the number of columns. For example, a figure containing subplots with dimensions 3 x 2 will hold 6 plots. 3 plots per row, and 2 plots per column.

Not every subplot needs to be used. Ex, if you were to plot only 5 plots, you would need dimensions 3 x 2 and one of the plots would never be populated.

The subplot’s dimensions are indicated on Lines 23-38 using the variable $self.num\_of\_rows$ and $self.num\_of\_cols$. Notice that these variables are surrounded by ’#’ indicating that this section needs to be modified by the user. The other section that needs to be modified by the user starts on Line 72 and is labeled Section 2.

In Section 2, the multiple instances child class myPlot are created and their object handles are placed inside the list $handle$. The order in which the myPlot object handles are placed inside the list $handle$ determines the order in which data needs to be passed in (this will be discussed later).

The syntax for the class myPlot is

```
myPlot(ax, gain, xlabel, ylabel, title, colors, line_styles, legend, grid).
```

All of these parameters have default values except the parameter ax. A description of each parameter is given below.

- ax - This is a handle to one of the axes on the figure.

- gain - This is used to scale the data that is passed to the plot. This can also be used to convert between units. Example, to convert from radians to degrees, the gain should be $\frac{180.0}{\pi}$. This parameter has a default value of 1.

- colors - This indicates the line colors. If there are multiple lines on a given axes, $colors$ can be a list of containing different colors. Example, colors = [’b’,’r’]. This parameter has a default value of [’b’,’r’,’g’,’c’,’m’,’y’,’b’]. There are more color option that can be found online.

- line_style - Indicates the line style. If there are multiple lines on a give axes, the first line style in $line\_style$ corresponds to the first line. The default value of this parameter is [’-’,’–’,’-.’,’:’].

- legend - The parameter is used to add a legend on the axes. It must be a tuple of strings that identify the data. Example, ("data1","data2","data3"). The default value is no legend.

- xlabel - The label of the x-axis
- ylabel - The label of the y-axis
- title - The title of the plot
- grid - Indicates if a grid is to be overlapped on the plot.

The axes's properties are mostly set up in lines 172-193. You can read the file for more information on these properties.

The parent class plotGenerator uses two functions to pass data to the child class myPlot: updateDataHistory and update_plots.

The function updateDataHistory updates the plot's data with new data, but it does not render the data to the plot. This function is passed the parameter new_t and new_data. new_t is a float containing the current simulation time and new_data is a list of lists containing data for each plot. The order of the data must be the same order in which the myPlot object handles were passed to the list $handles$.

In this example, there are two subplots created in Section two starting on line 71. The first subplot plots $theta_r$ and $theta$ as a function of time, and the second subplot plots $torque$ as a function of time. Since this is the order in which they myPlot classes were appended to the list $handle$, this is the order in which the data must be passed.

Continuing the example:

$$new\_data = [[theta_r, theta], [torque]]$$

Notice that new_data is a list of lists. The first inner list contains the data meant for the first subplot, and the second inner list is meant for the second subplot.

In Line 114, new_t is added to $time\_history$, and in lines 116-118, the plots are updated with the new data.

The function update_plots starts on line 121. It renders the plot's data to the axes. The two tasks performed by updateDataHistory and update_plots are kept separate to allow the data history to be updated at a different rate than the plots. This will increase efficiency.

The rest of the file will not be discussed, but it is recommended that you look thought it to understand how it works. Using the file is rather simple once it is understood since the user only needs to modify two sections of it.

signal_generator.py

In this section we will discuss the signal_generator.py file that can be used to generate signals with a specified amplitude, frequency and offset. This section uses an example based off of the inverted pendulum design study assignment 8 that is discussed in Appendix e Section e. The listing for this can be seen below.

```
import numpy as np
from scipy import signal

class Signals:
  ’’’ This class inherits the Signal class. It is used to organize
    1 or more signals of different types: square_wave,
    sawtooth_wave, triangle_wave, random_wave.’’’

  # You only need to modify the code inside ’#’s as shown below.
  ##################################################################
  #                       CODE TO BE MODIFIED
  ##################################################################
  def __init__(self):
    self.handle = []  # This is the handle to various signals

    ’’’ Down below you will create mySignal objects for each signal
    you need. The syntax for each object is as follows:
    self.Object1 = Signal(A, F,signal_type, phase)
    self.Object2 = Signal(A=1, F=0.1, signal_type, phase)
    ...
    self.ObjectN = Signal(A, F)

    The class MySignal parameters can be passed by reference or name.
    The parameters also have default values as shown below. This makes
    creating a myClass simple if the default values are wanted.

    self.handle.append(mySignal())

    For a description of each parameter, see the mySignal Class
    below.

    All mySignal objects need to be appended to the list self.handle ’’’

  #SECTION 1
  ##################################################################
    self.handle.append(mySignal(0.5,0.01)) # z
  ##################################################################

  # This function returns the values of the signal generator as
  # as a function of time. The order in which the values are returned is
  # the same order that the mySignal objects were appended to the
  # list self.handle.
  def getRefInputs(self,t):
    ref_inputs = []
    for i in range(len(self.handle)):
      ref_inputs.append(self.handle[i].signal(t))
    return ref_inputs


class mySignal:

  def __init__(self,A = 1,f = 1,offset = 0, signal_type = ’square_wave’, phase = 0):
    self.A = A                  # Amplitude of the signal
    self.f = f                  # Frequency of the signal, Hz
    self.phase = phase          # Phase of the signal, Radians
    self.offset = offset        # Offset of signal
```

```
57
58     if signal_type == 'square_wave':
59       self.signal = self.square_wave
60     elif signal_type == 'sawtooth_wave':
61       self.signal = self.sawtooth_wave
62     elif signal_type == 'triangle_wave':
63       self.signal = self.triangle_wave
64     elif signal_type == 'random_wave':
65       self.signal = self.random_wave
66     else:
67       print("input signal type not recognized")
68
69   def square_wave(self,t):
70       return self.A*signal.square((2*np.pi*self.f*t+self.phase), duty = 0.5) +self.offs
71
72   def sawtooth_wave(self,t):
73     return self.A*signal.sawtooth((2*np.pi*self.f*t + self.phase), width = 0)+self.offs
74
75   def triangle_wave(self,t):
76     return self.A*signal.sawtooth((2*np.pi*self.f*t + self.phase), width = 0.5)+self.of
77
78   def random_wave(self,t):
79     return self.A*np.random.rand()+self.offset
```

Listing b.4: invertedPendulum/hw_8/signal_generator.py

The parent class Signals organizes multiple instances of the child class mySignal. The class mySignal begins on line 50 of Listing b.4, and its syntax is

```
mySignal(A,f,offset,signal_type,phase)
```

A is the amplitude of the signal, f is the frequency of the signal in Hz, $offset$ is the offset of the signal, $signal\_type$ is the type of signal, and $phase$ is the phase of the signal. There are four different signal types that are available: square wave, sawtooth wave, triangle wave, and random wave. The random wave signal can represent random noise.

The class mySignal is instantiated in the parent class on line 34. Notice how this section is surrounded by '#' indicating that it is a section that the user needs to modify. In Section 1, all of the mySignal objects are created and placed in a list of handles. The order in which the objects handles are placed in the list $handles$ determines the order in which the input reference is returned to the main function.

In this example there is only one reference input being generated with an amplitude of 0.5 and a frequency of 0.01 representing z_r. Lets say that there is another input reference such that Section 1 looks like

```
self.handle.append(mySignal(0.5,0.01)) # z
self.handle.append(mySignal(15*np.pi/180,0.2)) # Psi
```

then the order in which the reference signals are returned is $ref_inputs = [z_r, psi_r]$. This is shown in function getRefInputs on line 43.

For more information, read the provided file.

C

Connecting to the Hummingbird with ROS

Often when building complex systems that incorporate multiple executable programs, whether on a single machine or across a network of machines, they need to communicate by sharing data. This can be done using a program called ROS (Robot Operating System). ROS is like a bridge connecting two executable programs that are otherwise unable to communicate. ROS controls the type of information and the rate at which the information is communicated. This is very useful since it allows developers to break down complex systems into smaller pieces that are connected together with ROS. For example, the hummingbird (a remote machine) needs to receive pulse width modulation (PWM) commands from another computer (the host computer). To be truthful ROS is overly capable and complex for controlling the hummingbird. In addition ROS 1.0 is technically not a real time communication protocol. However, it will allow us to easily prototype and implement control algorithms in python, both in simulation and on the hummingbird hardware. ROS is a widely used research and development tool for robotics and autonomous systems. The labs have been structured so that much of its complexity will be hidden from the casual user. For those interested in learning more about ROS, these labs will provide an introduction.

ROS uses the catkin build system. A build system creates software targets (e.g., libraries, executables, scripts, interfaces, etc.) from raw source code that can be used by an end user. The purpose of this appendix is to teach you the basics of ROS and set up a catkin workspace that will contain all of your code. You will also connect to the hummingbirds, visualize the encoder values, view the ROS network layout, and more. This walkthrough is based on ROS kinetic or later version in an Ubuntu 16.04 or later environment. We are currently using ROS melodic and Ubuntu 18.10.

Note: Please do not connect to the hummingbirds using a wireless connection. Wifi is much slower than a direct Ethernet connection, and can have an average lag time of 100 ms which will likely lead to instability of your control implementation. This could lead you to believe there is a bug in your code when the real issue is the lag in the wireless connection.

Setting Up Your catkin Workspace

The first step in getting started with your own ROS setup is to create a catkin workspace. A catkin workspace is simply a folder (directory) where you can modify, build, and install, ROS packages. In this tutorial we will name our catkin workspace hummingbird_ws, however, feel free to use a different name. Our hummingbird_ws will have three subfolders (subdirectories): build, devel, and src. Our attention will be focused primarily on src where our source code will reside.

Start by opening up a terminal and using the keyboard shortcut ctrl+alt+t. Navigate to the directory where you want to create your workspace using the cd command. In this tutorial, we will place and initialize the workspace in the home directory using the following commands in the terminal.

```
cd ~/
mkdir -p hummingbird_ws/src
cd hummingbird_ws/src
catkin_init_workspace
```

Next we need the hummingbird packages that are located on the BYU Magicc Gitlab. (It is recommended that you go to this webpage and look at the different packages' README.md files for more information about each package.) In the source directory of your workspace we need to add some ROS packages: hb_st, hb_msgs, hb_viz. The hb_student directory will eventually contain the hummingbird parameters and your controllers. The hb_student directory will also eventually contain the hummingbird dynamics for simulation. The hb_msgs directory contains all of the custom ROS messages for the hummingbird workspace, and the hb_viz directory contains the animation code.

