Jumping Robot Control System

Kyle Blanset, Nathaniel Seymour

Opportunity:

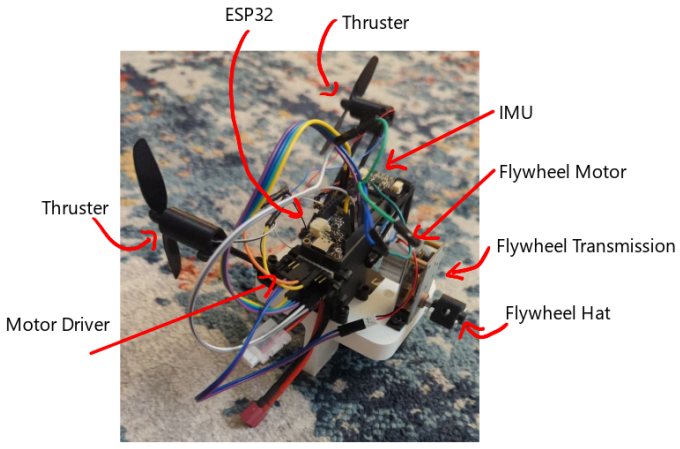
The objective of this design is to engineer a balancing system that can restore the position of an object in flight. The application for our balancing system was to balance an unstable monolegged hopper.

High Level Strategy:

The strategy taken to accomplish the system's task was to implement a flywheel and pair of thrusters to generate a righting torque. Angular position and velocity were measured with a 9 Degree-Of-Freedom sensor. These measurements were used to calculate the error from the desired position for motor movement.

The designed system is changed from the initial proposal in two ways. Firstly, the thrusters replaced one of the flywheels for positional control. This change was made because it is not limited by the top speed like a flywheel may be. Weight being a constant challenge with this design, a set of thrusters is more weight efficient as well. The set of thrusters also alleviated space constraints from having two large flywheels. The second large change was the implementation of a more sophisticated IMU. The original plan for measuring the absolute angle was to estimate position using a real time operating system and the angular velocity that was output from an IMU. This was discovered to introduce too much built up error over time, and a new IMU that was capable of measuring absolute angle was used.

Integrated Device:



Function Critical Decisions:

Mass Moment of Inertia:

$$I_{robot} = I_{body} + I_{thrusters} + I_{leg}$$

$I_{body} = \frac{1}{12} m_{body}(h^2 + w^2) = 2.825 * 10^{-4} kg * m^2$ approximated as a rectangular prism
Where $m_{body} = 0.3kg,\ h = 0.08m,\ and\ w = 0.07m$

$I_{thrusters} = 2 * (I_{rods} + I_{thrustermotor})$
$I_{rods} = 2 * \frac{1}{3} m_{rod} L_{rod}^2 = 3.2 * 10^{-5} kg * m^2$ approximated as rods about their ends
Where $m_{rod} = 0.015kg,\ L_{rod} = 0.08m$
$I_{thrustermotor} = m_{motor} L^2$
Where $m_{motor} = 0.008kg\ and\ L = L_{rod}$

$I_{leg} = \frac{1}{3} m_{leg} L_{leg}^2 = 8.64 * 10^{-4} kg * m^2$ approximated as a rod about its end
Where $m_{leg} = 0.1kg\ and\ L_{leg} = 0.2m$

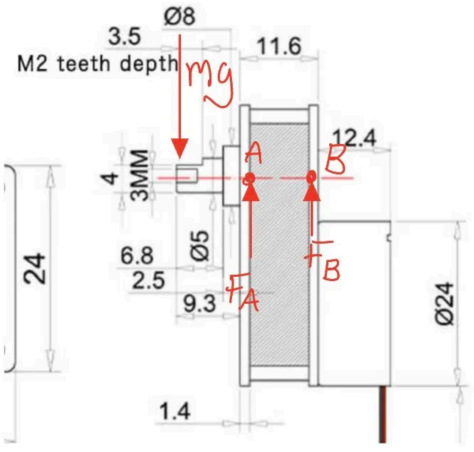
For $I_{flywheel} \geq I_{robot}$
$I_{flywheel} = 2 * (\frac{1}{3} * m_{rod} * L_{rod}^2) + m_{weights} L_{rod}^2$
Where $L_{rod} = 0.01m\ and\ m_{rod} = 0.015kg$

Solving for $m_{weights}$, m = 25g on each end of the flywheel rod.

