Edinburgh Napier
University

Advanced Control

MEC10104

Pololu Inverted Pendulum Robot Control.

Olivier Chaligne - 40292302.

November 2019.

Abstract:

This report explains the steps taken to design two system controls for a Pololu inverted pendulum robot.

It describes the various mechanical and electrical components useful to the robot balancing applications.

It offers theoretical background into PID and Multivariable controllers with closed-loop diagrams examples of implementation.

Two coded programmes are given accompanied by explanations and comments.

Finally, the system response analysis demonstrates how the basic Proportional Derivative controller was implemented to control the robot motors using the angle change to realise the balancing motion. Then, on the basis of the first controller and system behaviour observation, how a further two controllers (P and PD) were implemented to keep the robot going into a straight line but also reversing back to point of origin of its displacement.

Table of Content

Introduction.

As “Industry 4.0” dawns on us, with even more possibilities of automating processes with more complex methods, it is important to understand how current control methods are designed and how they fit within current solutions. The subject of this study is the applications of two controller methods on a Pololu Balboa robot. Although the purpose/function of the robot can be seen as menial it offers complex enough dynamics to justify the use of complex controlling methods. The purpose of the robot is to balance around angle and mitigate the effects of any disturbances.
This report will describe the components available within the Pololu Balboa kit as well as their use in the application. Then, theoretical descriptions accompanied by closed-loop diagrams of the selected controller methods will be given. And then, the report will describe the steps taken to design and apply the two controlling methods. Finally, an evaluation and comparison of both controllers will be realised.

I. Description of System Components.

This section will describe the components constituting the Pololu Balboa robot. Special attention will be given to develop on the interesting features and usefulness for the required application (in Appendix A)

A. Mechanical Components.

Although limited, the mechanical components are an integral part in the functionality of the robot. The mechanical components are listed below:

- Chassis.
- Wheels and gears system.
- Bumper cage system.

B. Electrical & Electronics Components.

This section will describe the electrical and electronics components allowing to interface with, programme, monitor and control the robot. It will also introduce the electro-mechanical components allowing the robot to move and position itself.
All components are mounted on the control PCB allowing for ease of manufacturing, space optimisation and rapid communication between the components.

The Electrical and Electronics components are listed below:

- ATmega32U4 AVR microcontroller
- USB Micro-B connector
- LEDs
- push buttons
- Buzzer
- LCD header
- Texas Instruments DRV8838 motors
- quadrature coders
- ST LIS3MDL a 3-axis magnetometer
- ST LSM6DS33 combining an accelerometer and gyroscope
- Power system.

II. Controller Types and Closed-Loop Diagrams.

This section will present the two controller methods selected for the inverted pendulum robot application. First it will introduce the theory aspects of the controllers and then relate them to the robot system and application.

A. PID Controller.

PID Controllers variations are among the most used closed-loop control methods for industrial processes. It is estimated that more than half of industrial controllers use a PID control system (Ogata,1997, p.669). They function by calculating an error from a defined setpoint and the system output. This error is then used to calculate the new system output aimed at reducing the gap between the setpoint and error. This is achieved through the use of “control laws” (Ellis, 2012, p.5). These laws are defined by a mathematical model based on three parameters: Proportional, Integral and Derivative.

Each parameter is determined using a gain and a calculated variable obtained from the system:

- Proportional: Kp (proportional gain) multiplied by the error (setpoint - system output). This term acts as an amplifier (Ogata, 1997, p.215). Essentially, the aim is to amplify error to improve the reaction time of the system.
- Integral: Ki (integral gain) multiplied by integral error (or cumulative error). The purpose of the integral term is to get rid of the steady state error. (Ogata, 1997, p.219)
- Derivative: Kd (proportional gain) multiplied by rate of change of the error. This term improves the sensitivity of the controller. Additionally, it helps controlling the overshoot created by the proportional term hence it acts as a dampener.(Ogata, 1997, p.225)

As described, each parameter, within the PID controller, complement one another to offer a reactive and robust control allowing to drive the system whilst mitigating for disturbances. (Ellis, 2012, p.5).

Fig. II.A.1 demonstrates where the PID controller would be placed within a closed-loop system.

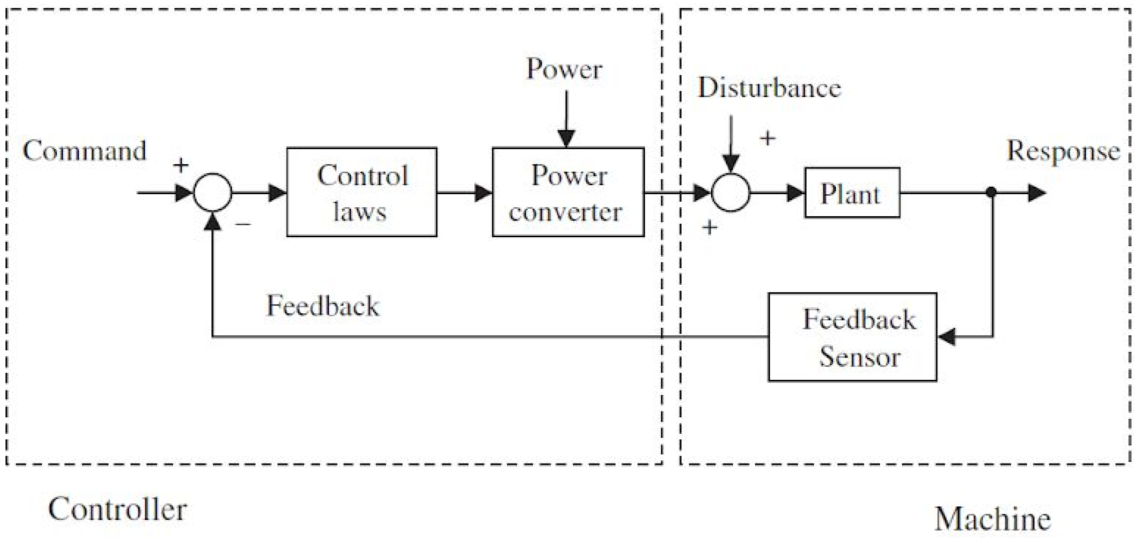

Fig. II.A.1 *The General Control System.* (Ellis, 2012, p.5)

Using the control system diagram, applied to the Pololo robot, command represents the setpoint as the best desired balance angle. The PID controller constitutes the control laws. The power converters in the application are the motor drives converting electrical power to torque. The plant is the robot being moved by the wheels through the motors. The disturbance represents the forces acting on the robot trying to balance. The feedback sensor here is the fusion of the gyroscope and accelerometer which measure the motion of the axis and provide a signal back to the PID to calculate the error.
Fig. II.A.2 offers a diagram representation of the application of a PID controller to the Pololu robot inverted pendulum system.

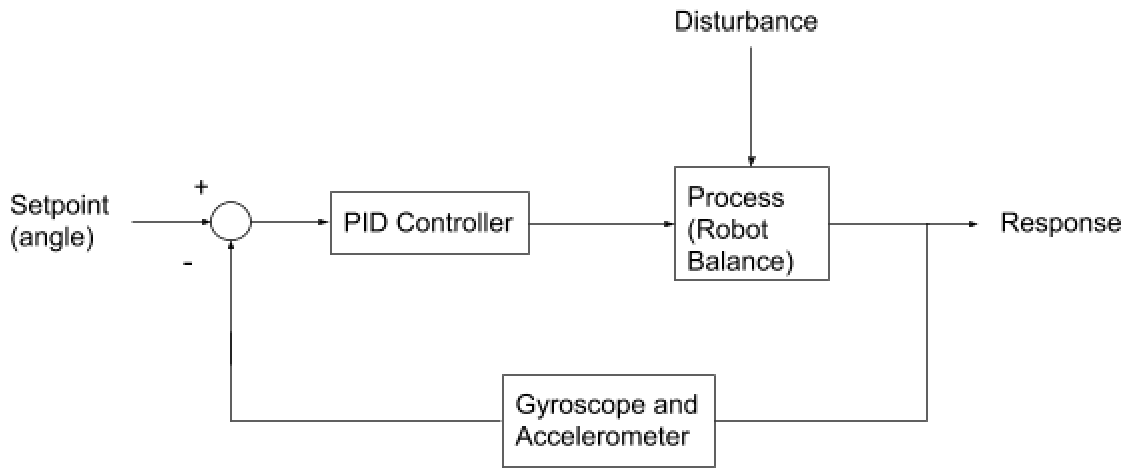

Fig. II.A.2 *PID Controller Pololu System Diagram*.

