QEA II Module 2: Robolympics

Braden Oh, Kyle Bertram

1 Athlete Demographics

The first challenge that we decided to complete was the "Rocky Stand Still!" challenge, in which the Rocky robot is challenged to stand upright for as long as possible while remaining within a 2ft x 2ft square box marked on the floor. A rough control algorithm that allows the Rocky robot to stand is given in the following series of steps:

1. Compute the error between the desired and actual angles of the robot
2. Calculate a velocity necessary to correct the error
3. Compute the error between the desired and actual velocities of the robot
4. Calculate a motor PWM signal that compensates for the velocity error
5. Measure the actual motor velocity achieved by a given PWM signal
6. Measure the final angle of the robot
7. Integrate the distance traveled as the robot drifts
8. Adjust the desired angle of the robot backwards to compensate for drift and repeat

Implicit in this series of steps are a set of controllers and transfer functions that can be organized into a block diagram as shown in Figure 1.

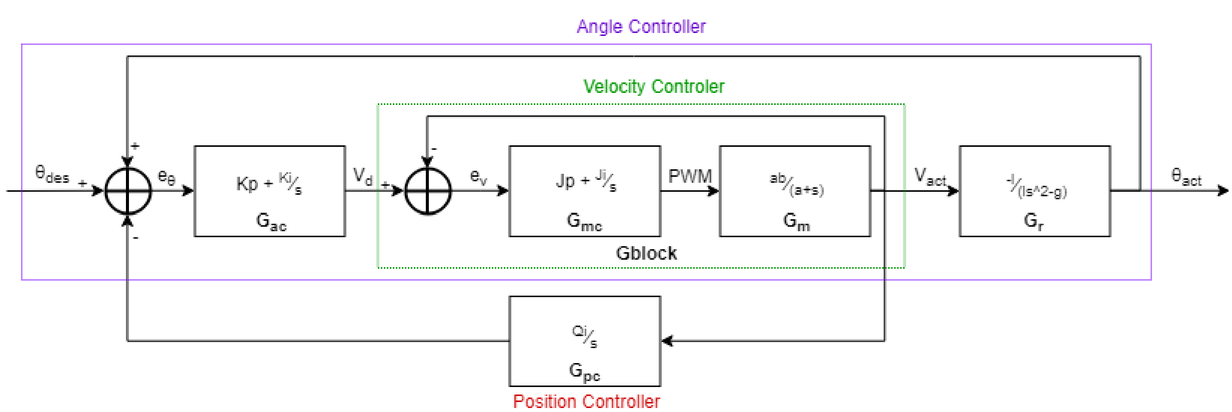

Figure 1: Detailed system block diagram showing transfer function expressions in the s-domain. Explanations of the blocks and transfer functions are given below.

A brief definition of the variables used in the Figure 1 block diagram are given in the following table. Full explanations are given further below the table.

Term	Description
K_p	Proportional term weight for PI angle controller
K_I	Integral term weight for PI angle controller
J_p	Proportional term weight for PI velocity controller
J_I	Integral term weight for PI velocity controller
a	Experimentally determined motor parameter
b	Experimentally determined motor parameter
Q_I	Weight for velocity integrator (net distance)
l	Length from axle to robot center of mass
g	Gravitational acceleration
θ_{des}	Desired robot angle value
θ_{out}	Actual (measured) robot angle value
v_{des}	Desired robot velocity
PWM	PWM signal sent to control motor speed
v_{act}	Actual (measured) wheel velocity

As shown in Figure 1, a given desired angle (usually zero radians, which is standing upright), θ_{des}, is fed into the system. Steps 1 and 8 are completed together by computing the angle error term, e_θ, which is given by the expression

$$e_\theta = \theta_{des} - \theta_{out} + \frac{Q_I}{s} v_{out}$$

where the angle error given by the difference between the actual and desired angles is added to a proportion of the total distance traveled (integral of v_{out}) to yield some final desired angle, e_θ. Step 2 is accomplished by passing this actual desired angle through a PI angle controller, denoted by the block G_{AC}. This PI controller has values K_p and K_I for the proportional and integral term coefficients, respectively, and outputs a desired velocity, v_{des}.

Step 3 is completed by simply subtracting the actual velocity of the robot from the desired velocity and step 4 is completed by passing that difference through a PI motor controller, denoted by the block G_{MC}. This PI controller has values J_p and J_I for the proportional and integral term coefficients, respectively, and outputs a PWM signal, PWM.

An implicit block appears between steps 4 and 5, acknowledging that the PWM signal sent to one of Rocky's motors does not result in an instantaneous velocity change, but rather an actual velocity determined by an s-domain function of the form $\frac{ab}{s+a}$, where a and b are experimentally derived constants unique to the motor. We used provided constant values $a = 14$ and $b = \frac{1}{400}$. To this end, a PWM signal passed into a motor behaving according to this function, G_M, will result in an actual velocity output, v_{act} that can be measured directly as a derivative of encoder data.

Another implicit block appears between steps 5 and 6, acknowledging that Rocky's final angle is a function of its actual velocity. This final angle is determined by an s-domain function of the form $\frac{-s}{ls^2-g}$, where l is the "effective length" of the inverted pendulum, the distance between the axis of rotation and the center of mass of the pendulum, and g is acceleration due to gravity. Our Rocky's value is $l = 9.42$cm, as it has 100g of mass at the top of the pendulum. To this end, a velocity applied by the motors on a Rocky robot according to this function, G_R, will result in an actual final angle, θ_{out}, that can be measured directly via a gyroscope.

