MEC ENG 102B Final Project Report
Little Buddy

Group 15: Luan Chuang, Katherine How,
Kaitlyn Lee, Kyle Miller

December 5, 2023

1 Project Overview

1.1 Project Goal

Our project focuses on general purpose automation. We initially wanted to build a two-legged walking robot that can walk freely. However, we revised this idea to account for time and difficulty limitations, and instead developed a bipedal, mini "acrobot". This acrobot mimics movements similar to that of a singular leg but simplified into a double inverted pendulum actuated at the upper joint. Our team chose this project because we were interested in learning the controls required for under-actuated robotic mechanisms.

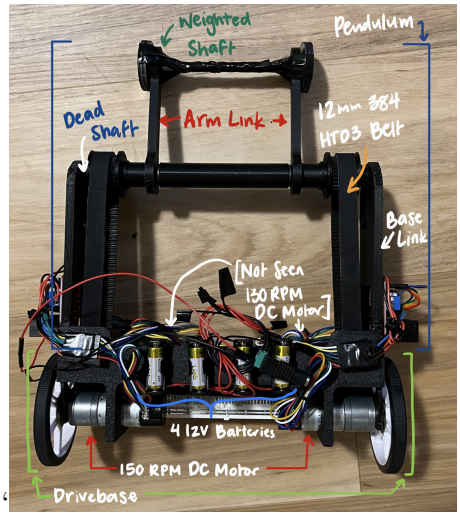
1.2 High Level Strategy

We decided to have a two level linkage driven by a motor and a pulley belt. The bottom wheels would be controlled by two separate motors. The controls are designed so that the robot could automatically adjust its position to account for the top weight. It acts as a double inverted pendulum, with the wheels/body as the first linkage and a pulley-driven arm as the second. We attach an adjustable mass at the top of the second linkage. Our initial desired functionality was for the robot to be able to balance completely on its own, as well as being able to translate without falling over. We intended for the robot to be autonomous and battery powered. Our achieved functionality was that we were able balance the robot and have it be controlled autonomously.

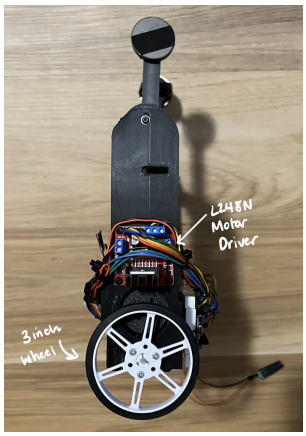
2 Integrated Physical System

More photos of the design, including the CAD model can be found in the Appendix 2.

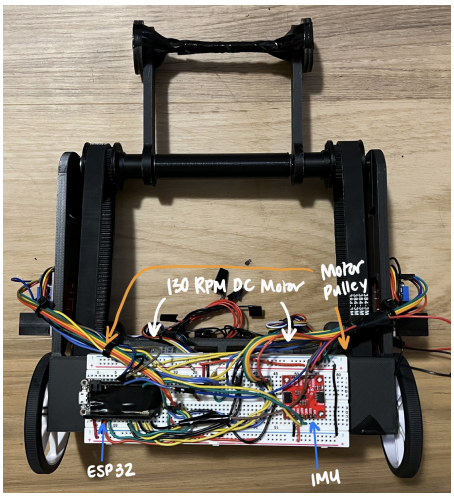
2.1 Physical Model



((a)) Front View



((b)) Side View



((c)) Electronics View

3 Function-Critical Design Decisions

The entire robot can be split into two bodies of a dual pendulum. The base of the robot is the first link, and the pivoting arm of the robot is the second link. This definition was important in our center of mass calibrations to optimize inertia values in the code.

3.1 Linkage 1: Base

Notably, the two wheels driving our entire acrobot were not included in our definition of the first linkage because it was not a pivoting body. Everything else attached to the 3D printed base pivoted with the robot, so was a part of the first linkage. We designed the CAD of the base to be able to support a breadboard for our electronics as well as house 4 motors. Each motor was inserted into the 3D printed base with heatset inserts. The height of the two beams on either side were dependent on the length of

the second link so that there would be enough clearance for a full 360 degree rotation. The lengths of these supports were also limited by off-the-shelf timing belt specifications.

3.1.1 Subsystem: Drive Base

These two wheels are directly driven by a single motor each. The motors are supported by the bushings (which experience a reaction force of 25N) which is within the bushing's stiffness (250N at 120 RPM).

3.1.2 Motor Selection

Since we originally simulated just an acrobat (1 DoF, no wheels), the system could work with any motor torque that surpasses friction (able to add Energy to the system), so to pick our actuators we set time based goals. With a 3.5" Arm Link and a 0.25 kg load, our target is a linkage actuator that can swing the system from inverted to standing in two swings, and a drive actuator that can handle the reaction forces at base speed of 15in/s (using this as a benchmark).

For our linkage actuator calculations, we have simulated the motor torque and velocity needed to needed to swing the linkage from low to high in two swings as shown **here**.

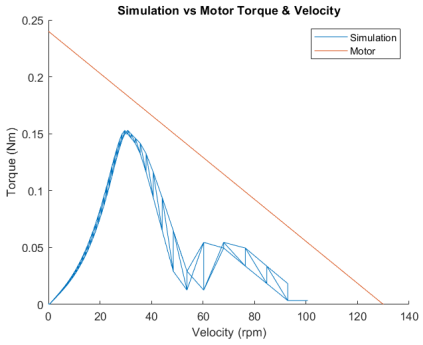
From this simulation data, we found an actuator with a 0.12 Nm stall torque and a 130 rpm peak, which we plan on having two of for a total of 0.24 Nm. This roughly matches the recommended stall torque of 1/0.4 times the max static load we could see.

$$\tau_l = \frac{1}{0.4} mrg = \frac{1}{0.4}(0.25kg)(0.09m)(9.8\frac{m}{s^2}) = 0.27Nm$$

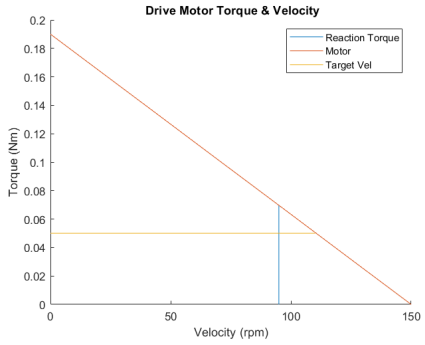
Using our 2D model simulation to estimate the required torques and velocities needed by our linkage motor for our system to reach a stable position, the resulting graph can be see in Fig. 2(a).

The reaction torque needed by the wheel motors can be calculated as a ratio of the wheel and arm radius as follows: $\tau_m = \frac{1.5in}{3.5in}\tau_l = 0.10Nm$

For our drive motor, we have picked a desired velocity of 15 inches per second which corresponds to about 90 RPM. We calculated the conversion like so: $\omega_w = \frac{15in/s}{1.5in}\frac{60s}{1min} = 90RPM$ We see how the desired criteria fit to the selected motor curve in in Fig. 2(b).



