Self Inverting Robot: An Experiment in Unstable Stability

Shashank Swaminathan
Engineering Systems Analysis
Olin College of Engineering

Colin Snow
Engineering Systems Analysis
Olin College of Engineering

I. Overview

The goal of this project was to create a robot which is capable of starting hanging down and subsequently inverting itself to balance in an upright position. Our belief was that from the standard downwards pendulum position, the robot could oscillate its base and eventually move its effective pendulum mass to the upright position. From there, we could begin the standard inverted state balancing algorithm. Hence, to accomplish our goal, we followed a two step approach:

1) We focused on getting the robot to balance in the inverted state reliably while subject to slight external disturbances
2) We focused on writing an algorithm to return the robot to an approximate upright position when not in the inverted state.

We accomplished the first task with a reasonable degree of success - for a video of the robot balancing, click here. Afterwards, we focused on trying to accomplish the second portion of our project. To give space below the robot base for the pendulum to hang when not in the inverted state, we made a 'track' suspended between two tables, with space for the robot to swing through - Figure 1 shows the robot balancing on said track. Once more, we were able to reliably get the robot to balance on the track - here's a video of the robot on tracks.

After balancing, we tried to get the inverting behavior to work. We knew that oscillating the system's base would oscillate the pendulum as well, and we also knew an oscillatory input close to a system's natural frequency would excite the system the most. Hence, we tried to write an oscillating algorithm to move the robot's base in a square-wave pattern at the robot's pendulum's natural frequency. Unfortunately, we were unable to do so, for a combination of several factors:

- The gyroscope that kept track of angle was not able to handle the drift associated with self-inverting behavior well.
  - The accelerometer was slightly offset from the axle, making reading angle measurements from it a non-trivial endeavour.
- The type of rails used to suspend the robot in air, the spacing of said rails, and the type of the robot's wheels, did not provide much traction or constraint on the robot's movement. Hence, forcing the robot to remain parallel on and to the rails was difficult.
- One of our computers would always break the robot's board's bootloader when connected, limiting how often we could upload and test our code

Overall, however, we were able to formulate a system implementation and control flow that showed potential to work, and we believe that given enough time we would be able to successfully implement the self-inverting inverted pendulum.

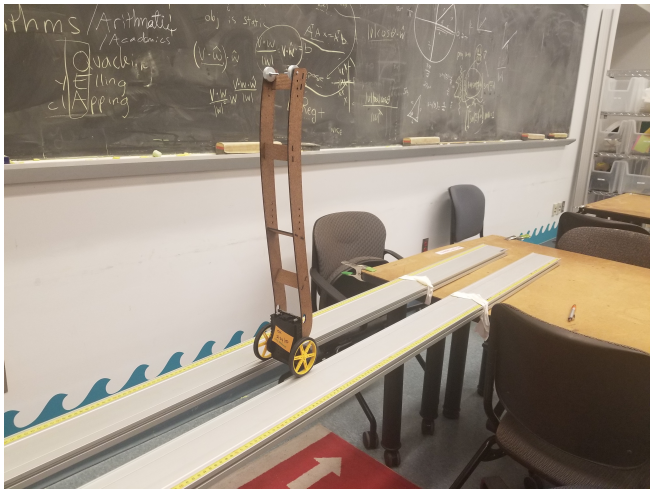

Fig. 1: The robot balancing on rails.

II. System Implementation

In order to invert the robot, we must first have a stable algorithm to make it balance once it is vertical. This is achieved by modelling the dynamics of the robot for small angles from vertical, and using this system to decide a motor input that keeps the robot balanced.

A. Deriving the system function

The system function for the robot is a complex combination of the inverted pendulum and the time response of the motors. We want to create a system which is both quick to respond to changes in angle and also has no error after a long time, necessitating the use of both proportional and integral control.

For a simple pendulum at small angles from the vertical, we can derive the governing differential equation:

$$g\theta - \ell\ddot{\theta} = a$$

Where θ is the angle of the robot from the vertical and a is the acceleration of the cart. From this equation, we can derive the system function which relates velocity input to θ output:

$$H(s) = \frac{s}{g - \ell s^2} = \frac{-s/\ell}{(s - \sqrt{\frac{g}{\ell}})(s + \sqrt{\frac{g}{\ell}})}$$

