Augmented Arthropod - Self-Balancing Mech

by tech_support

by Grzegorz Lochnicki and Nicolas Kubail Kalousdian

The year is 2048.

Sixty percent of the human population is already augmented. Internally, through biological modifications, and externally, through modular exoskeletons. The biodiversity of the world shrinks as the gap between species grows.

However, in the fringes of society, there is an idea permeating the minds of the populace. Some believe that bio-diversity is already a thing of the past. Dead. Others believe that through augmentation and artificial adaptation evolution can be altered from a natural process to a cybernetic one...

Governments have been augmenting various animals for espionage and warfare for at least five decades. In an attempt to secure the data developed from these experiments, they saved it outside of the grid into an archaic physical form, a solid state drive. The data you are about to witness was recovered from such a drive.

It is believed that this drive was en-route to a low orbit server station, XXX. However, it seems that it never reached its intended destination. Burn marks on the drive's capsule indicate that there was a malfunction on the autonomous delivery drone that carried it.

It was lost to the whims of the earth's orbit. Until now. The manuscript you are about to read was found deep in the emergent rain-forest of Siberia. Some critical parts of the experiment's results are missing, however, most of instructional data remains intact.

Chassis:

- 24 x M6 Nuts
- 24 x 20mm M6 Washers
- 1 x 1m M6 Threaded Rod
- 1 x 300x250x8mm Plywood sheet
- 1 x Small transparent tape
- 4 x M4 Bolts
- 4 x 7mm M4 Spacers
- 4 x 5mm Wood Screws

- 1 x Small Cable-Tie
- 1 x Strong Double Sided Table
- 2 x Kitchen Sponge
- 2 x Rubber Bands

Wheel Axle:

- 6 x M6 Nuts
- 2 x M6 Washers
- 2 x 30mm M6 Threaded Rod
- 2 x M6 Couplers with Set Screws
- 8 x 5mm Wood Screws

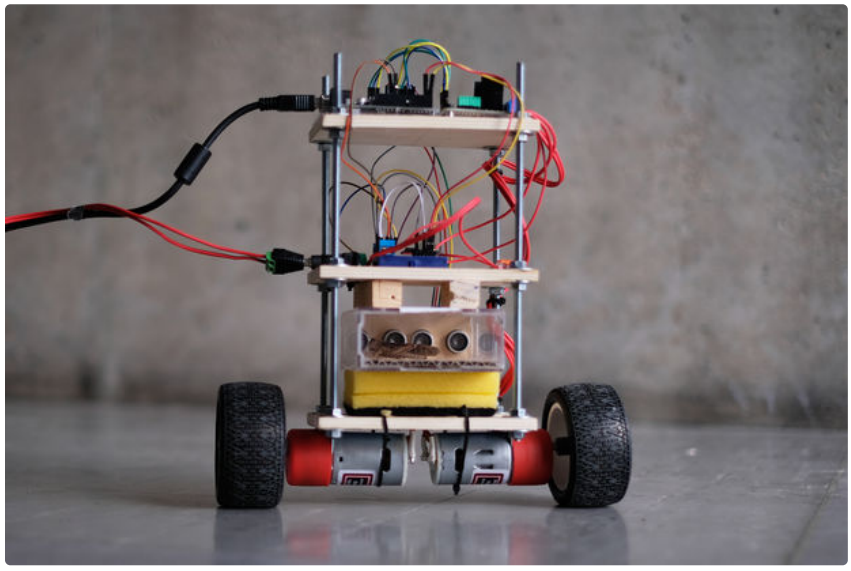
Control Center:

- 1 x 250x250x2mm Vivak (plexi) sheet
- 1 x 9cm by 4 cm piece of lightweight mesh fabric
- 1 x Insect (Arthropod, Caelifera, etc..)

Electronics:

- 1 x Arduino Uno
- 1 x L298N Motor Driver
- 1 x BNO055 Absolute Orientation Sensor
- 2 x HC-SR04 Ultrasonic Sensor
- 2 x 919D Series DC Motors (11:1 Gear Ratio)
- 1 x Female Barrel Plug
- 1 x Male Barrel Plug
- 1 x 12V 4A< Power Supply
- 2 x Double Screw Terminals
- 1 x Wago Connector
- Jumper Cables
- 1 x 20m Red-Black Power Wire (0.4mm)
- 1 x Mini-Breadboard

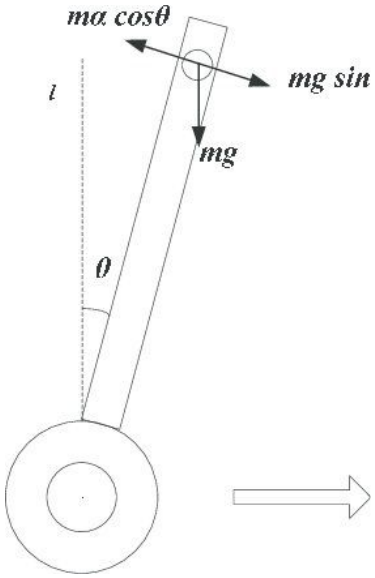
https://youtu.be/esiqaSfpYD4



Step 1: Theory

The problem of a self-balancing robot is that of an inverted pendulum. In order to counter-act the force of the robot falling either forwards or backwards we need a mechanism which will keep its center of gravity directly above the point in which it pivots. This pivot point would be our wheel axle. Our strategy of counter-action will be conducted by driving the robot's wheels in the direction in which it is falling.

However, the problem dosnt stop there. If we have a simple feedback loop which checks which direction the robot is falling and drive the wheels in that direction our robot will inherently oscillate and crash.



So our strategy will involve implementing a PID controller to drive the wheels back and forth in a controlled, mathematical manner, one that responds to the direction in which the robot is falling, the speed with which it falls, the amount it has tilted so far, and the relationship between all of these three variables over time.

The specifics on how this is implemented will be further explained in the PID section of this instructable.