((a)) Linkage motor specifications



((b)) Drive Motor Specification

Based on our motor calculations, we needed a motor that captures and exceeds the reaction torque and target velocity. Thus, we chose a motor with a max angular velocity of 150 RPM.

3.2 Linkage 2: Pivot

The upper linkage is actuated by two 130RPM DC gearmotors that drive a 12mm HTD3 timing belt at a 1:1 ratio. The adjustable weights can be slid onto the round standoff at the top of the link and held in place by a flange. The transmission is arranged as follows in Fig. 5(a).

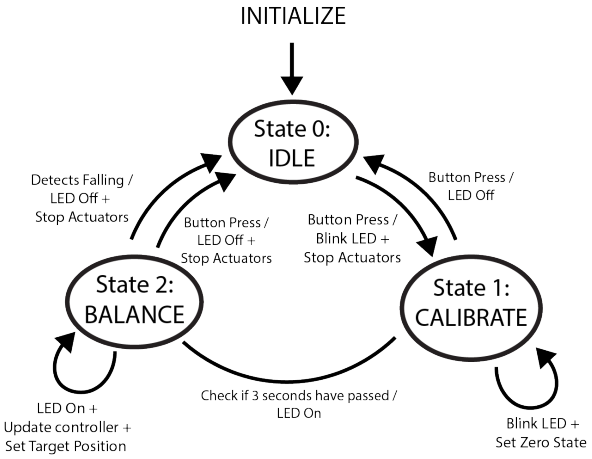
3.2.1 Designing the Transmission

To determine the belt sizes for the transmission, we used a timing belt length calculator to pick our belt. The timing belt calculator takes into account the desired size of and spacing between the pulleys to get an off-the-shelf belt size with enough pretension to avoid slipping. Using the calculator, we adjusted the desired center-to-center distance until we got a belt that we could purchase off the shelf (384 mm HTD3 Belt).

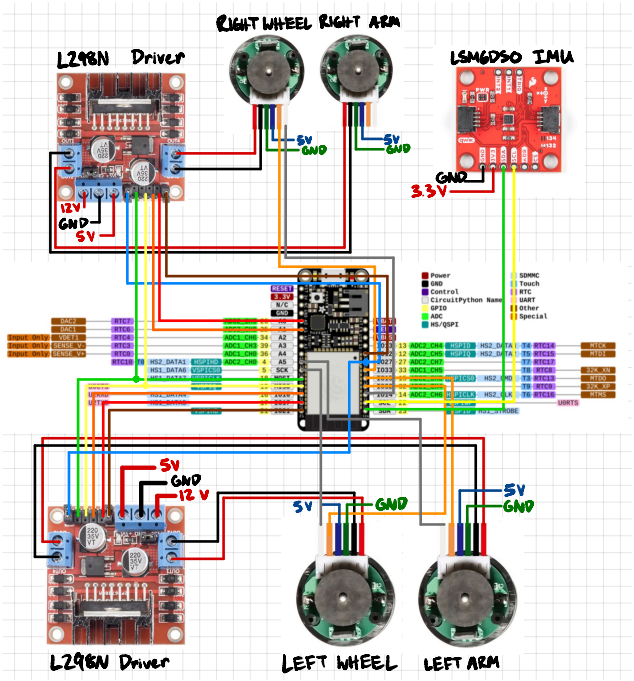
To reduce the friction between the linkages and the dead shaft, we lightly press-fitted oil-embedded bearings onto each of the ends of the arms. Additionally, to decrease friction between the spacers and linkages, we added thrust bearings. To increase the tension of the belts, we added a simple roller tensioner to each belt that mounted directly onto the base, composed of a 1.5" long 8-32" screw, 3D printed roller, and two locknuts to hold the tensioner in place on the base.

4 State Transition Diagram, Circuit Diagram

The state transition diagram is shown in Fig 3(a). The acrobot was wired as follows in Fig 3(b).



((a)) Our State Transition Diagram



((b)) Electrical Schematic

5 Reflections

Our team worked together well to distribute work for the manufacturing, design, control, and electronics of our project, and collaborated throughout to make the acrobot a success. Once our design was done, we were able to quickly assemble the acrobot. Upon reflection, we would have benefited from having a shared calendar for our deadlines. Sometimes, we would miss deadlines due to a lack of communication over our necessary deliverables. Also, it would have been better if we finalized our CAD and began printing our linkages earlier so that we could iterate the base and pulleys more. Better internal communication and a tighter design timeline would've helped us have more time for better controls implementation. We also should have taken advantage of staff knowledge and asked for more help with design choices.

6 Appendices

6.1 Appendix 1: Bill of Materials

Category	Listed Name	Quantity	Vendor	Cost
Mechanical	4mm Rigid Flange Coupling	2	Amazon	$9.99
	12 mm HTD3 Timing Belt (384mm)	2	Amazon	$6.99
	1/4” Shaft Thrust Bearing	4	McMaster	$12.08
	1/4” Shaft Oil-Pressed Bearing	4	McMaster	$2.28
	1/4” Steel Round Shaft	1	McMaster	$1.71
	M3x15 Steel Screws	14	McMaster	$6.40
	M3 Washer	2	McMaster	$2.19
	3in Wheels for 4mm D Shaft	1	Pololu	$9.95
	1/4” Round, Female 8-32 Threaded 4” Standoff	1	McMaster	$7.10
	8-32 x3/8” Steel Screws	2	Had Them	$0
	M3 Heatset Inserts	4	McMaster	$16.81
	Steel Weights	12	Jacobs Scraps	$0
	8-32 x 1.5” Steel Screws	2	Home Depot	$5.62
	8-32 Locknut	4	Home Depot	$0.80
	1/4” Washer	2	McMaster	$9.99
3D Prints	HTD3 Pulley Coupled Link	2	N/A	$0
	Large Spacer	1	N/A	$0
	Side Spacers	2	N/A	$0
	HTD3 Motor Pulley	2	N/A	$0
	Base Structure	1	N/A	$0
	Tensioner Roller	2	N/A	$0
Electrical	ESP32 Huzzah 32	1	Lab Kit	$0
	HiLetGo L298N Motor Driver Module	2	Amazon	$11.49
	Solid Core Wire	3ft	EECS Lab	$0
	LM317 Voltage Regulator	1	Lab Kit	$0
	$0.1\mu F$ Capacitor	1	Lab Kit	$0
	$1\mu F$ Capacitor	1	Lab Kit	$0
	Resistors (Assorted)	1	Lab Kit	$0
	DC Metal Gearmotor 130 RPM w/Encoder	2	Amazon	$14.88
	DC Metal Gearmotor 150 RPM w/Encoder	2	Amazon	$14.88
	12V A23 Battery	4	Amazon	$6.98
	A23 Battery Case	4	Amazon	$6.49
	Adhesive Velocro	Cut up	Amazon	$6.69
	SparkFun LSM6DSO IMU Sensor	1	Lab Kit	$0
	2 Pin Tactile Push Button	1	Lab Kit	$0
	Potentiometer	1	Lab Kit	$0
			Total Cost	$167.68

Table 1: Complete Bill of Materials

To view the BOM with links to products included, view this sheet.