To add these packages, navigate to the source directory of your workspace (you should already be there). Now clone the packages using the following commands.

```
git clone https://magiccvs.byu.edu/gitlab/hummingbird/hb_student.git
git clone https://magiccvs.byu.edu/gitlab/hummingbird/hb.git
```

Now, build your workspace by using the command catkin_make in your workspace directory.

```
cd ..
catkin_make
```

For more information on the individual packages or setting up your workspace, take a look at the different packages on the BYU Magicc GitLab.

ROS Basics

In this section, we will discuss the ROS workspace, ROS packages, ROS nodes and ROS topics. The majority of the work regarding initializing the hummingbird workspace, writing the packages, and communicating over the ROS framework has been implemented for you. However, a brief overview of ROS and the packages contained in the hummingbird workspace will allow you to test your work and understand where the controller fits into the ROS framework.

Hummingbird Workspace

A ROS workspace is the outermost directory that organizes the project's components: source files, build files, etc. The `hummingbird_ws` contains multiple packages in the source folder

- `hb`
    - `hb_common`: Contains common code on top of which students will build for the entire semester.
    - `hb_ros_driver`: Code that interfaces with the low-level micro controller on the hummingbird platform.
    - `hb_msgs`: Stores the custom ROS messages used for the hummingbird project.
    - `hb_viz`: An RVIZ package to visualize the position of the hummingbird based on the hummingbird state. This package can also publish different topics: hummingbird states, hummingbird reference states, and hummingbird command.
- `hb_student`: This is where you will put code to simulate the dynamics of the hummingbird. This is also where you will put your code to control the hummingbird as we learn different control methods.

The majority of your work will take place in the `hb_student` directory.

You should also notice a `build` and `devel` folder in your workspace. If you don't see them, you have not built your workspace using `catkin_make`. For help doing this, refer to the above section or any of the `README.md` files in the packages.

The `build` folder contains the compiled code. The `devel` folder contains several `setup.*sh` files that are responsible for adding the workspace path to a variable that ROS uses to locate workspaces. To include these variables in your current shell session, you will need to source them by running

```
$ source <path_to_workspace>/devel/setup.bash
```

Listing c.1: Add workspace to ROS path

You can tell that this command worked by looking at the environment variable **ROS_PACKAGE_PATH**. This variable shows the paths of all current sourced ROS packages. This variable can be viewed by executing

```
$ echo $ROS_PACKAGE_PATH
```

Listing c.2: ROS_PACKAGE_PATH

The contents of the variable `ROS_PACKAGE_PATH` should be printed to the terminal. Make sure that the path to your workspace has been added. If you see it, the workspace has been properly created. The file setup.bash will need to be sourced at the beginning of every bash session (or whenever you open a new terminal). You can automate this by adding the source command to your `.bashrc`. You can do this by executing the command

```
$ echo "source <path_to_workspace>/devel/setup.bash" >> ~/.bashrc
```

Listing c.3: ROS_PACKAGE_PATH

Remember, if you do not source this file, ROS will not be able to find anything in the ROS packages including nodes, messages, services, etc., so don't forget this! **If putting the command in your .bashrc file does not seem to work, you will need to manually source that file for every terminal you open instead.** If you are interested in learning more about catkin workspaces and how to create them, you can find more information in this tutorial.

ROS Packages

As mentioned above, a ROS workspace can contain one or more packages. A package is a module that can contains various pieces of software to perform a certain function. Each package that ROS handles has certain dependencies that must be met in order to properly compile. These dependencies are listed in each package's CMakeLists.txt and package.xml files. For example, if you looked at the hb_viz package's CMakeLists.txt, you will see that it depends on hb_msgs. In fact, all of the hummingbird packaged depend on hb_msgs.

To complete these labs, you will not be required to make any packages or modify package.xml or CMakeLists.txt. The repo that you cloned is self-contained and will only need to be modified to complete the labs, however, you can find more information on ROS packages, including how to make your own package, in this tutorial.

ROS Messages

A way that programs (of the same or different languages) can communicate over ROS is through using ROS messages. You can think of message as a user defined data type that contains information that needs to be communicated between two packages. A message file is a simple text file that consists of one or more field type(s) and field name(s). Some of the different field types that can be used are:

- int8, int16, int32, int64 (plus uint*)
- float32, float64

- string
- time, duration

If multiple packages depend on the same messages, it is convenient to create another package that contains only the messages. This will bypass the need to create the same messages in every package. Instead, the package containing the messages can be included as a dependency by the other packages. To see how this is implemented with the hummingbird, navigate to the directory **hummingbird_ws/src/hb/hb_msgs/msg** and look at the files **State.msg** and **Command.msg**. You can see what is in the file by running

```
$ rosmsg show State
```

As you can see, the State message contains floats describing the state of the hummingbird and the **Command** message contains PWM commands sent to the right and left motors of the hummingbird. Lastly, the **ReferenceState.msg** message is used to communicate the desired yaw and pitch values.

Roscore

Roscore is a collection of programs and nodes that are required to make ROS run. Before any node can run, roscore needs to be started. To start roscore, open up another terminal or another tab in the current terminal (press **Ctrl+Shift+t** while in the current terminal) and add run this command

```
$ source <path_to_workspace>/devel/setup.bash
```

to initialize bash functions that will be used to setup your ROS environment. Note that you can also add this command to your .bashrc file.

Now enter the following commands in the terminal.

```
$ roscore
```

The first line is a function that sets the ros master to the local machine. The second line starts roscore.

When working with a network of computers that communicate through ROS only one computer needs to have roscore running, and this computer is known as the ros master. By running roscore on your local computer, by default your computer is the ros master.

If you are not working from a computer in the hummingbird lab, you need to make sure that your computer is the ROS master. To check, type

```
$ echo $ROS_MASTER_URI
```

If **http://localhost:11311** is printed to the screen then your computer is the ROS master and you can go ahead and start roscore by entering the command

```
$ roscore
```

Otherwise you computer is not the ROS master and needs to be changed to be the ROS master. To do this, enter the command

```
$ export ROS_MASTER_URI=http://localhost:11311
$ roscore
```

If there are problems in the future, make sure that **ROS_MASTER_URI** is set properly in all your terminals. One way to do this is to also add an explicit command to your .bashrc file (which is run every time a new terminal is opened). You can do this as follows:

```
$ echo "export ROS_MASTER_URI=http://localhost:11311" >> ~/.bashrc
```

More information about roscore can be found here.

ROS Nodes

A ROS node is an executable file that uses ROS to communicate with other nodes via messages or other means. ROS nodes are a powerful tool in abstraction, since they isolate the implementation of a single node from how it communicates with other nodes. This is inherently what makes ROS a very modular system. Later in this tutorial, we will talk about tools to visualize the node network and see how they communicate.

Starting a Node

To start the node, use the command **rosrun**. An example is shown below. Note: since you have not yet implemented the controller, this command will assuredly throw some errors.

```
$ rosrun hb_common controller.py
```

The command's syntax is

```
rosrun ros_package_name executable_name
```

More information on rosrun can be found here. To verify that a node is running, open up another terminal and type the command

```
$ rosnode list
```

This command lists all of the currently running nodes. To shutdown the node and roscore, use the command `Ctrl+c` in the terminals that you used to start the node and roscore. You can also use the `rosnode kill node_name` command to kill the node. Nodes, however, are of little use unless they can communicate with each other. This communication is performed using messages and instantiations of these messages are called topics.

ROS Topic

When a node sends or receives information using a message, it sends (publishes) or receives (subscribes) that information to or from a ROS Topic (a name used to identify the type of message). A ROS Topic is like an intermediary that holds a queue of ROS messages. When a node subscribes to the topic, the oldest message is popped off the queue and given to the node. When a node publishes a message to the topic, it is added to the queue. If the queue is full when a new message is being published to it, the oldest message will be discarded.

A ROS Topic can have multiple nodes subscribing (subscribers) to it and multiple nodes publishing (publisher) to it. In their most basic form, a subscriber pulls information off of a topic and a publisher puts new information on a topic. For example, controller code you write as a ROS node will subscribe to the `/hb_state` topic and publishes to the `/Command` topic. Students will not have concern themselves with the nuances of publishers and subscribers to complete these labs but further information about them is found in these tutorials.

Current topics running on the ROS system can be viewed using the `rostopic list` command. If you want to see the information being published to a certain topic you can use the `rostopic echo` command. We will test to make sure we are getting information from the hummingbirds using this command. First open up a terminal, navigate to your workspace, source the setup.bash file if needed and run the following command.

```
$ roslaunch hb_viz visualize.launch
```

This will start the visualization node and GUI. On the upper left hand side you will see sliders, a button and a drop down menu. This area is called a panel and you can interface with it to change the states of the hummingbird. First make sure that the enable button is selected. Now move the sliders and see the animation move with it.

Now let's look at the states using the `rostopic echo` command, by opening up another terminal, navigating to your workspace, sourcing the setup.bash file if needed and run the following command

```
$ rostopic echo /hb_state
```

You should see the states being printed. If you move the sliders, you will see these values change.

If you want to see all of the topics available, run

```
$ rostopic list
```

In the Hummingbird RViz panel switch to the `hb_command` topic by changing Layout Type. Now click on the enable button to start publishing commands. You can use the sliders to adjust the Force and Torque being applied to the hummingbird.

In a new terminal source the devel/setup.bash file and run.

```
rostopic echo /hb_command
```

You should see the left and right motor pwm commands being printed to your screen. To stop the topic being written to the terminal, kill the process with **ctrl + c**.

ROS also has another utility `rqt_plot` that provides plotting. When `rqt_plot` is run it brings up a window displaying a live plot of a chosen topic. Try it out by running

```
$ rqt_plot /hb_command
```

Now move the adjust the sliders in the RViz panel to see the plot change. You can learn more about `rqt_plot` here.

ROS provides a utility to visualize the current nodes and topics in the ROS network. This utility is called `rqt_graph`. While running the dynamics and the visualization that you launched earlier, run the command `rqt_graph`. This will pull up a window displaying the ros nodes and rostopics currently running. `rqt_graph` is an extremely useful debugging tool and it is recommended that you use it in future labs when you run into errors. More info can be found here.

You can stop any process (besides "roscore") in any of your terminals before moving on to the next part by typing **ctrl + c**. at the same time in the terminal.

ROS Launch

A nice tool that ROS provides is a ROS Launch file. A Launch file is used to organize the execution of one or more nodes across a network. You will use the `roslaunch` command many times throughout the labs, but all of the launch files have been written for you; however, you will have to edit the controller launch files to specify your specific controller node that you wish to run. A basic tutorial on launch files is found here.