B. Multi-input/Single Output System: Multi-Controller.

The complexity of systems, like the Pololu and advanced industrial processes, often require several sub-processes to be in control to operate optimally. These sub-processes are monitored by sensor, hence the system can then be controlled by different controllers each with different setpoints with the output errors being summed up and fed to the system output. Seborg (1989) qualifies these systems as multivariable control in which all errors contribute to the overall control strategy.
In the case of the Pololu, the angle controller causes the robot to drift naturally. This is due to the drive action used to correct the angle fall. Therefore, using the wheel encoders we can calculate the displacement error. Finally, a PD controller will calculate the displacement error and output the proportional and derivative gains used to control the drives bringing the robot at the start position. A Proportional controller can also be added to the system to compensate for any drive and gear system being stronger than the other. Fig. II.A.3 demonstrates the application of a multi-controller system. More details will be given about the system application design in the following part.

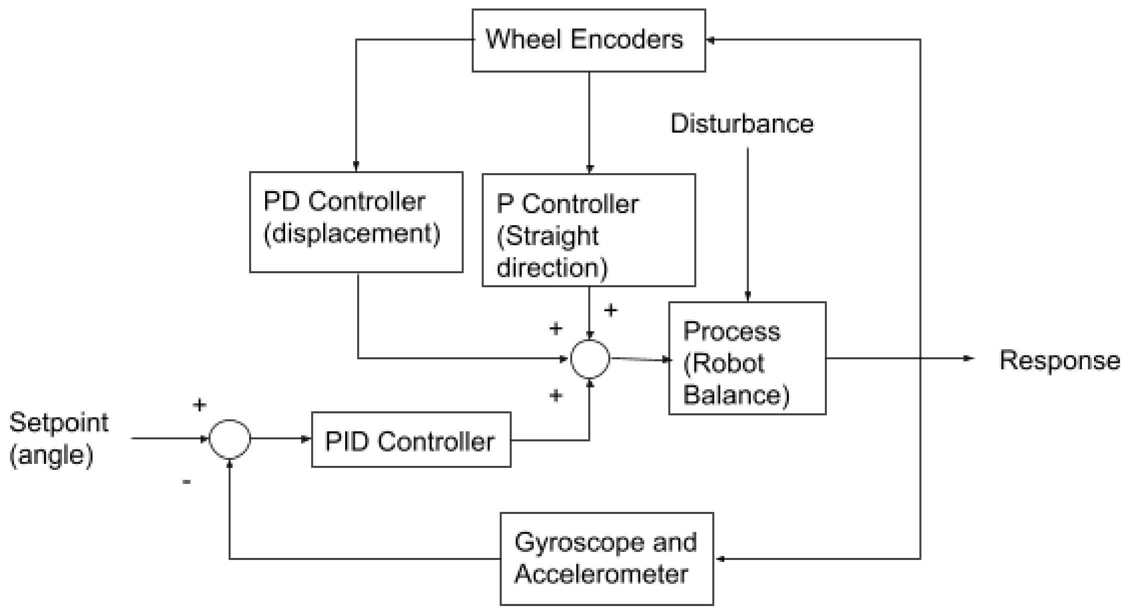

Fig. II.A.3 *Multi-Controller Pololu System Diagram*.

III. Description of Control System Design Process.

This part of the report will summarise the steps taken to develop the final Multi-Control system. The two controller methods code can be found in appendix B.

Step.	**Description, Observations and Comments.**
System backbone and driving motors.	This step involves: ● Including the various libraries required to facilitate the programming work. ● Creating the various variables which will be used to store information and calculations. ● Using "motors.setSpeeds(motorspeed,motorspeed)" to test the motor drives. ● Angle of action was defined between 60 and 135 degrees. Beyond this range the motor speed is set to 0 as a precaution. ● The main loop of the programme is to run at 100Hz. By calling a variable "Current_time" and "previous time" it is possible to measure running time and constrain the loop execution time: "if ((uint16_t)(current_time - previous_time) < 10) { return; }". Observation: The left drive requires a speed value which is negative to go forward and spin the same way as the right wheel.
Sensor signal reading and calibration.	This step involves: ● Creating variables to store and process information relating to the angle signal coming from the LSM6 chip (Gyroscope and Accelerometer). ● Initialising and checking the sensor is functional. ● The gyroscope and accelerometer are then reset to the default to then be configured by setting the registers using the LSM6DS33 datasheet (STMicroelectronics, 2015). - Gyroscope is set to data rate 208Hz (0101) and 1000dps (1000): "imu.writeReg(LSM6::CTRL2_G, 0b01011000);"

	- Accelerometer is set to data rate 208Hz (0101) 16 g and anti aliasing filter at 400Hz (0100): "imu.writeReg(LSM6::CTRL1_XL, 0b01010100);" Observation: Using readings (obtained with "imu-.read()") sent to the serial port it was possible to see that at rest values on the X and Z axis were variable on the accelerometer and gyroscope respectively. It was necessary to correct the readings. The average error was calculated (in the setup phase), over 50 readings, for the Accelerometer and Gyroscope signals. These values were then taken out of each reading to allow for a calibrated value to be obtained.
Angle calculations.	This step involves: ● Calculating the angle in degree using the calibrated accelerometer signals for axis x and z: "Tilting_angle = (atan2(ax,az)) * 180/PI;" ● The angle reading is then compared to the setpoint angle allowing to calculate the error: ● Using the calibrated signal from the gyroscope it is possible to measure the rate of change of the angle. The signal is returned in degrees per second by multiplying the reading by (1/29). This allows a better use of computational resources. The continuous readings are then compounded into one variable.
Proportional (Balance) Control Implementation.	This step involves: ● Creating the variable "Kpa" which will store the proportional gain value. ● The variable "Pa" is used to store the proportional control action: "Pa = Kpa*error;" ● This value is then passed to "output_Ang" the output of the controller. Itself passed to the motor control through "motorspeed"(constrained between -300 and 300, the max and min speed of the motors). Observation: As the angle error increases the motor speed increases until saturation at +/-300. Around the angle setpoint the motor slows and "stops" due to the error value being very small.
Derivative (Balance) Control Implementation.	This step involves: ● Creating the variable to store the rate of change of the angle ("rateOfError" and "angleRate" respectively). ● The rateOfError is then multiplied by the derivative gain "Kda" and stored in the controller variable "Da": "Da =

	Kd*rateOfError;" ● Da is then added to the output of the angle controller.
Integral (Balance) control Implementation.	Observation: The implementation of an integral was considered. Because of the build up in the I- controller and the very small impact this controller had, it was decided not to use this type of control.
PD (Balance) Controller Tuning.	Using the Ziegler-Nichols tuning rules second method, described by Ogata (1997) it was possible to obtain a self-balancing system: First, the proportional gain Kpa is raised until the system self-oscillates around the setpoint. From there, the derivative gain Kda was progressively risen, to help control the reaction time and utilise the dampening effect of the D term of the controller, until the system offers a stable enough reaction.
Proportional (Direction) Control Implementation and Tuning.	From observation of the system running, it is noticeable the robot tends to turn to one side, hence one of the drives is stronger than the other. To keep it running in a straight line a P controller is implemented. The error is calculated using the difference in displacement between the two wheel encoders. This error is then fed to the motor speed. Each wheel has a ratio applied to allow the right additional to keep the robot going straight : "motors.setSpeeds(motorspeed+0.4*countdiff , -motorspeed-0.65*countdiff)"

Proportional (Displacement) Control Implementation.	This step is similar to the implementation of the proportional term (Kpd) for the angle error calculation which has its own proportional gain setting. The setpoint is the starting point on the wheel encoders.
	
