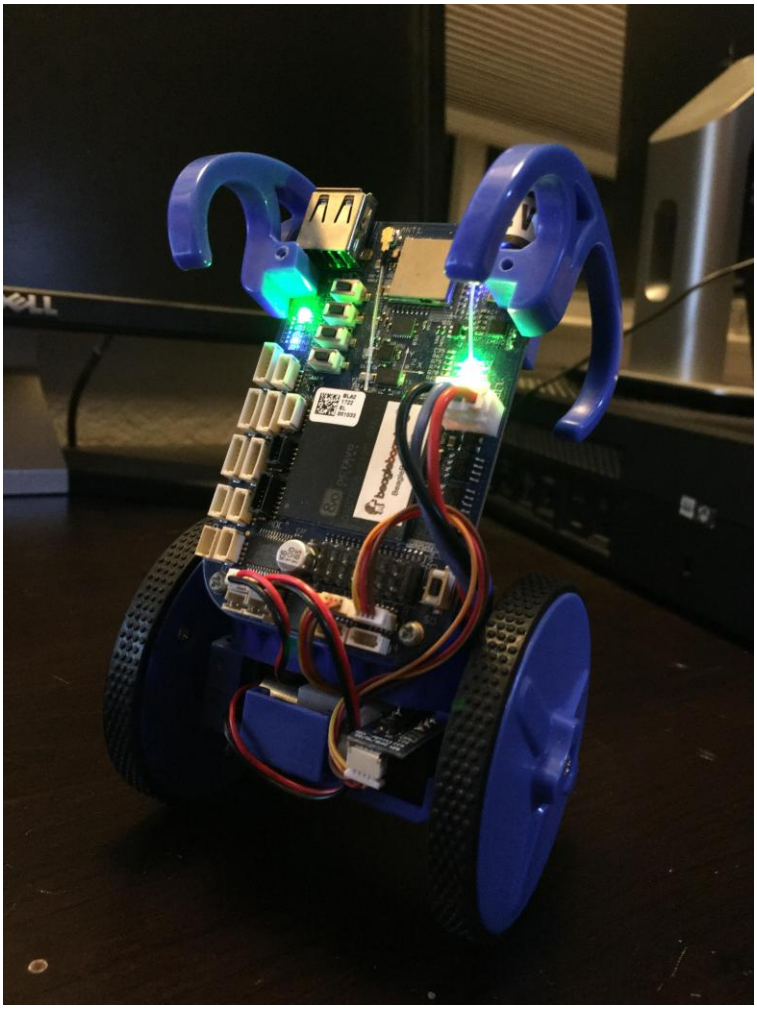
BeagleBone Blue eduMIP

Lead-Lag Controller Design to Balance an Inverted Pendulum Robot



Robert Ketchum

Abstract

The equations of motions of an eduMIP inverted pendulum were derived, and its system equations linearized. MATLAB was used to derive the two transfer functions from the input torque to the output pitch body angle and the pitch body angle to the output wheel angular velocity, respectively. To have the eduMIP robot balance itself using only its motors, two lead-lag compensators were designed with MATLAB utilizing the theory of successive loop closure. These two controllers were converted to discrete-time and implemented in the system using C-Programming Language. A video of the balancing robot can be viewed at: https://tinyurl.com/eduMIP .

Table of Contents

Figures and Tables

Introduction

The system under study is an inverted pendulum robot. At first glance, this genre of robots are not inherently useful due to their natural instability, and as such are not the most practical vehicle design for fuel efficiency, stability, nor carrying capacity. Despite all of these obvious flaws, they are a classic hardware choice for the study of control systems engineering. To be able to use programming and math to keep the bot upright is to fly in the face of nature and stabilize a system which by all intuition should topple over. In this sense, inverted pendulum robots are proof of the importance of control system engineering and its use in understanding and harnessing the power of the world around us.

The BeagleBone Blue eduMIP

The inverted pendulum chosen for this study is the eduMIP BeagleBone Blue Inverted Pendulum kit, developed by the UC San Diego Flow Control and Coordinated Robotics Labs [1].

The eduMIP consists of a mounted BeagleBone Blue on a main chassis, a battery pack for power, bumpers to prevent damage to the board, and two DC motors attached to the wheels with optical encoders for measuring wheel rotation. The BeagleBone has an onboard 9 degree of freedom Inertial Measurement Unit (IMU), which may be used to measure the pitch, yaw, and roll of the bot.

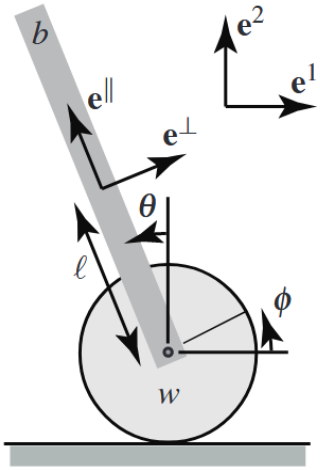


Figure 1: Coordinate axes of the eduMIP. The inertial frame is given by $e^I = \begin{Bmatrix} e^1 \\ e^2 \end{Bmatrix}$ and the body coordinate axes are: $e^M = \begin{Bmatrix} e^{\parallel} \\ e^{\perp} \end{Bmatrix}$ [2]

System Variables

The variables used in the system are listed as follows [3]:

Table 1: eduMIP Variables	
Variable Symbol	Name
θ	Body pitch angle. 0° defines the body upright and perpendicular to the ground
φ	Average wheel rotation angle. 0° defines the initial average wheel angles
τ	Torque from the wheel motors
x	Translation in x-direction
P	Force translated from the body to the wheels
$F_{friction}$	Friction force
F_{Normal}	Normal force
u	Ratio from the motor output torque and the stall torque. Also the controller output
τ_{motor}	Net torque output by one motor
ω	motor angular velocity

The constants used in the system are listed in Table 2 [3]:

Table 2: eduMIP Known Constants			
Variable Symbol	Name	Value	Units
α	Gear ratio of a motor	35.555	--
k	Motor constant ($\frac{\tau_{stall}}{\omega_{no\ load}}$)	1.7045×10^{-6}	$\frac{Nm}{sec}$
τ_{stall}	Motor Maximum (or stall) torque output	0.003	Nm
$\omega_{no\ load}$	Motor no load angular velocity	1760	$\frac{rad}{sec}$
l	Length from the wheel center to the center of mass of the body	4.77	cm
r	Wheel radius	3.4	cm
g	Gravity	9.81	$\frac{m}{s^2}$
m_b	Mass of the body	180	g
m_w	Mass of each wheel	27	g
I_b	Moment of inertia of the body	2.63×10^{-4}	$kg\ m^2$
I_w	Moment of inertia of the wheels	1.22×10^{-4}	$kg\ m^2$

Engineering Assumptions

In analyzing the system, we may make the following assumptions:

1) The two wheels spin at the same angular velocity
2) The pitch angle is small, such that $sin(\theta') \simeq \theta'$, $cos(\theta') \simeq 1$
3) There are no forces in the lateral direction and no torques about the yaw or roll axis, so the system is only in 2 dimensions

4) The pitch angle deviation is not changing rapidly with respect to time, so $(\frac{d\theta}{dt})^2 \simeq 0$
5) The robot wheels do not slip and are instead stuck to the ground, so $F_{friction} = F_x$

Derivation of the Equations of Motion

Lagrangian Mechanics

For systems where it is difficult to visualize all forces acting on bodies, one may find it easier to use Lagrangian mechanics to derive the equations of motion. The process for deriving the equations of motion for the eduMIP is given below [4] [5].

The Lagrangian function is defined as:

$$L = KE - PE$$

Where KE is the total possible kinetic energy in the system, and PE is the total possible potential energy in the system.

The kinetic energy in the system is composed of:

-The linear motion of the wheels: $KE_1 = \frac{1}{2} m_w (\frac{dx}{dt})^2$

-The rotational motion of the body about the wheels: $KE_2 = \frac{1}{2} I_b (\frac{d\theta}{dt})^2$

-The linear motion of the body: $KE_3 = \frac{1}{2} m_b (v_b)^2$

Note that there are two orthogonal components of v_b, decomposed in the inertial reference frame, e^I, as:

$$v_b = e^I \begin{Bmatrix} \frac{dx}{dt} + l\frac{d\theta}{dt}cos(\theta) \\ l\frac{d\theta}{dt}sin(\theta) \end{Bmatrix}$$

For our purposes, we only care about the magnitude squared of the body velocity:

$$\left|\left|v_b\right|\right|^2 = [\frac{dx}{dt} + l\frac{d\theta}{dt}cos(\theta)]^2 + [l\frac{d\theta}{dt}sin(\theta)]^2 = (\frac{dx}{dt})^2 + l^2(\frac{d\theta}{dt})^2 + 2l\frac{dx}{dt}\frac{d\theta}{dt}cos(\theta)$$

The total potential energy of the system is then:

$$KE = \sum_{i=1}^{3} KE_i = \frac{1}{2}m_w(\frac{dx}{dt})^2 + \frac{1}{2}I_b(\frac{d\theta}{dt})^2 + \frac{1}{2}m_b[(\frac{dx}{dt})^2 + l^2(\frac{d\theta}{dt})^2 + 2l\frac{dx}{dt}\frac{d\theta}{dt}cos(\theta)]^2$$

The only potential energy in the system is the gravitational potential energy of the inverted pendulum body:

$$PE = l\,m_b\,g(1 - cos(\theta))$$

The Lagrangian function is then:

$$L = \frac{1}{2}m_w\left(\frac{dx}{dt}\right)^2 + \frac{1}{2}I_b\left(\frac{d\theta}{dt}\right)^2 + \frac{1}{2}m_b\left[\left(\frac{dx}{dt}\right)^2 + l^2\left(\frac{d\theta}{dt}\right)^2 + 2l\frac{dx}{dt}\frac{d\theta}{dt}cos(\theta)\right]^2 - l\,m_b\,g(1$$

The Lagrange equations of motion for the x-coordinate and the body angle are given by:

$$\frac{d}{dt}\left(\frac{\partial L}{\partial\left(\frac{dx}{dt}\right)}\right) - \frac{\partial L}{\partial x} = F_x$$

$$\frac{d}{dt}\left(\frac{\partial L}{\partial\left(\frac{d\theta}{dt}\right)}\right) - \frac{\partial L}{\partial \theta} = -\tau$$

Where F_x and T are the nonconservative through-variables representing the force on the wheels by the ground and the torque on the body by the wheels, respectively.

The x-coordinate Lagrange equation may be simplified as follows:

$$\frac{d}{dt}\left(\frac{\partial L}{\partial\left(\frac{dx}{dt}\right)}\right) - \frac{\partial L}{\partial x} = F_x$$

$$\frac{d}{dt}\left(m_w\frac{dx}{dt} + m_b\frac{dx}{dt} + m_b l\frac{d\theta}{dt}cos(\theta)\right) = F_x$$

$$m_w\frac{d^2x}{dt^2} + m_b\frac{d^2x}{dt^2} + m_b l\frac{d^2\theta}{dt^2}cos(\theta) - m_b l\left(\frac{d\theta}{dt}\right)^2 sin(\theta) = F_x$$

The bot does not have sensors to determine its x-position. Rather, it uses motor encoders to determine the wheel angle. We therefore must use the following relationship to convert the linear position to the wheel angle: $x = r\varphi$. Continuing the derivation:

$$m_w r\frac{d^2\varphi}{dt^2} + m_b r\frac{d^2\varphi}{dt^2} + m_b l\frac{d^2\theta}{dt^2}cos(\theta) - m_b l\left(\frac{d\theta}{dt}\right)^2 sin(\theta) = F_x$$

The torque on the bot provided by the wheel motors is used to both spin the wheels and move the bot forward: $\tau = I_w\frac{d^2\varphi}{dt^2} + F_x r$. Using this relationship, we obtain our first equation of motion:

$$\tau = \left[I_w + (m_b + m_w)\,r^2\right]\frac{d^2\varphi}{dt^2} + m_b\,r\,l\,cos(\theta)\,\frac{d^2\theta}{dt^2} - m_b\,r\,l\,sin(\theta)\left[\frac{d\theta}{dt}\right]^2 \quad (I)$$

Likewise, the body angle Lagrange equation may be solved as:

$$\frac{d}{dt}\left(\frac{\partial L}{\partial(\frac{d\theta}{dt})}\right) - \frac{\partial L}{\partial \theta} = -\tau$$

$$\frac{d}{dt}\left(I_b \frac{d\theta}{dt} + m_b l^2 \frac{d\theta}{dt} + m_b \frac{dx}{dt} l cos(\theta)\right) - \left(-m_b l \frac{dx}{dt}\frac{d\theta}{dt} sin(\theta) + m_b l g sin(\theta)\right) = -\tau$$

$$\left(I_b + m_b l^2\right)\frac{d^2\theta}{dt^2} + m_b \frac{d^2x}{dt^2} l cos(\theta) - m_b l \frac{dx}{dt}\frac{d\theta}{dt} sin(\theta) + m_b l \frac{dx}{dt}\frac{d\theta}{dt} sin(\theta) - m_b l g sin(\theta)) = -\tau$$

Again, using the wheel angle and linear motion relationship, we obtain our second equation of motion:

$$-\tau = m_b\, r\, l\, cos(\theta)\, \frac{d^2\varphi}{dt^2} + \left[I_b + m_b\, l^2\right] \frac{d^2\theta}{dt^2} - m_b\, g\, l\, sin(\theta) \quad \text{(II)}$$

Newtonian Mechanics

The Lagrangian derived equations of motion may be verified using Newtonian mechanics. These derivations and the following linearizations are done in Examples 17.9 and 17.10 of Numerical Renaissance, by Thomas Bewley [2].