6.2 Appendix 2: CAD Model

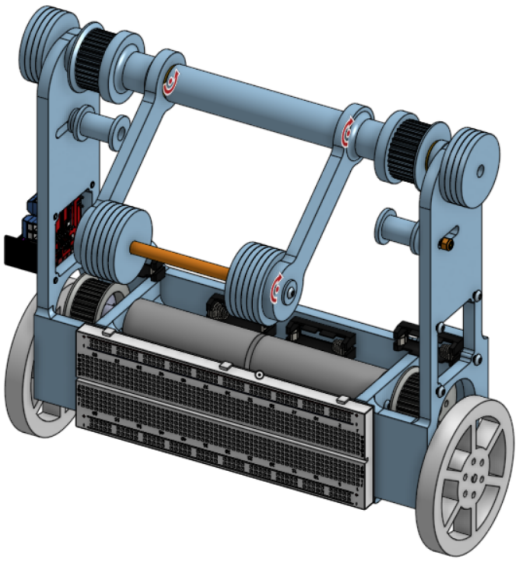
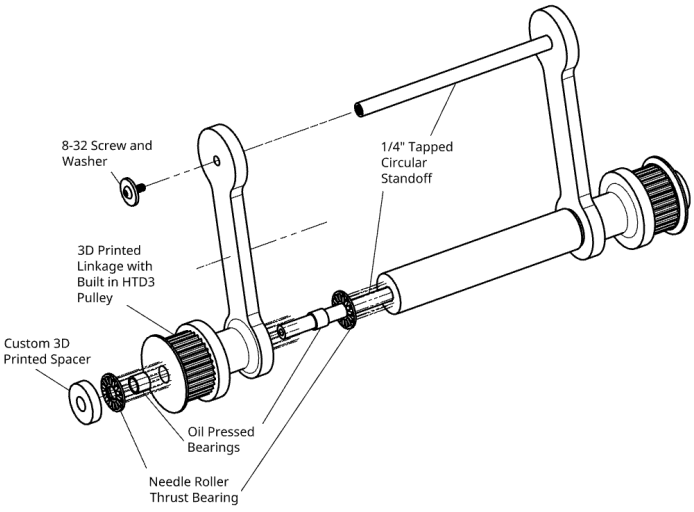
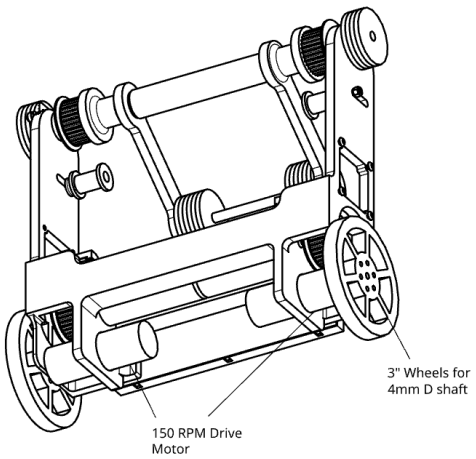


Figure 4: Full CAD Model

Belts aren't depicted in the CAD model.