Step 2: Constructing the Chasis the Axle and the Control Center

The Chassis:

1. Cut the plywood sheet into 3 pieces, each 9cm wide and 14.5cm long, these will become the three platforms of the robot's chassis.
2. Drill 6mm holes at the corners of each platform, 10mm in from the edges (see diagrams).
3. Label the three platforms: Top, Middle, and Bottom.
4. Drill out holes for the Arduino Uno, and the L298N Driver board on the Top platform.
5. Measure out the center point of the Middle platform, and mark out an area for the mini-breadboard to get taped down to, as well as a place for the wire connector (see diagrams).
6. Drill holes for the motor to be mounted on the Bottom platform, as well as the holes for the wires to come up from the motors (see diagrams).
7. Cut down the M6 threaded rod into four 25cm pieces.
8. Slide each piece into the corresponding holes of the platforms, securing both sides of each hole with an M6 washer and an M6 nut.
9. Your platforms should be around 9 cm apart, with the remaining length sticking out of the Top platform.
10. Measure the angle of inclination of each platform and adjust each nut in order to have them all level to the ground.
11. Cut longitudinally one of the kitchen sponges in half and attach each of the halves with rubbers to the top platform as bumpers. This step is only for testing.

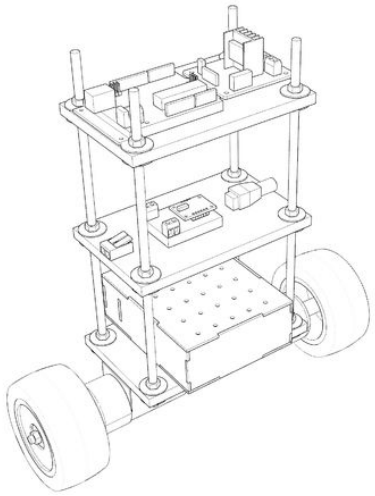
The Axle:

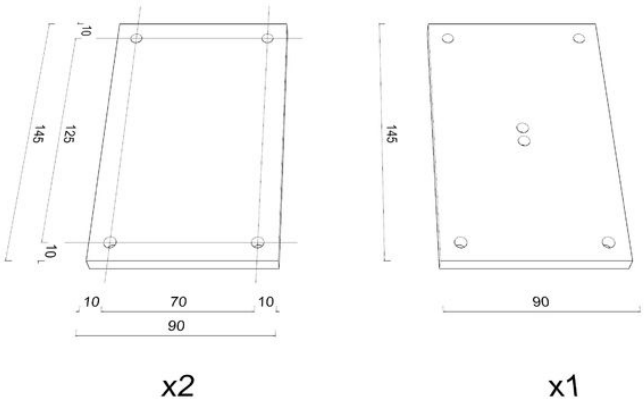
1. Drill a 6mm hole through the center of your wheels.
2. Slide the 30mm M6 threaded rod through the hole. Secure the outside end with an M6 washer and an M6 nut, and secure the inside end with another M6 nut.
3. Attach the coupler to the inside end of the M6 threaded rod and secure it in place by tightening the set screw.
4. Attach the open end of the coupler to the motor shaft, and secure it in place by tightening the set screw. Make sure both wheels are the same distance from the motor itself.
5. Cut out two 40 cm lengths of the Power Wire from your roll and solder them to your motor's terminals.
6. Take your 5mm wood screws, and attach the motors to the Bottom platform, following the guide holes you made during the chassis construction phase.
7. Thread the motors Power Wires through the center hole of the Bottom platform.

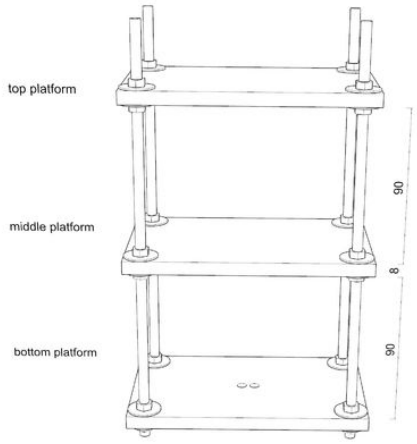
The Control Center:

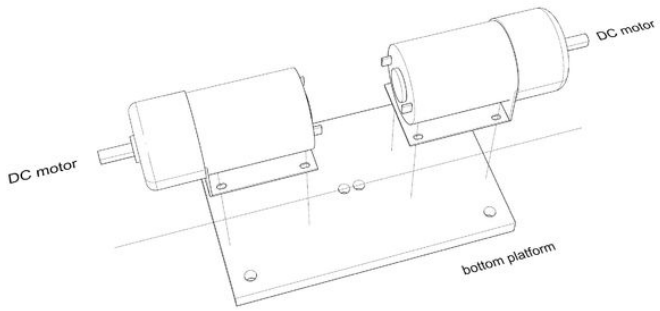
1. Download the dxf file at the top of this section (bug lounge cut file.dxf).
2. Laser cut the file out of 2mm plexiglass.
3. Assemble the parts according to the diagram at the top of this section.

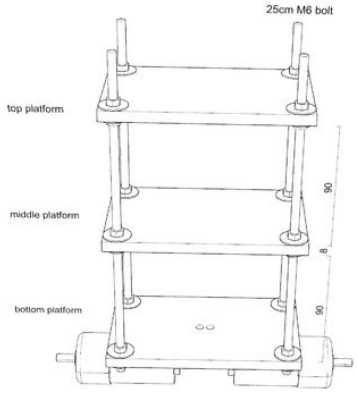
4. Put kitchen sponge on the bottom platform (you can glue or double tape it if you want).
5. Put control center on the sponge.
6. Squeeze the sponge and put two small pieces of wood (around 15x15 mm) as a distance between control center and middle platform. It is easy removable way to put in and out our control box. Our implementation on the picture above.