The Kinematics equations of the body in the inertial frame are given by:

$$e^I \begin{cases} m_b \frac{d^2x}{dt^2} - m_b\, l\, cos(\theta) \frac{d^2\theta}{dt^2} + \left(\frac{d\theta}{dt}\right)^2 = -P_x \\ -m_b\, l\, sin(\theta) \frac{d^2\theta}{dt^2} - m_b\, l\, cos(\theta) \left(\frac{d\theta}{dt}\right)^2 = -g\, m_b - P_y \end{cases} \begin{matrix} (1) \\ (2) \end{matrix}$$

Rotating the reference frame by θ so that the reference frame is attached to the eduMIP body yields:

$$e^M \begin{cases} m_b\, cos(\theta) \frac{d^2x}{dt^2} - m_b\, l \frac{d^2\theta}{dt^2} = -P_x\, cos(\theta) - P_y\, sin(\theta) - g\, m_b\, sin(\theta) \\ -m_b\, sin(\theta) \frac{d^2x}{dt^2} - m_b\, l \frac{d^2\theta}{dt^2} = P_x\, sin(\theta) - P_y\, cos(\theta) - g\, m_b\, cos(\theta) \end{cases} \begin{matrix} (3) \\ (4) \end{matrix}$$

The Torque about the wheels is given by the following :

$$I_w \frac{d^2\varphi}{dt^2} = \tau - r\, F_{friction} \quad (5)$$

Torque equation about the robot body:

$$I_b \frac{d^2\theta}{dt^2} = -\tau - P_x\, l\, cos(\theta) - P_y\, l\, sin(\theta) \quad (6)$$

Kinematics equations of the wheels in the inertial frame:

$$e^I \left\{ \begin{array}{ll} m_w \frac{d^2x}{dt^2} & = P_x - F_{friction} \\ m_w \frac{d^2y}{dt^2} & = P_y + F_{normal} - g\, m_w \end{array} \right\} \begin{array}{l} (7) \\ (8) \end{array}$$

The relationship between wheel rotation and position is given by:

$$-\varphi\, r = x \quad (9)$$

Combining equations (1), (5), (7), and (9) yields the simplified equation for torque about the wheels of the bot, which is our first nonlinear system equation:

$$\tau = \left[I_w + (m_b + m_w)\, r^2 \right] \frac{d^2\varphi}{dt^2} + m_b\, r\, l\, cos(\theta) \frac{d^2\theta}{dt^2} - m_b\, r\, l\, sin(\theta) \left[\frac{d\theta}{dt} \right]^2 \quad (I)$$

Our second nonlinear system equation, the torque about the body of the bot, is obtained by combining equations (3), (6), and (9):

$$-\tau = m_b\, r\, l\, cos(\theta) \frac{d^2\varphi}{dt^2} + \left[I_b + m_b\, l^2 \right] \frac{d^2\theta}{dt^2} - m_b\, g\, l\, sin(\theta) \quad (II)$$

Linearization of the Equations of Motion

Our pitch angle, wheel angle, and wheel torque can be written as the sum of the average and the current deviation from the average, or: $\theta = \bar{\theta} + \theta' \varphi = \bar{\varphi} + \varphi' \tau = \bar{\tau} + \tau'$.

Setting the average pitch angle, wheel angle, and torque as zero ($\bar{\theta} = \bar{\tau} = \bar{\varphi} = 0$), the system equations can then be written as follows:

$$\tau = \left[I_w + (m_b + m_w)\, r^2 \right] \frac{d^2\varphi'}{dt^2} + m_b\, r\, l\, cos(\theta') \frac{d^2\theta'}{dt^2} - m_b\, r\, l\, sin(\theta') \left[\frac{d\theta'}{dt} \right]^2 \quad (I')$$

$$-\tau = m_b\, r\, l\, cos(\theta') \frac{d^2\varphi'}{dt^2} + \left[I_b + m_b\, l^2 \right] \frac{d^2\theta'}{dt^2} - m_b\, g\, l\, sin(\theta') \quad (II')$$

For small angles, $cos(\theta') \simeq 1$, and $sin(\theta') \simeq \theta'$

Also, assuming that the pitch angle deviation is not changing rapidly with respect to time, $(\frac{d\theta'}{dt})^2 \simeq 0$

Our final linearized state equations are given as follows:

$$\tau = \left[I_w + (m_b + m_w)\, r^2\right] \frac{d^2\varphi'}{dt^2} + m_b\, r\, l\, \frac{d^2\theta'}{dt^2} \qquad (\mathrm{I}^*)$$

$$-\tau = m_b\, r\, l\, \frac{d^2\varphi'}{dt^2} + \left[I_b + m_b\, l^2\right] \frac{d^2\theta'}{dt^2} - m_b\, g\, l\, \theta' \qquad (\mathrm{II}^*)$$

Motor Characteristics and Control Inputs

While we have derived the linear equations of motions for the robot, we must characterize the eduMIP's DC motors.

The general equation for a linear DC motor is: $\tau_{motor} = \tau_{stall} - k\,\omega$. This relationship is most commonly displayed as a graph, as seen below in Figure 3.

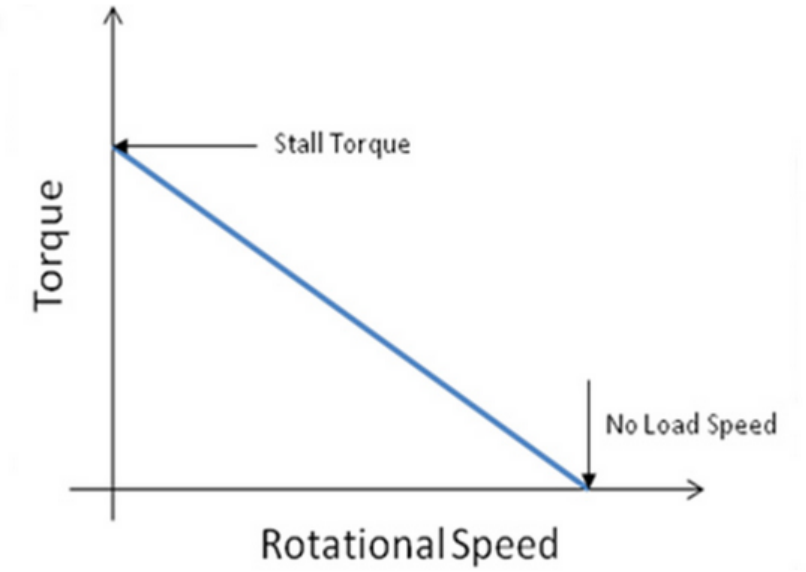


Figure 2: DC Motor Torque and Angular Velocity Relationship [6].

The control input for the system, u, is the normalized ratio of maximum torque to each motor. As such, it is bound by: $-1 \leq u \leq 1$, and takes the form of: $\tau_{motor} = u\tau_{stall} - k\,\omega$ [1].

Accounting for both wheels and the gear ratio, the total torque on the robot by the motors is: $\tau = 2\alpha(u\tau_{stall} - k\,\omega)$.

[1] It may seem that this motor command should account for the motor torque output rather than just the stall torque, resulting in the incorrect control input equation: $\tau_{motor} = u(\tau_{stall} - k\,\omega)$. There are a few reasons why this is not the case. For one, if the motor is operating at the no-load speed, such that $\tau_{stall} = k\,\omega$, a command may be sent to stop the wheels. The incorrect derivation does not allow for any command in such a case. From a controller design perspective, a normalized command also allows for easy controller design, as the same command tells the ratio of maximum motor torque, angular velocity, voltage, and current.

The translation of the motor angular velocity to motion of the wheel and body angle is given by: $\omega = \alpha(\frac{d\varphi}{dt} - \frac{d\theta}{dt})$. The subtraction of the body angle accounts for the conservation of angular momentum in the system, and is the key to why the system may be controlled at all.

Combining the two above motor equations gives an equation relating the wheel torque with the dynamics of the body and the wheels: $\tau = 2\alpha(u\tau_{stall} + k\alpha\frac{d\theta}{dt} - k\alpha\frac{d\varphi}{dt})$.

Controllable Linearized Equations of Motion

Inserting the motor torque equation into the linearized equations of motion, we obtain:

$$2\alpha(u\tau_{stall} + k\alpha\frac{d\theta}{dt} - k\alpha\frac{d\varphi}{dt}) = \left[I_w + (m_b + m_w)r^2\right]\frac{d^2\varphi}{dt^2} + m_b rl\frac{d^2\theta}{dt^2}$$

$$-2\alpha(u\tau_{stall} + k\alpha\frac{d\theta}{dt} - k\alpha\frac{d\varphi}{dt}) = \left[I_b + m_b l^2\right]\frac{d^2\theta}{dt^2} + m_b rl\frac{d^2\varphi}{dt^2} - m_b gl\theta$$

Single Input, Single Output Plant Equation Derivations

The pair of equations of motion each contain one input and two outputs. As such, this system is of type Single Input, Multiple Output (SIMO). To simplify these equations into a pair of Single Input, Single Output (SISO) equations, we must group terms of output variables. The most simple way to do this is by converting the equations into the Laplace Domain, as demonstrated below:

$$2\alpha(u\tau_{stall} + k\alpha\frac{d\theta}{dt} - k\alpha\frac{d\varphi}{dt}) = \left[I_w + (m_b + m_w)r^2\right]\frac{d^2\varphi}{dt^2} + m_b rl\frac{d^2\theta}{dt^2} \xrightarrow{\mathcal{L}}$$

$$2\alpha(U\tau_{stall} + k\alpha S\Theta - k\alpha S\Phi) = \left[I_w + (m_b + m_w)r^2\right]S^2\Phi + m_b rlS^2\Theta$$

$$2\alpha\tau_{stall}U = \left\{\left[I_w + (m_b + m_w)r^2\right]S^2 + 2\alpha^2 kS\right\}\Phi + \left\{m_b rlS^2 - 2\alpha^2 kS\right\}\Theta$$

$$-2\alpha(u\tau_{stall} + k\alpha\frac{d\theta}{dt} - k\alpha\frac{d\varphi}{dt}) = \left[I_b + m_b l^2\right]\frac{d^2\theta}{dt^2} + m_b rl\frac{d^2\varphi}{dt^2} - m_b gl\theta \xrightarrow{\mathcal{L}}$$

$$-2\alpha(U\tau_{stall} + k\alpha S\Theta - k\alpha S\Phi) = \left[I_b + m_b l^2\right]S^2\Theta + m_b rlS^2\Phi - m_b gl\Theta$$

$$-2\alpha\tau_{stall}U = \left\{\left[I_b + m_b l^2\right]S^2 - 2\alpha^2 kS - m_b gl\right\}\Theta + \left\{m_b rlS^2 + 2\alpha^2 kS\right\}\Phi$$

We must now take turns cancelling the body angle and wheel angle,Θ and Φ, respectively. To simplify the algebra, the following variables will be used:

$$A = \left[I_w + (m_b + m_w)r^2\right]S^2 + 2\alpha^2 kS$$
$$B = m_b rlS^2 - 2\alpha^2 kS$$
$$C = 2\alpha\tau_{stall}$$
$$E = m_b rlS^2 + 2\alpha^2 kS$$
$$D = \left[I_b + m_b l^2\right]S^2 - 2\alpha^2 kS - m_b gl$$
$$F = -2\alpha\tau_{stall}$$

The equations of motion then become:

$$CU = A\Phi + B\Theta$$
$$FU = D\Phi + E\Theta$$

Solving for the body angle and the torque duty cycle:

$$\Phi = \frac{CU - B\Theta}{A} = \frac{FU - E\Theta}{D}$$
$$U = \frac{A\Phi + B\Theta}{C} = \frac{D\Phi + E\Theta}{F}$$

Rearranging into our two equations, we obtain:

$$\left[\frac{C}{A} - \frac{F}{D}\right]U = \left[\frac{B}{A} - \frac{E}{D}\right]\Theta$$
$$\left[\frac{E}{F} - \frac{B}{C}\right]\Theta = \left[\frac{A}{C} - \frac{D}{F}\right]\Phi$$

$$G_1(s) = \frac{\Theta}{U} = \left[\frac{\frac{C}{A} - \frac{F}{D}}{\frac{B}{A} - \frac{E}{D}}\right] = \frac{Body\ Angle}{Normalized\ Motor\ Duty\ Cycle}$$

$$G_2(s) = \frac{\Phi}{\Theta} = \left[\frac{\frac{E}{F} - \frac{B}{C}}{\frac{A}{C} - \frac{D}{F}}\right] = \frac{Wheel\ Angle}{Body\ Angle}$$

$G_1(s)$is our SISO plant equation relating the normalized torque of the motor to the body angle of the eduMIP. $G_2(s)$is the plant equation relating the body angle to the wheel angle. Plugging in the known values for our robot from Table 2, we obtain our final SISO transfer equations:

$$G_1(s) = \frac{-882.7\,s}{s^3+44.15s^2-192.8s-2299}$$

$$G_2(s) = \frac{-1.476s^2+128.9}{s^2}$$

Note that these values were computed in MATLAB. The code for these transfer functions, as well as the full design of the controllers, including plots, is given in Appendix B.