In order to ensure there will be no steady state error, we apply a PI controller to the system which, using Black's formula, makes the system function:

$$\frac{\Theta(s)}{D(s)} = \frac{1}{1 - H(s)K(s)}$$

Where:

$$K(s) = K_p + \frac{K_i}{s} + \frac{K_p s + K_i}{s}$$

However, this function only holds true when we can directly specify the velocity of the robot at any time. Because our motors are obviously not ideal and can not instantly change velocity, we need to include a function that relates the velocity input into the actually velocity output. The system function for the motors can be approximated as:

$$H_m(s) = \frac{V(s)}{V_c(s)} = \frac{ab}{s + a}$$

Where a and b are constants that describe the steady state and time-dependent responses of the motor. If we want to ensure that this motor function also has a fast response to inputs and zero steady state error, we can apply a PI controller to the motor as well, which yields the system function, again from Black's formula:

$$\frac{V(s)}{V_d(s)} = \frac{H_m(s)}{1 - H_m(s)J(s)}$$

where:

$$J(s) = J_p + \frac{J_i}{s}$$

Substituting this function into the function for the entire system, we can rewrite the total response to a change in θ as:

$$\frac{V_s}{D(s)} = \frac{M_c(s)K(s)}{1 - M_c(s)K(s)H(s)}$$

This system is an effective model of the dynamics we are trying to capture, however it causes the function $X(s)$ (the integral of $V(s)$) to have a pole at zero, which causes the system to be marginally unstable in position. We can fix this in the same way we have done in the other two subsystems by applying a PI controller to the position function. This results in three sets of proportional and integral coefficients, but the proportional coefficient for the position is redundant, which yields five total parameters. Using these parameters, we can write the total system function as:

$$\frac{\Theta(s)}{D(s)} = \frac{ls^5 + (al + J_p abl)s^4 + (J_i abl - g)s^3 +}{ls^5 + (al + J_p abl)s^4 + (K_p ab - g + J_i abl)s^3 +}$$

$$\frac{(C_i abl - ag - J_p abg)s^2 - J_i abgs - C_i abg}{(K_i ab - ag + C_i abl - J_p abg)s^2 - J_i abgs - C_i abg}$$

Which describes the θ output for any input disturbance. It is important to note that this is only true for small values of θ and values for the motor input that do not exceed the motor's max speed.

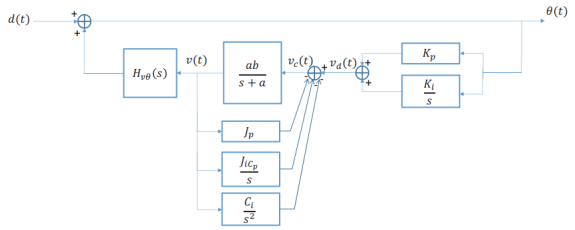

Fig. 2: Block diagram of the entire control system.

Now that we have the system function, we need to decide on a set of parameters for all of the PI coefficients and the motor coefficients such that the system is stable and responds well to disturbances. This can be achieved by defining the poles which are the roots of the denominator of the system function and then calculating coefficients to find those roots. However, to find these coefficients we first need to find the coefficients a and b for the motors, as these can not be set arbitrarily.

B. Motor Characterization and Effective Length

To find the values for a and b, we must run an experiment on the robot to determine how it responds to a unit step input. This is achieved by placing the robot inverted on a track and running it for three seconds, recording the motor velocities from encoder data. Once this data is collected, it can be fit to find coefficients of a and b which best describe the response with the system function as described before:

$$H_m(s) = \frac{ab}{s + a}$$

Figure 3 shows data collected from the motors n order to determine these coefficients. We found that a value of 11.41 for a and .0029 for b best characterize this first-order response and therefore provide an accurate approximation for the motors.

In addition to finding these motor coefficients, we must also determine the effective length of the robot. Because the robot is not perfectly represented by an idealized mass on a massless rod, we can instead use a good approximation for the effective length calculated from the natural frequency. We know that for an ideal pendulum the formula which relates frequency to length is:

$$F_n = \sqrt{\frac{g}{\ell}}$$

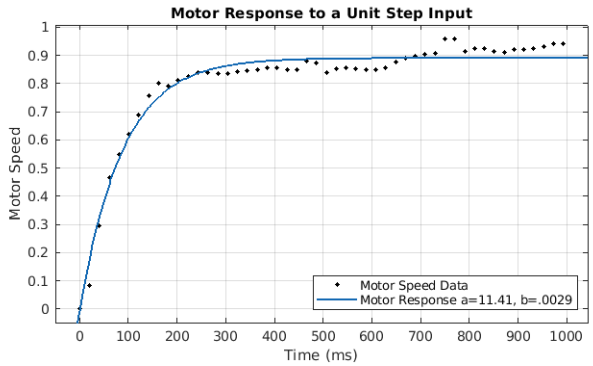

Fig. 3: Motor response to a unit step input. Data was collected by running the robot at full speed and recording its velocity curve.