((a)) Linkage 2 Transmission



((b)) Drivebase

6.3 Appendix 3: Full Code

Listing 1: Main.ino: Our State Machine and Target Control

```
#include <Arduino.h>
#include "Controller.h"

#define BTN 34
#define LED_PIN 13
#define POT 39

// State Machine
enum Case { IDLE, CALIBRATE, BALANCE };
Case c = IDLE;
volatile bool buttonIsPressed = false;
volatile bool buttonT = false;            // check timer interrupt 2
volatile bool ΔT = false;            // check timer interrupt 1 (for 10ms)
volatile bool ΔT2 = false;             // check timer interrupt 4 (for 12ms)
volatile bool calibrateT = false;            // check timer interrupt 3

// Controller
Controller controller;
const float pi = 3.14159;

// Timers
hw_timer_t* timer0 = NULL; // main timer (for balancing) - 10 milliseconds
hw_timer_t* timer1 = NULL; // button timer (in between button presses) - 1 second
hw_timer_t* timer2 = NULL; // calibrate timer (for determining calibration time) - 3 ...
    seconds
hw_timer_t* timer3 = NULL; // balancing timer for delay - 12 milliseconds
portMUX_TYPE timerMux0 = portMUX_INITIALIZER_UNLOCKED;
portMUX_TYPE timerMux1 = portMUX_INITIALIZER_UNLOCKED;
portMUX_TYPE timerMux2 = portMUX_INITIALIZER_UNLOCKED;
portMUX_TYPE timerMux3 = portMUX_INITIALIZER_UNLOCKED;

//Initialization ------------------------------------
void IRAM_ATTR buttonISR() {
  buttonIsPressed = true;
}

void IRAM_ATTR onTime0() {
  portENTER_CRITICAL_ISR(&timerMux0);
  ΔT = true;  // the function to be called when timer interrupt is triggered
  portEXIT_CRITICAL_ISR(&timerMux0);
}

void IRAM_ATTR onTime1() {
  timerStop(timer1);
}

void IRAM_ATTR onTime2() {
  portENTER_CRITICAL_ISR(&timerMux2);
  calibrateT = true;  // the function to be called when timer interrupt is triggered
  portEXIT_CRITICAL_ISR(&timerMux2);
}

void IRAM_ATTR onTime3() {
  timerStop(timer3);
}

void setup() {
```

```
57    Serial.begin(9600);
58    pinMode(BTN, INPUT);
59    pinMode(POT, INPUT);
60    pinMode(LED_PIN, OUTPUT);
61    attachInterrupt(BTN, buttonISR, RISING);
62
63    controller.init();
64
65    // initilize timers
66    timer0 = timerBegin(0, 80, true);              // timer 0, MWDT clock period = 12.5 ...
          ns * TIMGn_Tx_WDT_CLK_PRESCALE -> 12.5 ns * 80 -> 1000 ns = 1 us, countUp
67    timerAttachInterrupt(timer0, &onTime0, true);  // edge (not level) triggered
68    timerAlarmWrite(timer0, 10000, true);        // 10000 * 1 us = 10 ms, autoreload true
69
70    timer1 = timerBegin(1, 80, true);              // timer 1, MWDT clock period = 12.5 ...
          ns * TIMGn_Tx_WDT_CLK_PRESCALE -> 12.5 ns * 80 -> 1000 ns = 1 us, countUp
71    timerAttachInterrupt(timer1, &onTime1, true);  // edge (not level) triggered
72    timerAlarmWrite(timer1, 1000000, true);          // 1000000 * 1 us = 1 s, ...
          autoreload true
73
74    timer2 = timerBegin(2, 80, true);              // timer 2, MWDT clock period = 12.5 ...
          ns * TIMGn_Tx_WDT_CLK_PRESCALE -> 12.5 ns * 80 -> 1000 ns = 1 us, countUp
75    timerAttachInterrupt(timer2, &onTime2, true);  // edge (not level) triggered
76    timerAlarmWrite(timer2, 3000000, true);          // 3000000 * 1 us = 3 s, ...
          autoreload true
77
78    timer3 = timerBegin(3, 80, true);              // timer 2, MWDT clock period = 12.5 ...
          ns * TIMGn_Tx_WDT_CLK_PRESCALE -> 12.5 ns * 80 -> 1000 ns = 1 us, countUp
79    timerAttachInterrupt(timer3, &onTime3, true);  // edge (not level) triggered
80    timerAlarmWrite(timer3, 12000, true);          // 12000 * 1 us = 12 ms, autoreload true
81
82    // at least enable the timer alarms
83    timerAlarmEnable(timer0);  // enable
84    timerAlarmEnable(timer1);  // enable
85    timerAlarmEnable(timer2);  // enable
86    timerAlarmEnable(timer3);  // enable
87    timerStop(timer0);
88    timerStop(timer1);
89    timerStop(timer2);
90    timerStop(timer3);
91  }
92
93  void loop() {
94    controller.imu.update();
95
96    switch (c) {
97      case IDLE:
98        ledOff();
99        if (checkForButtonPress()) { // Check if button has been pressed
100         timerStart(timer2);        // Reset Calibrate timer
101         c = CALIBRATE;             // Change state to calibrate
102       }
103
104       controller.stopActuators();
105       break;
106
107     case CALIBRATE:
108       if (checkForButtonPress()) {  // Check if button has been pressed
109         ledOff();
110         controller.stopActuators();
111         c = IDLE;                   // Change state to idle
```

```
112       }
113       timerStart(timer2);
114       if (calibrateT) {                  // Check if 3 seconds has passed in calibrate ...
              mode (using timer 2)
115         portENTER_CRITICAL(&timerMux2);
116         calibrateT = false;              // reset calibrateT flag to false
117         portEXIT_CRITICAL(&timerMux2);
118
119         timerStop(timer2);
120         ledOn();
121         controller.zeroState();          // Calibrate Sensors
122         c = BALANCE;                     // Change state to balancing
123       }
124
125       flashLight();                      // Blink the LED
126       controller.stopActuators();
127
128       break;
129
130     case BALANCE:
131       ledOn();
132
133       if (checkForButtonPress() || controller.falling()) { // Check if button has ...
              been pressed, or if the bot has tilted too much
134         ledOff();
135         controller.stopActuators();
136         c = IDLE;                                          // Change state to idle ...
              (turn off actuators)
137       }
138       timerStart(timer0);
139       if (!ΔT) {
140         timerStart(timer3);
141         if(timerStarted(timer3)) {} //if this is triggered, don't do anything ...
              (consistent motor pulse, and prevents too small ms for deriving velocity)
142
143         portENTER_CRITICAL(&timerMux3);
144         ΔT2 = false;
145         portENTER_CRITICAL(&timerMux3);
146
147         controller.LW.update();
148         controller.RW.update();
149         controller.A2.update();
150         float potVal = analogRead(POT);
151         float targetPos = ((potVal/4095-0.5) / 5);
152         float target[6] = {targetPos, 0, 0, 0, 0, 0};
153         controller.setTarget(target); // Run actuators to stabilize at position & ...
              velocity targets
154       }
155       timerStop(timer0);
156
157       portENTER_CRITICAL(&timerMux0);
158       ΔT = false;
159       portENTER_CRITICAL(&timerMux0);
160
161       break;
162   }
163 }
164
165 bool checkForButtonPress() {
166   if (timerStarted(timer1)) {
167     buttonIsPressed = false;
```

```
  }
  if (buttonIsPressed) {
    buttonIsPressed = false;
    timerStart(timer1);
    return true;
  }
  return false;
}

void flashLight() {
  if ((millis() - timerReadMilis(timer2)) / 500 % 2) {
    ledOn();
  } else {
    ledOff();
  }
}

void ledOn() {
  digitalWrite(LED_PIN, HIGH);
}

void ledOff() {
  digitalWrite(LED_PIN, LOW);
}
```

Listing 2: Controller.h: Our Control System (Actuator Feedback Loop)

```
#include "Motor.h"
#include "IMU.h"
#include "K.h"

class Controller {
  private:
    // Physical Properties
    const float pi = 3.14159;
    const float r = 0.04;

    // Controls Variables
    const float Δ = 0.1;
    float state[6];
    float P[3] = {0, 0, 0};
    float target[6];
    const float fallingBound = 0.8*pi/2;
    float integralTimer;

  public:
    // Objects
    Motor LW;
    Motor RW;
    Motor A2;
    IMU imu;

    Controller() {}

    void init() {
      LW.init(14, 4, 21, 16, 17, 40);
      RW.init(12, 15, 33, 5, 18, 40);
      A2.init(19, 32, 27, 25, 26, 45);
      imu.init();
    }
```

```
34
35      void zeroState() {
36        LW.setRef();
37        RW.setRef();
38        A2.setRef();
39        imu.setRef();
40        updateState();
41        integralTimer = millis();
42      }
43
44      void stopActuators() {
45        LW.setPWM(0, 0);
46        RW.setPWM(0, 0);
47        A2.setPWM(0, 0);
48      }
49
50      void setTarget(float newTarget[]) {
51        for (int i = 0; i < 6; i++) {
52          target[i] = newTarget[i];
53        }
54        updateState();
55        controlActuators();
56      }
57
58      void updateState() {
59        state[0] = r * (-imu.getPos());// + (LW.getPos() + RW.getPos()) / 2);
60        state[1] = imu.getPos();
61        state[2] = A2.getPos();
62        state[3] = r * (-imu.getVel()); //+ (LW.getVel() + RW.getVel()) / 2);
63        state[4] = imu.getVel();
64        state[5] = //A2.getVel();
65
66        // State Transformation
67        state[1] = H[1][1] * state[1] + H[1][2] * state[2];
68        state[4] = H[4][4] * state[4] + H[4][5] * state[5];
69      }
70
71      void controlActuators() {
72        float TW = 0;
73        float TA2 = 0;
74        for (int i = 0; i < 6; i++) {
75          TW += K[0][i] * (target[i] - state[i]);
76          TA2 += K[1][i] * (target[i] - state[i]);
77        }
78        LW.setPWM(TW, 0.15);
79        RW.setPWM(TW, 0.15);
80        A2.setPWM(TA2, 0.4);
81      }
82
83      bool falling() {
84        return (abs(imu.getPos()) > fallingBound);
85      }
86  };
```