Step 7 is completed by integrating the actual velocity of the system, v_{act}, by passing it through an s-domain function of the form $\frac{1}{s}$ and then multiplying this integral by a scale factor, Q_I. Step 8 is achieved as this weighted velocity integral is saved and used in the calculation for the actual desired angle on the next pass through the control loop (step 1 for the next pass through the control loop).

A net transfer function for this system could be calculated at this point, but a simplification can be made first; the dotted region in Figure 1 is a subsystem used for "cruise control", and can be treated as a single block with a single transfer function. A block diagram for this subsystem is shown in Figure 2.

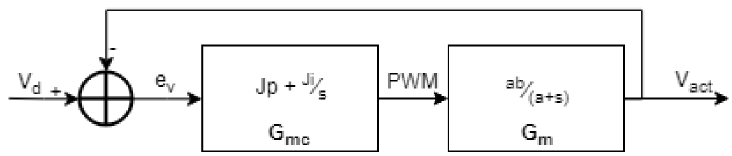

Figure 2: Block diagram of the motor/motor-controller subsystem showing frequency response equations in the s-domain.

The overall flow function for this subsystem block is given by the expression:

$$v_{act} = (v_{des} - v_{act})G_{MC}G_M$$

From here the transfer function for the subsystem, G_b can be derived:

$$G_b = \frac{v_{act}}{v_{des}} = \frac{G_{MC}G_M}{1 + G_{MC}G_M}$$

This entire cruise control subsystem can then be replaced in the overall block diagram by a single block of transfer function G_b which takes in a desired velocity, v_{des}, and outputs an actual velocity, v_{act}. This simplified block diagram is given in Figure 3.

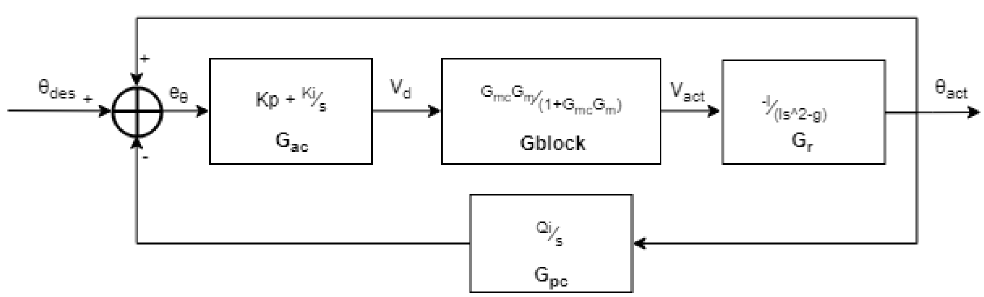

Figure 3: Simplified system block diagram with G_b replacing the transfer function for the motor/motor-controller subsystem. This diagram also includes frequency response equations in the s-domain.

This simplified block diagram is easier to find the overall transfer function for. To do this, we begin by taking the overall flow function for the system, which is given by the expression,

$$\theta_{out} = e_\theta G_{AC} G_b G_R$$

where

$$e_\theta = \theta_{desired} + \frac{Q_I}{s} v_{out} - \theta_{out}$$

which yields the flow function,

$$\theta_{out} = (\theta_{desired} + \frac{Q_I}{s} v_{out} - \theta_{out}) G_{AC} G_b G_R$$

which can be re-arranged into the transfer function

$$\frac{\theta_{out}}{\theta_{in}} = \frac{G_{AC} G_b G_R}{1 + G_{AC} G_b G_R - G_{AC} G_b \frac{Qi}{s}}$$

The s-domain equations for each block are re-iterated in the following table:

Term	Block Name	Variable Representation
Angle Controller	G_{AC}	$K_p + \frac{K_I}{s}$
Motor Controller	G_{MC}	$J_p + \frac{J_I}{s}$
Motor	G_M	$\frac{ab}{s+a}$
Robot	G_R	$\frac{-s}{ls^2-g}$
Motor Subsystem	G_b	$\frac{G_{MC}G_M}{1+G_{MC}G_M}$

Substituting these expressions in for the G terms of the transfer function $\frac{\theta_{out}}{\theta_{in}}$ yields a fraction whose denominator can be analyzed to determine the poles of the system. An overview of this analysis is given in the next section, "Athlete Performance Information."

2 Athlete Performance Information

Once we had the transfer function relating $\frac{\theta_{out}}{\theta_{in}}$, we determined the values K_I, K_p, J_I, J_p and Q_I, that are used in the control blocks. The values that are used to control the behaviour of Rocky are:

Variable	Value	Description
K_I	-88	Proportional term weight for PI angle controller
K_p	-100	Integral term weight for PI angle controller
J_I	70	Proportional term weight for PI velocity controller
J_p	8	Integral term weight for PI velocity controller
Q_I	-.3	Weight for velocity integrator (net distance)

Constants calculated based on our model of Rocky are:

Constant	Value	Description
a	14	Experimentally determined motor parameter
b	1/400	Experimentally determined motor parameter
l	.0942	Effective length of Rocky(distance between axle and COM)
g	9.8	Gravity