	Observation:
	The readings from the encoders being different, the error is calculated from the average displacement reading from each of the encoders:
	"displ_left += leftcounter - prev_leftcounter;
	displ_right += rightcounter - prev_rightcounter;
	displacement = (displ_left + displ_right)/2;".

Derivative (Displacement) Control	This part of the controller calculates the speed of each wheel using difference between the counter and previous counter readings. The system speed is then calculated from the

Implementation.	average of left and right speeds. This value is then multiplied by the derivative gain “Kdd” and fed to the motorspeed variable. Observation: The system speed would then normally be divided by the elapsed time. Though this case, our main loop is constrained to execute every 10ms (100Hz) hence the execution time is constant.
PD (Displacement) Controller Tuning.	This controller was tuned using the same technique as previously described for the angle controller.

IV. Design Evaluation.

This section will describe the system performance through data acquired from the robot whilst balancing. The readings being realised thanks to a USB cable connected to the computer, the setup creates unwanted disturbances on the robot due to the weight of the cable.

A. PD Angle Controller Evaluation.

This controller uses the angle error and angle rate to control the motorspeed of the robot. Fig. IV.A.1 demonstrates the correlation between the change in the angle and controller output.

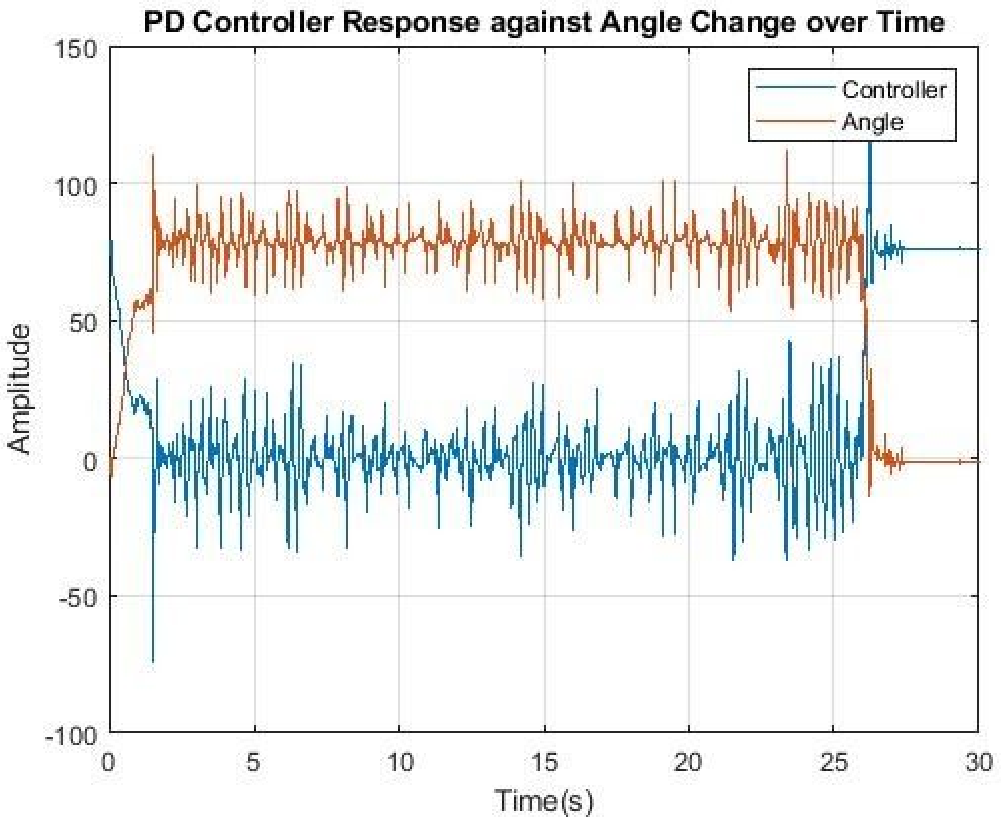

Fig. IV.A.1 *PD Controller Output against Angle change over Time.*

Fig. IV.A.2 shows how the robot control response of the system oscillates around the setpoint at 78.8 degrees.

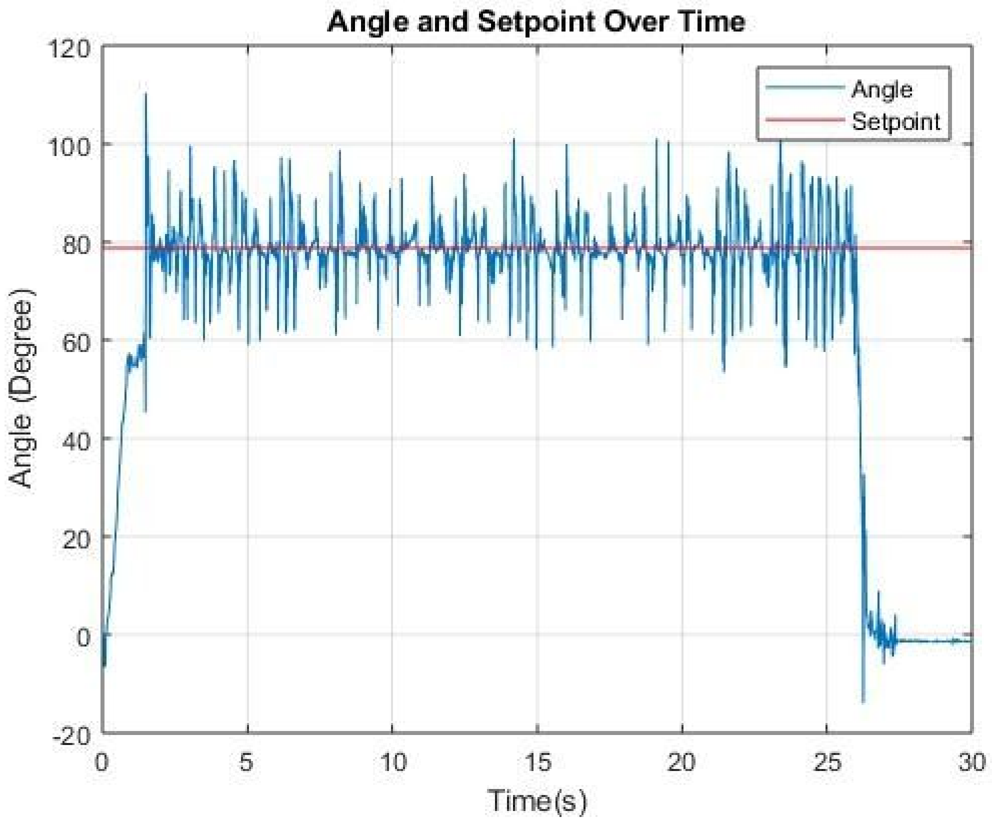

Fig. IV.A.2 *Angle and Setpoint over Time.*

Fig. IV.A.3 shows how as the system approaches the setpoint the motorspeed reduces. It also demonstrates the contribution of each controller to the system control. It is noticeable the peaks of the reaction of the P and D controller are directly inverted demonstrating how the derivative term works to dampen the proportional control.