Listing 3: Motor.h: Our Motor and Integrated Encoder Class

```
1  #include <ESP32Encoder.h>
2
3  class Motor {
4    private:
```

```
    int ENC_A, ENC_B, PWM, IN_1, IN_2, ticks;
    float tickTimer, tickRate, radToTick;

    ESP32Encoder encoder;

  public:
    Motor() {}

    void init(int ENC_A, int ENC_B, int PWM, int IN_1, int IN_2, int reduction) {
      ESP32Encoder::useInternalWeakPullResistors = UP;

      this->PWM = PWM;
      this->IN_1 = IN_1;
      this->IN_2 = IN_2;
      pinMode(PWM, OUTPUT);
      pinMode(IN_1, OUTPUT);
      pinMode(IN_2, OUTPUT);
      digitalWrite(IN_1, LOW);
      digitalWrite(IN_2, LOW);

      encoder.attachFullQuad(ENC_A, ENC_B);
      encoder.setCount(0);

      radToTick = (2 * 3.1415 * reduction);

      tickTimer = millis();
    }

    void update() {
      tickRate = (encoder.getCount() - ticks) / (millis() - tickTimer) * 1000;
      ticks = encoder.getCount();
      tickTimer = millis();
    }

    void setPWM(float voltage, float frictionVoltage) {
      int pwm = constrain(int(255 * voltage), -255, 255);
      int frictionpwm = constrain(int(255 * (abs(voltage) + abs(frictionVoltage))), ...
          0, 255);
      analogWrite(PWM, abs(frictionpwm));
      if (abs(pwm) < 1) {
        digitalWrite(IN_1, LOW);
        digitalWrite(IN_2, LOW);
      } else if (pwm > 0) {
        digitalWrite(IN_1, HIGH);
        digitalWrite(IN_2, LOW);
      } else {
        digitalWrite(IN_1, LOW);
        digitalWrite(IN_2, HIGH);
      }
    }

    float getPos() {
      return encoder.getCount() / radToTick;
    }

    float getVel() {
      return tickRate / radToTick;
    }

    void setRef() {
      encoder.clearCount();
```

```
65        ticks = 0;
66      }
67  };
```

Listing 4: IMU.h: Our Wrapper and Position Integrator for the LSM6DSO

```
1   #include "SparkFunLSM6DSO.h"
2   #include "Wire.h"
3   #include "Kalman.h"
4
5   class IMU {
6     private:
7       float accY, accZ;
8       float gyroX, gyroXangle;
9       float angRef;
10      float timer;
11
12      Kalman kalman;
13
14    public:
15      LSM6DSO myIMU;
16
17      IMU() {}
18
19      void init() {
20        Wire.begin(23, 22);
21        delay(10);
22        myIMU.begin();
23        myIMU.initialize(BASIC_SETTINGS);
24      }
25
26      void update() {
27        accY = myIMU.readFloatAccelY();
28        accZ = myIMU.readFloatAccelZ();
29        gyroX = myIMU.readFloatGyroX() / 131;
30
31        float dt = (millis() - timer) / 1000;
32        timer = millis();
33
34        double roll  = atan2(accY, accZ) * RAD_TO_DEG;
35
36        if ((roll < -90 && gyroXangle > 90) || (roll > 90 && gyroXangle < -90)) {
37          kalman.setAngle(roll);
38          gyroXangle = roll;
39        } else
40          gyroXangle = kalman.getAngle(roll, gyroX, dt);
41      }
42
43      float getPos() {
44        return (gyroXangle - angRef) * DEG_TO_RAD;
45      }
46
47      float getVel() {
48        return gyroX * DEG_TO_RAD;
49      }
50
51      void setRef() {
52        accY = myIMU.readFloatAccelY();
53        accZ = myIMU.readFloatAccelZ();
54
```

```
55        double roll = atan2(accY, accZ) * RAD_TO_DEG;
56        kalman.setAngle(roll);
57
58        gyroXangle = roll;
59        angRef = gyroXangle;
60
61        timer = millis();
62      }
63  };
```

Listing 5: K.h: Our Gains and State Transformation Computed via MATALB LQR Cost Functions

```
1  const float K[2][6] = {{-6.179, 30.4414, -0.64335, -5.2686, 11.493, -1.2754}, ...
       {-0.95826, 3.9051, 1.26747, -0.62286, 2.4848, -0.26032}};
2  const float H[6][6] = {{1, 0, 0, 0, 0, 0}, {0, 0.46639, 0.018222, 0, 0, 0}, {0, 0, 1, ...
       0, 0, 0}, {0, 0, 0, 1, 0, 0}, {0, 0, 0, 0, 0.18413, 0.15157}, {0, 0, 0, 0, 0, 1}};
```

6.4 Appendix 4A: 2D Model

6.4.1 Variables & Definitions

Parameter	Value	Units	Description
L_1		m	Length of link 1
L_2		m	Length of link 2
θ_1		rad	Angle of link 1 from vertical
θ_2		rad	Angle of link 2 from link 1
θ_w		rad	Angle of wheel from link 1
p_0			Base of link 1 & mounting point of wheel
m_w		kg	Mass of the wheel
m_1		kg	Mass of link 1
m_2		kg	Mass of link 2
F_x		N	Translational force provided by the wheel
r	0.04	m	Radius of wheel
τ_2		Nm	Motor torque on link 2
D_0^w			Center of mass of the wheel
D_0^1			Center of mass of link 1
D_0^2			Center of mass of link 2
x		m	X position of wheel from initialization
L_{c1}		m	Length to center of mass of link 1
L_{c2}		m	Length to center of mass of link 2
$\dot{D}_0^w$			Velocity of the center of mass of the wheel
$\dot{D}_0^1$			Velocity of the center of mass of link 1
$\dot{D}_0^2$			Velocity of the center of mass of link 2
J_{vi}			Linear velocity Jacobian of coordinate frame
ω_i			Angular velocity of relevant coordinate frame
J_{wi}			Angular velocity Jacobian of relevant joint
J_{vw}			Linear velocity Jacobian of wheel coordinate frame
J_{v1}			Linear velocity Jacobian of link 1 coordinate frame
J_{v2}			Linear velocity Jacobian of link 2 coordinate frame
J_{ww}			Angular velocity Jacobian of wheel coordinate frame
J_{w1}			Angular velocity Jacobian of link 1 coordinate frame

J_{w2}			Angular velocity Jacobian of link 2 coordinate frame
Q			Lagrangian State vector (x position of wheel, angle of link 1, angle of link 2)
M			Inertial matrix
V			Coriolis & centrifugal force vector
G			Gravitational force vector
T			External force vector
P			Potential energy