The first step in determining these parameters was to determine the parameters for the cruise control subsystem, G_b. Since G_b is pair of blocks with one input and one output, the poles of the transfer function of this subsystem are independent of the rest of the system. Therefore we can analyze this subsystem independently, and derive an equation with two roots and then solve for a relationship between the two unknowns, J_I and J_p. The transfer function for G_b, with all functions subbed in is:

$$\frac{V_{act}}{V_{des}} = \frac{ab(J_I + J_p s)}{s^2 + (a + abJ_p)s + abJ_I}$$

The denominator of this equation can be used to find the poles of this transfer function, which can tell us a lot about the behavior of this system, such as whether it is stable or unstable, and what type of damping the system has. The roots of the denominator are:

$$s = \frac{-a - abJ_p \pm \sqrt{(a + abJ_p)^2 - 4abJ_I}}{2}$$

Now we know that the system is critically damped when it has two identical, real roots. These conditions are met when the discriminant is zero, thus

$$\begin{aligned} 0 &= (a + abJ_p)^2 - 4abJ_I \\ 4abJ_I &= (a + abJ_p)^2 \\ J_I &= \frac{(a + abJ_p)^2}{4ab} \end{aligned}$$

This gives us a relationship between J_I and J_p that lead to a critically damped system. With this relationship between J_I and J_p, J_I can be set to an arbitrary value with a respective J_p, leaving us with an overall transfer function that has only three unknowns.

To determine the remaining unknowns, we repeated this process of seeking poles by solving for the roots of the denominator but this time of the overall transfer function, $\frac{\theta_{out}}{\theta_{des}}$. This is achieved by substituting the known G equations into the overall transfer function derived in section 1,

$$\frac{\theta_{out}}{\theta_{in}} = \frac{G_{AC}G_bG_R}{1 + G_{AC}G_bG_R - G_{AC}G_b\frac{Qi}{s}}$$

When solving for the zeroes of the denominator of this expression, we are left with six distinct roots, and so algebraically solving for a critically damped system is extremely impractical. To find values that were both stable and close to being critically damped, we implemented iterative testing in both Mathematica and onboard Rocky itself.

The first step in this process was to use our predetermined J_I and J_p and make logical guesses for K_I, K_p, and Q_I to get numerical values for the poles. Once we had this initial ballpark estimate, we continually adjusted those values until we reached a state of all negative poles, which is a known property of a stable system.

Once we had a full set of J/K values, we implemented them into Rocky and observed the actual performance of the robot. If we were sufficiently close to a stable system, we tuned the values to improve performance, all while checking that the poles remained stable using Mathematica. As you can see in Figure 4, all of the poles are negative, leaving us with a stable system.

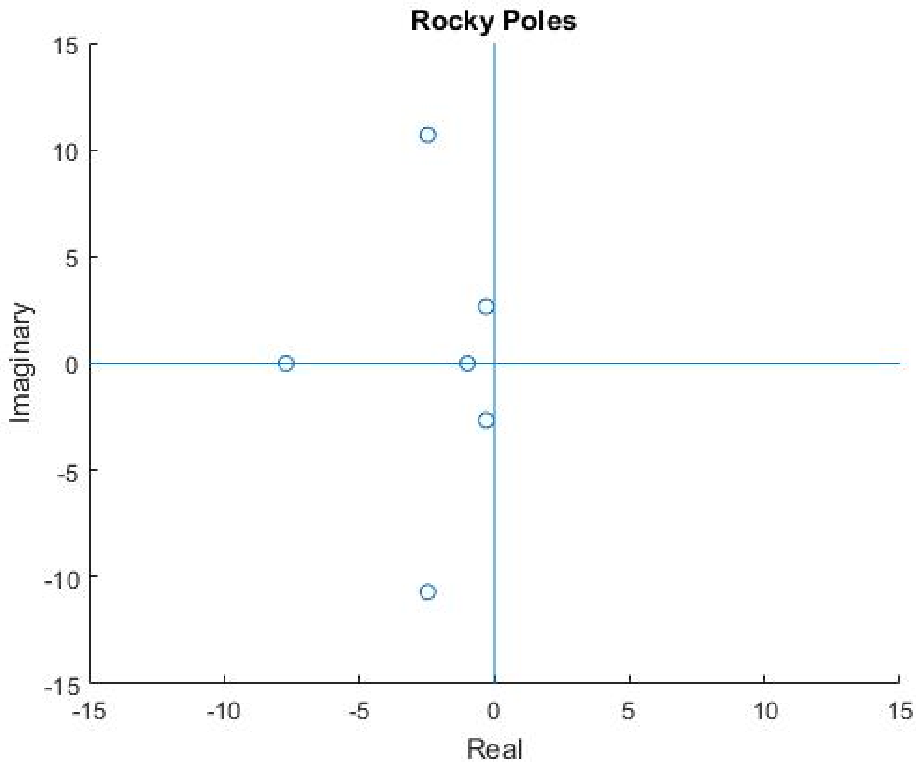

Figure 4: This is a graph of the poles used to make Rocky stand still.

3 Training Session Description

As explained at the end of section 2, Rocky's training sessions consisted of iterative guess-and-check using a combination of quantitative analysis in Mathematica and qualitative analysis made by observing the actual behavior of the Rocky robot system.

Once we had a robot that would stand up, we then moved to implement a velocity controller, to reduce the runaway found in the robot. This can be observed in our velocity control block in Figure 1. With the velocity controller in place, Rocky would now stay upright, but would drift significantly over time. To fix this issue we implemented the position control term, as can also be seen in Figure 1. At this point we had all of the mathematics and Arduino implementation to have Rocky complete the survivor challenge, however we got stuck in trying to tune Rocky to actually follow this control algorithm, and not violently oscillate out of control.