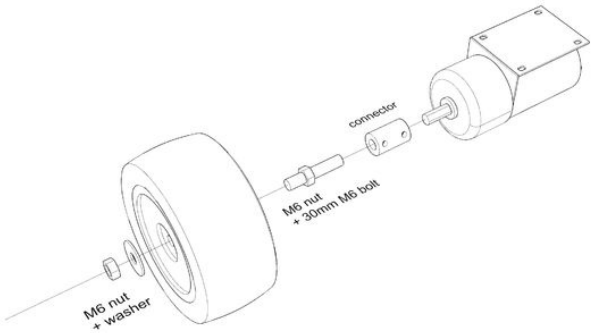


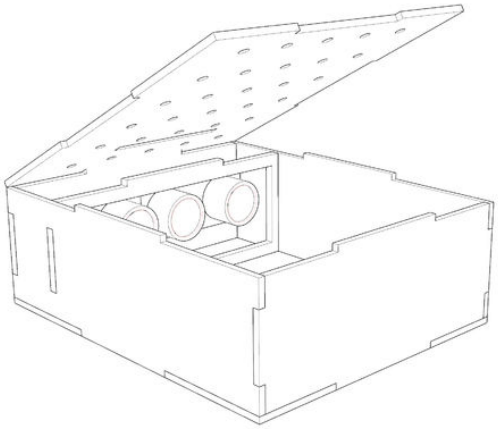


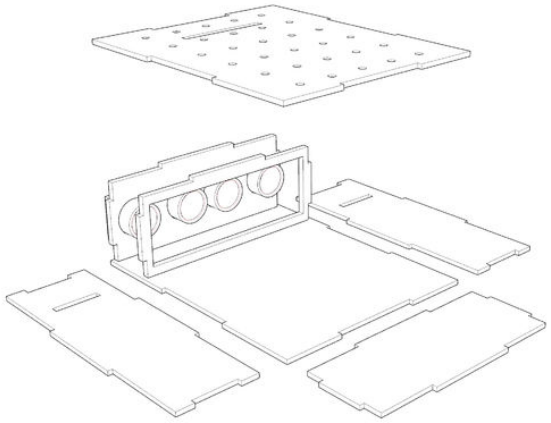


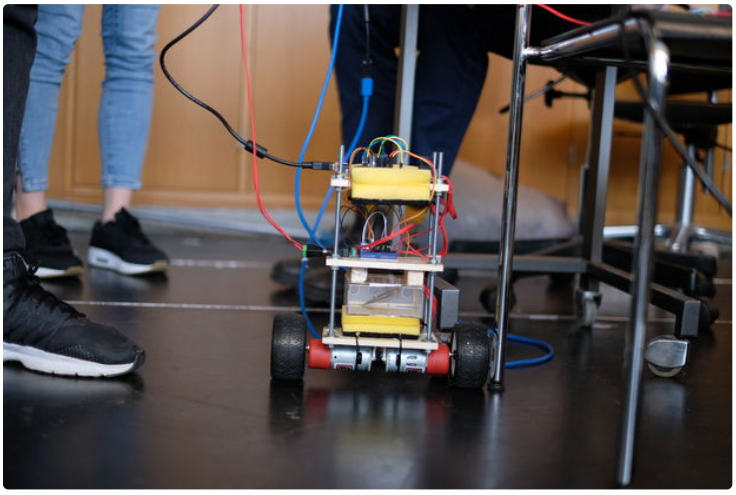












https://www.instructabl...

View in 3D Download

Step 3: Building the Circuit

Motor Driver

The enable pins of the L298N are used to control the speed of your motor using PWM (Pulse Width Modulation), while the In1-4 pins of the driver are used to switch the motor's directions. Below are the instructions which describe the Fritzing diagram at the top of this section.

1. Connect the EnA pin of the L298N to the Arduino's digital pin 6.
2. Connect the In1 pin to the digital pin 5, and the In2 pin to digital pin 3.
3. Connect the EnB pin of the L298N to the Arduino's digital pin 11.
4. Connect the In3 pin to the digital pin 13, and the In4 pin to digital pin 12.
5. Remove the 5V_EN jumper, in order to supply the Arduino with power from the driver as well.
6. Connect the 5V screw terminal from the L298N to the Arduino's Vin pin.
7. Connect one of your motors' positive and negative power wires to the MotorA screw terminals.
8. Connect the other motor's positive and negative power wires to the MotorB screw terminals.
9. Cut out another piece of the Power Wire and connect the red wire to the VMS pin of the L298N, and

the black wire to the GND pin of the L298N. The other end of the red wire should connect to the Wago connector.

10. Place a screw terminal at the end of the mini-breadboard.
11. Cut out another piece of the Power Wire and connect it to the female barrel jack. The red end should then be connected to the Wago connector to complete the circuit all the way to the VMS pin of the L298N, while the black end will go into the screw terminal we placed in the mini-breadboard earlier.
12. Connect the Arduino's GND pin in the same line as the female barrel jack in the mini-breadboard. This will ensure that our system's grounds are all connected.
13. Place another screw terminal on the mini-bread board, and connect the other end of the wire we previously placed in the L298N's GND pin into this terminal. Make sure that this is also connected to the ground line we established in the previous step. Our ground circuit should now be complete. (Look at the images if this part gets confusing).

BNO055 Absolute Orientation Sensor

The BNO055 is a 9 degree of freedom sensor. It fuses data from an accelerometer, gyroscope and magnetometer into absolute 3D orientation. The BNO055 uses I2C communication so we will be wiring it to the Arduino Uno's A5 and A4 pins. This would change depending on the kind of Arduino you choose to use.

1. Solder a header strip into the IMU's breakout board.
2. Place the IMU on the mini-breadboard.
3. Using a jumper cable, connect the Arduino's 5V pin to the mini-breadboard.
4. Connect the IMU's Vin pin inline with the 5V cable coming from the Arduino on the mini-breadboard.
5. Connect the IMU's GND pin inline with the GND pin coming from the Arduino on the mini-breadboard.
6. Take a longer jumper cable and run that from the IMU's SCL pin to the Arduino's A5 pin (which doubles as the SCL pin).
7. Take another long jumper cable and run that from the IMU's SDA pin to the Arduino's A4 pin (which doubles as the SDA pin).

HC-SR04 Ultrasonic Sensors

*The HC-SR04 sensor is an ultrasonic ranging module that provides a measurement function ranging from 2cm to 400cm with a 3mm accuracy. It works on a basic principles of sending pulses, and detecting the time it takes to receive the pulse back. The distance measured by this pulse can be broken down into a simple equation: Distance = (High Level Time * Velocity of Sound) / 2*

1. Connect the VCC pin of the HC-SR04 inline with the 5V cable coming from the Arduino on the mini-breadboard.
2. Connect the GND pin of the HC-SR04 inline with the GND cable coming from the Arduino on the mini-breadboard.
3. Connect the Trig pin of the HC-SR04 to the Arduino's Digital 4 pin.

4. Connect the Echo pin of the HC-SR04 to the Arduino's Digital 2 pin.
5. Repeat steps 1 through 4 with the second HC-SR04 but this time use Digital pin 7 for Trig, and Digital pin 8 for Echo.