Linearized Continuous-Time Controller Design

Subsequent Loop Closure

While there are multiple types of controllers we may use for lineraized SIMO systems, such as a Linear Quadratic Regulator (LQR) or direct pole placement via full state feedback, for the purposes of this project, subsequent loop closure for our two SISO equations was chosen for a few reasons. For one, it allows for easy design of individual controllers for each of the output variables via classical control techniques with transfer functionsG_1 and G_2. These individual controllers have the added benefit of easy testing in the physical system, as the inner loop may be tested in the eduMIP separately from the outer loop. Lastly, the computational power of the BeagleBone Blue allowed the inner loop to run at the necessary order of magnitude higher than the outer loop (200 Hz vs 20 Hz).

Inner Loop Overview

Figure 4 below shows the inner control loop of the system. The Plant, G_1, inputs the normalized motor duty cycle, u, and outputs the eduMIP body angle, θ. As the goal of the controller is to keep the body upright, the motors should only output a torque if the current body angle, θ, is different than the reference body angle, θ_r. For this reason, the controller, D_1, inputs the body error angle,$\theta_e = \theta_r - \theta$, and outputs the normalized duty cycle.

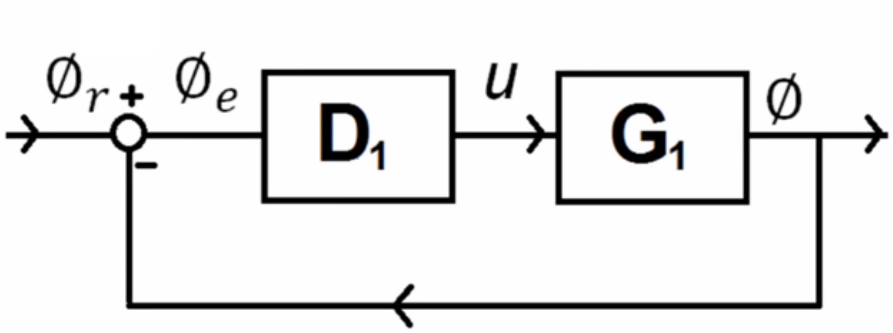


Figure 3: Inner Control Loop with Feedback

Inner Loop Stability

By using the `pole` and `zero` functions in MATLAB, we may obtain the poles and zeros of our inner plant, G_1, as shown below in Table 3. A positive pole value shows that the plant is unstable. This is intuitively pleasing; the robot will fall over if its body is placed upright.

Table 3: Inner Loop Plant Poles and Zeros		
Pole or Zero Number	Pole Locations	Zero Locations
1	-47.206	0
2	-5.617	None
3	8.670	None

These poles and zeros are better visualized on a Root Locus plot, seen below in Figure 5. The three poles are shown as crosses, and the single zero is shown as a circle. A simple positive increase in feedback to the system will not improve stability, as shown by the green line diverging positively from the unstable pole. As the system's zero is on the imaginary axis, a simple negative gain will still result in the system still having a positive pole. Additional poles and zeros are therefore needed to stabilize the system.

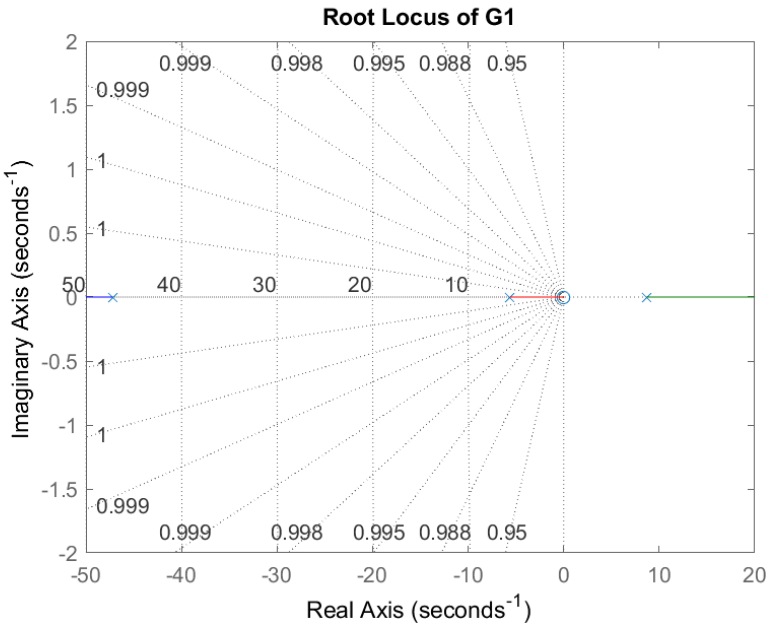


Figure 4: Root Locus Plot of Inner Loop Plant G_1, where the crosses represent poles and the circle represents the single zero. The system is clearly unstable.

Inner Loop Controller Design

Utilizing G_1 as the plant, a controller may be designed to balance the body angle of the robot. A Lead-Lag Compensator was chosen due to its benefits over PID control--namely the Lag Compensator's lack of integrator windup and Lead Compensator's high frequency attenuation. Likewise, a Lead-Lag compensator allows for easier control over the phase of the system.

A generalized Lead-Lag compensator uses two zeros and two poles and takes the form:

$$D_{lead-lag}(s) = K\frac{(s+z_{lead})}{(s+p_{lead})}\frac{(s+z_{lag})}{(s+p_{lag})}$$

Where $z_{lead}<p_{lead}$ and $z_{lag}>p_{lag}$ [7]. The poles and zeros were designed first, followed by the gain K.

The following relationships were used to find the Lead Compensator pole and zero locations: $z_{lead} = \frac{\omega_c}{\sqrt{\sigma}}, p_{lead} = \omega_c\sqrt{\sigma}$.

The ratio of the magnitude of the pole to that of the zero, $\sigma = \frac{p_{lead}}{z_{lead}}$, was chosen to be 4. This provided a large frequency range over which the Lead Compensator was in effect, but was small enough so that the Lead zero was spaced far from the Lag zero [7] .

The gain crossover frequency, ω_c, is the frequency at which the magnitude of the open loop gain of the system is $0\ dB$. This was chosen to be $10\ Hz\ =\ 20\pi\ \frac{rad}{sec}$, as it was sufficiently slower than the inner loop speed of $200\ Hz$ such that the control system would not amplify high frequency noise in the system.

The Lag Compensator utilized pole-zero cancellation to "replace" the zero at the origin with one with a larger magnitude, thereby speeding up the system [7].
The chosen poles and zeros of the system are shown in Table 4 below:

Table 4: Inner Loop Chosen Lead-Lag Compensator Poles and Zeros		
	Pole Locations	Zero Locations
Lead Compensator	-125.66	-31.42
Lag Compensator	0	-20

The resultant system was able to be stabilized using the correct negative gain. A Root Locus plot, seen in Figure 6, demonstrates that a gain of larger than $K\ =\ -0.6$ will make all poles negative, stabilizing the system.

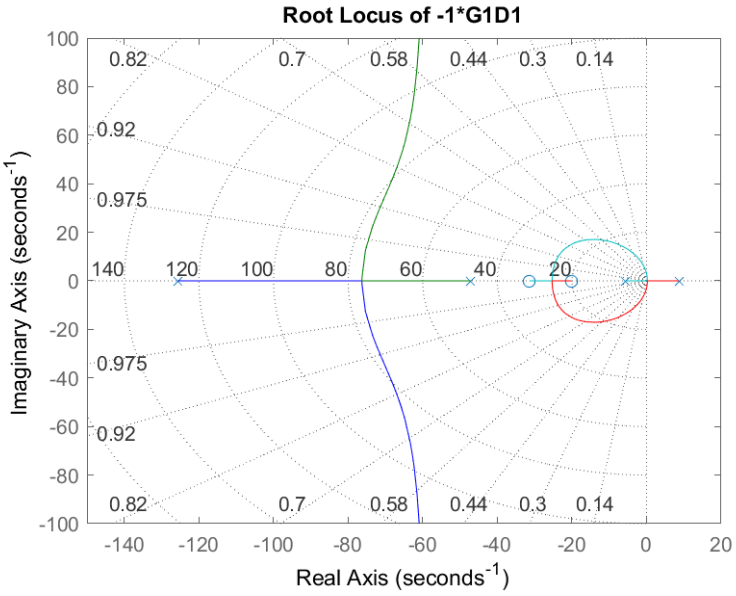


Figure 5: Root Locus Plot of the Inner Loop System. Note that the Root Locus branches demonstrated are those for negative feedback gain.

As our system utilized closed-loop feedback, the phase and magnitude of the gain may be used to check stability using Bode Plots. Of especial importance is ensuring that the phase is not 180° when the magnitude is at $0\ dB$, as this corresponds to an open loop gain of -1, or an undefined closed loop gain [7].

The Bode Plot of the controlled inner loop system is given below in Figure 7. The gain of the system was chosen to be $K = -10.5$, as it provided a gain crossover frequency of $\omega_c = 10\ Hz = 20\pi\ \frac{rad}{sec}$. The resultant system was very robust, as it had a phase margin of 47° and a gain margin of $24.8\ dB$. This means that the gain could be decreased by $24.8\ dB$ or the phase could be decreased by 47° before instability of the system.

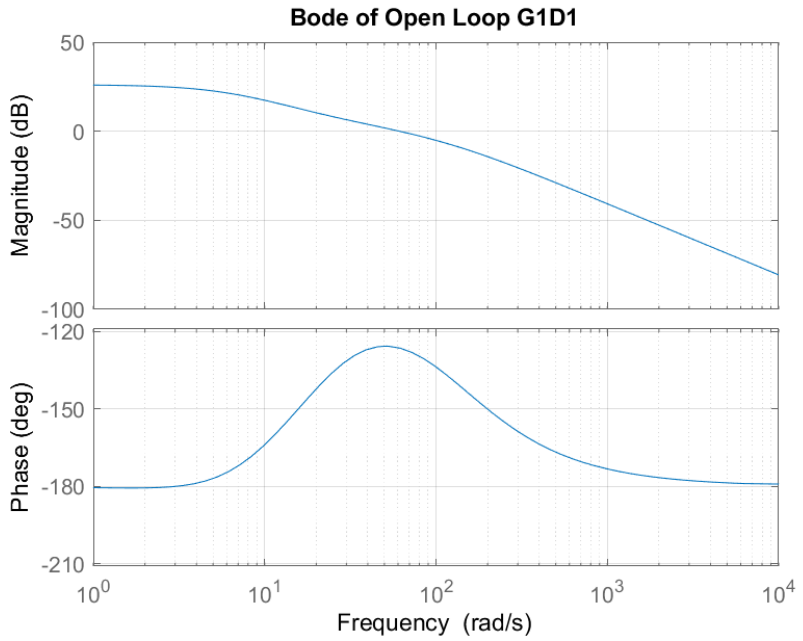


Figure 6: Bode Plot of the inner loop.

With the chosen lead and lag poles and zeros, we may compute the full Lead-Lag compensator transfer function, D_1, and the closed loop system, $T_1 = \frac{G_1 D_1}{G_1 D_1 + 1}$:

$$D_1(s) = -10.5\left(\frac{s+31.4}{s+125.7}\right)\left(\frac{s+20}{s}\right) = \frac{-10.5s^2 - 539.9s - 6597}{s^2 + 125.7s}$$

$$T_1(s) = \frac{9268 + 4.765\times10^5 s + 5.824\times10^6}{s^4 + 169.8s^3 + 1.462\times10^4 s^2 + 4.5\times10^5 s + 5.535\times10^6}$$

The stability of the controller may be further verified by taking the step response of the closed loop system, T_1, as shown in Figure 8. The system does not oscillate after reaching the peak value $0.05\ sec$ after the initial step. It also has a maximum overshoot of 21% above its settled amplitude of 1.05 . The rise time and settling time of the system are $0.02\ sec$ and$0.17\ sec$, respectively.