Solving for ℓ yields:

$$\ell = \frac{g}{F_n^2}$$

By placing the robot upside-down on the tracks again and instead reading from the gyro, we were able to find the period of the gyro data and determine the natural frequency to be 4.928 rad/s. Using the above formula, the effective length, ℓ, is .4038 m. Figure 4 shows our collected gyro data.

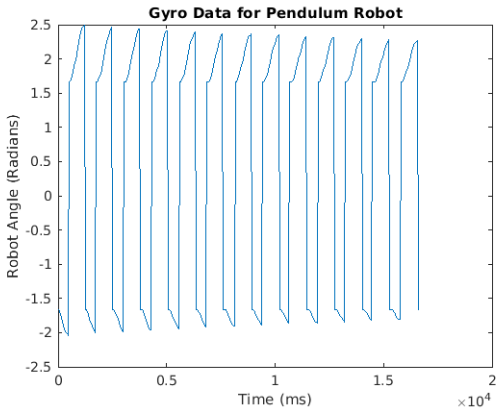

Fig. 4: Gyro data collected for analysis of the characteristic frequency. The data jumps values when passing straight down but this does not affect frequency.

With these three variables found, the only other values to find are those of the five coefficients.

C. Poles

In order to find all of the P and I coefficients, we need to construct the characteristic equation of the system such that it has poles which make it stable. We did this by choosing a set of poles and then using MATLAB to solve for the corresponding coefficients.

We chose the poles p1-p5 and then used these values to solve for the corresponding coefficients. The poles were selected in order to give the system a good response to quick disturbances while also having gentle movements near vertical to smooth out any oscillations. The poles at $-\frac{1}{2} \pm 2\pi i$ cause an oscillation at about the natural period of the robot, and gives the response an oscillatory component so that it can use more power and catch itself without worrying about overshooting. The poles at -1 give long-term stability to the system as they die out very slowly, creating

$$\begin{aligned} p1 &= -\frac{1}{2} + 2\pi i & J_p &= 168.9383 \\ p2 &= -\frac{1}{2} - 2\pi i & J_i &= -1508.8 \\ p3 &= -1 & K_p &= 2013.8 \\ p4 &= -1 & K_i &= 14013 \\ p5 &= -14 & C_i &= -719.6362 \end{aligned}$$

From these coefficients we can define a system which can respond to environmental disturbances and keep the robot balanced. Figure 5 shows the response of the system to a unit step input. This response shows damped oscillatory behavior, representing a slight oscillation in θ just after the disturbance and then a constant value in the long term, representing zero steady-state error.

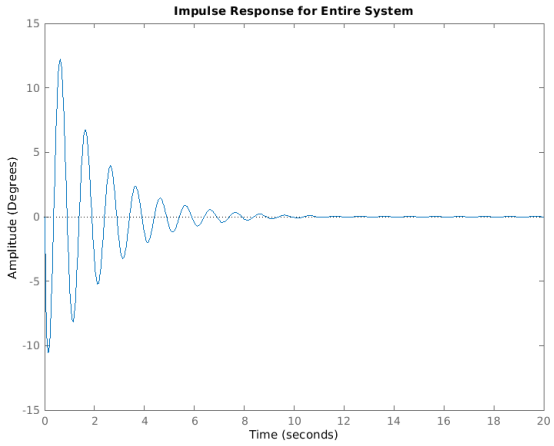

Fig. 5: Impulse response for the entire system after a disturbance.

III. Self Inverting

Now that we have derived a function to keep the robot balanced when within a few degrees of vertical, we must find a way to get it from upside down to within this range. As previously discussed, our system function is a linearization of the system dynamics around $\theta = 0$, so for angles far from this it is mostly useless. Instead we must consider a function which will repeatedly increase the energy of our system in order to bring it upright.

A. Natural Frequency

The best way to get the robot to vertical is to oscillate in phase with the robot's natural frequency, thus always pushing in such a way as to increase the total energy of the system. Fortunately, we already know the natural frequency of the system from the gyro experiment to be 4.928 rad/s, which corresponds to a period of 1.275 seconds.

In order to oscillate the robot at this frequency, we simply turn the motors to full positive power for 637 ms then to full negative for another 637 ms. In this way we slowly build energy until we get close to vertical, then let the balancing code take over.