Power Supply

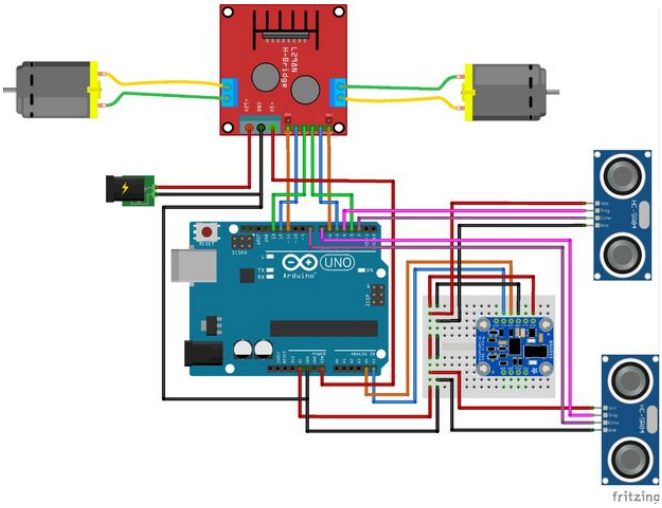
Our motors require 12V and around 2 Amps each, so we will be using an external power supply to provide this electricity. The arduino itself will be powered from the 5V output of the motor driver.

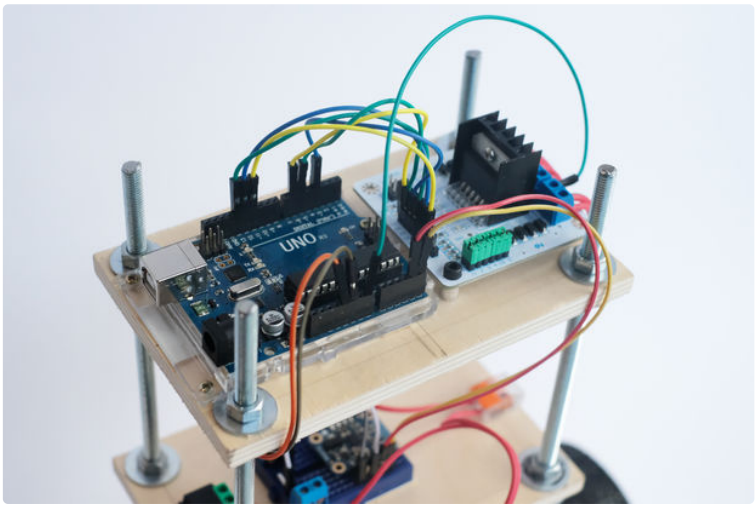
1. Cut out a 5m long strand from your power cable spool.
2. Strip the ends on either side. Attach on end to the power supply screw terminals, and the other end to your male barrel jack.

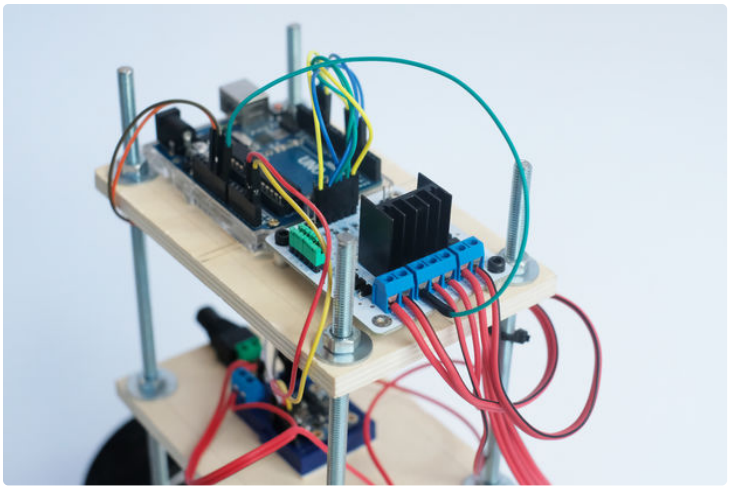
Assembly

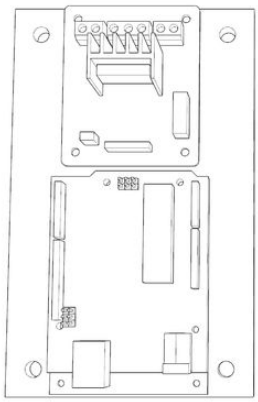
Assembling the electronic on to the chassis is simple. Simply follow the guide holes you made in the Chassis Construction step.

1. Take the 5mm wood screws and attach the Arduino to the Top platform using the mounting holes on the plastic case.
2. Take the 7mm spacers, place them underneath the L298N Motor Driver and slide the M4 bolts through the mounting holes and through the spacers.
3. Underneath the mini-breadboard there should be a patch of double sided tape. Remove the covering of this patch and stick the mini-breadboard in the center of the Middle platform. Make sure that the IMU is center on the paltform, you may need to adjust the breadboard in order to do so.
4. Take another piece of double sided tape and attach the Wago connector to the edge of the Middle platform.
5. Using a cable tie, secure the female barrel to one of the threaded rod pillars.
6. For testing purposes, cut the cleaning sponge in half and attach each half to the sides of the Top platform, using the rubber bands to secure it in place. You may remove this later once your robot stands on its own, but until then this will keep our electronics from getting damaged.