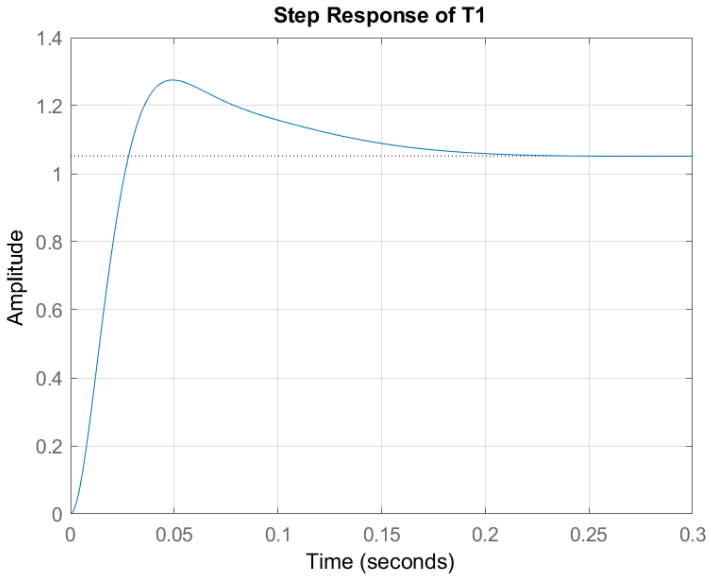


Figure 7: Step Response of the Inner Loop System. The system is stable.

Outer Loop Overview

Figure 9 below shows the control loop of the full system. As we are using subsequent loop closure, the inner loop may be ignored in the design of the outer loop controller. The Plant, G_2, inputs the eduMIP body angle, θ, and outputs the current bot wheel angle, φ . The wheel reference angle, φ_r, is set to zero so that the outer loop controller only acts when the wheel angle deviates from its starting position. The controller, D_2, inputs this wheel error angle, $\varphi_e = \varphi_r - \varphi$, and outputs the new body reference angle, θ_r. This reference angle is then fed back into the inner loop controller.

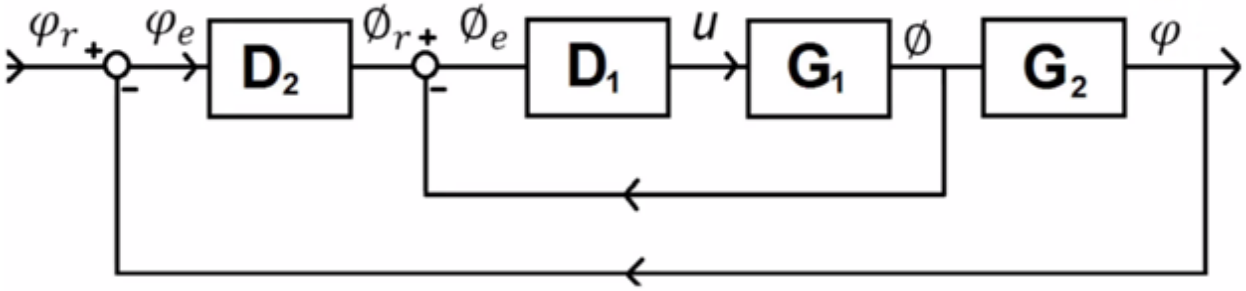


Figure 8: Outer Control Loop with Feedback.

Outer Loop Stability

The outer loop plant, G_2, has poles and zeros according to Table 5:

Table 5: Outer Loop Plant Poles and Zeros		
Pole or Zero Number	Pole Location(s)	Zero Location(s)
1	0	9.345
2	0	-9.345

These Poles and zeros are graphed on a Root Locus plot in Figure 10. Simple positive or negative gain will not stabilize the system, as one of the poles at the origin will be attracted to the positive zero, destabilizing the system.

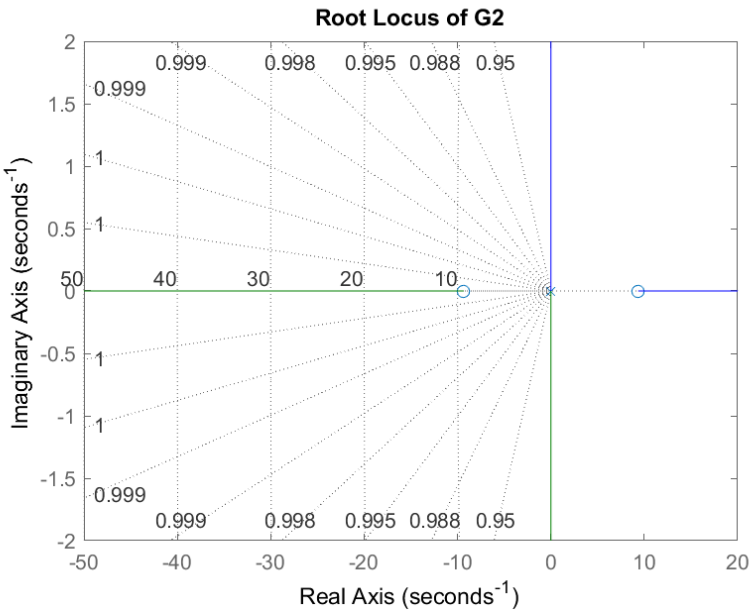


Figure 9: Root Locus of Outer Loop Plant G_2. The branches show simple positive gain. A simple negative gain will attract the two poles at the origin to the zeros along the real axis.

The Bode Plot of G_2, as shown in Figure 11, demonstrates that the phase of the plant is fully inverted at a constant 180°.

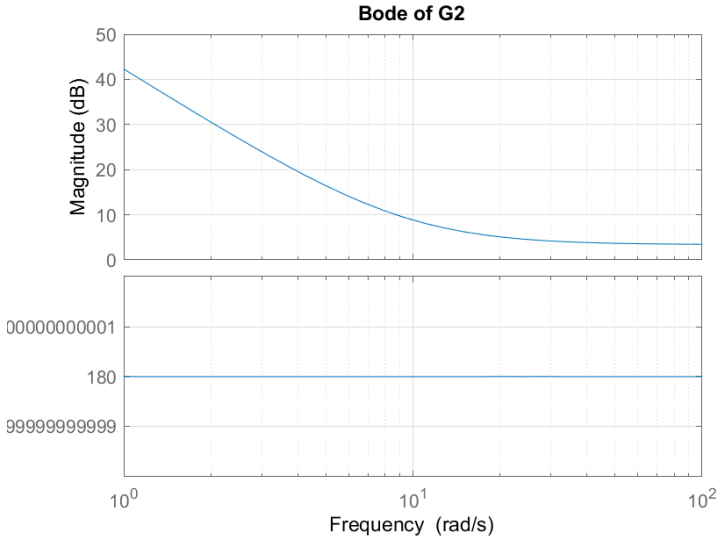


Figure 10: Bode Plot of the Outer Loop Plant, G_2

Outer Loop Controller Design

To ensure that the phase of the controlled system only fully inverted to 180° once, the number of zeros in the controller was chosen to be less than the number of poles. The lead-lag compensator therefore did not include a lag zero. The result was that the system had a phase which approached 180° at low frequencies, increased due to the lead zero at moderate frequencies, sharply decreased due to the two lead and lag poles--passing through 180° in the process--before approaching 90° at high frequencies. This behavior may be seen in the phase Bode plot of Figure 12.

Finally, the gain K of the controller was adjusted for a gain crossover frequency of $\omega_c = 1\,Hz = 2\,\pi\frac{rad}{sec}$. This value was chosen to be a factor of 20 lower than the $20\,Hz$ sampling frequency of the outer loop. It was found that a gain of $K = 7$ met this specification.

The robustness of the system may be checked using the gain and phase margin, or the value of the magnitude when the phase crossed 180° and the phase when the gain was $0\,dB$, respectively. A gain of $K = 7$ led to a 25° phase margin and a $-4.2\,dB$ gain margin, more than enough to stabilize the system. The results are reflected in the Bode plots of Figure 12.

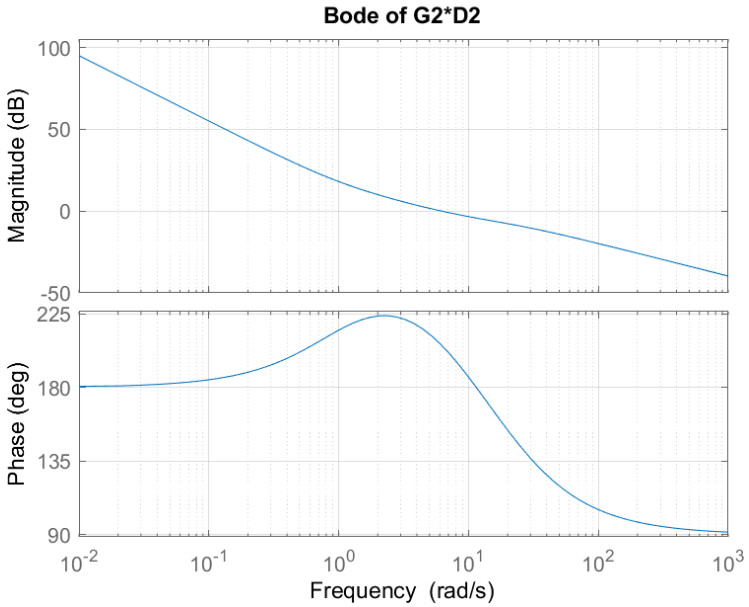


Figure 11: Bode Plot of the Open Loop System

The final chosen Pole and Zero locations are displayed in Table 6:

Table 6: Outer Loop Chosen Lead-Lag Compensator Poles and Zeros		
	Pole Locations	Zero Locations
Lead Compensator	-20	None
Lag Compensator	-8	-1

With the chosen lead and lag poles and zeros, we may compute the full controller transfer function, D_2, and the closed loop system, $T_2 = \frac{G_2 D_2}{G_2 D_2 + 1}$:

$$D_2(s) = 7\left(\frac{1}{s+20}\right)\left(\frac{s+1}{s+8}\right) = \frac{7s+7}{s^2+28s+160}$$

$$T_2(s) = \frac{-10.33s^3 - 10.33s^2 + 902.3s + 902.3}{s^4 + 17.67s^3 + 149.7s^2 + 902.3s + 902.3}$$

The Root Locus plot of the system, seen below in Figure 13, verifies that the closed loop system is stable, as suggested by the Bode plots.

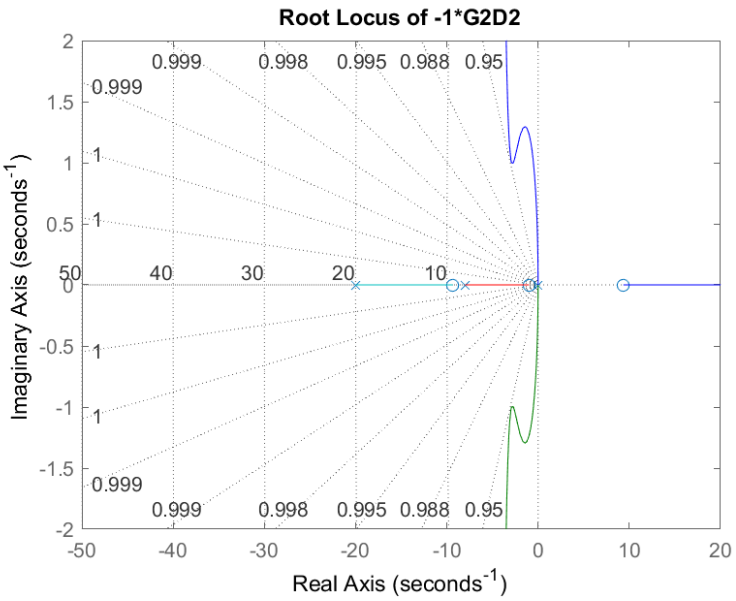


Figure 12: Root Locus of the Controlled Outer Loop System. Note that any positive gain will yield a stable system.

The stability of the outer loop may be further verified by step response of the closed loop system, as seen in Figure 14. Note that the system is of non-minimum phase. That is to say that a positive step input causes an initial decrease in amplitude of the system. This intuitively makes sense due to the conservation of angular momentum in our system--if the body falls over at a positive angle, the wheels must spin to a negative angle to "catch" the body.

The system does appear to be underdamped, as the step response oscillates after reaching its 69% overshoot $0.45\ sec$ after the initial step. The rise time, settling time, and settling amplitude of the system are $0.10\ sec$,$2.06\ sec$, and 1.00, respectively.