B. Challenges

1) Gyro Drift: The first issue we encountered in the implementation of this system was the tendency of the gyro to drift quite dramatically when moved through large angles. We found that after calibration the angle was extremely steady and accurate, but that moving the robot around in a circle caused an error of over 4 radians. We think this is due to a combination of sampling rate and just general sensor inaccuracies. The gyro samples at about 100 Hz and only measures angular velocity, so any small errors due to miscalculated velocity or slow sampling are quickly magnified. To help combat this effect, we used the accelerometer in addition to the gyro in order to provide a good reference for what is actually up and let the gyro do the fine tuning. In order to determine if we were close to vertical, we looked at the component of the robot acceleration that corresponded to the downward direction, and looked for that number to approach the acceleration of gravity. The acceleration for any angle from the vertical, θ, is:

$$accel = g\cos(\theta)$$

We then chose a small value for θ such that the small angle approximation from the balancing code holds well and oscillated the robot until that acceleration level was reached. Then we set the θ value at that moment to the given angle and used the gyro for fine positioning. With this strategy, we can use the gyro only when needed and only when the angle is not changing rapidly, which dramatically cuts down on the drift. However, inherent to this strategy is the assumption that the accelerometer is on the axle of the wheels, i.e. it does not experience additional acceleration due to the swinging of the pendulum. As it turned out, this assumption does not hold true. We attempted to correct for this by increasing and fine-tuning the threshold of downwards-axis acceleration to be greater than g (accounting for the increase due to centripetal acceleration).

2) Rails, Traction, and Spacing: Because the Rocky robots are by no means meant to attempt this kind of challenge, creating a setup which works was quite difficult. We used aluminum rails and placed the wheels as far out on the axles as possible, but this still did not leave much room for the rest of the robot to swing through the narrow gap. This led to many falls and other issues. Also, because the weight of the robot is not always on its wheels, we had problems with the limited traction on the track preventing us from getting up the energy we needed to stand up.

C. Final System Implementation Diagram

Figure 6 shows the finalized diagram of the system implementation, incorporating self-inverting algorithm, balancing algorithm, and accelerometer-based angle correction.

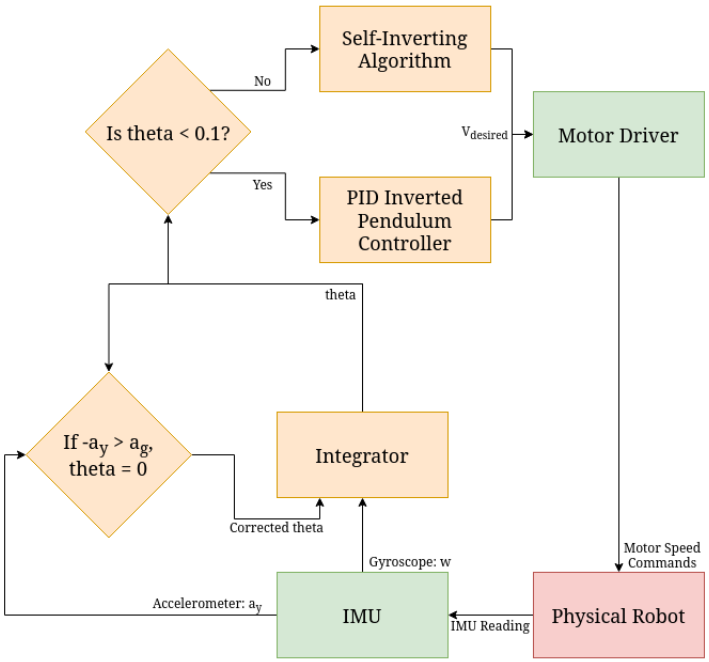

Fig. 6: Final Diagram of the System Implementation.

D. Results

In the end we were not able to make the robot self invert due to a combination of the challenges mentioned above and a general lack of time to work on it in the week before our departure. However, we are confident that with continued work we could achieve this goal, as long as the motors are strong enough and the robot does not loose traction on the track.

IV. Code

Here is a selection of the code used for our project - the full code base can be found on GitHub.

A. Determining system parameters

Finding natural frequency and effective length:

```
t = linspace(0,length(a)*50, length(a));

plot(t,a)

title("Gyro Data for Pendulum Robot")

xlabel("Time (ms)")
ylabel("Robot Angle (Radians)")

```

```
[pks, locs] = findpeaks(a)

freq = 2*pi / (mean(diff(locs))*50/1000)

% natural frequency is 4.9280

length_eq = 9.80665/(freq^2)

% Equivalent length is .4038
```

Finding coefficients a and b:

```
% Finding motor coefficients (x is motor data)
t = linspace(0,length(x)*20, length(x));
figure(2)
plot(t,x)
% Apply a two point moving average to reduce ...
    noise
average_speed = movmean((x(:,1)+x(:,2))/2,2);
hold on

% Plot Motor Response
plot(t,average_speed)

% Cut relevant part of data for fitting
average_speed_cut = average_speed(1:50);
t_cut = t(1:50);
```

B. *Driving the robot*

There are two versions of code shown here, the last version of the code that still let the robot exhibit inverted pendulum behavior, and the latest version of the code that does not work but incorporates the accelerometer-based correction to angle. Both depend on the Balance.cpp and Balance.h files found on the GitHub.