Let's start up the dynamics and visualization nodes by running the dynamics.launch file

```
roslaunch hb_common dynamics.launch
```

Note that this command will almost certainly fail. This is because the file used to simulate the dynamics is currently missing information that you will include in a future lab. Adding the dynamics is left as a later assignment.

You have now finished!

d

Parameters of the Hummingbird

The physical parameters of the hummingbird are given in the table below, where m_1, J_1 are the mass and inertia of the hummingbird head, rotors, and rolling shaft, m_2, J_2 are the mass and inertia of the pitching body, and m_3, J_3 are the mass and inertia of the hummingbird that only yaws. When the roll pitch and yaw angles are all zero, the position of the center of mass of m_1 is $(\ell_1, 0, 0)^\top$, the position of the center of mass of m_2 is $(\ell_2, 0, 0)^\top$, and the position of the center of mass of m_3 is $(\ell_{3x}, \ell_{3y}, \ell_{3z})^\top$. The distance from the center of rotation to the hummingbird head is ℓ_T and the distance from the center of the hummingbird head to the rotors is d.

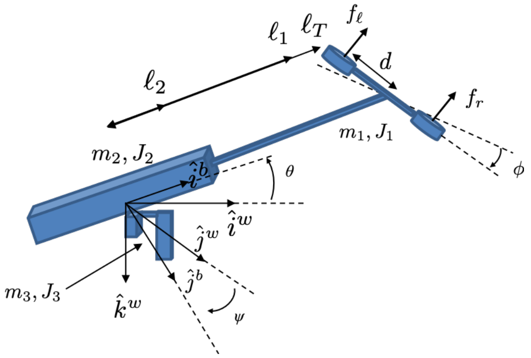


Figure 4-1: Hummingbird coordinate frames and nomenclature.

The hummingbird in the lab has the following parameters, which have been found experimentally:

The parameter value k_m is not given, but will be computed in one of the labs.

g	9.81	m/s^2
ℓ_1	0.247	m
ℓ_2	-0.039	m
ℓ_{3x}	-0.007	m
ℓ_{3y}	-0.007	m
ℓ_{3z}	0.018	m
ℓ_T	0.355	m
d	0.12	m
m_1	0.108862	kg
J_1	$\mathrm{diag}(0.000189, 0.001953, 0.001894)$	kg-m^2
m_2	0.4717	kg
J_2	$\mathrm{diag}(0.000231, 0.003274, 0.003416)$	kg-m^2
m_3	0.1905	kg
J_3	$\mathrm{diag}(0.0002222, 0.0001956, 0.000027)$	kg-m^2
k_m	$?$	N/PWM

e

Digital Implementation

This appendix will discuss how to implement the various controllers using Python.

PD Implementation

This section will demonstrate how to implement a PD controller provided that the states (the degrees of freedom and their derivatives) are known. First, we will look at a generic implementation of a PD controller, then we will look at an example using the robot arm design study, and finally we will look at an example that implements a successive loop PD controller using the inverted pendulum design study.

Generic PD Implementation

Listing e.1 shows a generic method to implement a PD controller.

```
import numpy as np
import param as P

def getForces(ref_inputs,states):
  # ref_inputs - the command input
  # states - the states of your system

  # unpack ref_inputs
  y_r = ref_inputs[0]

  # unpack states
  y = states[0]
  ydot = states[1]

  u_e = # Force equilibrium equation

  u = kp*(y_r-y)-kd*ydot + u_e

  return [u]
```

Listing e.1: Generic PD Implementation

On line 4 of listing e.1, the function *getForces* is declared. Notice that it takes in two arguments *ref_inputs* and *states*. The parameter *ref_inputs* is a list that contains all of the command inputs. The parameter *states* contains all of the states. In the generic case, there is only one command input (y_r) and one degree of freedom (y) with its associated derivative ($ydot$). In future assignments, the derivative will be assumed inaccessible and will need to be calculated, but this will be discussed in a later section.

The force applied to the system is a function of the command input, the current states, and equilibrium force/torque.

$$u = kp * (y\_r - y) - kd * ydot + u\_e$$

Where u is the calculated force, kp is the proportional gain, kd is the derivative gain, and u_e is the equilibrium force/torque. Once the applied force has been calculated, it is placed inside of a list and returned. This is shown on line 17 of listing e.1.

The generic main.py file associated with the PD controller is shown in listing e.2

```
import time
import sys
import numpy as np
import matplotlib.pyplot as plt
import param as P
from signal_generator import Signals  # Used for square wave
# from step import step_function      # Step input
from sim_plot import plotGenerator
import controllerPD as ctrl

# The Animation.py file is kept in the parent directory,
# so the parent directory path needs to be added.
sys.path.append(’..’)
from dynamics  import ________Dynamics
from animation import ________Animation

t_start = 0.0   # Start time of simulation
t_end = 10.0    # End time of simulation
t_Ts = P.Ts     # Simulation time step
t_elapse = 0.1  # Simulation time elapsed between each iteration
t_pause = 0.01  # Pause between each iteration





sig_gen = Signals()                   # Instantiate Signals class
plotGen = plotGenerator()             # Instantiate plotGenerator class
simAnimation = ________Animation()   # Instantiate Animate class
dynam = ________Dynamics()           # Instantiate Dynamics class
```

```
31
32 t = t_start               # Declare time variable to keep track of simulation time elap
33
34 while t < t_end:
35
36     # Get referenced inputs from signal generators
37   ref_input = sig_gen.getRefInputs(t)  # If using square wave
38   # ref_input = [step_function(t,T,A)] # If using step input
39
40   # The dynamics of the model will be propagated in time by t_elapse
41   # at intervals of t_Ts.
42   t_temp = t +t_elapse
43   while t < t_temp:
44
45     states = dynam.States()              # Get current states
46     u = ctrl.getForces(ref_input,states) # Calculate the forces
47     dynam.propagateDynamics(u)           # Propagate the dynamics of the model in time
48     t = round(t +t_Ts,2)                 # Update time elapsed
49
50   plt.figure(simAnimation.fig.number) # Switch current figure to animation figure
51   simAnimation.draw________(          # Update animation with current user input
52     dynam.Outputs())
53   plt.pause(0.0001)
54
55   # Organizes the new data to be passed to plotGen
56   new_data = [[ref_input[0],states[0]],
57         [states[1]],
58           [u[0]]]
59   plotGen.updateDataHistory(t, new_data)
60
61   # plt.figure(plotGen.fig.number)    # Switch current figure to plotGen figure
62   # plotGen.update_plots()              # Update the plot
63   # plt.pause(0.0001)
64
65   # time.sleep(t_pause)
66
67
68 plt.figure(plotGen.fig.number)
69 plotGen.update_plots()
70 plt.pause(0.001)
71
72 # Keeps the program from closing until the user presses a button.
73 print('done')
74 raw_input()
```

Listing e.2: Generic main.py for PD controller

The "______" in listing e.2 is used to indicate that the code can be used with any design study. Lines 6,7,37, and 38 show that the user can use either the step function or the signal generator to create the command input. The main thing to notice is on line 45 and 46. Notice that they dynamics function is returning all of the states, and they are being passed to the PD controller. Everything else should be familiar.

PD Implementation Example: Robot Arm

In creating the PD controller for the Robot Arm, the first step is selecting the proportional and derivative gains used in the controller. These gains can be calculated in the *param.py* file as shown in listing e.3.

```
# Inverted Pendulum Parameter File
import numpy as np

# Physical parameters of the inverted pendulum
m = 0.5   # Mass of arm, kg
ell = 0.3   # Length of arm, m
b = 0.01  # Damping coefficient, Nms
g = 9.8   # Gravity coefficient, m/s**2

# Simulation Parameters
Ts = 0.01
sigma = 0.05

# Initial Conditions
theta0 = 0.0*np.pi/180 # Initial theta position, rads
thetadot0 = 0.0

# Equilibrium tau at theta = 0
tau_e = m*g*ell/2

####################################################
#    PD Control: Pole Placement
####################################################

# Open Loop
# b0/(S**2 + a1*S + a0)
b0 = 3/(m*ell**2)
a1 = 3*b/(m*ell**2)
a0 = 0

# Desired Poles
p1 = -3.0
p2 = -4.0

# Desired Closed Loop
# S**2 + alpha1*S + alpha0
CL = np.polynomial.polynomial.polyfromroots((p1,p2))
alpha1 = CL[1]
alpha0 = CL[0]

# Gains
# b0*kp/[S**2 + (a1 + b0*kd)S + (a0 + b0*kp)]
kp = (alpha0-a0)/b0
kd = (alpha1-a1)/b0

print(’kp: ’,kp)
print(’kd: ’,kd)
```

Listing e.3: robotArm/hw_7/param.py

Lines 21-44 of listing e.3 shows how the PD gains were selected. These gains are used in the PD controller which is shown in listing e.4.

```
import numpy as np
import param as P

def getTorque(ref_inputs,states):
  theta = states[0]                         # Current theta
  thetadot = states[1]                      # Current thetadot
  theta_r = ref_inputs[0]                   # Command theta
  tau_e = P.m*P.g*P.ell*np.cos(theta)/2     # Equilibrium tau
  tau = P.kp*(theta_r-theta) \              # Controller
    - P.kd*thetadot +tau_e
  return [tau]
```

Listing e.4: robotArm/hw_7/controllerPD.py

As you can see, listing e.4 is very similar to the generic PD controller implementation discussed in subsection e. The differences are a change in the name of the function to reflect that the function is returning a torque and not a force, and the $states$ and ref_inputs parameters contain values associated with $theta$ instead of y.

To see all the files associated with this problem, visit the Control Book Website.

PD Successive Loop Implementation Example: Inverted Pendulum

The PD controller for the inverted pendulum is more complicated than the PD controller for the Robot Arm because its PD controller is designed using successive loop closure. We have to use successive loop closure since the transfer functions describing the motion for $theta$ and z are coupled. A diagram of the inner loop and outer loop of the controller is shown in figures 5-1 and 5-2.