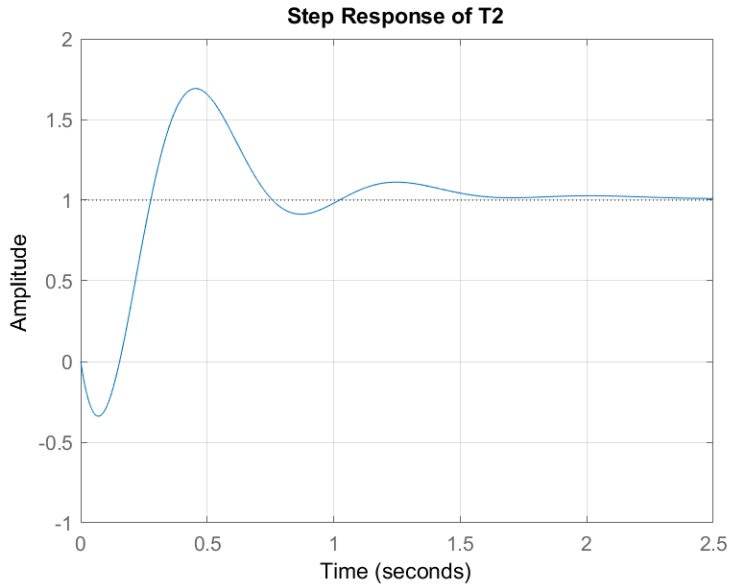


Figure 13: Step Response of the Outer Loop System with Unity Inner Loop Gain.

While the outer loop controller design ignored the inner control loop, the actual system must consider its dynamics. The full cascaded system transfer function is as follows:

$$T_{full} = \frac{G_2 T_1 D_2}{G_2 T_1 D_2 + 1} = \frac{-9.58\times10^{4}s^{6} - 6.94\times10^{6}s^{5} - 1.57\times10^{8}s^{4} - 7.56\times10^{8}s^{3} + 1.33\times10^{10}s^{2} + 1.19\times10^{11}s + 1.05\times10^{11}}{s^{9} + 218s^{8} + 2.35\times10^{4}s^{7} + 1.18\times10^{6}s^{6} + 3.13\times10^{7}s^{5} + 4.79\times10^{8}s^{4} + 4.67\times10^{9}s^{3} + 3.10\times10^{10}s^{2} + 1.19\times10^{11}s + 1.05\times10^{1}}$$

This full system is stable, as displayed by the step response in Figure 15 below. Similar to the outer loop step response, the full system is underdamped, as the step response oscillates after reaching its 95% overshoot $0.43\ sec$ after the initial step. The rise time, settling time, and settling amplitude of the system are $0.075\ sec$,$2.01\ sec$, and 1.01, respectively. Note that the inclusion of the inner loop transfer function caused more overshoot and oscillations in the full system step response as compared to only the outer loop T_2.

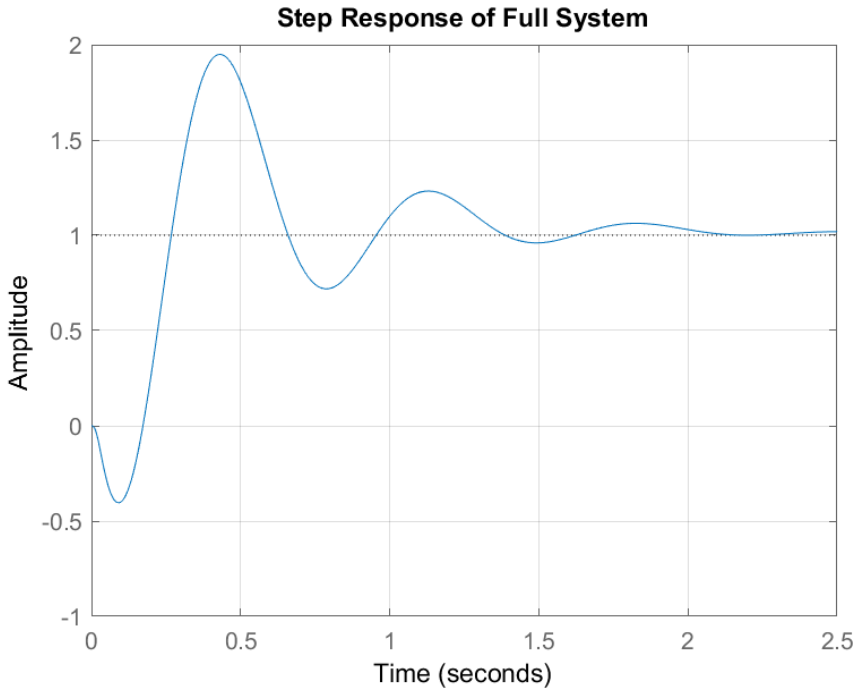


Figure 14: Cascaded System Step Response

Discrete-Time Controller Implementation

Continuous to Discrete Time--Tustin's Approximation with Prescaling

While the two Plants G_1 and G_2 are continuous as they represent a physical system in the real world, the controllers are implemented in discrete steps by the BeagleBone Blue's onboard microcontroller. For this reason, the two designed controllers need to be converted from continuous to discrete time.

This was accomplished by utilizing the `c2d` function in MATLAB with Tustin's Approximation. Tustin's Approximation maps our Transfer Functions from continuous to discrete time by utilizing the approximation: $z = e^{sT} \approx \frac{1+0.5sT}{1-0.5sT}$, where T is the sampling period. Prescaling the sampling period about each control system's most important frequency--its crossover frequency--ensures accurate discretization of the controllers [8].

The inner and outer loop discrete time controllers are:

$$D_1(z) = \frac{u}{\theta_e} = \frac{-9.048z^2+15.92z-6.994}{z^2-1.522z+0.5219}$$

$$D_2(z) = \frac{\theta_r}{\varphi_e} = \frac{0.09966z^2+0.004863z-0.0948}{z^2-z+0.2221}$$

To implement these controllers in software, we must rearrange the above equations. Recalling that the Z-Transform product corresponds to a discretized time step, we may shift the equations back in time and solve for our current time outputs to the equations. We obtain the following, which may be directly implemented into a software of our system for the inner and outer loop controller, respectively:

$$u_{(n)} = -9.048\,\theta_{e(n)} + 15.92\,\theta_{e(n-1)} - 6.994\,\theta_{e(n-2)} + 1.522\,u_{(n-1)} - 0.5219\,u_{(n-2)}$$

$$\theta_{r(n)} = 0.09966\,\varphi_{e(n)} + 0.004863\,\varphi_{e(n-1)} - 0.0948\,\varphi_{e(n-2)} + \theta_{r(n-1)} - 0.2221\,\theta_{r(n-2)}$$

Note that the subscript n represents the time step associated with each variable.

<u>eduMIP Controller Implementation</u>

Upon implementation of the above discrete-time controllers in the eduMIP, the robot would not balance. The gain of the controllers was too high, causing the duty cycle to maximize or minimize, with little in-between. The result was that the motors would shutter back and forth, and the robot would fall over. Decreasing the constant gain for the system inputs (the value K in the Lead-Lag equations) allowed the robot to balance. The modified controller transfer functions are as follows:

$$D_1 = -3.15\left(\frac{s+31.4}{s+125.7}\right)\left(\frac{s+20}{s}\right) = \frac{-3.15s^2-162s-1979}{s^2+125.7s}$$

$$D_2 = 0.7\left(\frac{1}{s+20}\right)\left(\frac{s+1}{s+8}\right) = \frac{0.7s+0.7}{s^2+28s+160}$$

Rearranging the equations as they are implemented in software:

$$u_{(n)} = 0.3\,(-9.048\,\theta_{e(n)} + 15.92\,\theta_{e(n-1)} - 6.994\,\theta_{e(n-2)}) + 1.522\,u_{(n-1)} - 0.5219\,u$$

$$\theta_{r(n)} = 0.1\,(0.09966\,\varphi_{e(n)} + 0.004863\,\varphi_{e(n-1)} - 0.0948\,\varphi_{e(n-2)}) + \theta_{r(n-1)} - 0.2221$$

It would seem that the lower gains slowed down the time response of the system, allowing the motors to more smoothly catch the falling body. As the only modification required to stabilize the system was the constant gain K, it is clear that the designed Lead-Lag Compensators worked as intended. What is less clear is why the modification was required. It is possible that the assumptions and simplifications used to derive the system model, such as linearization, affected the model's accuracy. It is also possible that the chosen crossover frequencies were too high, which may have been fixed by lowering the gain of the systems.

BeagleBone Blue Software

Low-Level Software: C-Code Framework

The above discrete-time controllers were implemented into the BeagleBone Blue eduMIP with assistance from the Robot Control Library [9]. See Appendix C for the documented C Code.

An interrupt was triggered at a rate of $200\ Hz$ which read the gyroscope and accelerometer data from the IMU. A complementary filter was then used to find the current pitch orientation of the eduMIP from the raw sensor readings. The interrupt also executed the inner control loop and wrote the output duty cycle to each of the two motors.

The outer loop was executed by a thread which ran at a constant $20\ Hz$. The thread read each wheel encoder value, averaged the values for use in the controller, and executed the outer loop controller, thereby updating the reference angle for the inner loop controller.

High-Level Software: State Machine

The state diagram of the eduMIP C Code is displayed below in Figure 16. Upon execution of the program, the eduMIP is in state `Paused`. In this state, the controllers are not executed, the motors are inoperational, and the display LED is solid red. If the "Pause" button is tapped--meaning that the button is pushed down for less than $0.5\,sec$, the state is changed to `Running`. In the `Running` state, the controllers are executed, the motors will attempt to balance the eduMIP on a flat surface, and the display LED is solid green. Tapping the "Pause" button will cause the state to change back to `Paused`. If at any point during either the `Paused` or `Running` states the "Pause" button is held down for longer than $0.5\,sec$, the state will change to `Exiting Program`. This state will cease motor movement and IMU sensor readings, reset the LED and exit the BeagleBone Blue out of the program.

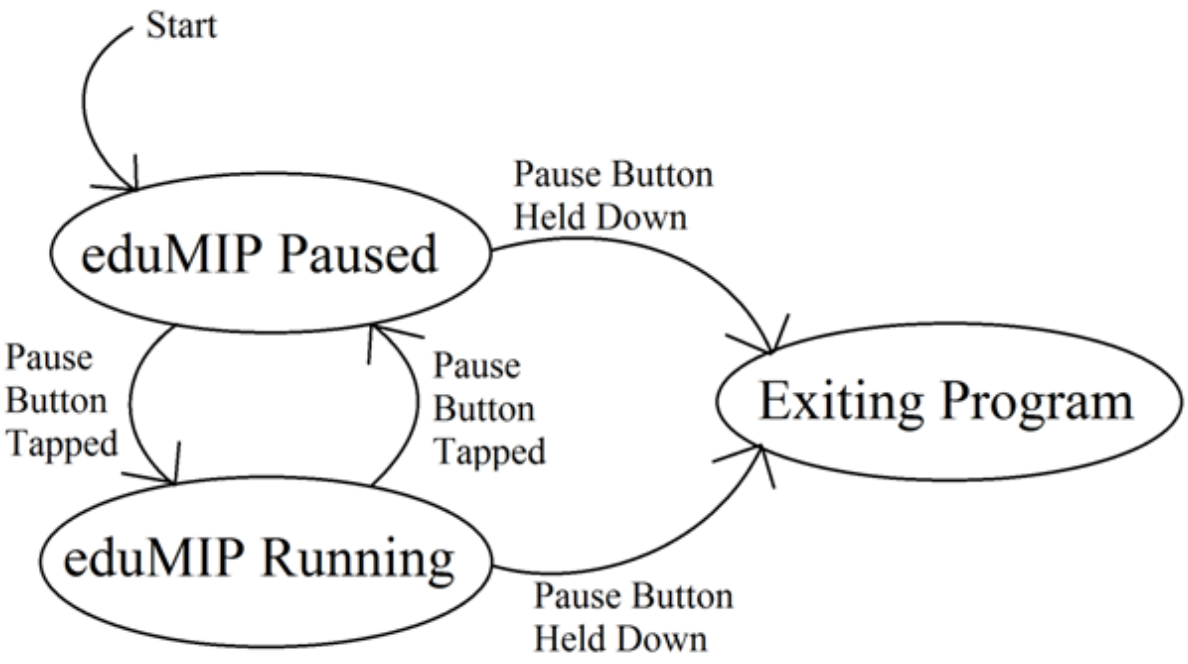


Figure 15: eduMIP State Diagram

Conclusion

The eduMIP was the perfect introduction to mechatronics. Other systems usually require fighting with software, hardware, and control theory simultaneously. This system has the advantage of eliminating, or at least minimizing, the many issues that hardware inevitably causes.

While Newtonian Mechanics are useful for many derivations, for this system Lagrangian Mechanics were found to be much more straightforward. In some cases, the Lagrangian derivation also helped with comprehension of the sources of the many forces in the multiple reference frames of the Newtonian derivation.

It is also important to stress how powerful of a tool Bode, Root Locus, and Step Input Plots are for SISO control system design. For this project, Root Locus Plots served as the primary control tool for the inner loop, while Bode Plots aided in design of the outer loop. The Step Input Plots were always useful for checking stability of the systems.