To determine the optimal set of values for our robot, we began with a set of values that we knew were stable in Mathematica,and implemented those values on the robot, and then observed how the robot reacted to disturbances and how quickly it came to a (relatively) stable balancing position, or how quickly it failed, if it didn't become stable. By solving for the J values in advance, the state space was reduced to three unknown variables, K_p, K_I, and Q_I. Given only these three control variables, the behavior of the robot's failure mode was fairly easy to classify into adjusting one of those variables:

1. Increase K_p: the robot fails to react quickly enough to a widening angle
2. Decrease K_p: the robot is "jerky" and responds too aggressively to a widening angle
3. Increase K_I: the robot badly overshoots a stable position
4. Decrease K_I: the robot attempts to respond to a widening angle but too sluggishly
5. Increase Q_I: The robot is running away
6. Decrease Q_I: The robot prioritizes staying in one position more than balancing

We would feed in an initial set of values that were stable in Mathematica and then incrementally change each parameter based on the rules above, logging each set of changes that we made in an Excel spreadsheet to track how the parameters changed over time. An excerpt of one of these spreadsheets is given in Figure **??**.

	A	B	C	D	E	F
1	Ki	Kp	Ji	Jp	Qi	Notes
2	-5000	-1350	-1500	-220		Feels like too strong of a response; the robot has narrow oscillations and stands, but drifts badly and occasionally hits max motor speed
3	-4800	-1200	-1000	-180		Still like too strong of a response; the robot has narrow oscillations and stands, but drifts badly and occasionally hits max motor speed
4	-4100	-720	-600	-100		Probably too weak of a response to theta; the robot has wide oscillations and falls over pretty quickly
5	-3800	-700	-600	-100	200	I'm dumb, there's definitely too weak of a kp response
6	-4000	-1000	-600	-200	200	Ooh it stands up. There's still a lot of drift, so we should increase Qi
7	-4000	-1000	-600	-200	700	Rocky won't stand up, probably too big
8	-4000	-1000	-600	-200	400	Still drifting, probably need to increase Qi again?
9	-4000	-1000	-600	-200	550	Hey, it tracks back and forth, but seems like unstable oscillations and falls. I'll try lowering Qi a little
10	-4000	-1000	-600	-200	500	Still oscillating and falling
11						Paul came in and showed us some values that are way off of our orders of magnitude. Here are some first tries at new ones
12	-15	15	-4	5	-0.2	Robot's not responding at all
13	-4000	-600	-450	-120	-0.1	This is rocking back and forth. There's still a bunch of drift, but it's rocking, at least
14	-4000	-600	-400	-120	-0.15	Robot's falling really quickly
15	-4000	-600	-450	-120	-0.15	Robot still falls pretty quickly
16	-4000	-600	-450	-120	-0.1	Robot is rocking back and forth staying upright, pretty large amplitude of oscillation but it was holding for a little while there...

Figure 5: An excerpt from one of our variable tuning tables. An initial set of parameters known to be stable in Mathematica is shown in row 1. Each subsequent row shows how the values evolve as a response to undesirable robot behavior (and continued to evolve well beyond what this image shows).

This method of trial-and-error incrimination resulted in a remarkably robust control system that could last for a fairly long time (¿3 mins) without drifting significantly from its initial position. This feat met the requirements of a professor-prescribed challenge dubbed "Survivor," and so Survivor is the first Olympic event our Rocky competes in. Video of Rocky achieving this 3 minute target is available online here.

An interesting attribute of our rather unique control parameters is that our system can recover from remarkably large disturbances from the outside (e.g. throwing a post-it pad at Rocky, kicking it gently, etc.). We have dubbed this "Trial by Ordeal" and present it as its own independent category with video available here.

A further interesting side effect of having a large Q_I value is that Rocky strongly wants to return to its initial position, even when it is pulled out to some initial displacement; to this end, Rocky behaves much like a damped spring-mass system, gradually returning to "equilibrium" (its starting position) with damped oscillations after being released from some initial displacement from that "equilibrium position." We have dubbed this "Physics Simulation" and present it as its own independent category with video available here.

Furthermore, by having such a robust system, we were able to make significant physical alterations to Rocky (i.e. attaching a phone and torch), allowing Rocky to carry the Olympic torch while streaming John Williams' famed Olympic March. We have dubbed this "Carrying the Torch" and present it as its own independent category with video available here.

With the survivor challenge under our belt we then moved on to attempt the sprint. Our algorithm for attempting the sprint was to measure Rocky's current position (from its initial position), increment a desired position variable to some displacement (e.g. 2cm), and then adjust Rocky's desired angle as a proportion of how far it is away from the new desired position. This is achieved with the help of a new magnitude-of-displacement coefficient, m_d, implemented by the expression

$$\theta_{des} = m_d(d_{desired} - d_{traveled})$$

where $d_{desired}$ is the (accumulating) global desired distance for the robot, $d_{traveled}$ is the (accumulating) total distance traveled by the robot, and m_d is a proportional coefficient with an experimentally derived value of 0.005.

When Rocky reaches the desired distance, $d_{desired} = d_{traveled}$, and so the desired angle equals zero. When the on-board computer detects that this desired distance is zero, it increments Rocky's $d_{desired}$ term and the cycle repeats.