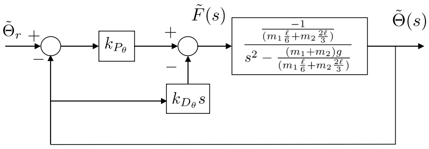

$\tilde{\Theta}_r$
+
−
k_{P_θ}
$\tilde{F}(s)$
+
−
$\frac{\frac{-1}{(m_1\frac{\ell}{6}+m_2\frac{2\ell}{3})}}{s^2-\frac{(m_1+m_2)g}{(m_1\frac{\ell}{6}+m_2\frac{2\ell}{3})}}$
$\tilde{\Theta}(s)$
$k_{D_\theta}s$

Figure 5-1: Inner Loop

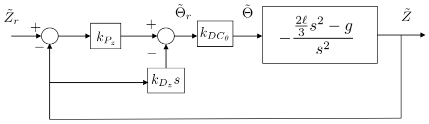


Figure 5-2: Outer Loop

From figures 5-1 and 5-2 notice how the input to the outer controller is z_r (position command input), and the output is $theta\_r$ (angle command input). Also notice that $theta\_r$ is the input to the inner controller and the output is F (force). Thus $theta\_r$ is a function of z_r, z, and $zdot$, and F is a function of $theta\_r$, $theta$, and $thetadot$.

$$theta\_r = z\_kp * (z\_r - z) - z\_kd * zdot$$

$$F = th\_kp * (theta\_r - theta) - th\_kd * thetadot$$

Where z_kp and z_kd are the PD gains for the outer control loop, and th_kp and th_kd are the PD gains for the inner control loop.

Once the force is calculated, it should be saturated or clipped to ensure that the output force is never greater than what the physical system can handle. Listing e.5 shows the implementation of the PD controller.

```
 1  import numpy as np
 2  import param as P
 3
 4  def saturate(limit,u):
 5    if abs(u) > limit:
 6      u = limit*np.sign(u)
 7    return u
 8
 9  def getForces(ref_inputs,states):
10
11    # unpack ref_inputs
12    z_r = ref_inputs[0]
13
14    # unpack states
15    z = states[0]
16    th = states[1]
17    zdot = states[2]
18    thdot = states[3]
19
20    # Compute theta_r using outer control loop
21    th_r = P.z_kp*(z_r-z)-P.z_kd*zdot
22
23    # Calculate the unsaturated force using the inner ctrl loop.
24    F_unsat = P.th_kp*(th_r-th)-P.th_kd*thdot
25
26    # Saturate the force
27    F_sat = saturate(P.F_max,F_unsat)
28
29    return [F_sat]
```

Listing e.5: invertedPendulum/hw_8/controllerPD.py

The controller starts on line 9 of listing e.5 where the parameters ref_inputs and $states$ are passed into the funciton *getForces*. ref_inputs is unpacked on line 12, and $states$ is unpacked on lines 15-18. The unsaturated force is calculated on line 24. The unsaturated force is then saturated on line 27 by passing it to the function *saturate*. Finally, the saturated force is returned on line 29.

To see all the files associated with this problem, visit the Control Book Website.

PID Implementation

An integrator can be added to the control system in order to track and compensate for constant disturbances. For the implementation of the PID controller discussed in this appendix, we assume that the controller does not have access to the derivatives of the degrees of freedom. This requires the controller to numerically calculate the derivatives. This section will discuss the theory needed to implement the PID controller, discuss the generic form of a PID controller, and then give an example of the PID controller using the Robot Arm design study.

Theory

In this book we work primarily with continuous time systems and continuous time controllers. However, almost all modern control strategies are implemented digitally on a micro-controller or a micro-processor, typically implemented using C-code. In this chapter, we describe how PID controllers can be implemented in a sampled data system using computer code.

A general PID control signal is given by

$$u(t) = k_P e(t) + k_I \int_{-\infty}^{t} e(\tau)d\tau + k_D \frac{de}{dt}(t),$$

where $e(t) = y_r(t) - y(t)$ is the error between the desired reference output $y_r(t)$ and the current output $y(t)$. Define $u_P(t) = e(t)$, $u_I(t) = \int_{-\infty}^{t} e(\tau)d\tau$, and $u_D(t) = \frac{de}{dt}(t)$. The PID controller is therefore given by $u(t) = k_p u_P(t) + k_I u_I(t) + k_D u_D(t)$. In the following paragraphs we will derive discrete sampled-time equivalents for $u_P(t)$, $u_I(t)$, and $u_D(t)$.

First consider a sampled time implementation of $u_P(t)$. When $t = nT_s$, where T_s is the sample period and n is the sample time, we have $u_P(nT_s) = e(nT_s)$, or using discrete time notation

$$u_P[n] = e[n].$$

To derive the sample-time implementation of an integrator, note that

$$\begin{aligned} u_I(nT_s) &= \int_{-\infty}^{nT_s} e(\tau)d\tau \\ &= \int_{-\infty}^{(n-1)T_s} e(\tau)d\tau + \int_{(n-1)T_s}^{T_s} e(\tau)d\tau \\ &= u_I((n-1)T_s) + \int_{(n-1)T_s}^{T_s} e(\tau)d\tau. \end{aligned}$$

As shown in Figure 5-3 a trapezoidal approximation of the integral between $(n-1)T_s$ and nT_s is given by

$$\int_{(n-1)T_s}^{T_s} e(\tau)d\tau \approx \frac{T_s}{2}\left(e(nT_s) + e((n-1)T_s)\right).$$

Therefore

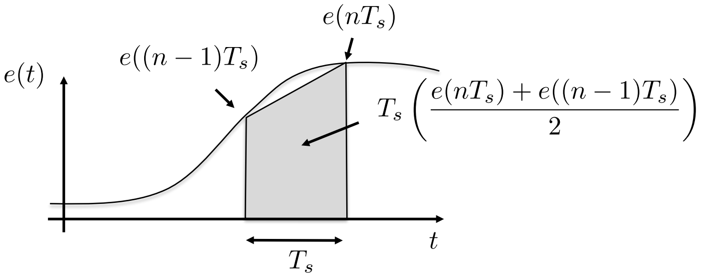


Figure 5-3: Trapezoidal rule for digital integration.

$$u_I(nT_s) \approx u_I((n-1)T_s) + \frac{T_s}{2}\left(e(nT_s) + e((n-1)T_s)\right).$$

Taking the z-transform we get

$$U_I(z) = z^{-1}U_I(z) + \frac{T_s}{2}(E(z) + z^{-1}E(z)).$$

Solving for $U_I(z)$ gives

$$U_I(z) = \frac{T_s}{2}\left(\frac{1+z^{-1}}{1-z^{-1}}\right)E(z). \tag{e.1}$$

Since in the Laplace domain we have

$$U_I(s) = \frac{1}{s}E(s),$$

we have derived the so-called Tustin approximation where s in the Laplace domain is replaced with the z-transform approximation

$$s \mapsto \frac{2}{T_s}\left(\frac{1-z^{-1}}{1+z^{-1}}\right), \tag{e.2}$$

where T_s is the sample period [**?**]. Taking the inverse z-transform of Equation (e.1) gives the discrete time implementation of an integrator as

$$u_I[n] = u_I[n-1] + \frac{T_s}{2}\left(e[n] + e[n-1]\right). \tag{e.3}$$

In the Laplace domain, a pure differentiator is given by $u_D(s) = sE(s)$. However, since a pure differentiator is not causal, the standard approach is to use the band-limited, or dirty derivative

$$U_D(s) = \frac{s}{\sigma s + 1} E(s),$$

where σ is small, and $\frac{1}{\sigma}$ defines the bandwidth of the differentiator. In practical terms, the dirty derivative differentiates signals with frequency content less than $\frac{1}{\sigma}$ radians per second. Using the the Tustin approximation (e.2), the dirty derivative in the z-domain becomes

$$\begin{aligned} u_D(z) &= \frac{\frac{2}{T_s}\left(\frac{1-z^{-1}}{1+z^{-1}}\right)}{\frac{2\sigma}{T_s}\left(\frac{1-z^{-1}}{1+z^{-1}}\right) + 1} E(z) \\ &= \frac{\left(\frac{2}{2\sigma+T_s}\right)(1 - z^{-1})}{1 - \left(\frac{2\sigma-T_s}{2\sigma+T_s}\right) z^{-1}} E(z). \end{aligned}$$

Transforming to the time domain, we have

$$u_D[n] = \left(\frac{2\sigma - T_s}{2\sigma + T_s}\right) u_D[n-1] + \left(\frac{2}{2\sigma + T_s}\right)(e[n] - e[n-1]) . \qquad \text{(e.4)}$$

Anti-Windup

A potential problem with a straight-forward implementation of PID controllers is integrator wind up. When the error $y\_r - y$ is large and a large error persists for an extended period of time, the value of the integrator, can become large, or "wind up." A large integrator will cause u, to saturate, which will cause the system to push with maximum effort in the direction needed to correct the error. Since the value of the integrator will continue to wind up until the error signal changes sign, the control signal may not come out of saturation until well after the error has changed sign, which can cause a large overshoot and may potentially destabilize the system.

Since integrator wind up can destabilize the system, it is important to add an anti-wind-up scheme. A number of different anti-wind-up schemes are possible. In this section we will discuss two schemes that are commonly used in practice. The first schemes is based on the fact that integrators are typically used to correct for steady state error and that overshoot is caused by integrating error during the transients of the step response. Therefore, the integrator can be turned off when the derivative of the error is large. This anti-windup scheme is implemented by an `if`-statement conditioned on the size of $\dot{e}$.

The second anti-windup scheme is to subtract from the integrator exactly the amount needed to keep u at the saturation bound. In particular, let

$$u_{\text{unsat}} = k_P e + k_D u_D + k_I u_I$$

denote the unsaturated control value before applying the anti-windup scheme. After the saturation block we have that

$$u = \text{sat}(u_{\text{unsat}}) = \text{sat}(k_P e + k_D u_D + k_I u_I).$$

The idea of anti-windup is to modify the output of the integrator from u_I to u_I^+ so that

$$k_P e + k_D u_D + k_I u_I^+ = \text{sat}(k_P e + k_D u_D + k_I u_I) = u.$$

Subtracting u_{unsat} from u gives

$$\begin{aligned} u - u_{\text{unsat}} &= (k_P e + k_D u_D + k_I u_I^+) - (k_P e + k_D u_D + k_I u_I) \\ &= k_I (u_I^+ - u_I). \end{aligned}$$

Solving for u_I^+ gives

$$u_I^+ = u_I + \frac{1}{k_I}(u - u_{\text{unsat}}),$$

which is the updated value of the integrator that ensures that the control signal remains just out of saturation. Note that if the control value is not in saturation then $u = u_{\text{unsat}}$ and the integrator value remains unchanged.