Due to inherent disparities between the derived model and the actual physical system due to engineering assumptions and linearization of the derived equations, practical implementation of the controller in software required some final tweaking. This is not to say that modelling systems is not a useful tool. Rather, the model of the system is just that: a model.

I believe that I will return to this project in the future. The next step will likely be to implement an LQR controller utilizing state space. I would also like to utilize state feedback linearization to attempt a nonlinear controller. The fact that we may balance a physical object with some software never ceases to amaze me.

Appendix A: References

[1] “EduMIP Kits - AVAILABLE NOW.” *UCSD Robotics*, UCSD Flow Control & Coordinated Robotics Labs, www.ucsdrobotics.org/edumip.

[2] Bewley, Thomas. “Chapter 17 Kinematics and Dynamics.” *Numerical Renaissance*, First ed., Renaissance Press, 2018, pp. 477–520. http://numerical-renaissance.com/NR.pdf

[3] Strawson, James “Balancing eduMIP”. https://www.coursehero.com/file/27260354/Balancing-EduMiPpdf/

[4] “Equations of Motion for the Inverted Pendulum (2DOF) Using Lagrange's Equations”. https://www.youtube.com/watch?v=Fo7kuUAHj3s

[5] “2.003SCRecitation 8 Notes: Cart and Pendulum (Lagrange)”. *Michigan Institute of Technology***.** https://ocw.mit.edu/courses/mechanical-engineering/2-003sc-engineering-dynamics-fall-2011/lagrange-equations/MIT2_003SCF11_rec8notes1.pdf

[6] The Torque Equation and the Relationship with DC Motors. https://www.motioncontroltips.com/torque-equation/

[7] Bewley, Thomas. “Chapter 19: Classical Control Systems.” *Numerical Renaissance*, First ed., Renaissance Press, 2018, pp. 586-588. http://numerical-renaissance.com/NR.pdf

[8] https://www.mathworks.com/help/control/ug/continuous-discrete-conversion-methods.html

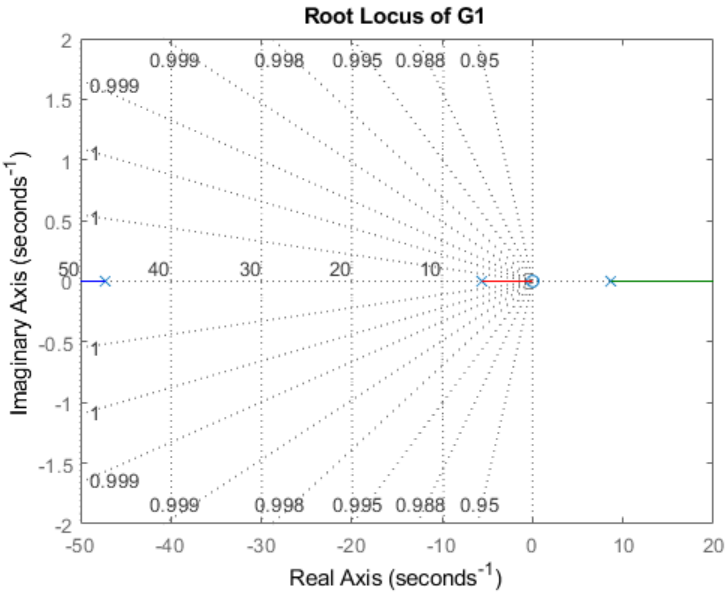
[9] “Robot Control Library Documentation”. *Strawson Design.* http://strawsondesign.com/docs/librobotcontrol/

Appendix B: MATLAB Code

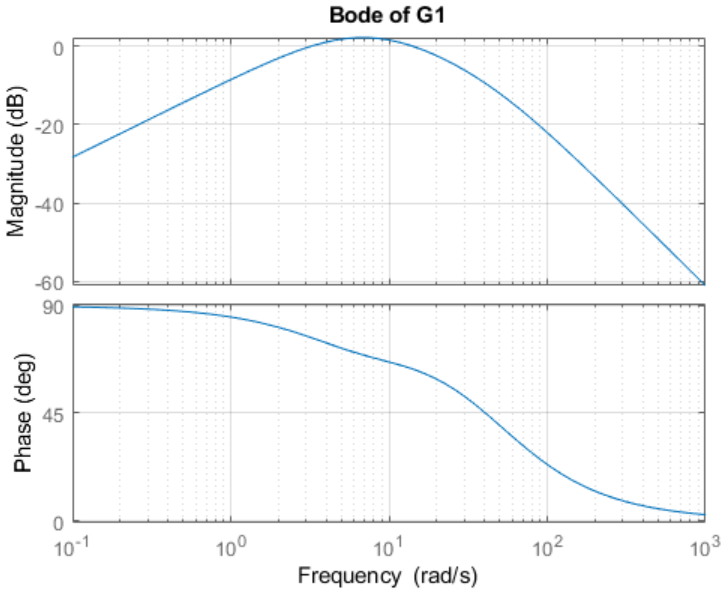
```
% . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
% .
% .  eduMIP BeagleBone Blue Controller Design 3.0
% .
% .  Title: edupMIP_Controller_Design_v3
% .  Description:  Designs two controllers to balance an inverted pendulum
% .  Author:  Robert Ketchum
% .  Date Created: April 7, 2021
% .  Last Modified: April 11, 2021
% . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .

close all  % Close all files
clear      % Clear all variables
clc        % Clear command line

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%PART 1: Characterize the robot's equations of motion
% and find the inner and outer loop plants: G1 and G2
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

Vb = 7.4;       % battery voltage [V]
wf = 1760;      % motor no load angular velocity [rad/sec]
st = 0.003;     % stall torque of motor sans gearbox [N m]
Gr = 35.5555;   % gearbox gear ratio [unitless]
Im = 3.6*10^-8; % motor armature moment of inertia [kg m^2]
rw = 0.034;     % wheel radius [r]
mw = 0.027;     % mass of each wheel [kg]
mb = 0.180;     % mass of robot body [kg]
L = 0.0477;     % length from center of mass to wheel axis [m]
Ib=0.000263;   % Robot body moment of inertia [kg m^2]
g=9.81;        % gravity [m/s^2]

k=st/wf;      % torque constant [N m/s]

Iw=2*((mw*rw^2)/2+Gr^2*Im); % wheel inertia [kg m^2]

%transfer functions to be used in G1
BA=tf([mb*rw*L -2*Gr^2*k 0],[Iw+(mb+mw)*rw^2 2*Gr^2*k 0]);
CA=tf([2*Gr*st],[Iw+(mb+mw)*rw^2 2*Gr^2*k 0]);
ED=tf([Ib+mb*L^2 2*Gr^2*k -mb*g*L],[mb*rw*L -2*Gr^2*k 0]);
FD=tf([-2*Gr*st],[mb*rw*L -2*Gr^2*k 0]);


G1=PolyDiv(CA-FD,BA-ED);
```

```
G1=minreal(G1);

pole1=pole(G1); % poles of G1
zero1=zero(G1); % zeros of G1

% transfer functions to be used in G2
EF = tf([Ib+mb*L^2 2*Gr^2*k -mb*g*L],[-2*Gr*st]);
BC = tf([mb*rw*L -2*Gr^2*k 0],[2*Gr*st]);
AC = tf([Iw+(mb+mw)*rw^2 2*Gr^2*k 0],[2*Gr*st]);
DF = tf([mb*rw*L -2*Gr^2*k 0],[-2*Gr*st]);

G2=PolyDiv(EF-BC,AC-DF);
G2=minreal(G2);

pole2=pole(G2); % poles of G2
zero2=zero(G2); % zeros of G2

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%PART 2: Design the inner loop controller: D1
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

DT1 = 200;           % sampling fz [Hz]
cross1 = (DT1)/20;   % crossover fz [Hz]
wc1 = cross1*2*pi;   % crossover fz [rad/sec]
alpha1 = 4;          % gives us that p1/z1=4

p1=wc1*sqrt(alpha1); % -1 * lead pole
z1=wc1/sqrt(alpha1); % -1 * lead zero
z2=20;               % -1 * lag zero
p2=0;                % -1 * lag pole
k1=-10.5;            % gain
D1=tf([1 (z1+z2) z1*z2],[1 (p2+p1) p1*p2]);
Dk1=D1*k1;           % Continuous Time Inner Loop Controller
G1D1=G1*D1*k1;       % Open Loop Transfer Function
T1=k1*G1*D1/(k1*G1*D1+1);
T1=minreal(T1);      % Closed Loop Transfer Function
T1_step_info = stepinfo(T1);

%Convert from continuous to Discrete Time using Tustin's Approximation with
%Prewarping
prewarp1=2*(1-cos(wc1/DT1))/(1/DT1*wc1*sin(wc1/DT1)); % Prewarp factor
dop1=c2dOptions('Method','tustin','PrewarpFrequency',prewarp1);
Dz1=c2d(Dk1,1/DT1,dop1); % Discrete Time Inner Loop Controller
```

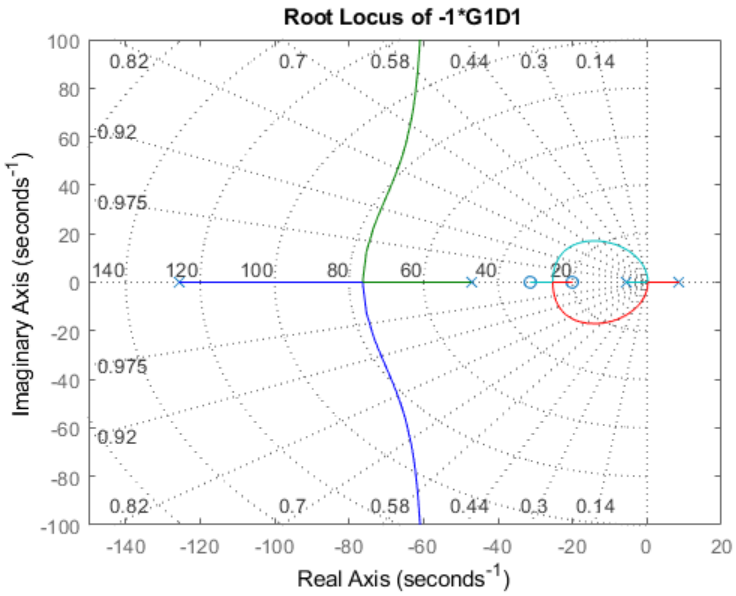
```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%PART 3: Design the outer loop controller, D2
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

DT2=20;          % sampling fz for outer loop [Hz]
cross2=(DT2)/20; % crossover fz [Hz]
wc2=cross2*2*pi; % crossover fz [rad/sec]

p4=tf([1],[1 20]);  % lead pole
z3=tf([1 1],[1]);   % lag zero
p3=tf([1],[1 8]);   % lag pole
k2=7;               % gain
D2=p3*z3*p4*k2;     % Continuous Time Controller 2
G2D2=G2*D2;         % Open Loop Transfer Function
T2=G2*D2/(G2*D2+1);
T2=minreal(T2);     % Closed Loop Transfer Function
T2_step_info = stepinfo(T2);

% Converting outer loop controller into Discrete Time
prewarp2=2*(1-cos(wc2/DT2))/(1/DT2*wc2*sin(wc2/DT2)); % prewarp factor
dop2=c2dOptions('Method','tustin','PrewarpFrequency',prewarp2);
Dz2=c2d(D2,1/DT2,dop2); % Outer loop discrete transfer function

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%PART 4: Full System Cascaded Transfer Function
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

L2=G2*T1*D2;
T2f=L2/(1+L2);
T2f=minreal(T2f); % Full system transfer function
T2f_step_info = stepinfo(T2f);

%%%%%%%%%%%%%%%%%%
%PART 4: Plotting
%%%%%%%%%%%%%%%%%%

%Plots for Inner Loop
figure(1)
rlocus(G1);
axis([-50 20 -2 2])
title("Root Locus of G1")
grid on
saveas(1,'G1_RL.png')
```



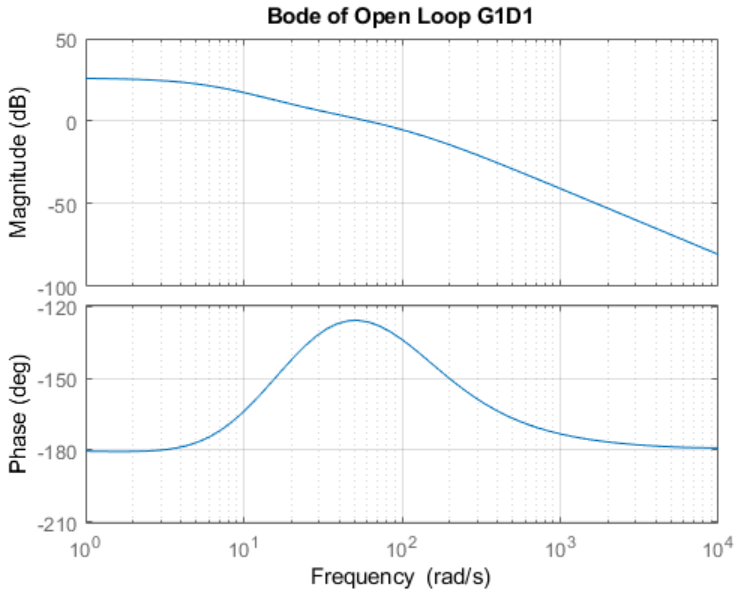

```
figure(2)
bode(G1);
title("Bode of G1")
grid on
saveas(2,'G1_Bode.png')
```