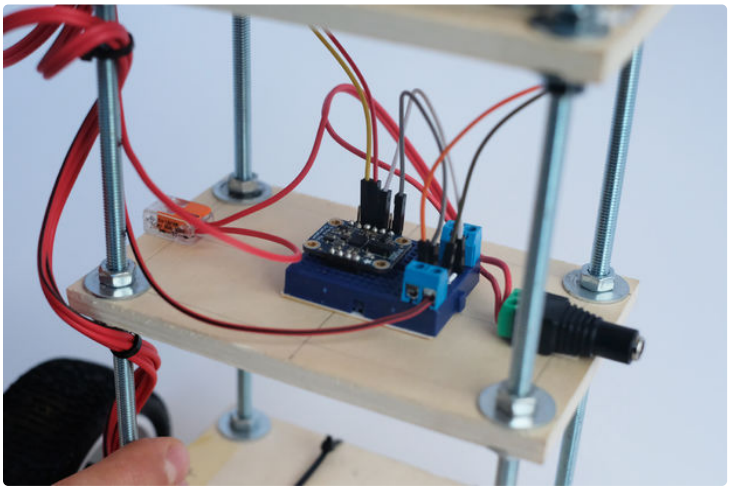


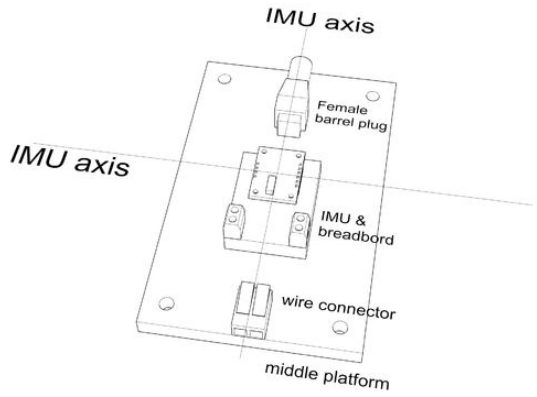


motor driver

Arduino UNO

top platform





Step 4: Coding: Setup Monster Class

In order to program our Monster in a way that can easily be built upon by other developers we will be implementing it as a class/library. A class consists of a header file (.h) and a source file (.cpp). The header file defines everything that will be inside of the class, while the source file consists of the actual code implementation.

We will start with the header file:

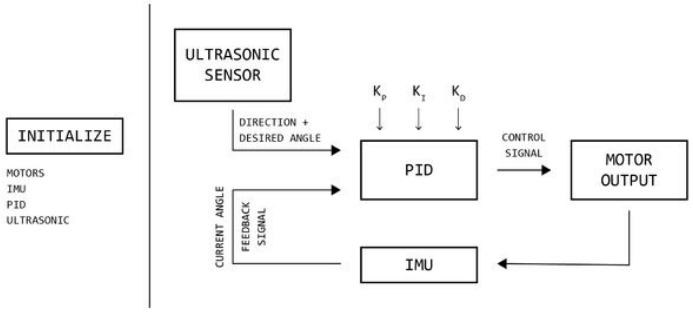
```
#ifndef Monstro_h
#define Monstro_h

#include "Arduino.h"

class Monstro {

  public:
    Monstro(int leftForward, int leftBackward, int leftSpeedPin,
        int rightForward, int rightBackward, int rightSpeedPin,
        int trigA, int echoA, int trigB, int echoB);

    // Behavior
    bool Update();
    void Initialize();

  private:

};
#endif
```

All we have done here is set up the header file with a constructor that takes in the pins we will use for interacting with our sensors and drivers. We will be adding to this later on as we introduce each of the constituent components.
The #include Arduino.h statement just ensures that we have access to the constants and types provided by the Arduino language.
We will be using the Update() function to call certain behaviors during the main loop, and the Initialize() function to make sure our sensors and motors are ready for action. More on this in later steps.

Our source file will reflect this header file:

```
#include "Arduino.h"
#include "monstro.h"

Monstro::Monstro(int leftForward, int leftBackward, int leftSpeedPin,
        int rightForward, int rightBackward, int rightSpeedPin,
        int trigA, int echoA, int trigB, int echoB)
{

}

// Behavior
void Monstro::Initialize() {
}
bool Monstro::Update() {
}
```

Again, all we have done here is set up the bare bones of the source file, while making sure we also include the Arduino.h reference here, and a reference to our header file so we have access to its definitions as well.



Step 5: Measuring Angle of Inclination (IMU)

Implementing the code for the BNO055 is quite simple, thanks to the libraries written by the programmers at Adafruit. We will be utilizing the Adafruit_BNO055 driver library as well as the Adafruit Unified Sensor library.

Let's begin by updating our header file to interact with the IMU.

```
#ifndef Monstro_h
#define Monstro_h

#include "Arduino.h"
#include <Adafruit_Sensor.h>
#include <Adafruit_BNO055.h>
#include <utility/imumaths.h>

class Monstro {

  public:
    Monstro(int leftForward, int leftBackward, int leftSpeedPin,
        int rightForward, int rightBackward, int rightSpeedPin,
        int trigA, int echoA, int trigB, int echoB);

    // Behavior
    bool Update();
    void Initialize();

    // IMU
    volatile double xTilt;
    volatile double yTilt;
    volatile double zTilt;

  private:

    // IMU
    Adafruit_BNO055 _bno;
    void initializeIMU();
    void readIMU();

};
#endif
```

We have added a few things to our header file.

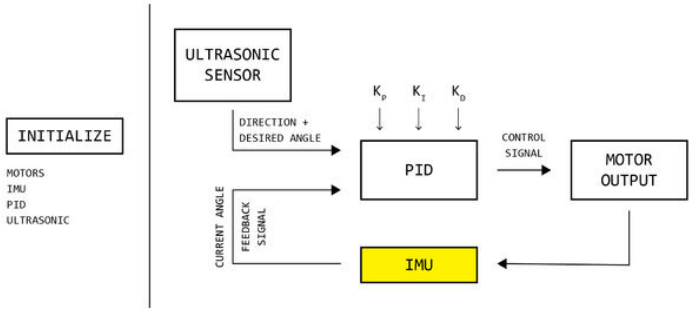
- First off, you will notice three include statements, which ensure we have access to the Adafruit libraries as well as the imumaths.h library whose functions we will require when implementing the IMU reading.
- We have also added public variables xTilt, yTilt, and zTilt. These are where we will be storing the data we retrieve from the IMU during each update cycle. Notice that we have marked them as volatile, this is because we will be using them in a timer interrupt later on in the tutorial.
- We have also added a BNO055 object (_bno), an initializer function to set it up, and a reading function to use during the update cycle.

Now lets implement these functions in the source file:

```
#include "Arduino.h"
#include "monstro.h"

Monstro::Monstro(int leftForward, int leftBackward, int leftSpeedPin,
        int rightForward, int rightBackward, int rightSpeedPin,
        int trigA, int echoA, int trigB, int echoB)
{

}

// Behavior
void Monstro::Initialize() {
  initializeIMU();
}
bool Monstro::Update() {
  readIMU();
}

// IMU
void Monstro::initializeIMU() {
  _bno = Adafruit_BNO055(55);
  if (!_bno.begin())
  {
    Serial.print("No BNO055 detected");
    while (1);
  }
  delay(1000);
  _bno.setExtCrystalUse(true);
}
void Monstro::readIMU() {
  sensors_event_t event;
  _bno.getEvent(&event);

  xTilt = event.orientation.x;
  yTilt = event.orientation.y;
  zTilt = event.orientation.z;
}
```

We have now implemented our IMU functionalities:

- We have included the IMU initialization inside of our master Initialize() function, and included the IMU reading inside of our master Update() function.
- We have also implemented the code for IMU initialization, where we interface with the BNO055
- And finally implemented the reading of our robots absolute orientation inside of our readIMU() function, which then assigns the three tilts to our internal variables.