Table 2: System parameters in order of appearance

6.5 Appendix 4B: 2D Model

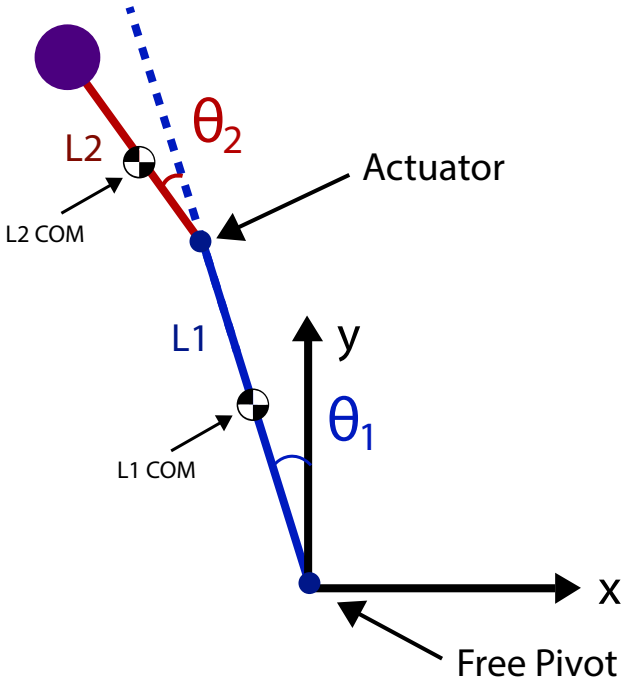


Figure 6: Kinematic model of our base system with relevant system parameters

6.5.1 Kinematics and Jacobians

Definitions:

$$c_1 = \cos\theta_1,\ c_{12} = \cos(\theta_1 + \theta_2),\ x = r(\theta_w - \theta_1),\ F_x = \frac{\tau_w}{r}$$

CoM Positions:

$$D_0^w = \begin{bmatrix} x \\ 0 \end{bmatrix},\ D_0^1 = \begin{bmatrix} x - L_{c1}s_1 \\ L_{c1}c_1 \end{bmatrix},\ D_0^2 = \begin{bmatrix} x - L_1 s_1 + L_{c2}s_{12} \\ L_1 c_1 - L_{c2}c_{12} \end{bmatrix}$$

CoM Velocities:

$$\dot{D_0^w} = \begin{bmatrix} \dot{x} \\ 0 \end{bmatrix},\ \dot{D_0^1} = \begin{bmatrix} \dot{x} - L_{c1}\dot{\theta_1}c_1 \\ -L_{c1}\dot{\theta_1}s_1 \end{bmatrix},\ \dot{D_0^2} = \begin{bmatrix} \dot{x} - L_1\dot{\theta_1}c_1 + L_{c2}(\dot{\theta_1} + \dot{\theta_2})c_{12} \\ -L_1\dot{\theta_1}s_1 + L_{c2}(\dot{\theta_1} + \dot{\theta_2})s_{12} \end{bmatrix}$$

Jacobians:

$$\dot{D_0^i} = J_{vi}\begin{bmatrix} \dot{\theta_w} \\ \dot{\theta_1} \\ \dot{\theta_2} \end{bmatrix},\ \omega_i = J_{\omega i}\begin{bmatrix} \dot{\theta_w} \\ \dot{\theta_1} \\ \dot{\theta_2} \end{bmatrix}$$

$$J_{vw} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix},\ J_{\omega w} = 0$$

$$J_{v1} = \begin{bmatrix} 1 & -L_{c1}c_1 & 0 \\ 0 & -L_{c1}s_1 & 0 \end{bmatrix},\ J_{\omega 1} = \begin{bmatrix} 0 & 1 & 0 \end{bmatrix}$$

$$J_{v2} = \begin{bmatrix} 1 & -L_1c_1 + L_{c2}c_{12} & L_{c2}c_{12} \\ 0 & -L_1c_1 + L_{c2}c_{12} & L_{c2}c_{12} \end{bmatrix},\ J_{\omega 2} = \begin{bmatrix} 0 & 1 & 1 \end{bmatrix}$$

6.5.2 Lagrangian Dynamics

State Matrix:

$$Q = \begin{bmatrix} x \\ \theta_1 \\ \theta_2 \end{bmatrix}$$

Base Equation:

$$M\ddot{Q} + V + G = T$$

Inertial Matrix:

$$M = \Sigma(m_i J_{vi}^T J_{vi} + J_{\omega i}^T I_{ci} J_{\omega i})$$

$$M = \begin{bmatrix} m_w + m_1 + m_2 & -m_1 L_{c1}c_1 + m_2(-L_1c_1 + L_{c2}c_{12}) & m_2 L_{c2}c_{12} \\ -m_1 L_{c1}c_1 + m_2(-L_1c_1 + L_{c2}c_{12}) & I_{c1} + I_{c2} + m_1 L_{c1}^2 + m_2(L_1^2 + L_{c2}^2 - 2L_1 L_{c2}c_2) & I_{c2} + m_2 L_{c2}(L_{c2} - L_1c_2) \\ m_2 L_{c2}c_{12} & I_{c2} + m_2 L_{c2}(L_{c2} - L_1c_2) & I_{c2} + m_2 L_{c2}^2 \end{bmatrix}$$

Coriolis and Centrifugal Vector:

$$V = \dot{M}\dot{Q} - \frac{1}{2}\begin{bmatrix} \dot{Q}^T \frac{\partial M}{\partial x}\dot{Q} \\ \dot{Q}^T \frac{\partial M}{\partial \theta_1}\dot{Q} \\ \dot{Q}^T \frac{\partial M}{\partial \theta_2}\dot{Q} \end{bmatrix}$$

$$\dot{M} = \begin{bmatrix} 0 & \sigma & -m_2 L_{c2}(\dot{\theta_1} + \dot{\theta_2})s_{12} \\ \sigma & 2m_2 L_1 L_{c2}\dot{\theta_2}s_2 & m_2 L_{c2}L_1\dot{\theta_2}s_2 \\ -m_2 L_{c2}(\dot{\theta_1} + \dot{\theta_2})s_{12} & m_2 L_{c2}L_1\dot{\theta_2}s_2 & 0 \end{bmatrix}$$

$$\sigma = L_{c1}\dot{\theta_1}s_1 + m_2(L_1\dot{\theta_1}s_1 - L_{c2}(\dot{\theta_1} + \dot{\theta_2})s_{12})$$

$$V = \begin{bmatrix} m_1 L_{c1}\dot{\theta_1}^2 s_1 + m_2(L_1\dot{\theta_1}^2 s_1 - L_{c2}(\dot{\theta_1} + \dot{\theta_2})^2 s_{12}) \\ m_2 L_1 L_{c2}\dot{\theta_2}(2\dot{\theta_1} + \dot{\theta_2})s_2 \\ -m_2 L_1 L_{c2}\dot{\theta_1}^2 s_2 \end{bmatrix}$$

Gravity Vector:

$$P = m_1 g L_{c1}c_1 + m_2 g(L_1c_1 - L_{c_2}c_{12})$$

$$G = \frac{\partial}{\partial Q}P = \begin{bmatrix} 0 \\ -m_1 g L_{c1}s_1 + m_2 g(-L_1s_1 + L_{c2}s_{12}) \\ m_2 g L_{c2}s_{12} \end{bmatrix}$$