This algorithm seems to work fairly well, but due to something strange in our implementation, the robot consistently fails at almost exactly 4 seconds of operation. Changing threshold values and parameters fails to mitigate the problem, and the serial data being offloaded by the robot does not have any glaring anomalies in it, so the issue remains mysterious. Professor Jeff Dusek observed the error being consistent for nearly an hour and eventually recommended that we submit our algorithm and implementation as-is in a separate Olympic category of "The 4-second Flop." We accepted this suggestion and present video of this performance online here.

4 Results Time Discussion

In summary, we have an algorithm that completes the "Rocky Stand Still!" challenge, as well as the individual categories of "Trial by Ordeal" and "Carrying the Torch", which demonstrate the robustness of our algorithm. We also have another self created event called, "The 4-second Flop", in which all of the mathematics and controls theoretically necessary to complete the full Sprint were developed, but after absurdly copious amounts of time of spent testing values, Rocky is still hindered at 4 seconds by what we presume is an implementation error. We decided to stop after reaching a point at which we deemed that we were no longer learning a proportional amount of useful information for the amount of time poured into the project. This decision was validated by Professor Jeff Dusek.

A Appendix A

A.1 Standing (Survior) Code

```
// This code should help you get started with your balancing robot.
// The code performs the following steps
// Calibration phase:
// In this phase the robot should be stationary lying on the ground.
```

```
// The code will record the gyro data for a couple of seconds to zero
// out any gyro drift.
//
// The robot has a hardcoded angle offset between the lying down and the
// standing up configuration. This offset can be modified in Balance.cpp around the lines below:
//
// // this is based on coarse measurement of what I think the angle would be resting on the flat surface.
// // this corresponds to 94.8 degrees
// angle = 94827-6000;
//
// Waiting phase:
// The robot will now start to integrate the gyro over time to estimate
// the angle. Once the angle gets within +/- 3 degrees of vertical,
// we transition into the armed phase. A buzzer will sound to indicate
// this transition.
//
// Armed phase:
// The robot is ready to go, however, it will not start executing its control
// loop until the angle leaves the region of [-3 degrees, 3 degrees]. This
// allows you to let go of your robot, and it won't start moving until it's started
// to fall just a little bit. Once it leaves the region around vertical, it enters
// the controlled phase.
//
// Controlled phase:
// Here you can implement your control logic to do your balancing (or any of the
// other Olympic events.


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

#define METERS_PER_CLICK 3.141592*80.0*(1/1000.0)/12.0/(162.5)
#define MOTOR_MAX 300
#define MAX_SPEED 0.75 // m/s
#define FORTY_FIVE_DEGREES_IN_RADIANS 0.78

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 vL, vR, totalDistanceLeft, totalDistanceRight;
float leftMotorPWM = 0;
float rightMotorPWM = 0;

void balanceDoDriveTicks();

extern int32_t displacement;
int32_t prev_displacement=0;

LSM6 imu;
Balboa32U4Motors motors;
Balboa32U4Encoders encoders;
Balboa32U4Buzzer buzzer;
```

```
Balboa32U4ButtonA buttonA;

// Instantiate values for all terms that we want to accumulate over
// time (i.e. collect the integrals of)
float fLaccum = 0.0;
float fRaccum = 0.0;
float eLaccum = 0.0;
float eRaccum = 0.0;
float vLoutaccum = 0.0;
float vRoutaccum = 0.0;

// Instantiate our control parameters
float ki = -88; // Integral angle controller
float kp = -100; // Differential angle controller
float ji = 70; // Integral velocity controller
float jp = 8; // Differential velocity controller
float qi = -.3; // Integral of velocity (position)
float theta_des = 0; // Desired angle (default to zero)

void updatePWMs(float totalDistanceLeft, float totalDistanceRight, float vL, float vR, float angleRad, float angleR

   // Calculate angle error in terms of theta_des, angleRad, and the average of the total distances
   float eL = theta_des-angleRad+(qi*(totalDistanceLeft+totalDistanceRight)/2);
   // Calculate the desired velocity for the left wheel
   float vtargetL = kp*eL+ki*eLaccum;
   // Calculate the error between desired and actual velocities
   float fL = vtargetL-vL;

   // Repeat the aforementioned steps for the right wheel
   float eR = eL;//theta_des-angleRad+(qi*(totalDistanceLeft+totalDistanceRight)/2);
   float vtargetR = kp*eR+ki*eRaccum;
   float fR = vtargetR-vR;

  // Add to the integrals of all terms we want to accumulate over time by adding the
  // value of each term times the duration of the current time step
   fLaccum += fL*dt;
   eLaccum += eL*dt;
   vLoutaccum += vL*dt;
   fRaccum += fR*dt;
   eRaccum += eR*dt;
   vRoutaccum += vR*dt;

   // Calculate the final PWM values for each wheel
   leftMotorPWM = jp*fL+ji*fLaccum;
   rightMotorPWM = jp*fR+ji*fRaccum;
}

uint32_t prev_time;

void setup()
{
  Wire.begin();

  Serial.begin(9600);
  Serial1.begin(9600);

  prev_time = 0;
  ledYellow(0);
  ledRed(1);
```

```
  balanceSetup();
  ledRed(0);
  ledGreen(0);
  ledYellow(0);
}

extern int16_t angle_prev;
int16_t start_flag = 0;
int16_t armed_flag = 0;
int16_t start_counter = 0;
void lyingDown();
extern bool isBalancingStatus;
extern bool balanceUpdateDelayedStatus;

void newBalanceUpdate()
{
  static uint32_t lastMillis;
  uint32_t ms = millis();

  if ((uint32_t)(ms - lastMillis) < UPDATE_TIME_MS) { return; }
  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 loop()
{
  uint32_t cur_time = 0;
  static uint32_t prev_print_time = 0; // this variable is to control how often we print on the serial monitor
  static float angle_rad; // this is the angle in radians
  static float angle_rad_accum = 0; // this is the accumulated angle in radians
  static float error_ = 0; // this is the accumulated velocity error in m/s
  static float error_left_accum = 0; // this is the accumulated velocity error in m/s
  static float error_right_accum = 0; // this is the accumulated velocity error in m/s

  cur_time = millis(); // get the current time in miliseconds



  newBalanceUpdate(); // run the sensor updates. this function checks if it has been 10 ms since the previous

  if(angle > 3000 || angle < -3000) // If angle is not within +- 3 degrees, reset counter that waits for start
  {
    start_counter = 0;
  }