Step 6: Motor Control

Implementing the code for motor control will involve a bit more logic than the IMU code. This is because it will be receiving its values from the PID algorithm we will be writing later on in the tutorial.

So let's start by updating the header file:

```
#ifndef Monstro_h
#define Monstro_h

#include "Arduino.h"
#include <Adafruit_Sensor.h>
#include <Adafruit_BNO055.h>
#include <utility/imumaths.h>

class Monstro {

  public:
    Monstro(int leftForward, int leftBackward, int leftSpeedPin,
        int rightForward, int rightBackward, int rightSpeedPin,
        int trigA, int echoA, int trigB, int echoB);

    // Behavior
    bool Update();
    void Initialize();

    // IMU
    volatile double xTilt;
    volatile double yTilt;
    volatile double zTilt;

  private:

    // IMU
    Adafruit_BNO055 _bno;
    void initializeIMU();
    void readIMU();

    // Motors
    int _leftForward;
    int _leftBackward;
    int _leftSpeedPin;
    int _rightForward;
    int _rightBackward;
    int _rightSpeedPin ;
    void initializeMotors();
    void setMotors(int leftMotorSpeed, int rightMotorSpeed);

};
#endif
```

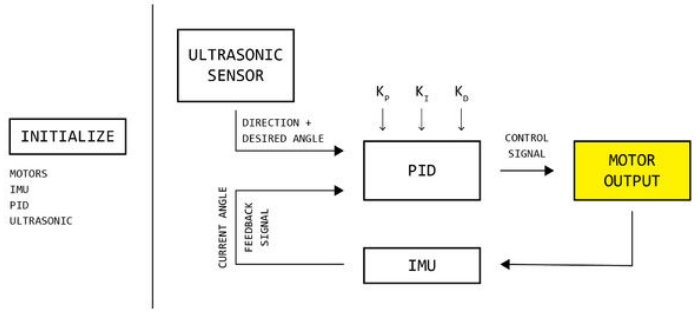
The new lines are at the bottom of the header files, under the comment "Motors". We are defining private variables which will hold a reference to the pins which each motor is controlled by (the direction pins, and the speed pins). We have also included two more functions, one to initialize the motors, and one to actually change the speed and direction of the motors.

Now lets update our source file to reflect these changes:

```
#include "Arduino.h"<br>#include "monstro.h"

Monstro::Monstro(int leftForward, int leftBackward, int leftSpeedPin,
        int rightForward, int rightBackward, int rightSpeedPin,
        int trigA, int echoA, int trigB, int echoB)
{
    _leftForward = leftForward;
    _leftBackward = leftBackward;
    _leftSpeedPin = leftSpeedPin;
    _rightForward = rightForward;
    _rightBackward = rightBackward;
    _rightSpeedPin = rightSpeedPin;
}

// Behavior
void Monstro::Initialize() {
  initializeIMU();
  initializeMotors();
}
bool Monstro::Update() {
  readIMU();
}

// IMU
void Monstro::initializeIMU() {
  _bno = Adafruit_BNO055(55);
  if (!_bno.begin())
  {
    Serial.print("No BNO055 detected");
    while (1);
  }
  delay(1000);
  _bno.setExtCrystalUse(true);
}
void Monstro::readIMU() {
  sensors_event_t event;
  _bno.getEvent(&event);

  xTilt = event.orientation.x;
  yTilt = event.orientation.y;
  zTilt = event.orientation.z;
}

// Motors
void Monstro::initializeMotors() {
  pinMode(_leftForward, OUTPUT);
  pinMode(_leftBackward, OUTPUT);
  pinMode(_leftSpeedPin, OUTPUT);
  pinMode(_rightForward, OUTPUT);
  pinMode(_rightBackward, OUTPUT);
  pinMode(_rightSpeedPin, OUTPUT);
}
void Monstro::setMotors(int leftMotorSpeed, int rightMotorSpeed) {
  if (rightMotorSpeed <= 0) {
    digitalWrite(_rightBackward, LOW);
    digitalWrite(_rightForward, HIGH);
    analogWrite(_rightSpeedPin, abs(rightMotorSpeed));
  }
  else {
    digitalWrite(_rightBackward, HIGH);
    digitalWrite(_rightForward, LOW);
    analogWrite(_rightSpeedPin, rightMotorSpeed);
  }
  if (leftMotorSpeed <= 0) {
    digitalWrite(_leftBackward, LOW);
    digitalWrite(_leftForward, HIGH);
    analogWrite(_leftSpeedPin, abs(leftMotorSpeed));
  }
  else {
    digitalWrite(_leftBackward, HIGH);
    digitalWrite(_leftForward, LOW);
    analogWrite(_leftSpeedPin, leftMotorSpeed);
  }
}
```

The updates are the following:

- We have now assigned our private pin variables values inside of the constructor, so that these can be changed by the user depending on their particular setup.
- We have added our motor initialization into our general Initialize() function.
- We have implemented the motor initialization routine, which involves setting the pins to Output.
- And we have defined our function which will actually actuate the motors setMotors(). Depending on the values passed into this function (-255 to 255) the motors will start spinning at different speeds and in different directions. These values will be generated by the PID algorithm in the next section.



Step 7: PID Algorithm Implementation

Now I will introduce the more complex part of the code: the PID controller algorithm.

This type of algorithm is used in many automatic control applications. It can regulate all sorts of processes, from flow and temperature to leveling and speed. Essentially it is a closed feedback loop which takes in a variable as an **input** and produces an **output** in an attempt to drive the **input** to a specific **set point**.

PID stands for Proportional, Integral, and Derivative. Each of these terms affect the controllers response in different ways. Together they will produce an output which will drive our motors to keep our robot balanced.