We want to have a high rotational inertia on the flywheel and a high torque output motor because we wanted to apply as large of a restoring torque as possible on our robot when it is in the air while avoiding spinning up the flywheel. The motor we chose has a maximum torque of 4.5kg.cm with a stall torque of 16kg.cm and gear ratio of 512:1. These specifications are good considering we need to keep the mass of the robot as small as possible, and the motor has good speed and torque for its size. In theme with selecting light weight strong components, the thruster motors were sufficiently strong to provide a restoring force, when operated at their proper voltage and not on an underpowered battery.

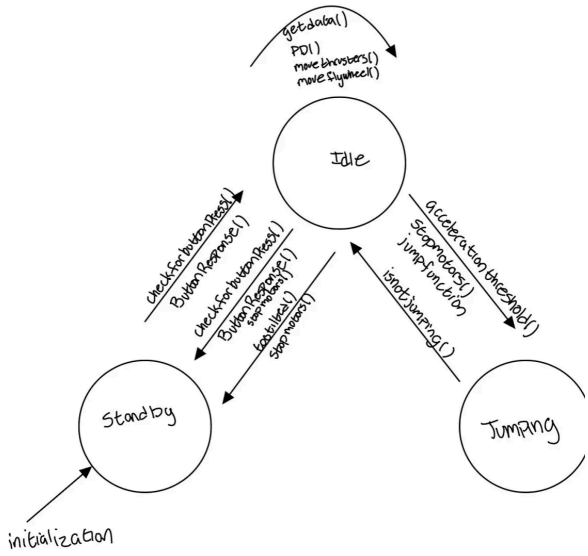
Flywheel Gearbox Forces:

$\Sigma F_y = -m_{flywheel} g + F_A + F_B = 0$
$\Sigma M_A = m_{flywheel} g * R_1 - F_B * R_2$
Where
$m_{flywheel} g = 0.638N,\ R_1 = 0.0044m,\ and\ R_2 = 0.0102m$
$F_A = 0.363N\ and\ F_B = 0.275N$

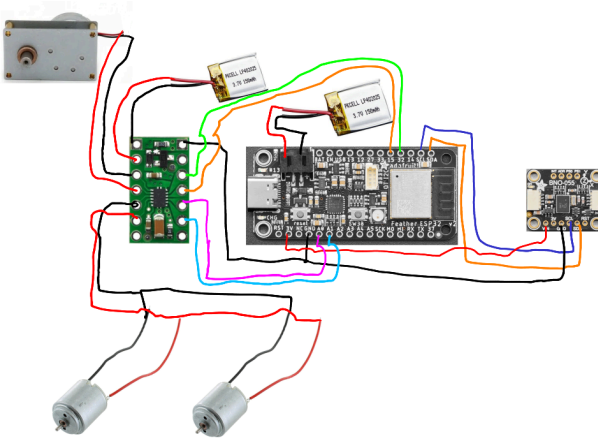


These forces are small enough that the built-in bushing on the motor gearbox won't have problems with this load.

State Transition Diagram:



Circuit Diagram:



Reflection:

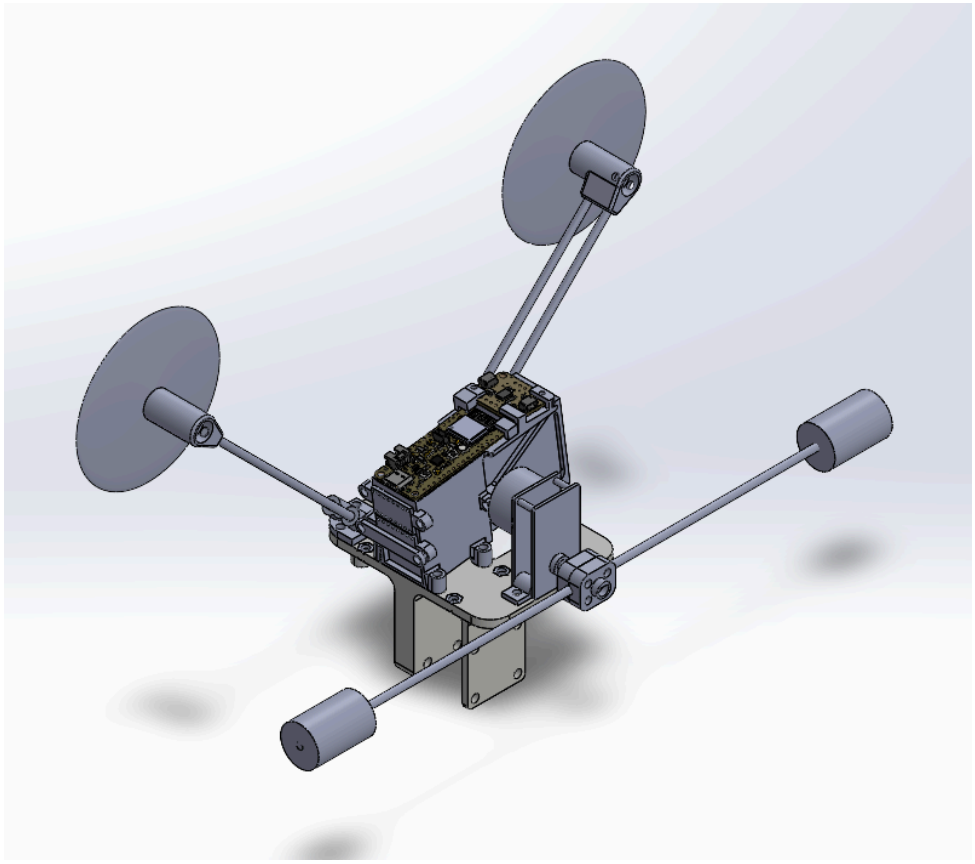
We learned the power of iteration in this project. We spent a good amount of time trying to model and calculate what these specifications should be, but it wasn't until we built the subsystems could we actually see if what we had conceptualized was adequate or not. We think that there is an importance for simple models to get relative orders of magnitude correct for certain function relevant systems, however, when the systems are complex and difficult to model it is really important to get to testing as soon as possible because a more complex model will take a lot more time and is still an approximation. The real world tests are invaluable to improving the design for the desired behavioral characteristics. We wish we had purchased our motors sooner and prototyped sooner.

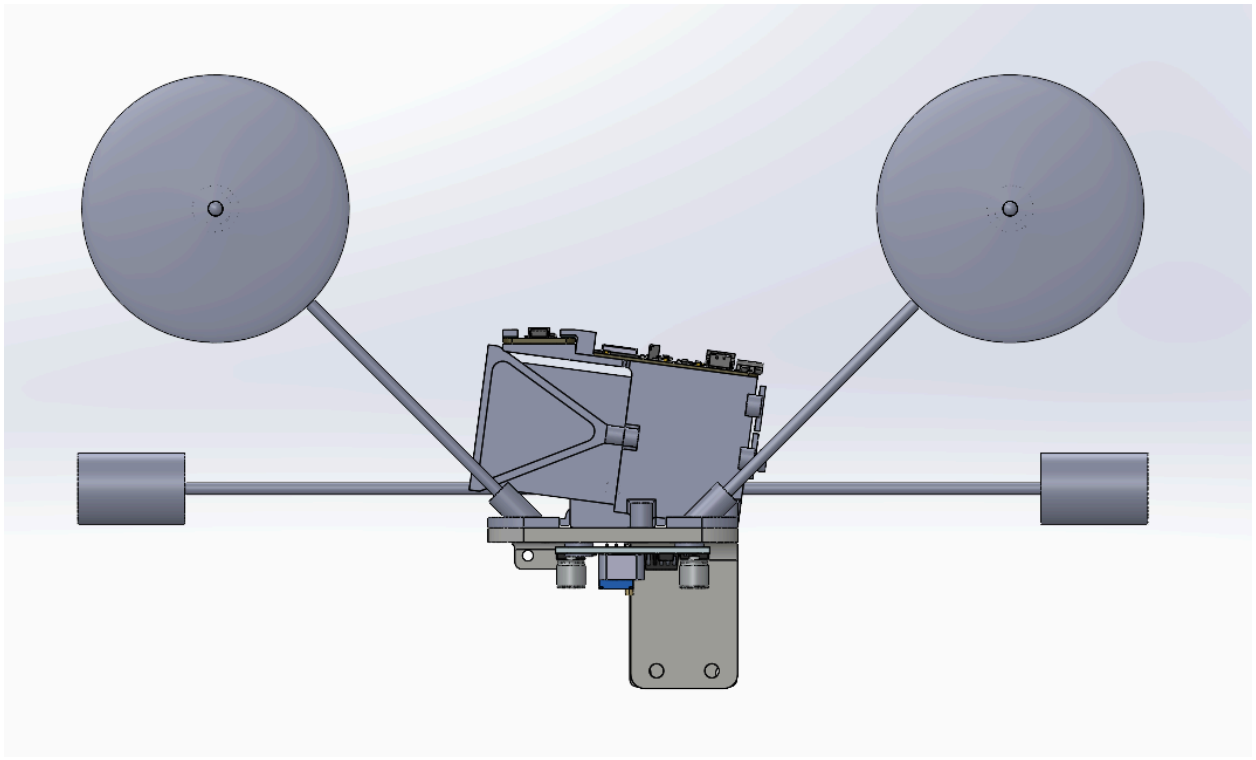
Appendix:

Bill of Materials:

	A	B	C	D
1	Electronics			
2	Name	Quantity	Price	Link
3	ESP32	1	0	NA
4	DRV8833 Dual Motor Driver Carrier	1	0	Pololu - DRV8833 Dual Motor Driver Carrier
5	IMU BNO055	2	30	Adafruit 9-DOF Absolute Orientation IMU Fusion Breakout - BNO055 : ID 2472
6	LM2596 DC to DC Buck Converter	1	8	Amazon.com: 5 Pack LM2596 DC to DC Buck Converter 3.0-40V to 1.5-35V Power Supply Step Down Module
7	Motor with Gearbox	1	8	Motor with gear box
8	Motor with Fan	2	12	Amazon.com: Hobbypower 8520 Coreless Brushed Motor Set 53000rpm 8.5x20mm + 75mm CW CCW Propeller
9	Battery, 3S	1	23.69	Amazon.com: Gens ace 11.1V 1000mAh 3S 45C LiPo Battery Pack with Deans Plug for RC Plane Car Boat Truck Heli Airplane
10	Batter, LiPO	2	0	https://www.adafruit.com/product/4237?gQT=1
11				
12	Hardware			
13	M2x10 Bolt	12	0	NA
14	M2 Nut	12	0	NA
15	M3x6 Bolt	6	0	NA
16	Flywheel weights	2	8.8	uxcell 20mm Precision Chrome Steel Bearing Balls G25 3pcs: Amazon.com
17	Flywheel Rod	1	2.33	https://www.mcmaster.com/8920K14/
18	Flywheel Hat Stock	1	3.64	https://www.mcmaster.com/9146T55-9146T551/
19	M3 heat set inserts	6	0	NA
20	1 kg PETG	1	0	NA
21	Two part epoxy	1	0	NA
22				
23		**TOTAL**	96.46	

CAD:





Code:

```
#include <Wire.h>
#include <Adafruit_Sensor.h>
#include <Adafruit_BNO055.h>
#include <utility/imumaths.h>
#include <Arduino.h>
#define BTN 38
#define BIN_1 25
#define BIN_2 26
#define AIN_1 32
#define AIN_2 15

const int freq = 50000;
const int freq2 = 25000;
const int ledChannel_1 = 1;
const int ledChannel_2 = 2;
const int ledChannel_3 = 3;
const int ledChannel_4 = 4;
const int resolution = 8;
int MAX_PWM_VOLTAGE = 230;
int motor_PWM;
int thruster_PWM;

/* Set the delay between fresh samples */
#define BNO055_SAMPLERATE_DELAY_MS (10)
#define readability (1)

Adafruit_BNO055 bno = Adafruit_BNO055(55, 0x28);
sensors_event_t orientationData , angVelocityData , accelerometerData, gravityData;

//button setup
volatile bool buttonIsPressed = false;
hw_timer_t *debounceTimer = NULL;    // timer for debouncing
volatile bool debounceComplete = true;

hw_timer_t *jumpingTimer = NULL;

void IRAM_ATTR resetDebounce();
void IRAM_ATTR resetJumping();

// Error values
float error_theta_x = 0;
float error_theta_y = 0;
float error_theta_z = 0;

float error_theta_dot_x = 0;
float error_theta_dot_y = 0;
float error_theta_dot_z = 0;

float theta_x_goal = 0;
float theta_y_goal = 0;
float theta_z_goal= 0;
```

```
52
53  float theta_dot_x_goal = 0;
54  float theta_dot_y_goal = 0;
55  float theta_dot_z_goal= 0;
56
57  //angular position
58  float theta_x = 0;
59  float theta_y = 0;
60  float theta_z = 0;
61
62  //angular velocity
63  float theta_dot_x = 0;
64  float theta_dot_y = 0;
65  float theta_dot_z = 0;
66
67  //acceleration
68  float acc_x = 0;
69  float acc_y = 0;
70  float acc_z = 0;
71  float acc_mag = 0;
72
73  float gravityX = 0;
74
75  //PD
76  float pd_flywheel = 0;
77  float pd_thruster_1 = 0;
78  float pd_thruster_2 = 0;
79
80  // Gains
81  static int KpX = 40;
82  static int KpY = 40;
83  static int KpZ = 300;
84
85  static int KdX = 1;
86  static int KdY = 1;
87  static int KdZ = 1;
88
89  static int KiX = 10;
90  static int KiY = 10;
91  static int KiZ = 10;
92
93  // Threshhold values
94  static int y_thresh = 70;
95  static int z_thresh = 70;
96  static int accel_thresh = 20;
97
98  volatile bool Jumping = false;
99
```

```
101
102 //STATE MACHINE VARIABLES
103 enum State {
104   STANDBY,
105   IDLE,
106   JUMPING
107 };
108
109
110 State currentState = IDLE;
111
112 void IRAM_ATTR isr() {
113   if (debounceComplete) {
114     buttonIsPressed = true;
115     debounceComplete = false; // reset debounce flag
116     timerStart(debounceTimer); // start the debounce timer
117   }
118 }
119
120 void setup(void)
121 {
122   Serial.begin(115200);
123   Serial.println("Orientation Sensor Test"); Serial.println("");
124
125   /* Initialise the sensor */
126   if(!bno.begin())
127   {
128     /* There was a problem detecting the BNO055 ... check your connections */
129     Serial.print("Ooops, no BNO055 detected ... Check your wiring or I2C ADDR!");
130     while(1);
131   }
132   bno.setExtCrystalUse(true);
133
134   //setup button press hardware interrupt
135   pinMode(BTN, INPUT);
136   attachInterrupt(BTN, isr, RISING); // attach interrupt to button pin
137
138   // Initialize debounce timer with a 50 ms period
139   debounceTimer = timerBegin(1000000);   // Timer 0, 80 prescaler, count up
140   timerAttachInterrupt(debounceTimer, resetDebounce); // attach interrupt to timer
141   timerAlarm(debounceTimer, 50000, true, 0); // 50 ms debounce period (in microseconds)
142
143   ledcAttach(BIN_1, freq, resolution);
144   ledcAttach(BIN_2, freq, resolution);
145   ledcAttach(AIN_1, freq2, resolution);
146   ledcAttach(AIN_2, freq2, resolution);
147
148   // JUMPING TIMER
149   jumpingTimer = timerBegin(1000000);
150   timerAttachInterrupt(jumpingTimer, resetJumping);
151 }
```

```
void IRAM_ATTR resetDebounce() {
  debounceComplete = true; // allow the button to be pressed again
  timerStop(debounceTimer); // stop timer until next button press
}


void IRAM_ATTR resetJumping() {
  Jumping = false; // allow the button to be pressed again