```

```
  bool shouldPrint = cur_time - prev_print_time > 105;
  if(shouldPrint) // do the printing every 105 ms. Don't want to do it for an integer multiple of 10ms to not hog t
  {
        Serial.print(angle_rad);
        Serial.print("\t");
        Serial.print(angle_rad_accum);
        Serial.print("\t");
        Serial.print(leftMotorPWM);
        Serial.print("\t");
        Serial.print(rightMotorPWM);
        Serial.print("\t");
        Serial.print(vL);
        Serial.print("\t");
        Serial.print(vR);
        Serial.print("\t");
        Serial.print(totalDistanceLeft);
        Serial.print("\t");
        Serial.println(totalDistanceRight);

        prev_print_time = cur_time;
/* Uncomment this and comment the above if doing wireless
        Serial1.print(angle_rad);
        Serial1.print("\t");
        Serial1.print(angle_rad_accum);
        Serial1.print("\t");
        Serial1.print(PWM_left);
        Serial1.print("\t");
        Serial1.print(PWM_right);
        Serial1.print("\t");
        Serial1.print(vL);
        Serial1.print("\t");
        Serial1.println(vR);
       */
  }

  float delta_t = (cur_time - prev_time)/1000.0;

  // handle the case where this is the first time through the loop
  if (prev_time == 0) {
    delta_t = 0.01;
  }

  // every UPDATE_TIME_MS, check if angle is within +- 3 degrees and we haven't set the start flag yet
  if(cur_time - prev_time > UPDATE_TIME_MS && angle > -3000 && angle < 3000 && !armed_flag)
  {
    // increment the start counter
    start_counter++;
    // If the start counter is greater than 30, this means that the angle has been within +- 3 degrees for 0.3 seco
    if(start_counter > 30)
    {
      armed_flag = 1;
      buzzer.playFrequency(DIV_BY_10 | 445, 1000, 15);
    }
  }

  // angle is in millidegrees, convert it to radians and subtract the desired theta
  angle_rad = ((float)angle)/1000/180*3.14159;

  // only start when the angle falls outside of the 3.0 degree band around 0. This allows you to let go of the
```

```
  // robot before it starts balancing
  if(cur_time - prev_time > UPDATE_TIME_MS && (angle < -3000 || angle > 3000) && armed_flag)
  {
    start_flag = 1;
    armed_flag = 0;
    angle_rad_accum = 0.0;
    fLaccum = 0.0;
    fRaccum = 0.0;
    eLaccum = 0.0;
    eRaccum = 0.0;
    vLoutaccum = 0.0;
    vRoutaccum = 0.0;
  }

  // every UPDATE_TIME_MS, if the start_flag has been set, do the balancing
  if(cur_time - prev_time > UPDATE_TIME_MS && start_flag)
  {
    // set the previous time to the current time for the next run through the loop
    prev_time = cur_time;

    // speedLeft and speedRight are just the change in the encoder readings
    // wee need to do some math to get them into m/s
    vL = METERS_PER_CLICK*speedLeft/delta_t;
    vR = METERS_PER_CLICK*speedRight/delta_t;

    totalDistanceLeft = METERS_PER_CLICK*distanceLeft;
    totalDistanceRight = METERS_PER_CLICK*distanceRight;
    angle_rad_accum += angle_rad*delta_t;

    updatePWMs(totalDistanceLeft, totalDistanceRight, vL, vR, angle_rad, angle_rad_accum, delta_t);

    // if the robot is more than 45 degrees, shut down the motor
    if(start_flag && fabs(angle_rad) > FORTY_FIVE_DEGREES_IN_RADIANS)
    {
      // reset the accumulated errors here
      start_flag = 0; /// wait for restart
      prev_time = 0;
      motors.setSpeeds(0, 0);
    } else if(start_flag) {
      motors.setSpeeds((int)leftMotorPWM, (int)rightMotorPWM);
    }
  }

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

A.2 Running (Sprint) Code

```
// This code should help you get started with your balancing robot.
// The code performs the following steps
// Calibration phase:
// In this phase the robot should be stationary lying on the ground.
// The code will record the gyro data for a couple of seconds to zero
// out any gyro drift.
```

```
//
// The robot has a hardcoded angle offset between the lying down and the
// standing up configuration. This offset can be modified in Balance.cpp around the lines below:
//
// // this is based on coarse measurement of what I think the angle would be resting on the flat surface.
// // this corresponds to 94.8 degrees
// angle = 94827-6000;
//
// Waiting phase:
// The robot will now start to integrate the gyro over time to estimate
// the angle. Once the angle gets within +/- 3 degrees of vertical,
// we transition into the armed phase. A buzzer will sound to indicate
// this transition.
//
// Armed phase:
// The robot is ready to go, however, it will not start executing its control
// loop until the angle leaves the region of [-3 degrees, 3 degrees]. This
// allows you to let go of your robot, and it won't start moving until it's started
// to fall just a little bit. Once it leaves the region around vertical, it enters
// the controlled phase.
//
// Controlled phase:
// Here you can implement your control logic to do your balancing (or any of the
// other Olympic events.