1) Last working version:

```
/* Code by Shashank Swaminathan
 * For the ESA class project
 *
 * Last working version of the code
 * Has the Rocky staying up and inverted.
 * Built off the Rocky_Balancing_Starter_Code
 */
#include <Balboa32U4.h>
#include <Wire.h>
#include <LSM6.h>
#include "Balance.h"


extern int32_t angle_accum;
extern int32_t speedLeft;
extern int32_t driveLeft;
extern int32_t distanceRight;
extern int32_t speedRight;
extern int32_t distanceLeft;
extern int32_t distanceRight;
float speedCont = 0;
float displacement_m = 0;
int16_t limitCount = 0;
uint32_t cur_time = 0;
float distLeft_m;
float distRight_m;


extern uint32_t Δ_ms;
float measured_speedL = 0;
float measured_speedR = 0;
float desSpeedL=0;
float desSpeedR =0;
float dist_accumL_m = 0;
float dist_accumR_m = 0;
float dist_accum = 0;
float speed_err_left = 0;
float speed_err_right = 0;
float speed_err_left_acc = 0;
float speed_err_right_acc = 0;
float errAccumRight_m = 0;
float errAccumLeft_m = 0;
float prevDistLeft_m = 0;
float prevDistRight_m = 0;
float angle_rad_diff = 0;
// this is the angle in radians
float angle_rad;
// this is the accumulated angle in radians
float angle_rad_accum = 0;
// previous angle measurement
float angle_prev_rad = 0;
extern int32_t displacement;
int32_t prev_displacement=0;
uint32_t prev_time;

#define G_RATIO (162.5)



LSM6 imu;
Balboa32U4Motors motors;
Balboa32U4Encoders encoders;
Balboa32U4Buzzer buzzer;
Balboa32U4ButtonA buttonA;

// Replace the 0.25 with the value obtained
// from the Gyro calibration procedure
#define FIXED_ANGLE_CORRECTION (0.29)

void BalanceRocky()
{
    // Enter the control parameters here

    float Kp = 1903.8;
    float Ki = 13248;

    float Ci = -680.34;

    float Jp = 85.71;
    float Ji = -1426.4;

        // Control velocities for motors
    float v_c_L, v_c_R;
        // Desired velocity set by controller
    float v_d = 0;

    // Variables available to you are:
    // angle_rad  - angle in radians
    // angle_rad_accum - integral of angle
    // measured_speedR - RW speed (m/s)
    // measured_speedL - LW speed (m/s)
    // distLeft_m - distance traveled by LW
    // distRight_m - distance traveled by RW
         // Both distances are in meters
     // Integral of velocities
    // dist_accum - integral of the distance

        // Desired velocity from angle ...
            controller
        v_d = Kp*angle_rad + Ki*angle_rad_accum;

    // The next two lines implement the
        // feedback controller for the motor.
    // Two separate velocities are calculated.
    //
    // We use a trick here by criss-crossing
```

```
        // the distance from left to right
        // and right to left.
        // Ensures that the motors are balanced

    v_c_R = v_d - Jp*measured_speedR -
                Ji*distLeft_m  - dist_accum*Ci;
    v_c_L = v_d - Jp*measured_speedL -
                Ji*distRight_m - dist_accum*Ci;

    // save desired speed for debugging
    desSpeedL = v_c_L;
    desSpeedR = v_c_R;

    // motor control must be between +- 300.
        // Clips values to be within that range
    if(v_c_L > 300) v_c_L = 300;
    if(v_c_R > 300) v_c_R = 300;
    if(v_c_L < -300) v_c_L = -300;
    if(v_c_R < -300) v_c_R = -300;

    // Set the motor speeds
    motors.setSpeeds((int16_t) (v_c_L),
(int16_t)(v_c_R));

}



void setup()
{
  // Uncomment if your motors are reversed.
  // motors.flipLeftMotor(true);
  // motors.flipRightMotor(true);

  Serial.begin(9600);
  prev_time = 0;
  displacement = 0;
  ledYellow(0);
  ledRed(1);
  balanceSetup();
  ledRed(0);
  angle_accum = 0;

  ledGreen(0);
  ledYellow(0);
}



int16_t time_count = 0;
extern int16_t angle_prev;
int16_t start_flag = 0;
int16_t start_counter = 0;
int16_t invertDir = 0;
uint32_t inv_time = 0;
const uint8_t INV_TIME_MS = 650;
void lyingDown();
extern bool isBalancingStatus;
extern bool balanceUpdateDelayedStatus;

void UpdateSensors()
{
  static uint16_t lastMillis;
  uint16_t ms = millis()

  // Perform the balance updates at 100 Hz.