Generic PID Implementation

The generic PID implementation that will be shown in this section is not the only method that can implement a PID controller. The method used was chosen for several reasons: 1) to match closely to the implementation done in MATLAB, and 2) to have non-global persistent variables. Listing e.6 shows the generic PID implementation.

```
import sys
import numpy as np
import param as P

class controllerPID:
  ''' This class inherits other controllers in order to organize multiple controllers.'

  def __init__(self):
      # Instantiates the PID_ctrl object
      # Let x and y represent a degree of freedom
      self.xctrl=xPID_ctrl(x_kp,x_ki,x_kd,x0,x_limit)
      self.yctrl=yPID_ctrl(y_kp,y_ki,y_kd,y0,y_limit)
      self.zctrl=zPID_ctrl(z_kp,z_ki,z_kd,z0,z_limit)
      # kp is the proportional gain
      # ki is the integrator gain
      # kd is the derivative gain
      # x/y/z0 is the initial position of the state
      # limit is the maximum force/torque

  def getForces(self,ref_inputs,states):
      # ref_inputs is the referenced input
      # states is the current orientation
      # of the degrees of freedom.

      # Unpack ref_inputs
      x_r = ref_inputs[0]
      y_r = ref_inputs[1]

      # Unpack states
      x = states[0]
      y = states[1]
      z = states[2]

      # Calculate forces and z_r
      u1 = self.xctrl.PID_loop(x_r,x)  # Calculate the force output
      z_r = self.yctrl.PID_loop(y_r,y) # Calculate z_r
      u2 = self.zctrl.PID_loop(z_r,z)  # Calculate the force ouput
      return [u1,u2]


class xPID_ctrl:
  def __init__(self,kp,ki,kd,y0):
      self.differentiator = 0.0    # Difference term
      self.integrator = 0.0        # Integrator term
      self.x_d1 = x0               # Delayed x output
      self.error_d1 = 0.0          # Delayed error
      self.kp = kp                 # Proportional control gain
      self.ki = ki                 # Integrator control gain
      self.kd = kd                 # Derivative control gain
      self.limit                   # Maximum force/torque

  def PID_loop(self,x_r,x):
      # Compute the current error
      error = x_r - x

      # Update Differentiator
      a1 = (2*sigma - Ts)/(2*sigma+Ts)
      a2 = 2/(2*sigma+Ts)
      self.differentiator = a1*self.differentiator
                          + a2*(error -self.error_dl)

      # Update Integrator
      if abs(self.differentiator) <0.05:
        self.integrator += (P.Ts/2.0)*(x - self.x_d1)

      # Update error_dl
      self.error_dl = error
      self.x_d1 = x
      # The controllers designed on feedback linearization
      # will have an equilibrium force since
      # u_tilde = u - u_e
      # u = u_tilde + u_e
      u_e = # Equilibrium force

      # Calculate the unsaturated force/torque
      u_unsat = self.kp*error + self.ki*self.integrator \
              - self.kd*self.differentiator +u_e

      # Saturate the force/torque
      u_sat = self.saturate(u_unsat)

      # Integrator anit-windup
      if self.ki != 0:
        self.integrator += P.Ts/self.ki*(u_sat-u_unsat)

      return u_sat


class yPID_ctrl:
  def __init__(self,kp,ki,kd,y0):
      self.differentiator = 0.0    # Difference term
      self.integrator = 0.0        # Integrator term
      self.y_d1 = y0               # Delayed y output
      self.error_d1 = 0.0          # Delayed error
      self.kp = kp                 # Proportional control gain
      self.ki = ki                 # Integrator control gain
      self.kd = kd                 # Derivative control gain
      self.limit                   # Maximum force/torque

  def PID_loop(self,y_r,y):
      # Compute the current error
      error = y_r - y

      # Update Differentiator
      a1 = (2*sigma - Ts)/(2*sigma+Ts)
      a2 = 2/(2*sigma+Ts)
      self.differentiator = a1*self.differentiator
                          + a2*(error -self.error_dl)

      # Update Integrator
      if abs(self.differentiator) <0.05:
        self.integrator += (P.Ts/2.0)*(error+self.error_d1)

      # Update error_dl
      self.error_dl = error

      # The controllers designed on feedback linearization
      # will have an equilibrium force since
      # u_tilde = u - u_e
      # u = u_tilde + u_e
      u_e = # Equilibrium force

      # Calculate the unsaturated force/torque
      u_unsat = self.kp*error + self.ki*self.integrator \
              + self.kd*self.differentiator +u_e

      # Saturate the force/torque
      u_sat = self.saturate(u_unsat)

      # Integrator anit-windup
      if self.ki != 0:
        self.integrator += P.Ts/self.ki*(u_sat-u_unsat)

      return u_sat

    def saturate(self,u):
      if abs(u) < self.limit:
        u = self.limit*np.sign(u)
      return u


class zPID_ctrl:
  def __init__(self,kp,ki,kd,y0):
      self.differentiator = 0.0    # Difference term
      self.integrator = 0.0        # Integrator term
      self.z_d1 = z0               # Delayed z output
      self.error_d1 = 0.0          # Delayed error
      self.kp = kp                 # Proportional control gain
      self.ki = ki                 # Integrator control gain
      self.kd = kd                 # Derivative control gain
      self.limit                   # Maximum force/torque

  def PID_loop(self,z_r,z):
      # Compute the current error
      error = z_r - z

      # Update Differentiator
      a1 = (2*sigma - Ts)/(2*sigma+Ts)
      a2 = 2/(2*sigma+Ts)
      self.differentiator = a1*self.differentiator
                          + a2*(error -self.error_dl)

      # Update Integrator
      if abs(self.differentiator) <0.05:
        self.integrator += (P.Ts/2.0)*(error+self.error_d1)

      # Update error_dl
      self.error_dl = error

      # The controllers designed on feedback linearization
      # will have an equilibrium force since
      # u_tilde = u - u_e
      # u = u_tilde + u_e
      u_e = # Equilibrium force

      # Calculate the unsaturated force/torque
      u_unsat = self.kp*error + self.ki*self.integrator \
              + self.kd*self.differentiator +u_e

      # Saturate the force/torque
      u_sat = self.saturate(u_unsat)

      # Integrator anit-windup
      if self.ki != 0:
        self.integrator += P.Ts/self.ki*(u_sat-u_unsat)

      return u_sat
```

```
188
189    def saturate(self,u):
190      if abs(u) < self.limit:
191        u = self.limit*np.sign(u)
192      return u
193
194    def saturate(self,u):
195      if abs(u) < self.limit:
196        u = self.limit*np.sign(u)
197      return u
```

Listing e.6: Generic PID Controller

The generic PID implementation uses a hierarchy of classes. The parent class, *conrollerPID* organizes multiple *PID_ctrl* classes in order to compute the forces and torques required for the system to reached its desired orientation. The *controllerPID* class instantiates a *PID_ctrl* class for each degree of freedom the system has. In this example, there are three degrees of freedom whoes PID controllers are instantiated on lines 9-19.

Lines 20-38 implement the PID controllers using the function *getForces*. This function is similar to the function *getForces* that was used in the PD controller implementation. First the command inputs are unpacked, and then the states. Notice that there are only two command inputs in this generic example. This is to show how to implement a single loop and succesive loop controller. Also, notice that the derivatives of the degrees of freedom are not included in the parameter $states$. The last part of this functon calls other functions to implement PID control.

The first *PID_ctrl* class begins on line 41. When this class is instantiated, the following are variables are also instantiated,

- $self.differentiator$ - The derivative of the degree of freedom.
- $self.integrator$ - The integral of the error.
- $self.x\_d1$ - The value of the degree of freedom in the last time step.
- $self.error\_d1$ - The value of the error in the last time step
- $self.kp$ - The proportional gain.
- $self.ki$ - The integral gain.
- $self.kd$ - The derivative gain.
- $self.limit$ - The limit on force/torque or another degree of freedom.

The function *PID_loop* is the function that uses the variables discussed above to implement the PID controller. The reference input and the current state are passed into this function. Using these parameters, the error is calculated first. The error is then used to calculate the derivative and the integral terms. Notice that the integrator is inside of an if statement. This is to help improve the integrator by not integrating over the error until the system is close to steady state.

One the intergral and derivative terms are calculated, $error\_d1$ is updated,the force is calculated and then saturated. The last portion of the PID controller contains the integrator anti-windup. The if statement prevents the anti-windup from being implemented if the integrator gain is zero.

PID Implementation Example: Robot Arm

This section gives an example of how to implement the PID controller using the Robot Arm design study. Since the robot arm only has one degree of freedom $theta$, it only needs one control loop. The listing for the controller is shown in listing e.7.

```
import sys
import numpy as np
import param as P

class controllerPID:
  ''' This class inherits other controllers in order to organize multiple controllers.'

  def __init__(self):
      # Instantiates the PD_ctrl object
      self.thetaCtrl=thetaPID_ctrl(P.kp,P.kd,P.ki,P.theta0,P.tau_max)
      # kp is the proportional gain
      # kd is the derivative gain
      # ki is the integral gain
      # theta0 is the initial position of the state
      # P.error_max is the maximum error before saturation

  def getTorque(self,y_r,y):
      # y_r is the referenced input
      # y is the current state
      theta_r = y_r[0]
      theta = y[0]
      tau = self.thetaCtrl.thetaPID_loop(theta_r,theta) # Calculate the force output
      return [tau]


class thetaPID_ctrl:
  def __init__(self,kp,kd,ki,theta0,limit):
      self.thetadot = 0.0    # Difference term
      self.integrator = 0.0        # Integrator term
      self.theta_d1 = theta0       # Delayed y output
      self.error_d1 = 0.0          # Delayed error
      self.kp = kp                 # Proportional control gain
      self.kd = kd                 # Derivative control gain
      self.ki = ki                 # Integral control gain
      self.limit = limit           # Max torque

  def thetaPID_loop(self,theta_r,theta):
      # Compute the current error
      error = theta_r - theta

      # Update thetadot
      a1 = (2*P.sigma - P.Ts)/(2*P.sigma+P.Ts)
      a2 = 2/(2*P.sigma+P.Ts)
      self.thetadot = a1*self.thetadot \
                          + a2*(theta-self.theta_d1)

      # Update Integrator
      if abs(self.thetadot) <0.01:
        self.integrator += (P.Ts/2.0)*(error+self.error_d1)

      # Update error_d1
      self.error_d1 = error
      self.theta_d1 = theta
      # The controllers designed on feedback linearization
      # with have an equilibrium force since
      # tau_tilde = T - tau_e
      # T = tau_tilde + tau_e
      tau_e = P.m*P.g*P.ell*np.cos(theta)/2

      # PD Control to calculate T
      tau_unsat = self.kp*error - self.kd*self.thetadot + \
                self.ki*self.integrator + tau_e

      tau_sat = self.saturate(tau_unsat)

      if self.ki != 0:
        self.integrator += P.Ts/self.ki*(tau_sat-tau_unsat)

      return tau_sat


  def saturate(self,u):
    if abs(u) > self.limit:
      u = self.limit*np.sign(u)
    return u
```

Listing e.7: robotArm/hw_10/controllerPID.py

To see all the files associated with this problem, visit the Control Book Website.