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

#define METERS_PER_CLICK 3.141592*80.0*(1/1000.0)/12.0/(162.5)
#define MOTOR_MAX 300
#define MAX_SPEED 0.75 // m/s
#define FORTY_FIVE_DEGREES_IN_RADIANS 0.78

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 vL, vR, totalDistanceLeft, totalDistanceRight;
float leftMotorPWM = 0;
float rightMotorPWM = 0;
float imu_ax_average = 0.0;
float alpha_imu_ax = 0.1;

void balanceDoDriveTicks();

extern int32_t displacement;
int32_t prev_displacement = 0;

LSM6 imu;
Balboa32U4Motors motors;
Balboa32U4Encoders encoders;
Balboa32U4Buzzer buzzer;
```

```
Balboa32U4ButtonA buttonA;

// Instantiate values for all terms that we want to accumulate over
// time (i.e. collect the integrals of)
float fLaccum = 0.0;
float fRaccum = 0.0;
float eLaccum = 0.0;
float eRaccum = 0.0;
float vLoutaccum = 0.0;
float vRoutaccum = 0.0;

// Instantiate our control parameters
float ki = -88;
float kp = -100;
float ji = 70;
float jp = 8;
float qi = -.3;
float theta_des = 0;
float md = 0.005;
float dDesired = 0;

void updatePWMs(float realDistanceLeft, float realDistanceRight, float vL, float vR, float angleRad, float angleRad

  // Define local total distance variables that look at the difference between the
  // system total distance (realDistanceL/R) and the accumulated distance that we
  // don't care about anymore (vL/Routaccum)
  float totalDistanceLeft = realDistanceLeft - vLoutaccum;
  float totalDistanceRight = realDistanceRight - vRoutaccum;

  // Calculate a theta_des value that is some proportion of how far
  // we currently are from the target distance
  theta_des = md * (dDesired - (totalDistanceLeft + totalDistanceRight) / 2);

  // If theta_des is less than zero, we've overshot the target distance
  // and so can increment the step we want to be at, dDesired
  if (theta_des <= 0) {
    dDesired = 0.1;

    // Also we want to increment our voutaccum terms so that the position
    // check term doesn't try to pull us backwards
    vLoutaccum = totalDistanceLeft;
    vRoutaccum = totalDistanceLeft;

    // Next we want to reset any variables that are integrals that are
    // needed in order to move forward to the next waypoint
    eLaccum = 0.0;
    eRaccum = 0.0;
    fLaccum = 0.0;
    fRaccum = 0.0;

    // And finally we re-calculate theta_des with these new values in mind
    theta_des = md * ((totalDistanceLeft + totalDistanceRight) / 2 - dDesired);
  }


  // Calculate angle error in terms of theta_des, angleRad, and the average of the total distances
  float eL = theta_des - angleRad + (qi * (totalDistanceLeft + totalDistanceRight) / 2);
  // Calculate the desired velocity for the left wheel
  float vtargetL = kp * eL + ki * eLaccum;
```

```
  // Calculate the error between desired and actual velocities
  float fL = vtargetL - vL;

  // Repeat the aforementioned steps for the right wheel
  float eR = eL;//theta_des-angleRad+(qi*(totalDistanceLeft+totalDistanceRight)/2);
  float vtargetR = kp * eR + ki * eRaccum;
  float fR = vtargetR - vR;

  // Add to the integrals of all terms we want to accumulate over time by adding the
  // value of each term times the duration of the current time step
  fLaccum += fL * dt;
  eLaccum += eL * dt;
  vLoutaccum += vL * dt;
  fRaccum += fR * dt;
  eRaccum += eR * dt;
  vRoutaccum += vR * dt;

  // Calculate the final PWM values for each wheel
  leftMotorPWM = jp * fL + ji * fLaccum + 10;
  rightMotorPWM = jp * fR + ji * fRaccum;// + 12; //leftMotorPWM;;

  // Perform a check to verify that we're not asking for a PWM
  // outside the -300 to +300 range that the robto can handle
  if (leftMotorPWM < -300) {
    leftMotorPWM = -300;
  }
  else if (leftMotorPWM > 300) {
    leftMotorPWM = 300;
  }
  if (rightMotorPWM < -300) {
    rightMotorPWM = -300;
  }
  else if (rightMotorPWM > 300) {
    rightMotorPWM = 300;
  }
}

uint32_t prev_time;

void setup()
{
  Wire.begin();

  Serial1.begin(9600);
  Serial1.begin(9600);

  prev_time = 0;
  ledYellow(0);
  ledRed(1);
  balanceSetup();
  ledRed(0);
  ledGreen(0);
  ledYellow(0);
}

extern int16_t angle_prev;
int16_t start_flag = 0;
int16_t armed_flag = 0;
int16_t start_counter = 0;
```