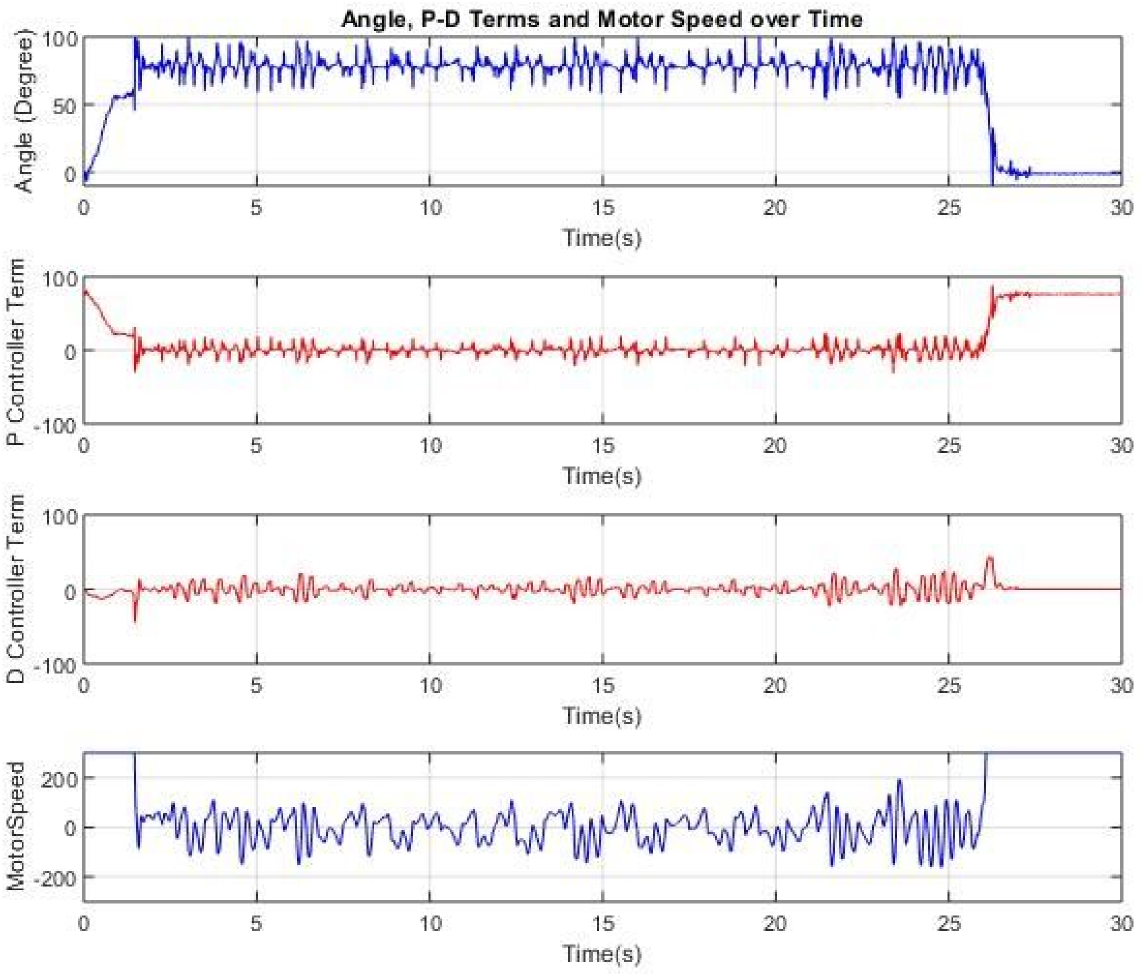

Fig. IV.A.3 *Angle, P and D Controllers and MotorSpeed over Time.*

B. PD-P-PD Multi-Controller Evaluation.

The analysis was realised after the addition of the wheel straightening control and displacement control to the basic angle control system.

Fig. IV.B.1, shows the multicontroller response to the angle and displacement disturbance. Between 2 and 10 seconds the system rises and balances. At 12 seconds the system receives a push. Although the value of the displacement seems disproportionately bigger, this is due to the encoder reading being large numbers. The system actually rapidly works against the push to bring it back to the setpoint.

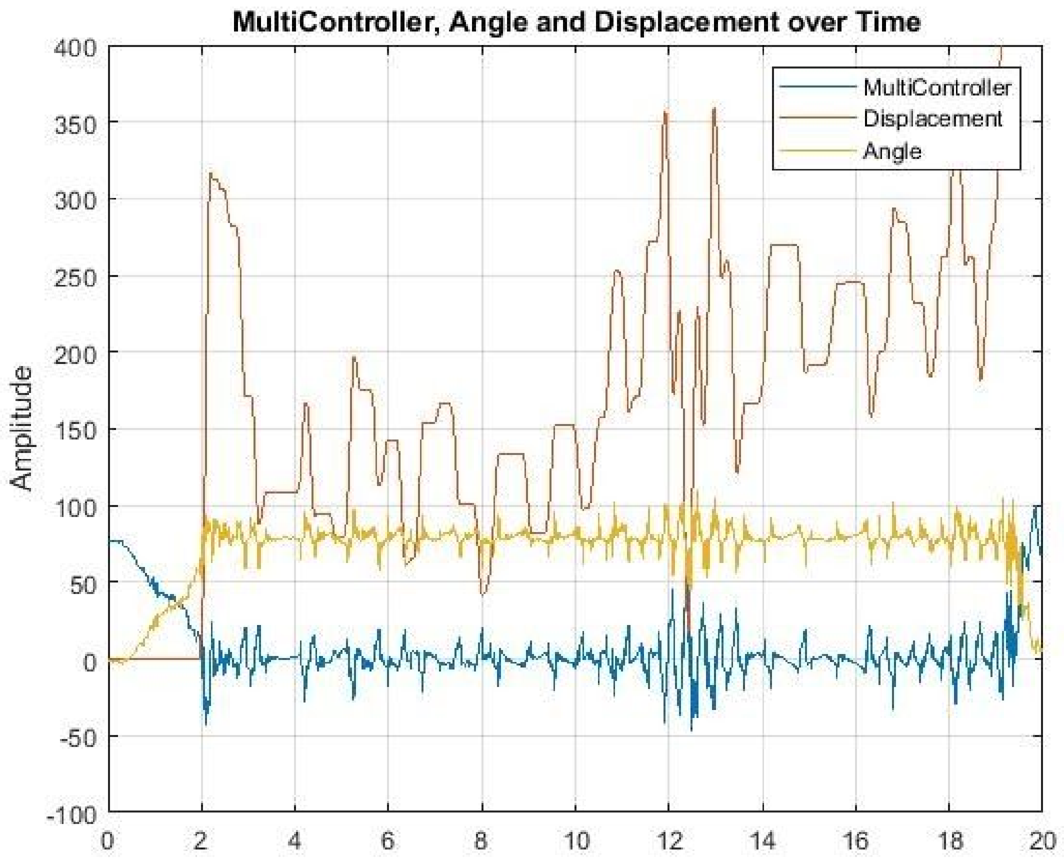

Fig. IV.B.1 *Angle, MultiController and Displacement over Time.*

Fig. IV.B.1, demonstrate the contribution of each of the controllers to the overall system control. The displacement action is relatively small. Essentially, the robot being in constant movement the displacement controller is used to nudge or tilt the robot towards the setpoint to close the displacement gap.

As with the previous controller, it is noticeable the motorspeed and angle are directly inverted. The displacement and angle controller responses are also inverted. As the robot moves to correct the angle the displacement controller pushes against the travel to slow and tilt the robot the other way.