SS Implementation

This section demonstrates how to implement a state space controller in python using the robot arm design study. The first step is to select the state space gains K and k_r. This can be done in the param.py file.

```
# Inverted Pendulum Parameter File
import numpy as np
import control as cnt

# Physical parameters of the inverted pendulum
m = 0.5    # Mass of arm, kg
ell = 0.3  # Length of arm, m
b = 0.01   # Damping coefficient, Nms
g = 9.8    # Gravity coefficient, m/s**2
```

```

# Simulation Parameters
Ts = 0.01
sigma = 0.05

# Initial Conditions
theta0 = 0.0*np.pi/180 # Initial theta position, rads
thetadot0 = 0.0

# Equilibrium tau at theta = 0
tau_e = m*g*ell/2


####################################################
#              State Space
####################################################
tau_max = 1                  # Max torque input, Nm

# State Space Equations
# xdot = A*x + B*u
# y = C*x
A = np.matrix([[0.0, 1.0],
         [0.0, -3.0*b/m/ell**2]])

B = np.matrix([[0.0],
             [3.0/m/ell**2]])

C = np.matrix([[1.0,0.0]])


# Desired Closed Loop tuning parameters
# S**2 + 2*zeta*wn*S + wn**2
tr = 0.8 #part a                   # Time rise, s
tr = 0.4 #
zeta = 0.707                       # Damping coefficient
wn = 2.2/tr

# S**2 + alpha1*S + alpha0
alpha1 = 2*zeta*wn
alpha0 = wn**2

# Desired Poles
des_char_poly = [1,alpha1,alpha0]
des_poles = np.roots(des_char_poly)

# Controllability Matrix
if np.linalg.matrix_rank(cnt.ctrb(A,B))!=2:
  print("The system is not controllable")
else:
  K = cnt.acker(A,B,des_poles)
  kr = -1.0/(C*np.linalg.inv(A-B*K)*B)

print(’K: ’, K)
print(’kr: ’, kr)
```

Listing e.8: robotArm/hw_11/param.py

In order to select the gains, the matrices A, B, and C need to be constructed from the equations of motion. This is shown for the robot arm in lines 31-37 of listing e.8. The desired poles are obtained from the tuning parameters used in hw_9. These calculations are shown in lines 40-53.

Before calculating the gains, the system needs to be check to ensure that it is controllable. This is done using the if-statement on line 56. If the system is controllable, then the gains will be calculated using the acker formula from the control library as shown on lines 59-60.

Now that the state space gains are calculated, they can be used in the state space controller. Listing **??** contains the code for the robot arm's SS controller.

```
import sys
import numpy as np
import param as P

class controllerSS:
  ''' This class inherits other controllers in order to organize multiple controllers.'

  def __init__(self):
      # Instantiates the SS_ctrl object
      self.thetaCtrl=thetaSS_ctrl(P.K,P.kr,P.theta0,P.tau_max)
      # K is the closed loop SS gains
      # kr is the input gain
      # theta0 is the initial position of the state

  def getTorque(self,y_r,y):
      # y_r is the referenced input
      # y is the current state
      theta_r = y_r[0]
      theta = y[0]
      tau = self.thetaCtrl.thetaSS_loop(theta_r,theta) # Calculate the force output
      return [tau]


class thetaSS_ctrl:
  def __init__(self,K,kr,theta0,limit):
      self.thetadot = 0.0          # Difference term
      self.theta_d1 = theta0          # Delayed state
      self.limit = limit           # Max torque
      self.K = K                   # SS gains
      self.kr = kr                 # Input gain

  def thetaSS_loop(self,theta_r,theta):

      # Update Differentiator
      a1 = (2*P.sigma - P.Ts)/(2*P.sigma+P.Ts)
      a2 = 2/(2*P.sigma+P.Ts)
      self.thetadot = a1*self.thetadot \
                          + a2*(theta-self.theta_d1)

      # Update theta_d1
      self.theta_d1 = theta

      # Construct the state
      x = np.matrix([[theta],
45                      [self.thetadot]])
46
47      # Compute the equilibrium torque tau_e
48      tau_e = P.m*P.g*P.ell*np.cos(theta)/2
49
50      # Compute the state feedback controller
51      tau_tilde = -self.K*x+self.kr*theta_r
52
53      # Compute total torque
54      tau = self.saturate(tau_e+tau_tilde)
55
56      return tau.item(0)
57
58
59
60  def saturate(self,u):
61    if abs(u) > self.limit:
62      u = self.limit*np.sign(u)
63    return u
```

Listing e.9: robotArm/hw_11/controllerSS.py

The file structure is very similar to the structure used for the PID implementation discussed in section e. The controller is instantiated on line 11. Notice that the gains that are passed in are K and k_r; these are needed to calculate the torque in the function *thetaSS_loop*.

The function *thetaSS_loop* takes takes in $theta_r$ and $theta$. Using $theta$, $thetadot$ is calculated on lines 35-38. Notice how the method used to calculate the derivative is different then the method used when implementing the PID controller. This is to expose you to another valid method. After the derivative is calculated, it is then used to form the state of the system x on lines 44-45. Using the state x and the gains K and *k_r*, the torque is calculated on line 51. The torque is then saturated and returned.

To see all the files associated with this problem, visit the Control Book Website.

SSI Implementation

This section demonstrates how to implement a state space with integrator controller in python using the inverted pendulum design study. The first step is to select the state space gains K and ki. This can be done in the param.py file.

```
# Inverted Pendulum Parameter File
import numpy as np
import control as cnt

# Physical parameters of the inverted pendulum
m1 = 0.25     # Mass of the pendulum, kg
m2 = 1.0      # Mass of the cart, kg
ell =  0.5    # Length of the rod, m
w = 0.5       # Width of the cart, m
```

```
h = 0.15      # Height of the cart, m
gap = 0.005   # Gap between the cart and x-axis
radius = 0.06 # Radius of circular part of pendulum
g = 9.8       # Gravity, m/s**2
b = 0.05      # Damping coefficient, Ns

# Simulation Parameters
Ts = 0.01
sigma = 0.05

# Initial Conditions
z0 = 0.0                # ,m
theta0 = 0.0*np.pi/180  # ,rads
zdot0 = 0.0             # ,m/s
thetadot0 = 0.0         # ,rads/s

####################################################
#                 State Space
####################################################
F_max = 5                    # Max Force, N
error_max = 1                # Max step size,m
theta_max = 30.0*np.pi/180.0 # Max theta
M = 3                        # Time scale separation between
                   # inner and outer loop

# State Space Equations
# xdot = A*x + B*u
# y = C*x
A = np.matrix([[0.0,0.0,               1.0,      0.0],
         [0.0,0.0,               0.0,      1.0],
         [0.0,-m1*g/m2,          b/m2,     0.0],
         [0.0,(m1+m2)*g/m2/ell, b/m2/ell, 0.0]])

B = np.matrix([[0.0],
         [0.0],
         [1.0/m2],
         [-1.0/m2/ell]])

C = np.matrix([[1.0,0.0,0.0,0.0],
         [0.0,1.0,0.0,0.0]])
Cr = C[0,:]

# Augmented Matrices
A1 = np.concatenate((
  np.concatenate((A,np.zeros((4,1))),axis=1),
  np.concatenate((-Cr,np.matrix([[0.0]])),axis=1)),axis = 0)

B1 = np.concatenate((B,np.matrix([[0.0]])),axis = 0)

# Desired Closed Loop tuning parameters
# S**2 + 2*zeta*wn*S + wn**2

th_tr = 0.5           # Rise time, s
th_zeta = 0.707       # Damping Coefficient
th_wn = 2.2/th_tr     # Natural frequency

# S**2 + alpha1*S + alpha0
```

```
67  th_alpha1 = 2.0*th_zeta*th_wn
68  th_alpha0 = th_wn**2
69
70  # Desired Closed Loop tuning parameters
71  # S**2 + 2*zeta*wn*S + wn**2
72
73  z_tr = th_tr*M      # Rise time, s
74  z_zeta = 0.707      # Damping Coefficient
75  z_wn = 2.2/z_tr     # Natural frequency
76  integrator_pole = -10.0
77
78  # S**2 + alpha1*S + alpha0
79  z_alpha1 = 2.0*z_zeta*z_wn
80  z_alpha0 = z_wn**2
81
82  # Desired Poles
83  des_char_poly = np.convolve(
84    np.convolve([1,z_alpha1,z_alpha0],[1,th_alpha1,th_alpha0]),
85    np.poly(integrator_pole))
86  des_poles = np.roots(des_char_poly)
87
88
89  # Controllability Matrix
90  if np.linalg.matrix_rank(cnt.ctrb(A1,B1))!=5:
91    print("The system is not controllable")
92  else:
93    K1 = cnt.acker(A1,B1,des_poles)
94    K = K1[0,0:4]
95    ki = K1[0,4]
96    print(’K1: ’, K1)
97    print(’K: ’, K)
98    print(’ki: ’, ki)
99
100
101 #################################################
102 #          Uncertainty Parameters
103 #################################################
104 UNCERTAINTY_PARAMETERS = True
105 if UNCERTAINTY_PARAMETERS:
106   alpha = 0.2;                                # Uncertainty parameter
107   m1 = 0.25*(1+2*alpha*np.random.rand()-alpha)  # Mass of the pendulum, kg
108   m2 = 1.0*(1+2*alpha*np.random.rand()-alpha)   # Mass of the cart, kg
109   L =  0.5*(1+2*alpha*np.random.rand()-alpha)   # Length of the rod, m
110   b = 0.05*(1+2*alpha*np.random.rand()-alpha)   # Damping coefficient, Ns
```

Listing e.10: invertedPendulum/hw_12/param.py

In order to select the gains, the matrices $A1$, $B1$, and Cr need to be constructed from A, B, and C that were created from the EOMs in the previous lab. This is shown for the robot arm in lines 50-59 of listing e.10. The tuning parameters are shown on lines 64-66 and 75-58. From which the desired poles are calculated on lines 85-88. The state space and integrator gains are finally calculated on lines 92-97.