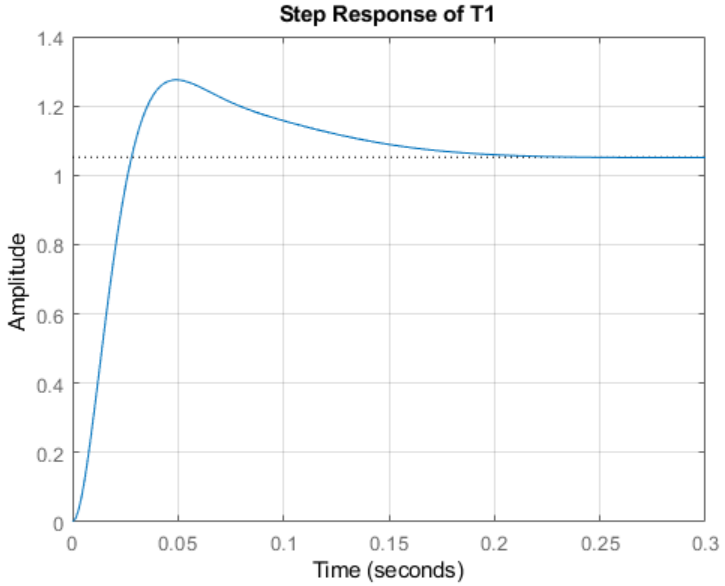
```
figure(3)
rlocus(G1D1/-k1);
grid on
title("Root Locus of -1*G1D1")
axis([-150 20 -100 100])
saveas(3,"G1D1_RL.png")
```



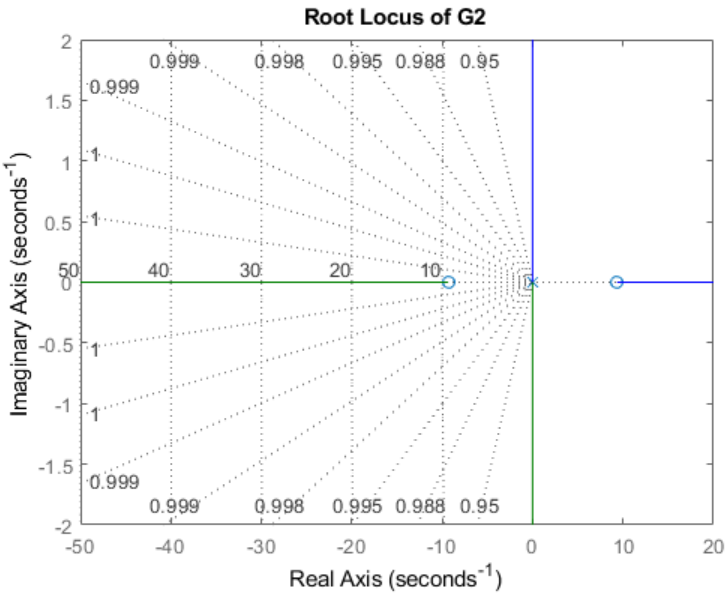

```
figure(4)
bode(G1D1)
grid on
title("Bode of Open Loop G1D1")
saveas(4,"G1D1_Bode.png")
```



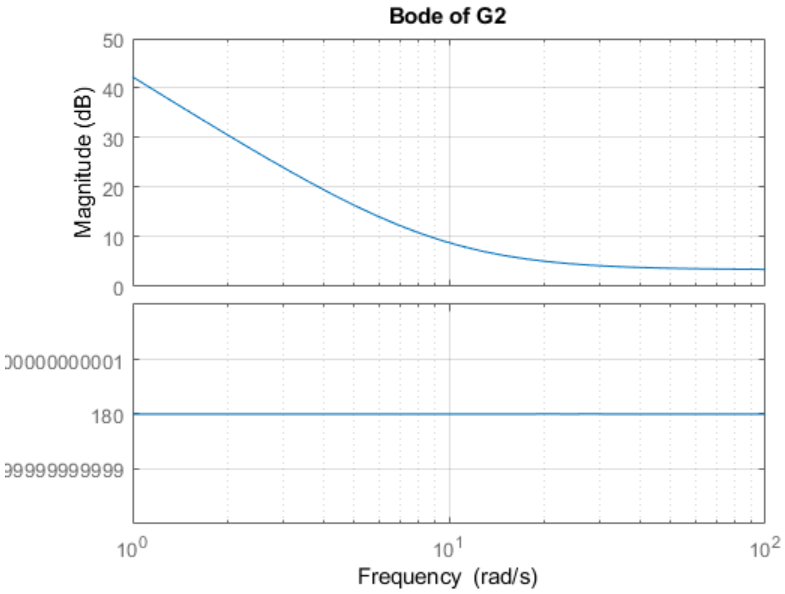

```
figure(5)
step(T1);
grid on
axis([0 0.3 0 1.4])
title("Step Response of T1")
saveas(5,"T1_Step.png")
```



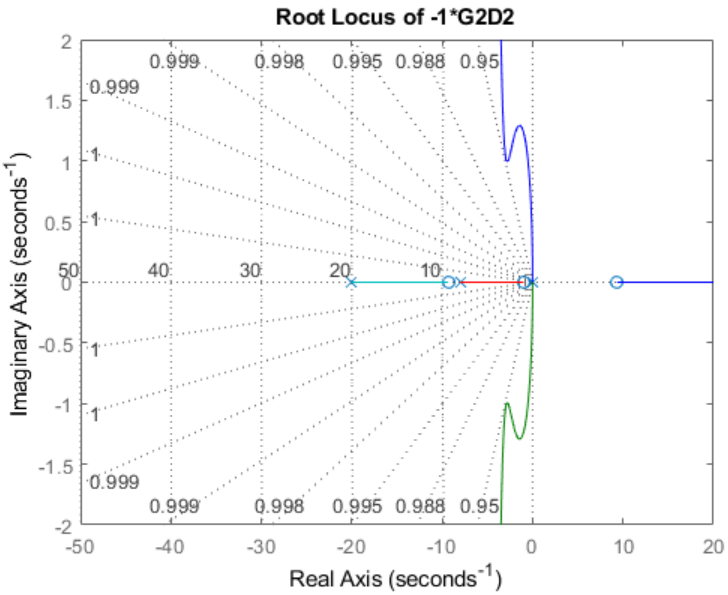
```
%Plots for Outer Loop
figure(6)
rlocus(G2);
grid on
title("Root Locus of G2")
axis([-50 20 -2 2])
saveas(6,"G2_RL.png")
```



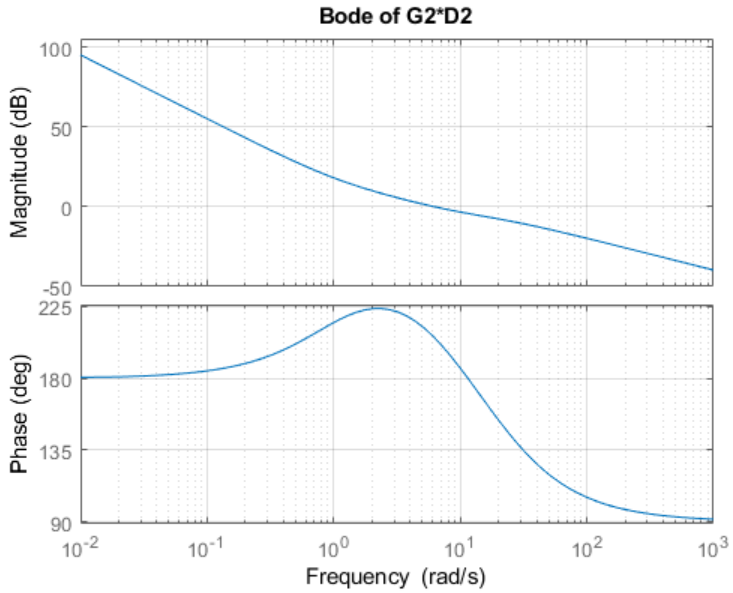

```
figure(7)
bode(G2);
title("Bode of G2")
grid on
saveas(7,"G2_Bode.png")
```



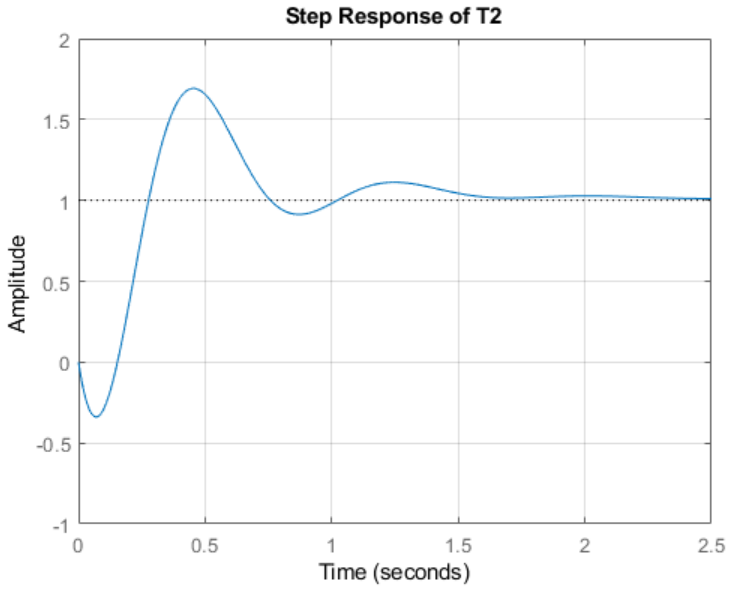

```
figure(8)
rlocus(G2D2/k2);
grid on
title("Root Locus of -1*G2D2")
axis([-50 20 -2 2])
saveas(8,"G2D2_RL.png")
```



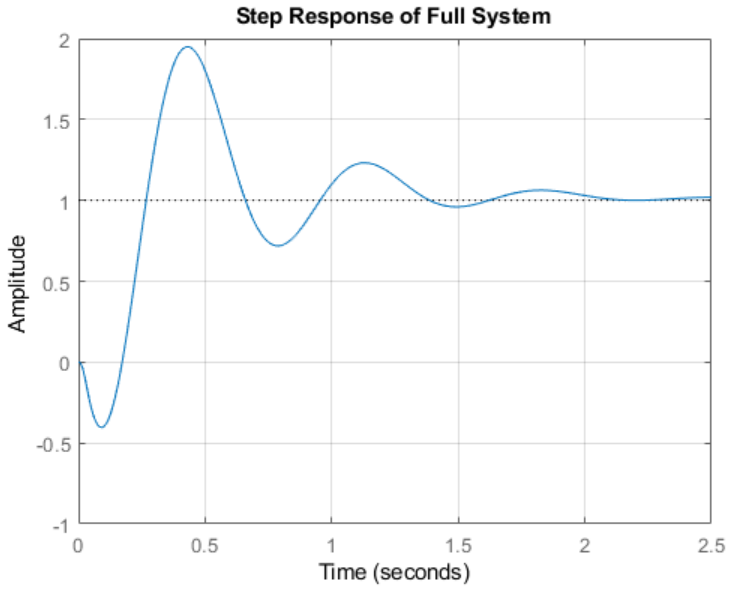
```
figure(9)
bode(G2D2)
grid on
title("Bode of G2*D2")
saveas(9,"G2D2_Bode.png")
```




```
figure(10)
step(T2);
title("Step Response of T2")
axis([0 2.5 -1 2])
grid on
saveas(10,"T2_Step.png")
```




```
%Plot for Full System
figure(11)
step(T2f)
axis([0 2.5 -1 2])
grid on
title("Step Response of Full System")
saveas(11,"T2f_Step.png")
```



```
 %%%%%%%%%%%%%%%%%%%%%%%%
 %PART 5: Function PolyDiv
 %%%%%%%%%%%%%%%%%%%%%%%%
 function [d,b]=PolyDiv(b,a)
 % function [d,b]=PolyDiv(b,a)
 % Perform polynomial division of a into b, resulting in d with remainder in the
modified b.
 % See <a href="matlab:NRweb">Numerical Renaissance</a>, Appendix A, for further
discussion.
 % Part of <a href="matlab:help NRC">Numerical Renaissance Codebase 1.0</a>, <a
href="matlab:help NRchapAA">Appendix A</a>; see webpage for <a href="matlab:help
NRcopyleft">copyleft info</a>.

 m=length(b); n=length(a); if m<n d=0; else
   if strcmp(class(b),'sym')|strcmp(class(a),'sym'), syms d, end
   for j=1:m-n+1, d(j)=b(1)/a(1); b(1:n)=PolyAdd(b(1:n),-d(j)*a); b=b(2:end);
end, end
 end % function PolyDiv
```