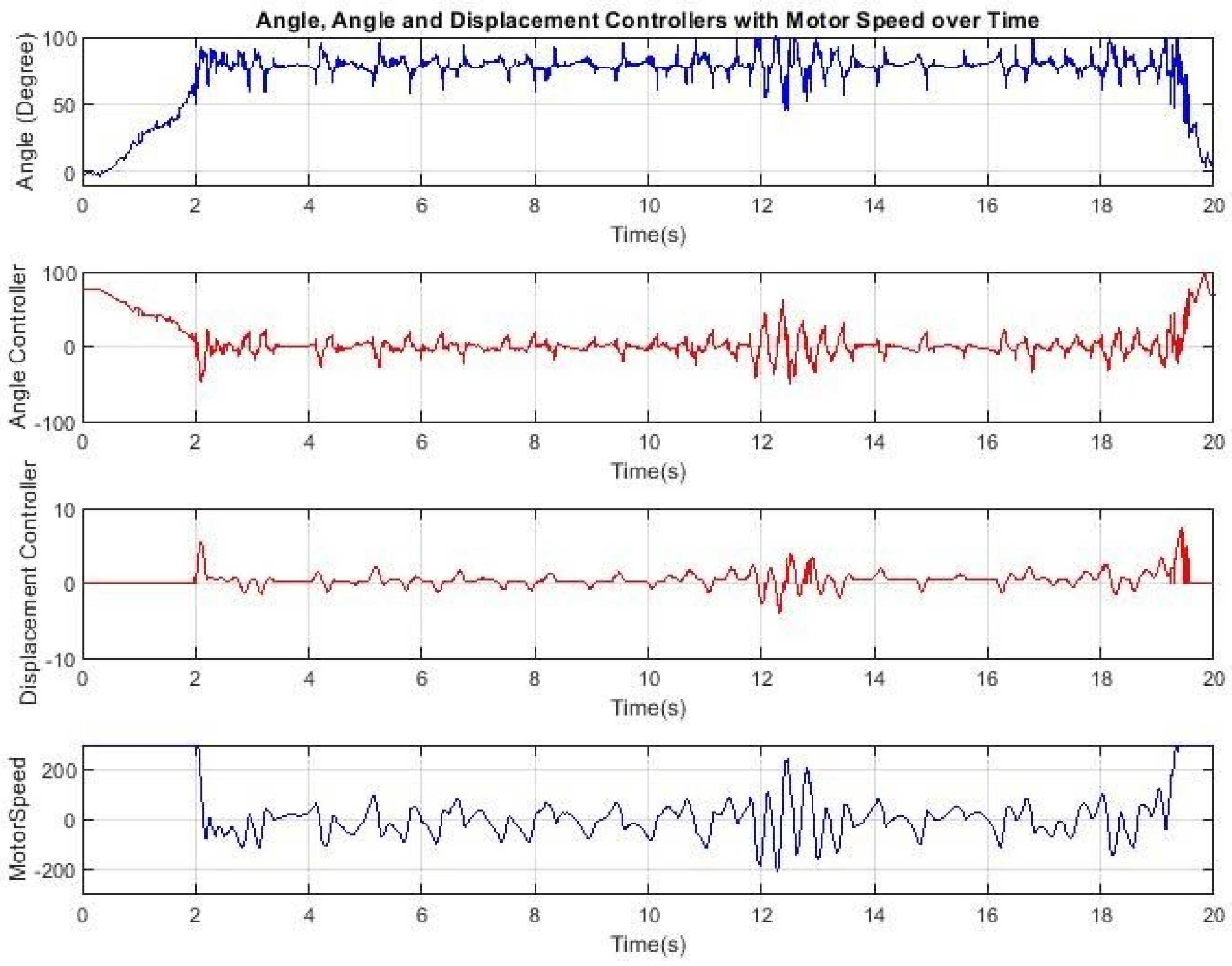

Fig. IV.B.2 *Angle, Separated Angle and Displacement Controllers and MotorSpeed over Time.*

V. Conclusion.

This report offered an explanation to the work undertaken to complete and offer solutions to balancing the inverted pendulum robot. A detailed explanation of the various components and their purpose was given. Then, the two controlled methods retained for the applications were introduced with closed-loop diagrams representing the potential system control implementation. And then, a description of the design process with explanation of the code implementation were given. Finally, measurements were realised to observe the system behaviour.

From the implementation and observation it was clear the basic angle control was not sufficient to provide a functional solutions. Although the system did balance, it was clear the displacement and turning did not offer a satisfactory response hence the implementation of the multicontrol controller.

Now that data is available the system could be further analysed and fine tuned in order to offer an optimal response. Moreover, other control methods (for instance cascade controller or feedforward) could be implemented to test if the system response could be improved.

VI. Bibliography.

Corporation, Pololu. (2019). *Pololu Balboa 32U4 Balancing Robot User’s Guide*. [ebook] Las Vegas: Pololu Corporation. Available at: https://www.pololu.com/docs/pdf/0J70/balboa_32u4_robot.pdf [Accessed 14 Nov. 2019].

Ellis, G. (2012). *Control system design guide*. 4th ed. Oxford: Butterworth-Heinemann.

Ogata, K. (1997). *Modern control engineering* (3rd ed., [International ed.]. ed.). Upper Saddle River, N.J. ; London: Pearson.

Seborg, D., Edgar, T., & Mellichamp, D. (1989). *Process dynamics and control*. New York: Wiley.

STMicroelectronics, “iNEMO inertial module: always-on 3D accelerometer and 3D gyroscope”, LSM6DS33 datasheet, Oct. 2015.

VII. Appendices.

Appendix A - Components Descriptions.

Mechanical Components.

Although limited the mechanical components an integral part in the functionality of the robot. First, the **chassis** provides a structure on which the other components are fitted (Control board, wheel and gear system, bumper cage, motors, battery packs...).
Secondly, the **wheels and gears system** mounted on the chassis (fig. I.A.1) allow for the robot to be mobile. The gears (fig. I.A.2) act as mechanical multiplicators whilst transferring the motion from the motor axis to the wheels.

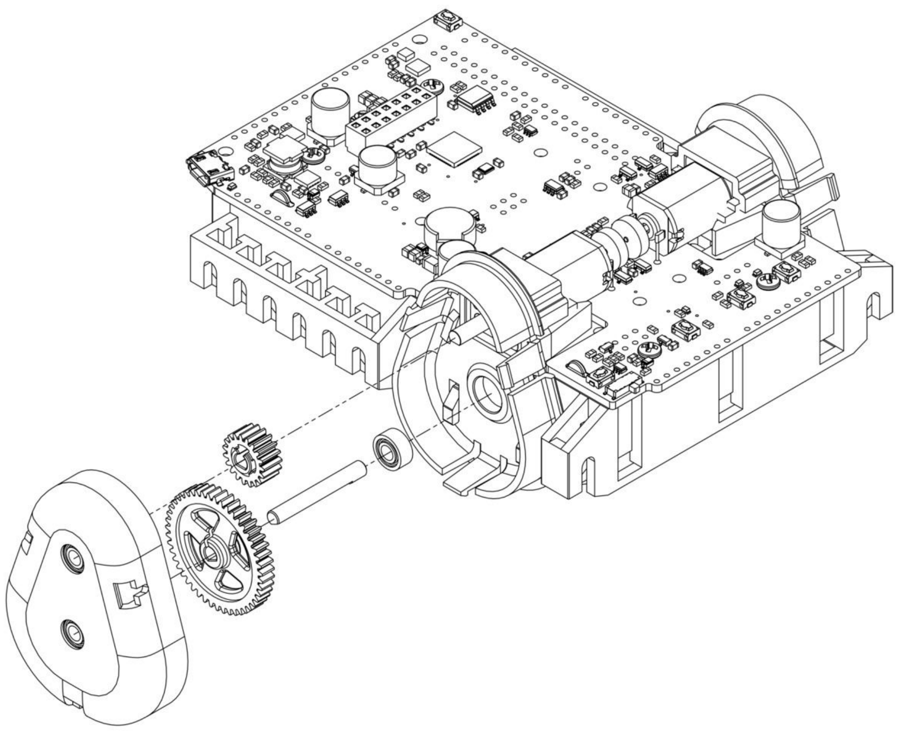

fig. I.A.1 - Mounted Gearbox. (Pololu Corporation, 2019, p.53).