Now that the gains are calculated, they can be used in the state space with integrator controller as shown in

```
import sys
import numpy as np
import param as P

class controllerSSI:
  ’’’ This class inherits other controllers in order to organize multiple controllers.’

  def __init__(self):
      # Instantiates the SS_ctrl object
      self.SSICtrl = SSI_ctrl(P.K,P.ki,P.z0,P.theta0,P.F_max)
      # K is the closed loop SS gains
      # ki is the integrator gain
      # y0 is the initial position of the state

  def getForces(self,y_r,y):
      # y_r is the referenced input
      # y is the current state
      z_r = y_r[0]
      z = y[0]
      theta = y[1]

      F = self.SSICtrl.SSI_loop(z_r,z,theta) # Calculate the force output
      return [F]


class SSI_ctrl:
  def __init__(self,K,ki,z0,theta0,limit):
      self.zdot = 0.0                 # Difference term
      self.thetadot = 0.0             # Difference term
      self.integrator = 0.0           # Integrator term
      self.z_d1 = z0                  # Last z term
      self.theta_d1 = theta0          # Last theta term
      self.error_d1 = 0.0
      self.K = K                      # Closed loop SS gains
      self.ki = ki                    # Integrator gain
      self.limit = limit              # Maxiumum force
      self.input_disturbance = 0.5 # Input disturbance, N


  def SSI_loop(self,z_r,z,theta):

      error = z_r-z

      # Update Differentiator
      a1 = (2*P.sigma - P.Ts)/(2*P.sigma+P.Ts)
      a2 = 2/(2*P.sigma+P.Ts)
      self.zdot = a1*self.zdot + a2*(z-self.z_d1)

      self.thetadot = a1*self.thetadot + a2*(theta-self.theta_d1)
      self.integrator += (P.Ts/2.0)*(error+self.error_d1)

      # Update delays
      self.z_d1 = z
      self.theta_d1 = theta
      self.error_d1 = error

```

```
57       # Construct the state
58       x = np.matrix([[z],
59                      [theta],
60                      [self.zdot],
61                      [self.thetadot]])
62 
63       # Compute the state feedback controller
64       F_unsat = -self.K*x -self.ki*self.integrator + self.input_disturbance
65 
66       F_sat = self.saturate(F_unsat)
67 
68       if self.ki !=0:
69         self.integrator += P.Ts/self.ki*(F_sat-F_unsat)
70       return F_sat.item(0)
71 
72   def saturate(self,u):
73     if abs(u) > self.limit:
74       u = self.limit*np.sign(u)
75     return u
```

Listing e.11: invertedPendulum/hw_12/controllerSSI.py

Most of the format is the same as the state space controller implementation done in the previous lab except for the addition of the integrator on line 50. The integrator is taken into account when computing the force in line 64, and the integrator anti-windup is implemented on lines 68-69; all of which should be familiar to you by now.

To see all the files associated with this problem, visit the Control Book Website

Observer

This section will demonstrate how to implement an observer to be used with a SSI controller. We will use the inverted pendulum hw 12 design study as a basis and build off of it. We will begin by finding the estimator gain L, then we will add the observer to the SS controller and name it *controllerObs.py*, and finally we will modify the *main.py* file and *sim_plot.py* file to plot the estimated states.

Observer Param File

The estimator gain will be calculated in the *param.py* file.

```
# Inverted Pendulum Parameter File
import numpy as np
from scipy.signal import place_poles as place
import control as cnt

# Physical parameters of the inverted pendulum
m1 = 0.25      # Mass of the pendulum, kg
m2 = 1.0       # Mass of the cart, kg
ell =  0.5     # Length of the rod, m
w = 0.5        # Width of the cart, m
h = 0.15       # Height of the cart, m
```

```
gap = 0.005   # Gap between the cart and x-axis
radius = 0.06 # Radius of circular part of pendulum
g = 9.8       # Gravity, m/s**2
b = 0.05      # Damping coefficient, Ns

# Simulation Parameters
Ts = 0.01
sigma = 0.05

# Initial Conditions
z0 = 0.0                # ,m
theta0 = 0.0*np.pi/180  # ,rads
zdot0 = 0.0             # ,m/s
thetadot0 = 0.0         # ,rads/s

####################################################
#                 State Space Feedback
####################################################
F_max = 5                    # Max Force, N
error_max = 1                # Max step size,m
theta_max = 30.0*np.pi/180.0 # Max theta
M = 3                        # Time scale separation between
                   # inner and outer loop

# State Space Equations
# xdot = A*x + B*u
# y = C*x
A = np.matrix([[0.0,0.0,               1.0,      0.0],
         [0.0,0.0,               0.0,      1.0],
         [0.0,-m1*g/m2,          b/m2,     0.0],
         [0.0,(m1+m2)*g/m2/ell, b/m2/ell, 0.0]])

B = np.matrix([[0.0],
         [0.0],
         [1.0/m2],
         [-1.0/m2/ell]])

C = np.matrix([[1.0,0.0,0.0,0.0],
         [0.0,1.0,0.0,0.0]])
Cr = C[0,:]

# Augmented Matrices
A1 = np.concatenate((
  np.concatenate((A,np.zeros((4,1))),axis=1),
  np.concatenate((-Cr,np.matrix([[0.0]])),axis=1)),axis = 0)

B1 = np.concatenate((B,np.matrix([[0.0]])),axis = 0)

# Desired Closed Loop tuning parameters
# S**2 + 2*zeta*wn*S + wn**2

th_tr = 0.5           # Rise time, s
th_zeta = 0.707       # Damping Coefficient
th_wn = 2.2/th_tr     # Natural frequency

# S**2 + alpha1*S + alpha0
th_alpha1 = 2.0*th_zeta*th_wn
```

```
th_alpha0 = th_wn**2

# Desired Closed Loop tuning parameters
# S**2 + 2*zeta*wn*S + wn**2

z_tr = 1.5           # Rise time, s
z_zeta = 0.707       # Damping Coefficient
z_wn = 2.2/z_tr      # Natural frequency
integrator_pole = -5.0

# S**2 + alpha1*S + alpha0
z_alpha1 = 2.0*z_zeta*z_wn
z_alpha0 = z_wn**2

# Desired Poles
des_char_poly = np.convolve(
  np.convolve([1,z_alpha1,z_alpha0],[1,th_alpha1,th_alpha0]),
  np.poly(integrator_pole))
des_poles = np.roots(des_char_poly)


# Controllability Matrix
if np.linalg.matrix_rank(cnt.ctrb(A1,B1))!=5:
  print("The system is not controllable")
else:
  K1 = cnt.acker(A1,B1,des_poles)
  K = K1[0,0:4]
  ki = K1[0,4]
  print(’K1: ’, K1)
  print(’K: ’, K)
  print(’ki: ’, ki)

#####################################################
#                  Observer
#####################################################

# Observer design
obs_th_wn = 5.0*th_wn
obs_z_wn = 5.0*z_wn
obs_des_char_poly = np.convolve([1,2.0*z_zeta*obs_z_wn,obs_z_wn**2],
                 [1,2.0*th_zeta*obs_th_wn,obs_th_wn**2])
obs_des_poles = np.roots(obs_des_char_poly)


if np.linalg.matrix_rank(cnt.obsv(A,C))!=4:
  print(’System Not Observable’)
else:
  L = place(A.T,C.T,obs_des_poles).gain_matrix.T
  print(’L:’, L)

##################################################
#           Uncertainty Parameters
##################################################
UNCERTAINTY_PARAMETERS = False
if UNCERTAINTY_PARAMETERS:
  alpha = 0.2;                                  # Uncertainty parameter
  m1 = 0.25*(1+2*alpha*np.random.rand()-alpha)  # Mass of the pendulum, kg
```

```
126   m2 = 1.0*(1+2*alpha*np.random.rand()-alpha)   # Mass of the cart, kg
127   ell =  0.5*(1+2*alpha*np.random.rand()-alpha)   # Length of the rod, m
128   b = 0.05*(1+2*alpha*np.random.rand()-alpha)   # Damping coefficient, Ns
```

Listing e.12: invertedPendulum/hw_13/param.py

The observer for the inverted pendulum is a MIMO system since it takes in the arguments z and $theta$ when trying to correct the estimated states. This can be seen in the equation below where $obs_states = np.matrix([[z], [theta]])$.

$$xhat = xhat + Ts * A * xhat + B * U + L(obs_states - C * xhat)$$

Because it is MIMO, the Ackerman algorithm will not work. Instead, the *place* function from the scipy library can be used. This library is imported on line 3 of listing e.12.

The calculation for the observer gain L begins on line 101 of listing e.12. The first step is to select the desired observer poles as shown in lines 106-110. The next step is to ensure that the system is observable using the functin *obsv* from the control library as shown on line 113. If the system is observable, then L is calculated using the *place* command.

ControllerObs File

The *controllerObs.py* for the inverted pendulum is shown in listing e.13

```
import sys
import numpy as np
import param as P

class controllerObs:
  ’’’ This class inherits other controllers in order to organize multiple controllers.’

  def __init__(self):
      # Instantiates the SS_ctrl object
      self.ObsCtrl = Obs_ctrl(P.K,P.ki,P.L,P.z0,P.theta0,P.F_max)
      # K is the closed loop SS gains
      # ki is the integrator gain
      # L is the observer gain
      # y0 is the initial position of the state

  def getForces(self,y_r,y):
      # y_r is the referenced input
      # y is the current state
      z_r = y_r[0]
      z = y[0]
      theta = y[1]

      F = self.ObsCtrl.Obs_loop(z_r,z,theta) # Calculate the force output
      return [F]

  def getObsStates(self):
    return self.ObsCtrl.getObsStates()
```

```


class Obs_ctrl:
  def __init__(self,K,ki,L,z0,theta0,limit):
      self.xhat = np.matrix([[0.0],  # z
                             [0.0],  # theta
                             [0.0],  # zdot
                             [0.0]]) # theta dot
      self.F_d1 = 0.0                # Delayed input
      self.integrator=0.0            # Integrator term
      self.error_d1 = 0.0            # Delayed error
      self.limit = limit             # Max torque
      self.K = K                     # SS gains
      self.ki=ki                     # Integrator gain
      self.L = L                     # Obs Gain
      self.input_disturbance= 0.0 #0.05    # Input disturbance


  def Obs_loop(self,z_r,z,theta):

      # Observer
      N = 10
      for i in range(N):
        self.xhat += P.Ts/N*(P.A*self.xhat + \
          P.B*(self.F_d1)+\
          self.L*(np.matrix([[z],[theta]]) - P.C*self.xhat))

      error = z_r-self.xhat.item(0)

      self.integrator += (P.Ts/2.0)*(error+self.error_d1)

      # Update delays
      self.error_d1 = error

      # Compute the state feedback controller
      F_unsat = -self.K*self.xhat - self.ki*self.integrator

      # Input disturbance. To motivate next weeks lab (disturbance observer)
      F_unsat = F_unsat + self.input_disturbance

      F_sat = self.saturate(F_unsat)

      self.F_d1 = F_sat

      if self.ki !=0:
        self.integrator += P.Ts/self.ki*(F_sat-F_unsat)

      return F_sat.item(0)

  def getObsStates(self):
    return self.xhat.tolist()

  def saturate(self,u):
    if abs(u) > self.limit:
      u = self.limit*np.sign(u)
    return u
```

Listing e.13: invertedPendulum/hw_13/controllerObs.py

The listing is similar to the controller listing of the last homework. On line 10, where the controller object is instatiated, the observer gain L is passed into the class. The declaration of the Obs_ctrl class begins on line 30. When the class is instantiated, the variable $self.xhat$ is created to hold the estimated states, the variable $self.F\_d1$ holds the last input to the system, L is the estimator gain, and all other variable instantiated should be familiar to you.