External Force Vector:

$$T = \begin{bmatrix} F_x \\ 0 \\ \tau_2 \end{bmatrix}$$

Motor Torque Model:

$$\tau = \frac{k}{R}(V - k\omega)$$

Max Torque ($\omega = 0$):

$$\tau_m = \frac{k}{R}V_m \rightarrow \frac{k}{R} = \frac{\tau_m}{V_m}$$

Max Velocity ($\tau = 0$):

$$0 = (V_m - k\omega_m) \rightarrow k = \frac{V_m}{\omega_m}$$

Motor Torque Model:

$$\tau = \frac{\tau_m}{V_m}(V - \frac{V_m}{\omega_m}\omega) = \tau_m(\frac{V}{V_m} - \frac{\omega}{\omega_m})$$

With Gear Ratio:

$$\tau = G\tau_m(\frac{V}{V_m} - G\frac{\omega}{\omega_m})$$

$$\omega_w = \frac{\dot{x}}{r} + \dot{\theta}_2$$

External Force Vector:

$$T = \begin{bmatrix} F_x \\ 0 \\ \tau_2 \end{bmatrix} = \begin{bmatrix} \frac{G_w\tau_m}{r}(V_w - G_w\frac{\omega_w}{\omega_m}) \\ 0 \\ G\tau_m(V_2 - G_2\frac{\omega_2}{\omega_m}) \end{bmatrix} = \begin{bmatrix} \frac{G_w\tau_m}{r}(V_w - G_w\frac{\frac{\dot{x}}{r}+\dot{\theta}_1}{\omega_m}) \\ 0 \\ G_2\tau_m(V_2 - G_2\frac{\dot{\theta}_2}{\omega_m}) \end{bmatrix}$$

Base Equation Expanded:

$$M\begin{bmatrix} \ddot{x} \\ \ddot{\theta}_1 \\ \ddot{\theta}_2 \end{bmatrix} + \begin{bmatrix} m_1L_{c1}\dot{\theta_1}^2s_1 + m_2(L_1\dot{\theta_1}^2s_1 - L_{c2}(\dot{\theta}_1 + \dot{\theta}_2)^2s_{12}) \\ m_2L_1L_{c2}\dot{\theta}_2(2\dot{\theta}_1 + \dot{\theta}_2)s_2 \\ -m_2L_1L_{c2}\dot{\theta_1}^2s_2 \end{bmatrix} + \begin{bmatrix} 0 \\ -m_1gL_{c1}s_1 + m_2g(-L_1s_1 + L_{c2}s_{12}) \\ m_2gL_{c2}s_{12} \end{bmatrix} = \begin{bmatrix} F_x \\ 0 \\ \tau_2 \end{bmatrix}$$

6.6 Appendix 4C: Control System

6.6.1 State Space Background

The method of control system is State Space, where q is the state vector, U is the input vector, A is the state matrix, and B is the input to state matrix. Through this linear system, we can compute an array of gains K, that determine our desired inputs given a current state q and state q_t:

$$\dot{q} = Aq + Bu$$

$$u = K(q_t - q)$$

In our case, the state space state vector is $q = \begin{bmatrix} Q \\ \dot{Q} \end{bmatrix}$, where Q is the Lagrangian state matrix. Our input vector is the voltages of our motors

$$u = \begin{bmatrix} V_w \\ V_2 \end{bmatrix}$$

. The target state vector q_t can be fixed (such as balancing the cart at an equilibrium point) or dynamic (such as following the trajectory of a pendulum swinging upwards).

6.6.2 LQR Background

https://www.mathworks.com/help/control/ref/lti.lqr.html

Our method of computing K is a Linear-Quadratic Regulator (LQR). We use MATLAB to compute this, it takes in an A, B, Q, and R matrices and computes K via the algebriac Riccati equation. Q is the State-cost weighted matrix. R is the Input-cost weighted matrix. Q and R are diagonal matrices, with squared cost values corresponding to state error and input effort respectively.

These values are selected by Bryson's rule, where the value should be $\frac{1}{\text{max error}^2}$. For example, our we've set 1 as our maximum voltage value, so our R matrix looks like $R = \begin{bmatrix} \frac{1}{1^2} & 0 \\ 0 & \frac{1}{1^2} \end{bmatrix}$. An exampled of a Q value could be the max error for the first link state at $\frac{\pi}{6}$. The Q value would then be $\frac{6}{\pi}^2$. The Q values provide the most opportunity for tuning a system, and offer infinite solutions for a system. Too large of Q values will be saturated by actuator capabilities, and too small of Q values will not cause a quick enough response (example, the cart tipping instead of recovering from a disturbance).

6.7 Appendix 4D: Jacobian Linearization

6.7.1 Overview

Our system is nonlinear, and we must linearize it to put into state space form. To do so, we must pick an equilibrium point q_0 and u_0 and can assume the system will act linear for small distances around the equilibrium points. To solve this problem, We instead consider the state space matrix of the change in state:

$$\dot{\tilde{q}} = A\tilde{q} + B\tilde{u}$$

$$\tilde{q} = q - q_0$$

$$\tilde{u} = u - u_0$$

We can calculate the A and B matrix via a Jacobian. As we can see, A and B are constant at a given equilibrium point. If we can keep the equilibrium point constant (such as balancing vertically), this can be very useful as modeling tools for LTI systems are well developed. For systems such as the acrobat swing up, the state changes very significantly and the equilibrium point must change, and so current state is a factor in the K gain matrix:

$$\dot{q} = f(q, u)$$

$$A = \frac{\partial f(q,u)}{\partial q}|_{q_0,u_0},\ B = \frac{\partial f(q,u)}{\partial u}|_{q_0,u_0}$$

For the given K value determined by using LQR on the linearized state space, we note that error is the same using an equilibrium point, and therefore we can use this K with just our state and target state.

$$\tilde{u} = K(\tilde{q_t} - \tilde{q}) = K((q_t - q_0) - (q - q_0)) = K(q - q_0)$$

6.7.2 Our System

For our Lagriangian system, we can determine our Jacobian and state space matrix as follows. Note Inertia matrix M well known to always be invertible.

$$\begin{bmatrix} \dot{Q} \\ \ddot{Q} \end{bmatrix} = f(Q, \dot{Q}, u) = \begin{bmatrix} \dot{Q} \\ M^{-1}(T - V - G) \end{bmatrix}$$

$$q = \begin{bmatrix} \dot{Q} \\ \ddot{Q} \end{bmatrix},\ \dot{\tilde{q}} = \frac{\partial f(q,u)}{\partial q}|_{q_0,u_0}\tilde{q} + \frac{\partial f(q,u)}{\partial u}|_{q_0,u_0}\tilde{u}$$

Since $\dot{Q}$ is equal to only itself, we know it's Jacobian relative to Q will be 0 and it's Jacobian relative to itself will be the identity matrix. Though not needed for computation, we can view this reduction as:

$$f_1(Q, \dot{Q}, u) = M^{-1}(T - V - G)$$

$$\begin{bmatrix} \dot{\tilde{Q}} \\ \ddot{\tilde{Q}} \end{bmatrix} = \begin{bmatrix} 0 & I \\ \frac{\partial f_1}{\partial Q}|_{Q_0,\dot{Q}_0,u_0} & \frac{\partial f_1}{\partial \dot{Q}}|_{Q_0,\dot{Q}_0,u_0} \end{bmatrix} \begin{bmatrix} \tilde{Q} \\ \dot{\tilde{Q}} \end{bmatrix} + \begin{bmatrix} 0 \\ \frac{\partial f_1}{\partial u}|_{Q_0,\dot{Q}_0,u_0} \end{bmatrix} \tilde{u}$$

Note: the Jacobian for the x position state is 0, so our system does not behave differently after translating. However, it does change with x velocity, as the wheel actuator requires more voltage to produce the same torque at a larger velocity.