  timerStop(jumpingTimer); // stop timer until next button press
}


void loop(void) {
  switch (currentState) {
    case STANDBY:
      Serial.println("State: STANDBY");

      // Transition condition to DATA_COLLECTION
      if (CheckForButtonPress()) {
        ButtonResponse();
        currentState = IDLE;
        delay(readability);
      }
      break;

    case IDLE:
      getData();

      if (CheckForButtonPress()) {
        ButtonResponse();
        stop_motors();
        currentState = STANDBY;
        }
      else if (too_tilted()) {
        Serial.println("Returning to STANDBY");
        stop_motors();
        currentState = STANDBY;
        }
      else if (acceleration_threshold()){
        Serial.println("Jumping");
        stop_motors();
        jumpfunction();
        currentState = JUMPING;
        }
      else {
        PD();
        move_thrusters();
        move_flywheel();
      }
      break;
```

```
    case JUMPING:
      Serial.println("State: Jumping");
      if (is_not_jumping()){
        currentState = IDLE;
      }
      break;


  }
}

void getData(void){
  bno.getEvent(&orientationData, Adafruit_BNO055::VECTOR_EULER);
  bno.getEvent(&angVelocityData, Adafruit_BNO055::VECTOR_GYROSCOPE);
  bno.getEvent(&accelerometerData, Adafruit_BNO055::VECTOR_ACCELEROMETER);
  bno.getEvent(&gravityData, Adafruit_BNO055::VECTOR_GRAVITY);

  theta_x = orientationData.orientation.x;
  theta_dot_x = angVelocityData.gyro.x;
  acc_x = accelerometerData.acceleration.x;

  theta_y = orientationData.orientation.y;
  theta_dot_y = angVelocityData.gyro.y;
  acc_y = accelerometerData.acceleration.y;

  theta_z = orientationData.orientation.z;
  theta_dot_z = angVelocityData.gyro.z;
  acc_z = accelerometerData.acceleration.z;
}

// EVENT CHECKERS

bool CheckForButtonPress() {
  return buttonIsPressed && debounceComplete;
}

bool too_tilted() {
  if (theta_y > y_thresh || theta_y < -y_thresh){
    return true;
  } else if (theta_z > z_thresh || theta_z < -z_thresh){
    return true;
  } else {
    return false;
  }
}
```

```
252  bool acceleration_threshold() {
253    acc_mag = sqrt(acc_x*acc_x + acc_y*acc_y + acc_z*acc_z);
254    if (acc_mag > accel_thresh) {
255      return true;
256    } else {
257      return false;
258    }
259  }
260
261  bool is_not_jumping() {
262    return !Jumping;
263  }
264
265  // SERVICES
266
267  void ButtonResponse() {
268    buttonIsPressed = false;
269    Serial.println("Pressed!");
270  }
271
272  void jumpfunction() {
273    Jumping = true;
274    timerStart(jumpingTimer);
275  }
276
277  void move_thrusters() {
278    if (pd_thruster_1 > 0) {
279      thruster_PWM = map(pd_thruster_1, 0, 1024, 0 , MAX_PWM_VOLTAGE);
280      ledcWrite(BIN_2, LOW);
281      ledcWrite(BIN_1, thruster_PWM);
282
283    } else if (pd_thruster_1 < 0) {
284      thruster_PWM = map(pd_thruster_1*(-1), 0, 1024, 0 , MAX_PWM_VOLTAGE);
285      ledcWrite(BIN_1, LOW);
286      ledcWrite(BIN_2, thruster_PWM);
287      }
288  }
289
290  void move_flywheel() {
291    if (pd_flywheel > 0) {
292      motor_PWM = map(pd_flywheel, 0, 1024, 0 , MAX_PWM_VOLTAGE);
293      ledcWrite(AIN_2, motor_PWM);
294      ledcWrite(AIN_1, LOW);
295
296    } else if (pd_flywheel < 0) {
297      motor_PWM = map(pd_flywheel*(-1), 0, 1024, 0 , MAX_PWM_VOLTAGE);
298      ledcWrite(AIN_1, motor_PWM);
299      ledcWrite(AIN_2, LOW);
300      }
301  }
```

```
303  void stop_motors() {
304    ledcWrite(BIN_1, LOW);
305    ledcWrite(BIN_2, LOW);
306    ledcWrite(AIN_1, LOW);
307    ledcWrite(AIN_2, LOW);
308  }
309
310  void PD() {
311    error_theta_z = theta_z_goal - theta_z;
312    error_theta_y = theta_y_goal - theta_y;
313
314    error_theta_dot_z = theta_dot_z_goal - theta_dot_z;
315    error_theta_dot_y = theta_dot_y_goal - theta_dot_y;
316
317    pd_flywheel = KpZ * error_theta_z + KdZ * error_theta_dot_z;
318    pd_thruster_1 = KpY * error_theta_y + KdY * error_theta_dot_y;
319
320  }
321
```