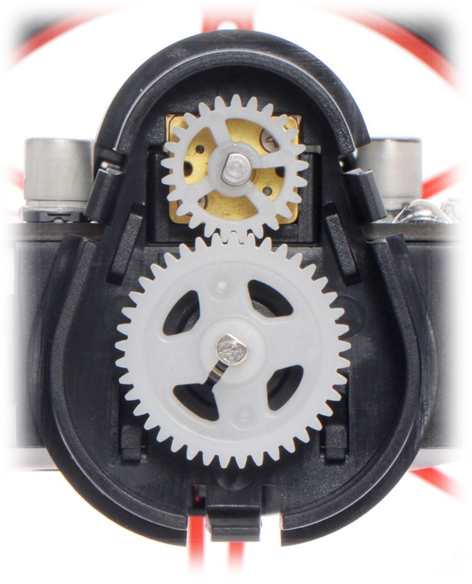

fig. I.A.2 - Gearbox transmission (top: motor axis, bottom: wheel axis) (Pololu Corporation, 2019, p.9).

Finally, the **bumper cage system** (fig. I.A.3) aims to mitigate the effect of shocks during operation of the robot. For instance, if the robot is in balance but comes in contact with an object or receives a push the bumper will act as a barrier protecting the control board. Also, when the robot experiences a fall the bumper will protect the chassis and avoid the risk of the robot falling on and damaging the electronic components. An additional protective plaque was fitted on the top of the robot to protect the control board during falls.

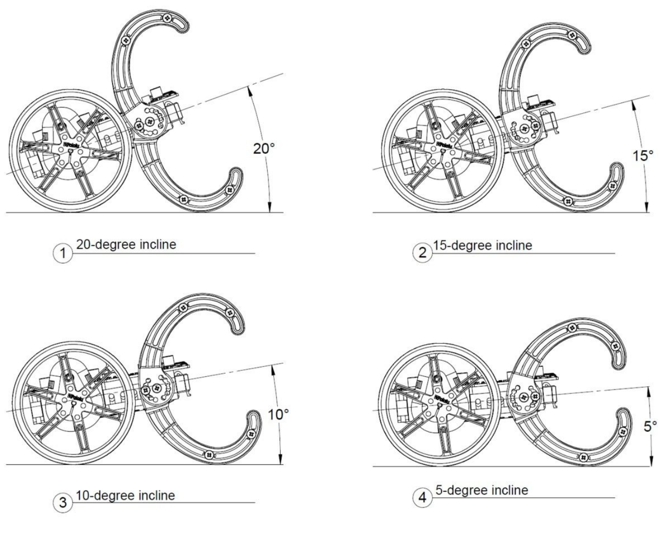

fig. I.A.3 Bumper Cage Mounting Configurations (Pololu Corporation, 2019, p.39).

Electrical & Electronics Components.

This section will describe the electrical and electronics components allowing to interface with, programme, monitor and control the robot. It will also introduce the electro-mechanical components allowing the robot to move and position itself.
All components are mounted on the control PCB allowing for ease of manufacturing, space optimisation and rapid communication between the components.

First, the **“ATmega32U4 AVR” microcontroller** from Atmel, it is the processing brain of the robot. It is coupled with a 16 MHz crystal oscillator. The combination of those two components and the addition of the “Arduino-compatible A-Star 32U4 USB bootloader” makes the control board similar to an Arduino hence can be used with the Arduino IDE (Pololu Corporation, 2019, p.14).
Secondly, the **USB Micro-B connector** allows to interface the robot control board with a computer to programme and then monitor the robot through the Arduino IDE. It can also be used to power the board, but not the motors, to obtain readings from the sensors whilst prototyping (Pololu Corporation, 2019, p.14).
The board also offers 3 coloured **LEDs** (Red, Green and Yellow) which can be used to display various states within the programme (i.e if the robot is within a specific angle range or the system is experiencing an error). The supplied pololu libraries allow for an easy use of the LEDs by calling the appropriate function (for example: “ledRed(1); //red led is set to high = On”). Two LEDs can be found: one green notifying the user that the USB port is powered (i.e the USB cable is connected) and one blue showing the system is receiving power from the battery pack.
The robot offers power and reset buttons allowing to boot the robot and reset the control board. Additionally it also offers the possibility to programme and use three **push buttons** for a user to interact with the programme and robot. The Balboa32U4 library provides function allowing for an easy use of the push buttons.
The board offers the possibility of using a **buzze**r to create sound. The sound is created by changing the output from low to high at a certain frequency which will pitch the sound to that frequency.

An **LCD header** is available to connect a screen if required. It will not be used for the application described in this report.

The robot can move thanks to **two Texas Instruments DRV8838 motors** which drive the gearmotors (fig I.B.1). The drives are programmed by setting the direction and speed within the code. The Balboa32U4 library provides function allowing for an easy use of the drives. The library offer the possibility to use "motors.setSpeeds()" to control the motors movement without the need to cater for each drive individually.

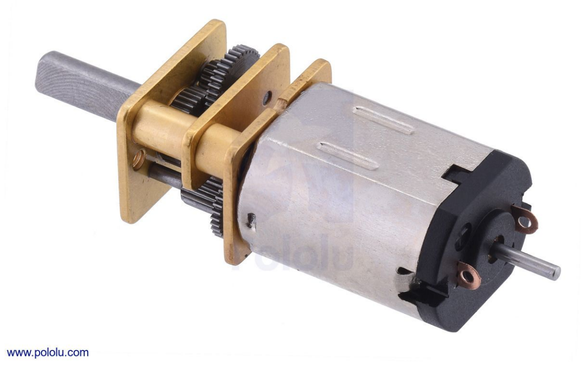

fig. I.B.1 Micro Metal gearmotor HPCB with extended motor shaft. (Pololu Corporation, 2019, p.9)

The drive axis are connected to **quadrature coders** (fig. I.B.2). The coders will count to 12 for each revolution of the motor axis. The signal of the encoder is used to measure the rotational speed and direction of the wheels (Pololu Corporation, 2019, p.18). The Balboa32U4 library offers functions to easily read the position information from the encoders.

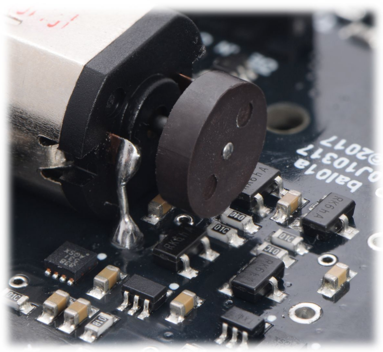

fig. I.B.2. *Balboa encoder, showing the magnetic disc and sensors (with one motor removed).* (Pololu Corporation, 2019, p.18)

The control board is equipped with two sensing chips the **ST LIS3MDL a 3-axis magnetometer** (not used for the application) and the **ST LSM6DS33 combining an accelerometer and gyroscope** (Pololu Corporation, 2019, p.20). The accelerometer and gyroscope (fig. I.B.3) signals (x, y, z axis) will be used to calculate the angle in order to control the robot. Again, the robot library provides functions to facilitate reading values from the sensors.

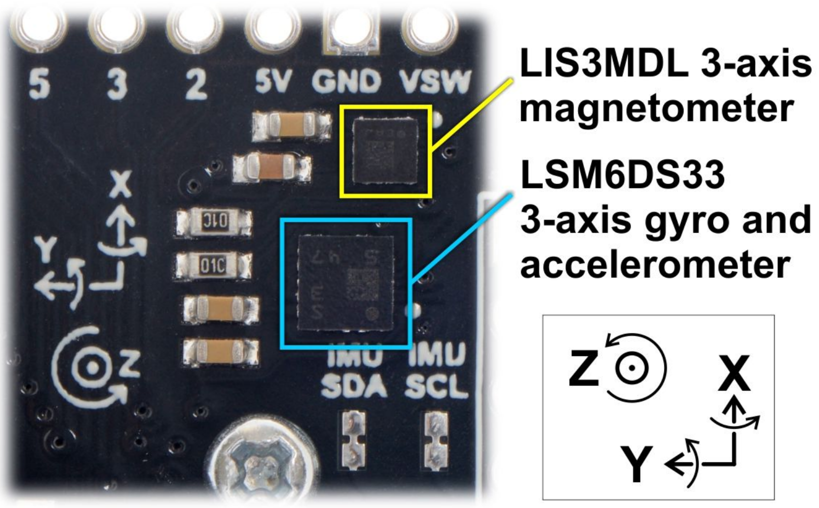
5
3
2
5V
GND
VSW
LIS3MDL 3-axis magnetometer
LSM6DS33 3-axis gyro and accelerometer
IMU SDA
IMU SCL
X
Y
Z

fig. I.B.3 *The Balboa 32U4's IMU and compass chips.* (Pololu Corporation, 2019, p.18)

Finally, **the power system** is composed of the battery pack requiring 6 batteries, a power switch circuit (ON/OFF), a MP4423H switching buck converter to regulate the 5V and 3.3V supplies and a TPS2113A power multiplexer to allow the switch between USB and battery power source. (Pololu Corporation, 2019)

Appendix B - Programme Code Controller 1 and 2.