6.7.3 Discretization

Since our simulation and real world processor operate discretely with a time step Δ, we must use the discrete form of the state space matrix and MATLAB's discrete LQR calculator. For a continuous system:

$$\dot{q}(t) = A_c q(t) + B_c u(t)$$

This equivalent discrete system is (note q on left instead of $\dot{q}$):

$$q[k+1] = A_d q[k] + B_d u[k]$$

$$A_d = I + \Delta A_c,\ B_d = \Delta B_c$$

For our Lagrangian system, this discrete model will be:

$$\begin{bmatrix} \tilde{Q} \\ \dot{\tilde{Q}} \end{bmatrix} = \left(I + \Delta \begin{bmatrix} 0 & I \\ \frac{\partial f_1}{\partial Q}|_{Q_0,\dot{Q}_0,u_0} & \frac{\partial f_1}{\partial \dot{Q}}|_{Q_0,\dot{Q}_0,u_0} \end{bmatrix} \begin{bmatrix} \tilde{Q} \\ \dot{\tilde{Q}} \end{bmatrix} \right) + \Delta \begin{bmatrix} 0 \\ \frac{\partial f_1}{\partial u}|_{Q_0,\dot{Q}_0,u_0} \end{bmatrix} \tilde{u}$$

The discretization is very useful if the model's linearization point must change. In this case, you can take the linearization point to be the previous time step when computing the next change in input $\tilde{u}[k+1]$:

$$\tilde{q}[k] = q[k] - q_0 = q[k] - q[k-1]$$

$$\tilde{u}[k] = u[k] - u_0 = u[k] - u[k-1]$$

6.8 Appendix 4E: 2D MATLAB Simulation

6.8.1 2D Simulation Control System Derivation

To help validate our project goal and learn more about the double inverted pendulum as a controls problem, we wanted to simulate a controllable double inverted pendulum to gain insight on the physics behind the system and the different approaches to controlling an underactuated system.

To develop this test simulation, we looked at various approaches used to solve this problem online. A notable exampled was this course from MIT on underactuated robots that explains in depth the dynamics behind the double inverted pendulum. Using these dynamic equations, we were able to successfully simulate the acrobot as shown in the video below. The MATLAB script and equations used to generate the video can be found in Appendix II - LQR Swing.

6.8.2 2D Simulation Demo Video

A screen recording of our 2D double inverted pendulum can be found linked here.

6.8.3 Lagrangian with Link Point Mass Simplification

$$I_1 = I_2 = m_w = x = \dot{x} = \ddot{x} = 0$$

$$M \begin{bmatrix} \ddot{\theta}_1 \\ \ddot{\theta}_2 \end{bmatrix} + \begin{bmatrix} m_2 L_1 L_{c2} \dot{\theta_2} (2\dot{\theta_1} + \dot{\theta_2}) s_2 \\ -m_2 L_1 L_{c2} \dot{\theta_1}^2 s_2 \end{bmatrix} + \begin{bmatrix} -m_1 g L_{c1} s_1 + m_2 g(-L_1 s_1 + L_{c2} s_{12}) \\ m_2 g L_{c2} s_{12} \end{bmatrix} = \begin{bmatrix} 0 \\ \tau_2 \end{bmatrix}$$

$$M = \begin{bmatrix} m_1 L_{c1}^2 + m_2(L_1^2 + L_{c2}^2 - 2L_1 L_{c2} c_2) & m_2 L_{c2}(L_{c2} - L_1 c_2) \\ m_2 L_{c2}(L_{c2} - L_1 c_2) & m_2 L_{c2}^2 \end{bmatrix}$$

Matches the professor Kazerooni's Lagrangian example, except for sign changes.

6.8.4 3D Simulation Control System Derivation

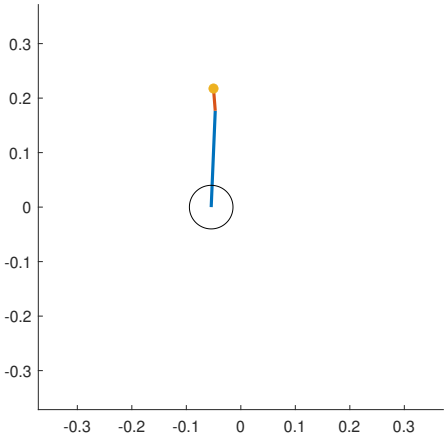


Figure 7: 3DoF Matlab Simulation

Using the derived state space equations and the 2D simulation shown above as a baseline, we were able to generate a MATLAB script to simulte the 3DoF version of this system that matches the hardware we built.

The system controller for the 3DoF simulation uses the same approach as the 2DoF one. A new Jacobian is calculated at each timestep & used to determine optimized controller gains using the LQR method.

The full code & equations used in this simulation can be found in Appendix IV - Matlab Simulation Main Scripts.

6.8.5 3D Simulation Demo Video

A screen recording of the 3DoF simulation can be found linked here.

6.9 Appendix 4F: Implementation

6.9.1 One Moving Link Lagrangian Simplification

$$I_1 = I_2 = m_2 = L_{c2} = \theta_2 = \dot{\theta}_2 = \ddot{\theta}_2 = 0,\ L_{c1} = L_1$$

$$\begin{bmatrix} m_w + m_1 & -m_1 L_1 c_1 \\ -m_1 L_1 c_1 & m_1 L_1^2 \end{bmatrix} \begin{bmatrix} \ddot{x} \\ \ddot{\theta}_1 \end{bmatrix} + \begin{bmatrix} m_1 L_1 \dot{\theta_1}^2 s_1 \\ 0 \end{bmatrix} + \begin{bmatrix} 0 \\ -m_1 g L_1 s_1 \end{bmatrix} = \begin{bmatrix} F_x \\ 0 \end{bmatrix}$$

Matches Slide 15 of UMass class pole-cart (which used the same coordinate frame as we did) - so hopefully some validation that our equations are right!