- The **proportional** is the main driving term in the controller. It changes the controller output in proportion to the error (in our case the difference between the measured angle, and the desired angle). If the error becomes larger, then the gain from this term will increase proportionally.
- The **integral** term affects our robots response to the error based on the error's accumulation over time. If the error is large for a given period of time the increase/decrease will happen at a fast rate.

Likewise if the error is small for a long period of time the changes will occur at a slower pace. You can think of this as the response based on how the system has behaved in the past.

- The **derivative** term produces an output based on the rate of change of the error. This translates to the difference between the current error and the previous error divided by the sampling period. This term will help in predicting how the balance of our robot will respond in the next reading. You can think of this term a predictive response on how the system will behave in the future.

So, now that we have a basic understanding of how the PID controller works in theory, let's go ahead and implement it into our class. We can start off by updating our header:

```
#ifndef Monstro_h
#define Monstro_h

#include "Arduino.h"
#include <Adafruit_Sensor.h>
#include <Adafruit_BNO055.h>
#include <utility/imumaths.h>

class Monstro {

  public:
    Monstro(int leftForward, int leftBackward, int leftSpeedPin,
        int rightForward, int rightBackward, int rightSpeedPin,
        int trigA, int echoA, int trigB, int echoB);

    // Behavior
    bool Update();
    void Initialize();
    void ComputeBalance();

    // IMU
    volatile double xTilt;
    volatile double yTilt;
    volatile double zTilt;

  private:

    // IMU
    Adafruit_BNO055 _bno;
    void initializeIMU();
    void readIMU();

    // Motors
    int _leftForward;
    int _leftBackward;
    int _leftSpeedPin;
    volatile int _leftSpeed = 0;
    int _rightForward;
    int _rightBackward;
    int _rightSpeedPin ;
    volatile int _rightSpeed = 0;
    void initializeMotors();
    void setMotors(int leftMotorSpeed, int rightMotorSpeed);

    // PID
    volatile float previous_error = 0, integral = 0;
    volatile int motorPower;
    float sampleTime = 0.005;
    double outMin, outMax;
    double _Kp, _Ki, _Kd;
    volatile float Setpoint = 0, Input, Output;
    void initializePID();
    void SetTunings(double Kp, double Ki, double Kd);
    void SetOutputLimits(double Min, double Max);

};
#endif
```

There is a lot of new code here, and most of it may at first glance be difficult to comprehend, so I will go into detail:

- As you can see we have added another public function to our class: ComputeBalance(). This function will be called in a timer interrupt and consists of our PID controller's algorithm.
- We have also included a few variables which we will use in the actual implementation, like the previous_error, and the integral which we need to store between iterations.
- The motorPower is what we will use to drive our motors when calling the setMotors() function in the main Update() loop.
- The sampleTime is how often we will be calling the ComputeBalance function, in seconds.
- The variables outMin and outMax will help us constrain our output to values our motors are able to read (in our case these will be -255 to 255, however there may be cases where we want to change these.
- _Kp, _Ki, and _Kd are our proportional, integral, and derivative constants. These are what each part of the algorithm will be multiplied by.
- The Setpoint is our desired angle, and if our robot wants to stay balanced it should be set to 0. The input is what we will read from our IMU's tilt, and the Output is what the PID algorithm will give us.
- We also have an initializer function, and two more functions to help us tune the algorithm.

Now lets get into the source code implementation of the PID controller. This part is by no means finished, and we have written a few variations of this algorithm, but for the time being this version seems to work the best with our current setup:

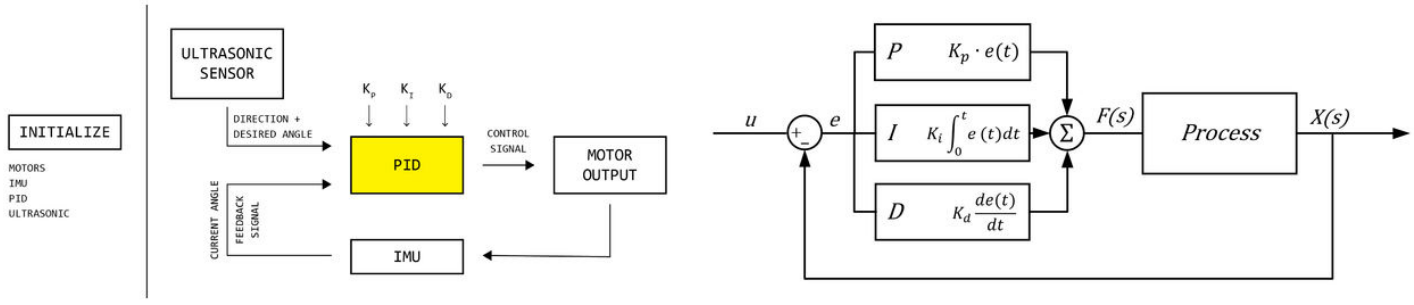
```
#include "Arduino.h"<br>#include "monstro.h"

Monstro::Monstro(int leftForward, int leftBackward, int leftSpeedPin,
        int rightForward, int rightBackward, int rightSpeedPin,
        int trigA, int echoA, int trigB, int echoB)
{
    leftForward = leftForward;
    _leftBackward = leftBackward;
    _leftSpeedPin = leftSpeedPin;
    _rightForward = rightForward;
    _rightBackward = rightBackward;
    _rightSpeedPin = rightSpeedPin;
}

// Behavior
void Monstro::Initialize() {
  initializeIMU();
  initializeMotors();
  initializePID();
}
bool Monstro::Update() {
  readIMU();
  setMotors(motorPower, motorPower);
}

// IMU
void Monstro::initializeIMU() {
  _bno = Adafruit_BNO055(55);
  if (!_bno.begin())
  {
    Serial.print("No BNO055 detected");
    while (1);
  }
  delay(1000);
  _bno.setExtCrystalUse(true);
}
void Monstro::readIMU() {
  sensors_event_t event;
  _bno.getEvent(&event);
```

```
  xTilt = event.orientation.x;
  yTilt = event.orientation.y;
  zTilt = event.orientation.z;
}

// Motors
void Monstro::initializeMotors() {
  pinMode(_leftForward, OUTPUT);
  pinMode(_leftBackward, OUTPUT);
  pinMode(_leftSpeedPin, OUTPUT);
  pinMode(_rightForward, OUTPUT);
  pinMode(_rightBackward, OUTPUT);
  pinMode(_rightSpeedPin, OUTPUT);
}
void Monstro::setMotors(int leftMotorSpeed, int rightMotorSpeed) {
  if (rightMotorSpeed <= 0) {
    digitalWrite(_rightBackward, LOW);
    digitalWrite(_rightForward, HIGH);
    analogWrite(_rightSpeedPin, abs(rightMotorSpeed));
  }
  else {
    digitalWrite(_rightBackward, HIGH);
    digitalWrite(_rightForward, LOW);
    analogWrite(_rightSpeedPin, rightMotorSpeed);
  }
  if (leftMotorSpeed <= 0) {
    digitalWrite(_leftBackward, LOW);
    digitalWrite(_leftForward, HIGH);
    analogWrite(_leftSpeedPin, abs(leftMotorSpeed));
  }
  else {
    digitalWrite(_leftBackward, HIGH);
    digitalWrite(_leftForward, LOW);
    analogWrite(_leftSpeedPin, leftMotorSpeed);
  }
}

// PID
void Monstro::initializePID() {
  SetOutputLimits(-250, 250);
  SetTunings(25, 0.5, 275);
}
void Monstro::ComputeBalance() {
  Input = zTilt;

  // Compute error variables
  float error = Input - Setpoint;

  // Calculate proportional component
  float proportional = error * _Kp;

  // Calculate integral component
  integral += error * _Ki;
  integral = constrain(integral, outMin, outMax); // limit wind-up

  // Calculate derivative component
  float derivative = (error - previous_error) * _Kd;

  // Save variables for next error computation
  previous_error = error;

  // Add up PID
  Output = proportional + integral + derivative;
  // Limit to PWM constraints
  Output = constrain(Output, outMin, outMax);

  // Motor control
  motorPower = Output;

  // give up if there is no chance of success
  if (Input < -40 || Input > 40) motorPower = 0;
}
void Monstro::SetTunings(double Kp, double Ki, double Kd) {
  _Kp = Kp;
  _Ki = Ki;
```

```
  _Kd = Kd;
}
void Monstro::SetOutputLimits(double Min, double Max) {
  if (Min > Max) return;
  outMin = Min;
  outMax = Max;
}
```