The estimated states, $xhat$ is updated on lines 47-52. Notice that in the rest of this function, $xhat$ is used instead of z or $theta$ since we are assuming that we do not always have access to them.

When tuning the controller, it is useful to plot the estimated states along with the states from they dynamics file. The functions $getObsStates$ on lines 26 and 73 allows the user to grab the estimated states. These states can then be passed into the sim_plot.py file through the main.py file in order to be plotted. To see this, download the files from the Control Book Website.

Observer With Disturbance Estimator

This section will demonstrate how to add a disturbance estimator to the observer that was discussed in the previous section, and will use the inverted pendulum design study. To implement the disturbance estimator in the controller, the disturbance gain Ld needs to be calcualated first. This will be done in the param file.

Param File

The *param.py* for the inverted pendulum is shown in listing e.14

```
# Inverted Pendulum Parameter File
import numpy as np
from scipy.signal import place_poles as place
import control as cnt

# Physical parameters of the inverted pendulum
m1 = 0.25      # Mass of the pendulum, kg
m2 = 1.0       # Mass of the cart, kg
ell =  0.5     # Length of the rod, m
w = 0.5        # Width of the cart, m
h = 0.15       # Height of the cart, m
gap = 0.005    # Gap between the cart and x-axis
radius = 0.06 # Radius of circular part of pendulum
g = 9.8        # Gravity, m/s**2
b = 0.05       # Damping coefficient, Ns

# Simulation Parameters
Ts = 0.01
sigma = 0.05
```

```

# Initial Conditions
z0 = 0.0                # ,m
theta0 = 0.0*np.pi/180  # ,rads
zdot0 = 0.0             # ,m/s
thetadot0 = 0.0         # ,rads/s

####################################################
#                 State Space Feedback
####################################################
F_max = 5                    # Max Force, N
error_max = 1                # Max step size,m
theta_max = 30.0*np.pi/180.0 # Max theta
M = 3                        # Time scale separation between
                   # inner and outer loop

# State Space Equations
# xdot = A*x + B*u
# y = C*x
A = np.matrix([[0.0,0.0,               1.0,      0.0],
         [0.0,0.0,               0.0,      1.0],
         [0.0,-m1*g/m2,          b/m2,     0.0],
         [0.0,(m1+m2)*g/m2/ell, b/m2/ell, 0.0]])

B = np.matrix([[0.0],
         [0.0],
         [1.0/m2],
         [-1.0/m2/ell]])

C = np.matrix([[1.0,0.0,0.0,0.0],
         [0.0,1.0,0.0,0.0]])
Cr = C[0,:]

# Augmented Matrices
A1 = np.concatenate((
  np.concatenate((A,np.zeros((4,1))),axis=1),
  np.concatenate((-Cr,np.matrix([[0.0]])),axis=1)),axis = 0)

B1 = np.concatenate((B,np.matrix([[0.0]])),axis = 0)

# Desired Closed Loop tuning parameters
# S**2 + 2*zeta*wn*S + wn**2

th_tr = 0.5           # Rise time, s
th_zeta = 0.707       # Damping Coefficient
th_wn = 2.2/th_tr     # Natural frequency

# S**2 + alpha1*S + alpha0
th_alpha1 = 2.0*th_zeta*th_wn
th_alpha0 = th_wn**2

# Desired Closed Loop tuning parameters
# S**2 + 2*zeta*wn*S + wn**2

z_tr = 1.5          # Rise time, s
z_zeta = 0.707      # Damping Coefficient
z_wn = 2.2/z_tr     # Natural frequency
```

```
integrator_pole = -5.0

# S**2 + alpha1*S + alpha0
z_alpha1 = 2.0*z_zeta*z_wn
z_alpha0 = z_wn**2

# Desired Poles
des_char_poly = np.convolve(
  np.convolve([1,z_alpha1,z_alpha0],[1,th_alpha1,th_alpha0]),
  np.poly(integrator_pole))
des_poles = np.roots(des_char_poly)


# Controllability Matrix
if np.linalg.matrix_rank(cnt.ctrb(A1,B1))!=5:
  print("The system is not controllable")
else:
  K1 = cnt.acker(A1,B1,des_poles)
  K = K1[0,0:4]
  ki = K1[0,4]
  print(’K1: ’, K1)
  print(’K: ’, K)
  print(’ki: ’, ki)

####################################################
#                  Observer
####################################################

A2 = np.concatenate((np.concatenate((A,B), axis = 1),
          np.zeros((1,5))),axis = 0)
C2 = np.concatenate((C,np.zeros((2,1))),axis = 1)

# Observer poles
obs_th_wn = 10.0*th_wn
obs_z_wn = 10.0*z_wn
obs_dist_pole = -1
obs_des_char_poly = np.convolve(
  np.convolve([1,2.0*z_zeta*obs_z_wn,obs_z_wn**2],
          [1,2.0*th_zeta*obs_th_wn,obs_th_wn**2]),
        np.poly(obs_dist_pole))
obs_des_poles = np.roots(obs_des_char_poly)



if np.linalg.matrix_rank(cnt.obsv(A2,C2))!=5:
  print(’System Not Observable’)
else:
  L2 = place(A2.T,C2.T,obs_des_poles).gain_matrix.T
  L = np.matrix(L2.tolist()[0:4])
  Ld = np.matrix(L2.tolist()[4])
  print(’L:’, L)
  print(’Ld:’, Ld)

#################################################
#          Uncertainty Parameters
#################################################
UNCERTAINTY_PARAMETERS = True
```

```
134 if UNCERTAINTY_PARAMETERS:
135   alpha = 0.2;                                # Uncertainty parameter
136   m1 = 0.25*(1+2*alpha*np.random.rand()-alpha)  # Mass of the pendulum, kg
137   m2 = 1.0*(1+2*alpha*np.random.rand()-alpha)   # Mass of the cart, kg
138   ell =  0.5*(1+2*alpha*np.random.rand()-alpha)   # Length of the rod, m
139   b = 0.05*(1+2*alpha*np.random.rand()-alpha)   # Damping coefficient, Ns
```

Listing e.14: invertedPendulum/hw_14/param.py

The first step in finding the disturbance gain Ld is to augment the state space matrices. This is shown on lines 105-107 of listing e.14. Then the observer and disturbance poles are selected on line 109-117. The selected poles are used to calculate the observer and disturbance gains as shown on lines 121-126.

ControllerObsD

Once the gains are seleceted, the controller file is modified to implement the disturbance estimator. The listing for the controll in shown in listinge.15

```
import sys
import numpy as np
import param as P

class controllerObsD:
  ''' This class inherits other controllers in order to organize multiple controllers.'

  def __init__(self):
      # Instantiates the SS_ctrl object
      self.ObsCtrl = ObsD_ctrl(P.K,P.ki,P.L,P.Ld,P.z0,P.theta0,P.F_max)
      # K is the closed loop SS gains
      # ki is the integrator gain
      # L is the observer gain
      # Ld is the disturbance observer
      # y0 is the initial position of the state

  def getForces(self,y_r,y):
      # y_r is the referenced input
      # y is the current state
      z_r = y_r[0]
      z = y[0] + np.random.normal(0,0.001,1).item(0)*0
      theta = y[1] + np.random.normal(0,0.001,1).item(0)*0

      F = self.ObsCtrl.Obs_loop(z_r,z,theta) # Calculate the force output
      return [F]

  def getObsStates(self):
    return self.ObsCtrl.getObsStates()


class ObsD_ctrl:
  def __init__(self,K,ki,L,Ld,z0,theta0,limit):
      self.xhat = np.matrix([[0.0],  # z
                             [0.0],  # theta
                             [0.0],  # zdot
                             [0.0]]) # theta dot
      self.dhat = 0.0                # Estimated Distrubance
      self.F_d1 = 0.0                # Delayed input
      self.integrator=0.0            # Integrator term
      self.error_d1 = 0.0            # Delayed error
      self.limit = limit             # Max torque
      self.K = K                     # SS gains
      self.ki=ki                     # Integrator gain
      self.L = L                     # Obs Gain
      self.Ld = Ld                   # Disturbance gain
      self.input_disturbance= 0.05    # Input disturbance


  def Obs_loop(self,z_r,z,theta):

      # Observer
      N = 10
      for i in range(N):
        self.xhat += P.Ts/N*(P.A*self.xhat + \
          P.B*(self.F_d1+self.dhat)+\
          self.L*(np.matrix([[z],[theta]]) - P.C*self.xhat))
        self.dhat += P.Ts/N*self.Ld*(np.matrix([[z],[theta]]) - P.C*self.xhat)


      #print(self.dhat)
      error = z_r-self.xhat.item(0)

      self.integrator += (P.Ts/2.0)*(error+self.error_d1)

      # Update delays
      self.error_d1 = error


      # Compute the state feedback controller
      F_unsat = -self.K*self.xhat - self.ki*self.integrator - self.dhat

      # Input disturbance.
      F_unsat = F_unsat + self.input_disturbance

      F_sat = self.saturate(F_unsat)

      self.F_d1 = F_sat

      if self.ki !=0:
        self.integrator += P.Ts/self.ki*(F_sat-F_unsat)
      return F_sat.item(0)

  def getObsStates(self):
    return self.xhat.tolist()

  def saturate(self,u):
    if abs(u) > self.limit:
      u = self.limit*np.sign(u)
    return u
```

Listing e.15: invertedPendulum/hw_14/controllerObsD.py

The first step is to augment the parameters passed into the class *ObsD_Ctrl* on line 32 to include the disturbance gain Ld. Also, notice that the variables $self.dhat$, $self.Ld$, and $self.input\_disturbance$ was added to the class. $dhat$ is the estimated disturbance, Ld is the disturbance gain, and $input\_disturbance$ is a modeled disturbance put into the system.

The estimated disturbance $dhat$ is updated on line 57, and used to estimate the states $xhat$ on lines 54-56. Notice that on line 69 the $input\_disturbance$ is added.

In addition to input disturbance, noise has been added to z and $theta$ on lines 21-22 to simulate noisy sensors.

To see all the files associated with this problem, download the files from the Control Book Website.