```
void lyingDown();
extern bool isBalancingStatus;
extern bool balanceUpdateDelayedStatus;

void newBalanceUpdate()
{
  static uint32_t lastMillis;
  uint32_t ms = millis();

  if ((uint32_t)(ms - lastMillis) < UPDATE_TIME_MS) {
    return;
  }
  balanceUpdateDelayedStatus = ms - lastMillis > UPDATE_TIME_MS + 1;
  lastMillis = ms;

  // call functions to integrate encoders and gyros
  balanceUpdateSensors();
  imu_ax_average = alpha_imu_ax * imu.a.x + (1 - alpha_imu_ax) * imu_ax_average;
  if (imu_ax_average < 0)
  {
    lyingDown();
    isBalancingStatus = false;
  }
  else
  {
    isBalancingStatus = true;
  }
}



void loop()
{
  uint32_t cur_time = 0;
  static uint32_t prev_print_time = 0; // this variable is to control how often we print on the serial monitor
  static float angle_rad; // this is the angle in radians
  static float angle_rad_accum = 0; // this is the accumulated angle in radians
  static float error_ = 0; // this is the accumulated velocity error in m/s
  static float error_left_accum = 0; // this is the accumulated velocity error in m/s
  static float error_right_accum = 0; // this is the accumulated velocity error in m/s

  cur_time = millis(); // get the current time in miliseconds



  newBalanceUpdate(); // run the sensor updates. this function checks if it has been 10 ms since the previous

  if (angle > 3000 || angle < -3000) // If angle is not within +- 3 degrees, reset counter that waits for start
  {
    start_counter = 0;
  }

  bool shouldPrint = cur_time - prev_print_time > 105;
  if (shouldPrint) // do the printing every 105 ms. Don't want to do it for an integer multiple of 10ms to not hog
  {
    Serial1.print(angle_rad);
    Serial1.print("\t");
    Serial1.print(angle_rad_accum);
    Serial1.print("\t");
```

```
    Serial1.print(leftMotorPWM);
    Serial1.print("\t");
    Serial1.print(rightMotorPWM);
    Serial1.print("\t");
    Serial1.print(vL);
    Serial1.print("\t");
    Serial1.print(vR);
    Serial1.print("\t");
    Serial1.print(totalDistanceLeft);
    Serial1.print("\t");
    Serial1.print(totalDistanceRight);
    Serial1.print("\t");
    Serial1.println(dDesired);

    prev_print_time = cur_time;
    /* Uncomment this and comment the above if doing wireless
            Serial1.print(angle_rad);
            Serial1.print("\t");
            Serial1.print(angle_rad_accum);
            Serial1.print("\t");
            Serial1.print(PWM_left);
            Serial1.print("\t");
            Serial1.print(PWM_right);
            Serial1.print("\t");
            Serial1.print(vL);
            Serial1.print("\t");
            Serial1.println(vR);
    */
  }

  float delta_t = (cur_time - prev_time) / 1000.0;

  // handle the case where this is the first time through the loop
  if (prev_time == 0) {
    delta_t = 0.01;
  }

  // every UPDATE_TIME_MS, check if angle is within +- 3 degrees and we haven't set the start flag yet
  if (cur_time - prev_time > UPDATE_TIME_MS && angle > -3000 && angle < 3000 && !armed_flag)
  {
    // increment the start counter
    start_counter++;
    // If the start counter is greater than 30, this means that the angle has been within +- 3 degrees for 0.3 seco
    if (start_counter > 30)
    {
      armed_flag = 1;
      buzzer.playFrequency(DIV_BY_10 | 445, 1000, 15);
    }
  }

  // angle is in millidegrees, convert it to radians and subtract the desired theta
  angle_rad = ((float)angle) / 1000 / 180 * 3.14159;

  // only start when the angle falls outside of the 3.0 degree band around 0. This allows you to let go of the
  // robot before it starts balancing
  if (cur_time - prev_time > UPDATE_TIME_MS && (angle < -3000 || angle > 3000) && armed_flag)
  {
    start_flag = 1;
    armed_flag = 0;
```

```
    angle_rad_accum = 0.0;
    fLaccum = 0.0;
    fRaccum = 0.0;
    eLaccum = 0.0;
    eRaccum = 0.0;
    vLoutaccum = 0.0;
    vRoutaccum = 0.0;
    theta_des = 0.0;
    dDesired = 0.0;
  }

  // every UPDATE_TIME_MS, if the start_flag has been set, do the balancing
  if (cur_time - prev_time > UPDATE_TIME_MS && start_flag)
  {
    // set the previous time to the current time for the next run through the loop
    prev_time = cur_time;

    // speedLeft and speedRight are just the change in the encoder readings
    // wee need to do some math to get them into m/s
    vL = METERS_PER_CLICK * speedLeft / delta_t;
    vR = METERS_PER_CLICK * speedRight / delta_t;

    totalDistanceLeft = METERS_PER_CLICK * distanceLeft;
    totalDistanceRight = METERS_PER_CLICK * distanceRight;
    angle_rad_accum += angle_rad * delta_t;

    updatePWMs(totalDistanceLeft, totalDistanceRight, vL, vR, angle_rad, angle_rad_accum, delta_t);

    // if the robot is more than 45 degrees, shut down the motor
    if (start_flag && fabs(angle_rad) > FORTY_FIVE_DEGREES_IN_RADIANS)
    {
      // reset the accumulated errors here
      start_flag = 0; /// wait for restart
      prev_time = 0;
      motors.setSpeeds(0, 0);
    } else if (start_flag) {
      motors.setSpeeds((int)leftMotorPWM, (int)rightMotorPWM);
    }
  }

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