Controller 1: PD Angle Controller Code.

```
// This program is a PD controlled self balancing inverted pendulum robot.
//
//Libraries:
```

```
#include <Wire.h>
#include <Balboa32U4.h>
#include <LSM6.h>

//Classes renaming for ease of use.
LSM6 imu; //Variable for the Sensors.
Balboa32U4Motors motors; //Variable for motors.
Balboa32U4Encoders encoders; //Variable for wheel encoders.

// -------------------Variable Declaration---------------------------------------//
//Variables for calibration
int32_t sumAx=0; //Summing variable for average error accelerometer x-axis.
int32_t cAx=0; //variable storing value for calibration of accelerometer x-axis.
int32_t sumGz=0; //Summing variable for average error gyroscope y-axis.
int32_t gYZ=0; //variable storing value for calibration of gyroscope y-axis.

//PD Gain constants - Controller Variables
double Kpa = 0.95; //Proportional Gain (Full charge - 0.6).
double Kda = 0.15; // Derivative Gain (Full charge - 0.10).
double Pa; //Proportional Controller Variable.
double Da; //Derivative Controller Variable.
double output_Ang; //controller summed output.

//Variables for error calculations.
double setpoint = 78.8; //78.8 degrees.
double Tilting_angle; //variable to store read/calculated calibrated angle.
double angleRate; //variable to store read/calculated calibrated change in angle.
uint16_t countdiff; //variable to calculate the encoder reading difference error.

//Time variables
uint16_t current_time; //variable to store time reading.
static uint16_t previous_time; //variable to store time reading.
```

```
//Error variables
double error; //Variable for proportional error.
double rateOfError; //Variable for derivative error.

//Motor related variables
double motorspeed; //Variable to store the motor speed value.
int16_t rightcounter; //Variable to store the right wheel encoder reading.
int16_t leftcounter; //Variable to store the left wheel encoder reading.

// -------------------End of Variable Declaration----------------------------------------//
// -------------------System Setup Start-----------------------------------------------//
void setup()
{
  // join I2C bus
Wire.begin();
  if (!imu.init())   // Initialise the imu sensor
  {
    // Verify connection - Failed to detect the LSM6.
    ledRed(1);
    while(1)
    {
         Serial.println(F("Failed to detect the LSM6.")); //error message if connection
failed.

      delay(100);
    }
  }
//-----------------Sensor Configuration Start-----------------------------------------//
  imu.enableDefault();
  // Set the gyro full scale to 1000 dps because the default
  // value is too low, and leave the other settings the same.
  imu.writeReg(LSM6::CTRL2_G, 0b01011000);
  // Set the accelerometer full scale to 16 g because the default
  // value is too low, and leave the other settings the same.
  imu.writeReg(LSM6::CTRL1_XL, 0b01010100);
//------------------Sensor Configuration End--------------------------------------------//
//------------------Calibration Calculation Start--------------------------------------//
//Calculating average error on accelerometer.
  for (int i=0; i<50 ; i++) //loop over 50 readings.
    {
    imu.read(); //Reading from the sensor.
    sumAx += imu.a.x; //Sum the readings.
    delay(1);
    }
  cAx =  sumAx/50; //Accelerometer value for calibration.

//Calculating average error on the gyro.
  for (int i = 0; i < 50; i++)//loop over 50 readings.
  {
    imu.read(); //Reading from the sensor.
    sumGz += imu.g.y; //Sum the readings.
    delay(1);
  }

 gYZ = sumGz / 50; //Gyroscope value for calibration.

  delay(1000); //Delay to give time to the system to settle.
}
//------------------Calibration Calculation End--------------------------------------//
// -----------------System Setup End-------------------------------------------------//
// -----------------Main Programme Loop Start-----------------------------------------//
void loop()
{
  current_time = millis(); //Reading start time.

// Perform the balance updates at 100 Hz.
if ((uint16_t)(current_time - previous_time) < 10) { return; } // Constrain the main loop
to a 100Hz or 10ms cycle.

  imu.read();// reading values from the imu
//Send values to display for monitoring.
  //Serial.print(Tilting_angle); Serial.print("\t");
  //Serial.print(Pa); Serial.print("\t");
  //Serial.print(Da); Serial.print("\t");
  //Serial.print(output_Ang); Serial.print("\t");
  //Serial.print(setpoint); Serial.print("\t");
  //Serial.print(motorspeed); Serial.print("\t");
  //Serial.print("\n");



// Motor Counters
  leftcounter = encoders.getCountsLeft(); //Reading left encoder.
  rightcounter = encoders.getCountsRight(); //Reading right encoder.
  countdiff = leftcounter - rightcounter;  //Calculating encoder error.

//Angle rate from gyroscope;
  angleRate = (imu.g.y - gYZ) / 29; //Angle rate calibrated value in degree/s.

//Angle value from accelerometer.
   double   ax = ((imu.a.x - cAx)*0.488/1000); //calibrated x-axis reading from
accelerometer
  double  az = ((imu.a.z)*0.488/1000); //y-axis reading from accelerometer
  Tilting_angle = (atan2(ax,az)) * 180/PI;    //Angle in degree
```

```
//PID Angle Control.
  error = setpoint - Tilting_angle; //Angle Error calculation.
  rateOfError = angleRate; //Angle rate error.
  Pa = Kp*error; //Proportional controller.
  Da = Kd*rateOfError; //Derivative controller.
  output_Ang = Pa + Da;//Controller summed output.

//Motor Control
  motorspeed += output_Ang; //motorspeed set as controller value
   motorspeed = constrain(motorspeed,-300,300); //constrain/ scale the motor speed
to avoid saturation.

  //For Angle:
  //Safety motor: if 60<Tilting_angle<135 then motor on else  motor off
  if (60 < Tilting_angle && Tilting_angle < 135)
    {
      ledYellow(1); //Indicator robot is in operating range.
      ledRed(0);
            motors.setSpeeds(motorspeed+0.4*countdiff  ,  -motorspeed-0.65*countdiff);
//Motors are on receiving motorspeed with added correction to keep the robot
straight.
      delay(1);
    }
  else
    {
      ledRed(1); //Indicator robot is beyond operating range.
      ledYellow(0);
      motors.setSpeeds(0,0); //Out of range then motor off.
      delay(1);

    }
    previous_time = current_time; //Reading end of loop time.
```

```
 }
// -----------------Main Programme Loop End-------------------------------------------//
// -----------------End of PD Control Programme---------------------------------------//
```