  balanceUpdateDelayedStatus = ms -
            lastMillis > UPDATE_TIME_MS + 1;
  lastMillis = ms;

  // call functions to integrate
    // encoders and gyros
  balanceUpdateSensors();

  if (imu.a.x < 0)
  {
    lyingDown();
    isBalancingStatus = false;
  }
  else
  {
    isBalancingStatus = true;
  }
}



void GetMotorAndAngleMeasurements()
{

    // convert distance calculation into meters
    // and integrate distance
    distLeft_m = ((float)distanceLeft)/
                ((float)G_RATIO)/12.0*80.0/
                1000.0*3.14159;
    distRight_m = ((float)distanceRight)/
                ((float)G_RATIO)/12.0*80.0/
                1000.0*3.14159;
    dist_accum += (distLeft_m+distRight_m)*
                0.01/2.0;

    // compute left and right wheel speed in ...
        meters/s
    measured_speedL = ...
        speedLeft/((float)G_RATIO)/
                12.0*80.0/1000.0*3.14159*100.0;
    measured_speedR = ...
        speedRight/((float)G_RATIO)/
                12.0*80.0/1000.0*3.14159*100.0;

    prevDistLeft_m = distLeft_m;
    prevDistRight_m = distRight_m;


    // this integrates the angle
    angle_rad_accum += angle_rad*0.01;
    // this is the derivative of the angle
    angle_rad_diff = (angle_rad-
angle_prev_rad)/0.01;
    angle_prev_rad  = angle_rad;
}

void balanceResetAccumulators()
{
    errAccumLeft_m = 0.0;
    errAccumRight_m = 0.0;
    speed_err_left_acc = 0.0;
    speed_err_right_acc = 0.0;
}

void loop()
{
  // Serial print controller
  static uint32_t prev_print_time = 0;
    // RW vs. LW encoder distance
  int16_t distanceDiff;
  static float del_theta = 0;
  char enableLongTermGyroCorrection = 1;

  cur_time = millis(); // current time in ms


  if((cur_time - prev_time) > UPDATE_TIME_MS){
            UpdateSensors(); // update sensors

            // calculate the angle in radians.
            // The FIXED_... comes from ...
                angle calib.
```

```
            // del_theta corrects for ...
                long-term drift
    angle_rad = ((float)angle)/1000/180*3.14159
                - FIXED_ANGLE_CORRECTION - ...
                    del_theta;

        // check if angle is within upright ...
            bounds
    if(angle_rad > 0.1 || angle_rad < -0.1)
    {
      start_counter = 0;
   }


  if(angle_rad > -0.3 &&
        angle_rad < 0.3 && ! start_flag)
  {
    balanceResetEncoders();
    start_flag = 1;
    buzzer.playFrequency(DIV_BY_10 | 445,
1000, 15);
    Serial.println("Starting");
    ledYellow(1);
  }

  // if start_flag, do balancing
  if(start_flag)
  {
    GetMotorAndAngleMeasurements();
    if(enableLongTermGyroCorrection)
                // Handle gyro drift
                // Assumes gyro is standing
                del_theta = 0.999*del_theta +
                        0.001*angle_rad;
    // Control the robot
    BalanceRocky();
  }
  else {
    if (cur_time - inv_time ≥ INV_TIME_MS) {
    // save the last time you blinked the LED
    inv_time = cur_time;

    // if the LED is off turn it on and vcvs:
    if (invertDir) {
      motors.setSpeeds(1,1);
      invertDir = 0;
    } else {
      motors.setSpeeds(-1,-1);
      invertDir = 1;
    }
    }
  }
  prev_time = cur_time;
  }
// if the robot is more than 45 degrees, ...
// OR consider: Just flip it back up
  if(start_flag && angle_rad > .78)
  {
    start_flag = 0;
  }
  else if(start_flag && angle < -0.78)
  {
    start_flag = 0;
  }

// kill switch
  if(buttonA.getSingleDebouncedPress())
  {
      motors.setSpeeds(0,0);
      while(!buttonA.getSingleDebouncedPress());
  }

// Print every 105 ms
// Avoid 10ms multiples to not hog processor
if(cur_time - prev_print_time > 103)
  {
        Serial.print(angle_rad);
        Serial.print("\t");
        Serial.print(distLeft_m);
        Serial.print("\t");
        Serial.print(measured_speedL);
        Serial.print("\t");
        Serial.print(measured_speedR);
        Serial.print("\t");
       Serial.println(speedCont);
       prev_print_time = cur_time;
  }
}
```