Appendix C: BeagleBone Blue C-Code

```
/** File  Name: edumipv3.c
 *
 * Project Name: eduMIP BeagleBone Blue Controller Design 3.0
 *
 * Author: Robert Ketchum (skeleton of file from Robot Control Library at strawsondesign.com)
 *
 * Description: Balances an eduMIP inverted pendulum robot using an onboard
 * IMU with two controllers which were created in Matlab.
 *
 * Date Created: November 2018
 *
 * Last Modified: 11 April 2021 -- updated controller
 *
 * To compile:
 * gcc -Wall edumipv3.c -o edumipv3 -lm -lrt -l:librobotcontrol.so.1
 *
 * To run:
 * sudo ./edumipv3
 *
 *
 **/

//Libraries to Include
#include <stdio.h>
#include <math.h>
#include <robotcontrol.h>

// function declarations
void on_pause_press();
void on_pause_release();
static void imu_interupt_function(void); //mpu interrupt routine
static void* my_outer_loop_thread(void* unused);   //background outer loop thread

//variable declerations
static rc_mpu_data_t mpu_data; //mpu data

double raw_gyro              = 0;         //raw gyro data [rad/sec]
double gyro_angle[2]         = {0, 0};    //euler-integrated gyro angle [rad]
double filtered_gyro         = 0;         //high passed gyro angle [rad]
double filtered_acc          = 0;         //low passed accelerometer angle [rad]
double current_angle         = 0;         //complementary filter output. robot pitch angle [rad]
double raw_accel             = 0;         //accelerometer pitch angle [rad]
double pitch_err[3]          = {0, 0, 0}; //robot pitch error angle with relation to vertical [rad]
double uk[3]                 = {0, 0, 0}; //duty cycle. Output of inner loop controller
int    encoder_reading[2]    = {0, 0};    //wheel angle [rad]
int    encoder_offset[2]     = {0, 0};    //wheel offset [rad]
double wheel_angle[3]        = {0, 0, 0}; //current wheel angles [rad]. the third is average of the two wheels
double ref_pitch_angle[3] = {0, 0, 0};    //outer loop controller output [rad]
double wheel_error[3]        = {0, 0, 0}; //wheel position error [rad]


//controller value declarations
const double inner_den[2] = {1.522, -0.5219};
const double inner_num[3] = {-9.048, 15.92, -6.994};
const double outer_den[2] = {1, -0.2221};
const double outer_num[3] = {0.09966, 0.004863, -0.0948};

//Constant declarations
const double PI            = 3.1415926;
const double WC            = 157.9;    //complimentary filter cutoff fz [rad/sec]
const double K1            = 0.3;      //inner loop scaling factor
const double K2            = 0.1;      //outer loop scaling factor
const double REV           = 341.8648; //encoder value for 1 wheel revolution [ticks/rev]
const double INNER_LOOP_DT = 0.005;    //inner loop step size [seconds]
const double OUTER_LOOP_DT = 50000;    //outer loop step size [microseconds]
const double OFFSET = 0.3457;          //balanced robot angle offset from vertical [rad]


#define I2C_BUS 2             // I2C data bus
#define GPIO_INT_PIN_CHIP 3   // Interrupt pin chip
#define GPIO_INT_PIN_PIN 21   // Interrupt pin


int main()
{
    // create our thread for the outer loop controller
    pthread_t thread_outer_loop = 0;
```

```
    // make sure another instance isn't running
    if(rc_kill_existing_process(2.0)<-2) return -1;

    // start balance stack to control wheel setpoints
    if(rc_pthread_create(&thread_outer_loop, my_outer_loop_thread, (void*) NULL, SCHED_OTHER,  0)){
        fprintf(stderr, "failed to start thread2\n");
        return -1;
    }

    // start signal handler so we can exit cleanly
    if(rc_enable_signal_handler()==-1){
        fprintf(stderr,"ERROR: failed to start signal handler\n");
        return -1;
    }

    // initialize pause button
    if(rc_button_init(RC_BTN_PIN_PAUSE, RC_BTN_POLARITY_NORM_HIGH,
                                RC_BTN_DEBOUNCE_DEFAULT_US)){
        fprintf(stderr,"ERROR: failed to initialize pause button\n");
        return -1;
    }

    //initialize encoder
    if(rc_encoder_eqep_init()){
        fprintf(stderr,"ERROR: failed to run rc_encoder_eqep_init\n");
        return -1;
    }

    //initialize mpu + imu
    rc_mpu_config_t conf = rc_mpu_default_config();
    conf.i2c_bus = I2C_BUS;
    conf.gpio_interrupt_pin_chip = GPIO_INT_PIN_CHIP;
    conf.gpio_interrupt_pin = GPIO_INT_PIN_PIN;
    conf.dmp_sample_rate = 200;                  // hz to execute imu interrput
    conf.dmp_fetch_accel_gyro=1;
    conf.orient = ORIENTATION_Y_UP;              // imu orientation for edumip
    if(rc_mpu_initialize_dmp(&mpu_data, conf)){
        printf("rc_mpu_initialize_failed\n");
        return -1;
    }
    rc_mpu_set_dmp_callback(&imu_interupt_function);
    int freq_hz = RC_MOTOR_DEFAULT_PWM_FREQ;

    //initialize motor
    if(rc_motor_init_freq(freq_hz)) return -1;

    // Assign functions to be called when button events occur
    rc_button_set_callbacks(RC_BTN_PIN_PAUSE,on_pause_press,on_pause_release);

    // make PID file
    rc_make_pid_file();
    printf("\n prints out orientation angles \n");

    // Keep looping until state changes to EXITING
    rc_set_state(PAUSED);
    while(rc_get_state()!=EXITING){
        // change LED colors based on state
        if(rc_get_state()==RUNNING){
            rc_led_set(RC_LED_GREEN, 1);
            rc_led_set(RC_LED_RED, 0);
        }
        else{
            rc_led_set(RC_LED_GREEN, 0);
            rc_led_set(RC_LED_RED, 1);
        }
        rc_usleep(1000000);
    }

    //turn off LEDs and close file descriptors
    rc_led_set(RC_LED_GREEN, 0);
    rc_led_set(RC_LED_RED, 0);
    rc_led_cleanup();
    rc_button_cleanup();    // stop button handlers
    rc_remove_pid_file();   // remove pid file LAST
    return 0;
}

static void* my_outer_loop_thread(__attribute__ ((unused)) void* ptr){
    while(rc_get_state()!=EXITING){
    // update steps and print out data if state is PAUSED or RUNNING
```

```
        if(rc_get_state()==RUNNING){
            //Only run controller if state is RUNNING

            //find wheel position from encoder value
            encoder_reading[0] = rc_encoder_read(2) - encoder_offset[0]; //read encoder value for each wheel
            encoder_reading[1] = rc_encoder_read(3) - encoder_offset[1];
            wheel_angle[0]     = encoder_reading[0];                     //change to double precision
            wheel_angle[1]     = encoder_reading[1];
            wheel_angle[0]     = wheel_angle[0] / REV;                   //change encoder value to radians
            wheel_angle[1]     = -wheel_angle[1] / REV;
            wheel_angle[2]     = (wheel_angle[0] + wheel_angle[1]) / 2;  //take average encoder value
            wheel_angle[2]     = wheel_angle[2] - pitch_err[0];          //subtracts off balanced IMU offset
            wheel_error[0]     = -wheel_angle[2];                        //wheel angle error

            //outer loop controller--updates robot reference angle for use in inner loop
            ref_pitch_angle[0] = K2*(outer_num[0]*wheel_error[0] + outer_num[1]*wheel_error[1] +   ...
                                 outer_num[2]*wheel_error[2])  + outer_den[0]*ref_pitch_angle[1] + ...
                                 outer_den[1]*ref_pitch_angle[2];
        }
        if(rc_get_state()==PAUSED){
            //If the state is PAUSED, store the encoder values so that
            //the outer loop controller is not messed up if the wheels are spun
            encoder_offset[0] = rc_encoder_read(2);
            encoder_offset[1] = rc_encoder_read(3);
        }
        //update steps
        ref_pitch_angle[2] = ref_pitch_angle[1];
        ref_pitch_angle[1] = ref_pitch_angle[0];
        wheel_error[2] = wheel_error[1];
        wheel_error[1] = wheel_error[0];

        printf("wheel error: %6.2f    Pitch Angle: %6.2f      pitch error: %6.2f   duty cycle: %6.2f \n ",...
                                 wheel_error[0]*RAD_TO_DEG, current_angle, pitch_err[0]*RAD_TO_DEG, uk[0]);
        rc_usleep(OUTER_LOOP_DT);
    }
    return NULL;
}


void imu_interupt_function(void){
    //Find orientation of robot from accelerometer+gyro data.
    //Then calculate the motor duty cycle from the inner loop control system.
    //Finally, apply duty cycle to motors.

    //accelerometer data
    raw_accel = atan2(mpu_data.accel[1],mpu_data.accel[2]);      //accelerometer data [rad]

    //gyro data
    raw_gyro      = mpu_data.gyro[0] * DEG_TO_RAD;
    gyro_angle[0] = gyro_angle[1] + INNER_LOOP_DT * raw_gyro;    //find angle using euler's integration method

    //complementary filter--high pass gyro data, low pass accelerometer data
    double HIGH_PASS = 1 / (INNER_LOOP_DT * WC / 2 / PI + 1);               //generic high pass filter
    double LOW_PASS  = (INNER_LOOP_DT * WC / 2 / PI) * HIGH_PASS;           //generic low pass filter
    filtered_acc  = LOW_PASS * raw_accel + (1 - LOW_PASS) * filtered_acc;    //filter accel data
    filtered_gyro = HIGH_PASS * filtered_gyro + HIGH_PASS*(gyro_angle[0] - gyro_angle[1]); //filter gyro data
    current_angle = filtered_gyro + filtered_acc;                           //add filtered angles together
    gyro_angle[1] = gyro_angle[0];                                          //update the previous gyro step

    //find error angle, duty cycle
    current_angle = current_angle - PI/2;                                   //angle with relation to vertical
    pitch_err[0]  = -ref_pitch_angle[0] - current_angle - OFFSET;           //robot error angle
    if ((((pitch_err[0] - pitch_err[1]) > 0.1) || ((pitch_err[1] - pitch_err[0]) > 0.1)) || ...
                                 (rc_get_state()!=RUNNING)){
        // Make sure the state is set to RUNNING to move the motors
        // Also prevent wheel motion during discontinuous angle reads.
        uk[0] = 0;
    }
    else {
        //inner loop controller
        uk[0] = K1*(inner_num[0]*pitch_err[0] + inner_num[1]*pitch_err[1] + inner_num[2]*pitch_err[2]) + ...
                                 inner_den[0] * uk[1] + inner_den[1] * uk[2];
    }
    if (uk[0] > 1){
        //max duty cycle
        uk[0] = 1;
    }
    if (uk[0] < -1){
        //max duty cycle
        uk[0] = -1;
    }
```

```
    if (pitch_err[0] * RAD_TO_DEG > 70){
        //stop wheels if robot knocked over
        uk[0] = 0;
    }
    if (pitch_err[0] * RAD_TO_DEG < -70){
        //stop wheels if robot knocked over
        uk[0] = 0;
    }

    //update steps
    pitch_err[2] = pitch_err[1];
    pitch_err[1] = pitch_err[0];
    uk[2]        = uk[1];
    uk[1]        = uk[0];

    //apply duty cycle
    rc_motor_set(3, uk[0]);
    rc_motor_set(2,-uk[0]);
    return;
}

void on_pause_release()
{
    //Make the Pause button toggle between paused and running states. Only works for short (< 0.5s) toggles
    if(rc_get_state()==RUNNING)     rc_set_state(PAUSED);
    else if(rc_get_state()==PAUSED) rc_set_state(RUNNING);
    return;
}

void on_pause_press()
{
/**
* If the user holds the pause button for 0.5 seconds, set state to EXITING which
* triggers the rest of the program to exit cleanly.
**/
    const int samples = 100;                    // check for release 100 times in this period
    const int us_wait = 500000;                 // total time to hold down button: 0.5 seconds
    for(int i=0; i<samples; i++){
        // keep checking to see if the button is still held down
        rc_usleep(us_wait/samples);
        if(rc_button_get_state(RC_BTN_PIN_PAUSE)==RC_BTN_STATE_RELEASED){
            // If the button is released in this time, set state to PAUSED or RUNNING
            return;
        }
    }
    printf("long press detected, shutting down\n");
    rc_set_state(EXITING); // exit program
    return;
}
```