Controller 2: Multi-Controller Code.

```
// This program is a Multi Controller controlled self balancing inverted pendulum robot.
//
//Libraries:
#include <Wire.h>
#include <Balboa32U4.h>
#include <LSM6.h>

//Classes renaming for ease of use.
LSM6 imu; //Variable for the Sensors.
Balboa32U4Motors motors; //Variable for motors.
Balboa32U4Encoders encoders; //Variable for wheel encoders.

// -------------------Variable Declaration--------------------------------------//
//Variables for calibration
int32_t sumAx=0; //Summing variable for average error accelerometer x-axis.
int32_t cAx=0; //variable storing value for calibration of accelerometer x-axis.
int32_t sumGz=0; //Summing variable for average error gyroscope y-axis.
int32_t gYZ=0; //variable storing value for calibration of gyroscope y-axis.

//Controllers Gain constants - Controller Variables.
double Kpa = 0.95; //Angle Proportional Gain (Full charge - 0.6).
double Kda = 0.15; // Angle Derivative Gain (Full charge - 0.10).
double Pa; //Angle Proportional Controller Variable.
double Da; //Angle Derivative Controller Variable.
double output_Ang; //Angle controller summed output.
double Kpd = 0.00225; //Displacement Proportional Gain (Full charge - 0.00225).
double Kdd = 0.2; //Displacement derivative Gain (Full charge - 0.2).
double Pd; //Displacement Proportional Controller Variable.
double Dd; //Displacement Derivative Controller Variable.
double output_dis; //Displacement controller summed output.

//Variables for angle error calculations.
double setpoint = 78.8; //78.8 degrees.
double Tilting_angle; //variable to store read/calculated calibrated angle.
double angleRate; //variable to store read/calculated calibrated change in angle.
uint16_t countdiff; //variable to calculate the encoder reading difference error.

//Time variables
uint16_t current_time; //variable to store time reading.
static uint16_t previous_time; //variable to store time reading.

//Error variables
double error; //Variable for proportional error.
double rateOfError; //Variable for derivative error.
int16_t error_dis;
int16_t Start_Pos = error_dis;
int16_t displacement;
double prev_error_dis;

//Motor related variables
double motorspeed; //Variable to store the motor speed value.
int16_t rightcounter; //Variable to store the right wheel encoder reading.
int16_t leftcounter; //Variable to store the left wheel encoder reading.
double speed_left ; //Variable to store left speed.
double speed_right; //Variable to store right speed.
double speedwheel; //Variable to store average speed.
int16_t prev_leftcounter; //Variable to store left encoder 2nd reading.
int16_t prev_rightcounter; //Variable to store right encoder 2nd reading.
double displ_left; //Variable to store left wheel displacement error.
double displ_right; //Variable to store right wheel displacement error.
// -------------------End of Variable Declaration---------------------------------------//
// -------------------System Setup Start-----------------------------------------------//
void setup()
{
  // join I2C bus
Wire.begin();
  if (!imu.init())   // Initialise the imu sensor
  {
    // Verify connection - Failed to detect the LSM6.
    ledRed(1);
    while(1)
    {
          Serial.println(F("Failed to detect the LSM6.")); //error message if connection
failed.

      delay(100);
    }
  }
//------------------Sensor Configuration Start-----------------------------------------//
  imu.enableDefault();
  // Set the gyro full scale to 1000 dps because the default
  // value is too low, and leave the other settings the same.
  imu.writeReg(LSM6::CTRL2_G, 0b01011000);
  // Set the accelerometer full scale to 16 g because the default
  // value is too low, and leave the other settings the same.
  imu.writeReg(LSM6::CTRL1_XL, 0b01010100);
//------------------Sensor Configuration End-------------------------------------------//
```

```
//------------------Calibration Calculation Start---------------------------------------//
//Calculating average error on accelerometer.
  for (int i=0; i<50 ; i++) //loop over 50 readings.
    {
    imu.read(); //Reading from the sensor.
    sumAx += imu.a.x; //Sum the readings.
    delay(1);
    }
  cAx =  sumAx/50; //Accelerometer value for calibration.

//Calculating average error on the gyro.
  for (int i = 0; i < 50; i++)//loop over 50 readings.
  {
    imu.read(); //Reading from the sensor.
    sumGz += imu.g.y; //Sum the readings.
    delay(1);
  }

 gYZ = sumGz / 50; //Gyroscope value for calibration.

  delay(1000); //Delay to give time to the system to settle.
}
//------------------Calibration Calculation End---------------------------------------//
// -----------------System Setup End-------------------------------------------------//
// -----------------Main Programme Loop Start-----------------------------------------//
void loop()
{

  current_time = millis(); //Reading start time.

// Perform the balance updates at 100 Hz.
```

```
if ((uint16_t)(current_time - previous_time) < 10) { return; } // Constrain the main loop
to a 100Hz or 10ms cycle.

  imu.read();// reading values from the imu
  //Send values to display for monitoring.
  //Serial.print(Tilting_angle); Serial.print("\t");
  //Serial.print(Pa); Serial.print("\t");
  //Serial.print(Da); Serial.print("\t");
  //Serial.print(output_Ang); Serial.print("\t");
  //Serial.print(displacement); Serial.print("\t");
  //Serial.print(setpoint); Serial.print("\t");
  //Serial.print(Pd); Serial.print("\t");
  //Serial.print(Dd); Serial.print("\t");
  //Serial.print(output_dis); Serial.print("\t");
  //Serial.print(motorspeed); Serial.print("\t");
  //Serial.print("\n");

// Motor Counters
  leftcounter = encoders.getCountsLeft(); //Reading left encoder.
  rightcounter = encoders.getCountsRight(); //Reading right encoder.
  countdiff = leftcounter - rightcounter;  //Calculating encoder error.
  displ_left += leftcounter - prev_leftcounter; //left wheel displacement error.
  displ_right += rightcounter - prev_rightcounter; //right wheel displacement error.
  displacement = (displ_left + displ_right)/2; //Displacement average error.
  speed_left = (leftcounter - prev_leftcounter); //left wheel speed error.
  speed_right = (rightcounter - prev_rightcounter);//right wheel speed error.
  speedwheel = (speed_left + speed_right)/2; //Speed average error.
  prev_leftcounter = leftcounter; //Reading left encoder.
  prev_rightcounter = rightcounter;//Reading right encoder.

//Angle rate from gyroscope;
  angleRate = (imu.g.y - gYZ) / 29; //Angle rate calibrated value in degree/s.
```

```
//Angle value from accelerometer.
   double   ax = ((imu.a.x - cAx)*0.488/1000); //calibrated x-axis reading from
accelerometer
  double  az = ((imu.a.z)*0.488/1000); //y-axis reading from accelerometer
  Tilting_angle = (atan2(ax,az)) * 180/PI;    //Angle in degree

//PID Angle Control.
  error = setpoint - Tilting_angle; //Angle Error calculation.
  rateOfError = angleRate; //Angle rate error.
  Pa = Kp*error; //Angle Proportional controller.
  Da = Kd*rateOfError; //Angle Derivative controller.
  output_Ang = Pa + Da; //Angle Controller summed output.

//PD Controller - Displacement.
  Pd = Kpd*displacement; //Displacement Proportional controller.
  Dd = Kdd*speedwheel; //Displacement Derivative controller.
  output_dis = Pd+Dd; //Displacement Controller summed output.

//Motor Control
   motorspeed += output_Ang + output_dis; //motorspeed set as sum of controller
values.
  motorspeed = constrain(motorspeed,-300,300);//constrain/scale the motor speed to
avoid saturation.

  //For Angle:
  //Safety motor: if 60<Tilting_angle<135 then motor on else  motor off
  if (60 < Tilting_angle && Tilting_angle < 135)
   {
     ledYellow(1); //Indicator robot is in operating range.
     ledRed(0);
```

```
          motors.setSpeeds(motorspeed+0.4*countdiff  ,  -motorspeed-0.65*countdiff);
//Motors are on receiving motorspeed with added correction to keep the robot
straight.
     delay(1);
   }
  else
   {
     ledRed(1); //Indicator robot is beyond operating range.
     ledYellow(0);
     motors.setSpeeds(0,0); //Out of range then motor off.
     delay(1);

   }
   previous_time = current_time; //Reading end of loop time.
 }
// -----------------Main Programme Loop End------------------------------------------//
// -----------------End of Multi-Controller Control Programme-------------------------//
```