2) Latest version with accelerometer-based θ correction:

```
/* Code by Shashank Swaminathan for the ESA ...
    class project
 * For the ESA inverted pendulum class project.
 *
 * This is the code to get the Rocky to ...
     self-invert.
 * Currently not a working version.
 * Built off the Rocky_Balancing_Starter_Code
 */

#include <Balboa32U4.h>
#include <Wire.h>
#include <LSM6.h>
#include "Balance.h"

extern int32_t angle_accum;
extern int32_t speedLeft;
extern int32_t driveLeft;
extern int32_t distanceRight;
extern int32_t speedRight;
extern int32_t distanceLeft;
extern int32_t distanceRight;
float speedCont = 0;
float displacement_m = 0;
int16_t limitCount = 0;
uint32_t cur_time = 0;
float distLeft_m;
float distRight_m;

extern uint32_t ∆_ms;
float measured_speedL = 0;
float measured_speedR = 0;
float desSpeedL=0;
float desSpeedR =0;
float dist_accumL_m = 0;
float dist_accumR_m = 0;
float dist_accum = 0;
float speed_err_left = 0;
float speed_err_right = 0;
float speed_err_left_acc = 0;
float speed_err_right_acc = 0;
float errAccumRight_m = 0;
float errAccumLeft_m = 0;
float prevDistLeft_m = 0;
float prevDistRight_m = 0;
float angle_rad_diff = 0;
// angle in radians
float angle_rad;
// acculumlated angle in radians
float angle_rad_accum = 0;
// previous angle measurement
float angle_prev_rad = 0;
// Unsigned upper limit on a.x
const float AX_THRESH = 1;
// signed threshold on a.g + a.y
const float AY_THRESH = -9.8;
```

```
extern int32_t displacement;
int32_t prev_displacement=0;
uint32_t prev_time;

#define G_RATIO (162.5)

LSM6 imu;
Balboa32U4Motors motors;
Balboa32U4Encoders encoders;
Balboa32U4Buzzer buzzer;
Balboa32U4ButtonA buttonA;

// Replace the 0.25 with the value obtained
// from the Gyro calibration procedure
#define FIXED_ANGLE_CORRECTION (0.24)

void BalanceRocky()
{

    // Enter the control parameters here

    float Kp = 1903.8;
    float Ki = 13248;

    float Ci = -680.34;

    float Jp = 85.71;
    float Ji = -1426.4;

        // Control velocities for motors
    float v_c_L, v_c_R;
        // Desired velocity set by controller
    float v_d = 0;

    // Variables available to you are:
    // angle_rad  - angle in radians
    // angle_rad_accum - integral of angle
    // measured_speedR - RW speed (m/s)
    // measured_speedL - LW speed (m/s)
    // distLeft_m - distance traveled by LW
    // distRight_m - distance traveled by RW
         // Both distances are in meters
     // Integral of velocities
    // dist_accum - integral of the distance

        // Desired velocity from angle ...
            controller
        v_d = Kp*angle_rad + Ki*angle_rad_accum;

        // The next two lines implement the
        // feedback controller for the motor.
    // Two separate velocities are calculated.
    //
    // We use a trick here by criss-crossing
        // the distance from left to right
        // and right to left.
        // Ensures that the motors are balanced

    v_c_R = v_d - Jp*measured_speedR -
                Ji*distLeft_m  - dist_accum*Ci;
    v_c_L = v_d - Jp*measured_speedL -
                Ji*distRight_m - dist_accum*Ci;

    // save desired speed for debugging
    desSpeedL = v_c_L;
    desSpeedR = v_c_R;

        // motor control must be between +- 300.
        // Clips values to be within that range.
    if(v_c_L > 300) v_c_L = 300;
    if(v_c_R > 300) v_c_R = 300;
    if(v_c_L < -300) v_c_L = -300;
    if(v_c_R < -300) v_c_R = -300;

    // Set the motor speeds
    motors.setSpeeds((int16_t) (v_c_L),
(int16_t)(v_c_R));

}

void setup()
{
  // Uncomment if your motors are reversed.
  // motors.flipLeftMotor(true);
  // motors.flipRightMotor(true);

  Serial.begin(9600);
  prev_time = 0;
  displacement = 0;
  ledYellow(0);
  ledRed(1);
  balanceSetup();
  ledRed(0);
  angle_accum = 0;

  ledGreen(0);
  ledYellow(0);
}



int16_t time_count = 0;
extern int16_t angle_prev;
int16_t start_flag = 0;
int16_t start_counter = 0;
int16_t invertDir = 1;
uint32_t inv_time = 0;
const uint8_t INV_TIME_MS = 650;
void lyingDown();
extern bool isBalancingStatus;
extern bool balanceUpdateDelayedStatus;

void UpdateSensors()
{
  static uint16_t lastMillis;
  uint16_t ms = millis();

  // Perform the balance updates at 100 Hz.
  balanceUpdateDelayedStatus = ms -
            lastMillis > UPDATE_TIME_MS + 1;
  lastMillis = ms;

  // call functions to integrate
    // encoders and gyros
  balanceUpdateSensors();

  if (imu.a.x < 0)
  {
    lyingDown();
    isBalancingStatus = false;
  }
  else
  {
    isBalancingStatus = true;
  }
}

void GetMotorAndAngleMeasurements()
{
    // convert distance calculation into meters
    // and integrate distance
    distLeft_m = ((float)distanceLeft)/
                ((float)G_RATIO)/12.0*80.0/
                1000.0*3.14159;
    distRight_m = ((float)distanceRight)/
                ((float)G_RATIO)/12.0*80.0/
                1000.0*3.14159;
    dist_accum += (distLeft_m+distRight_m)*
```

```
                0.01/2.0;

    // compute left and right wheel speed in ... meters/s
    measured_speedL = ...
        speedLeft/((float)G_RATIO)/
                12.0*80.0/1000.0*3.14159*100.0;
    measured_speedR = ...
        speedRight/((float)G_RATIO)/
                12.0*80.0/1000.0*3.14159*100.0;

    prevDistLeft_m = distLeft_m;
    prevDistRight_m = distRight_m;


    // this integrates the angle
    angle_rad_accum += angle_rad*0.01;
    // this is the derivative of the angle
    angle_rad_diff = (angle_rad-
angle_prev_rad)/0.01;
    angle_prev_rad  = angle_rad;

}

void balanceResetAccumulators()
{
    errAccumLeft_m = 0.0;
    errAccumRight_m = 0.0;
    speed_err_left_acc = 0.0;
    speed_err_right_acc = 0.0;
}

void loop()
{
  // Serial print controller
  static uint32_t prev_print_time = 0;
    // RW vs. LW encoder distance
  int16_t distanceDiff;
  static float del_theta = 0;
  char enableLongTermGyroCorrection = 1;

  cur_time = millis(); // current time in ms


  if((cur_time - prev_time) > UPDATE_TIME_MS){
            UpdateSensors(); // update sensors

            // calculate the angle in radians.
            // The FIXED_... comes from ... angle calib.
            // del_theta corrects for ... long-term drift
            angle_rad = ...
                ((float)angle)/1000/180*3.14159 ...
                - FIXED_ANGLE_CORRECTION - ...
                del_theta;

            // check if angle is within ... upright bounds
            if(angle_rad > 0.1 || angle_rad ...
                < -0.1)
            {
                    start_counter = 0;
            }

        if(abs(imu.a.x) < AX_THRESH &&
             imu.a.y < AY_THRESH && !start_flag)
        {
                balanceResetEncoders();
                angle = 0;
                start_flag = 1;
                BalanceRocky();
                buzzer.playFrequency(DIV_BY_10 ...
                    | 445,
1000, 15);
                Serial.println("Starting");
                ledYellow(1);
        }

  // if start_flag, do balancing
  if(start_flag)
  {
    GetMotorAndAngleMeasurements();
    if(enableLongTermGyroCorrection)
                // Handle gyro drift
                // Assumes gyro is standing
                del_theta = 0.999*del_theta+
                        0.001*angle_rad;
    // Control the robot
    BalanceRocky();
  }
  else {
    if (cur_time - inv_time ≥ INV_TIME_MS) {
      // save the last time LED was blinked
      inv_time = cur_time;
      invertDir = -1*invertDir;
      motors.setSpeeds(invertDir*100,
invertDir*100);
    }
  }
  prev_time = cur_time;
  }
// if the robot is more than 45 degrees, ...
// OR consider: Just flip it back up
  if(start_flag &&
          (angle_rad > .78 || angle_rad < -0.78))
  {
    start_flag = 0;
  }

// kill switch
  if(buttonA.getSingleDebouncedPress())
  {
      motors.setSpeeds(0,0);
      while(!buttonA.getSingleDebouncedPress());
  }

// Print every 105 ms
// Avoid 10ms multiples to not hog processor
if(cur_time - prev_print_time > 103)
  {
        Serial.print(angle_rad);
        Serial.print("\t");
        Serial.print(distLeft_m);
        Serial.print("\t");
        Serial.print(measured_speedL);
        Serial.print("\t");
        Serial.print(measured_speedR);
        Serial.print("\t");
       Serial.println(speedCont);
       prev_print_time = cur_time;
  }
}
```