Let's go over our changes:

- We included the setMotors() function inside of the Update() loop. This will ensure that each time the main loop runs our motors' rotation and speed is updated according to the output provided by the PID controller (motorPower).
- The initializePID() function is added to our master Initialize() function. It is used to set our constants, and set our minimum and maximum outputs.
- The ComputeBalance() function itself is where the PID calculations happen. It starts by taking in the zTilt of our robot as the input. Then we compute the error by checking on the difference between the input and our setpoint (which should be 0 if we are keeping the robot balanced). We then compute the proportional term by multiplying it against the error. Followed by adding the error*integral constant to our integral term, and constraining it to our maximum and minimum in order to limit the wind up this term can cause (in case our robot falls down for a while and we want to pick it back up without it acting crazy). The derivative is then calculated by checking on the difference between our current error and our last measured error, and multiplying that by our derivative constant. We then save our current error as our last error, and add up our terms together to compute our Output. This Output can now be assigned to the motorPower which will be read to drive our motors in the main Update() loop. Last but not least we also want to make sure to turn the motorPower to 0 in case our robot tilt is beyond a point where it can recover from, so that when it falls down it doesnt keep spinning its wheels and destroys itself.

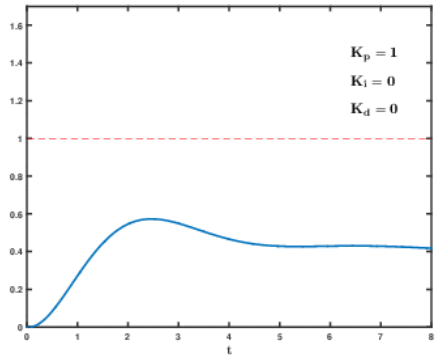


Step 8: Tuning PID Constants

There are some established mathematical strategies of tuning the PID constants, like the Ziegler-Nichols method, or the Cohen-Coon method. However, we have found it difficult to implement these methods in our system, and have therefore opted for tuning with a few simpler rules:

1. Set all the constants to zero. Then slowly increase _Kp until the robot starts to oscillate. Make sure that it will always correct itself if leaning to one side, even if it falls down to the other when doing so.
2. Increase _Kd in regular intervals until you notice the oscillations begin to decrease.
3. Increase _Ki so that the response is faster when the robot is really out of balance, and slower when it is only a little off its setpoint. This should improve the reaction time that was decreased when you increased _Kd.
4. Fine tune the constants from this point until the robot can maintain its balance indefinitely.

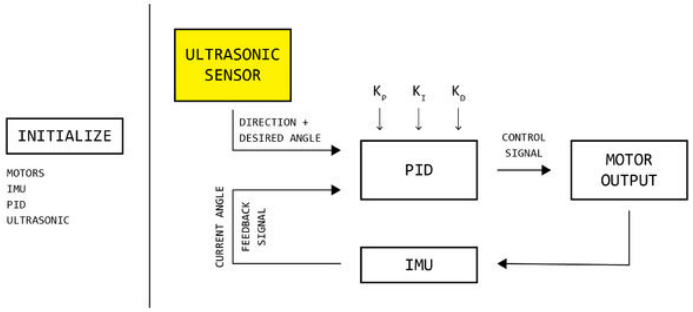
The gif uploaded to the beginning of this section can also serve as a visual guide as to the effect of each of these constants. We have found it very helpful as a visual tool for tuning.



Step 9: Ultrasonic Sensors

The ultrasonic sensor code is particular to each setup and bug type. Therefore we will not be including it in this instructable. However the overall logic is the same for all types of movement: change the Setpoint variable to be more than 0 and the robot will travel one way, change it to be below 0 and the robot will travel the other way. You can also multiply constants to each wheel's speed to get the robot to turn left and right.

(Update: Upon further consideration this section will be detailed soon)


INITIALIZE
MOTORS
IMU
PID
ULTRASONIC
ULTRASONIC SENSOR
DIRECTION + DESIRED ANGLE
CURRENT ANGLE
FEEDBACK SIGNAL
K_P
K_I
K_D
PID
CONTROL SIGNAL
MOTOR OUTPUT
IMU

Step 10: Using the Monster Library

Now that we have the library all coded up, we can use it in a simple Arduino sketch.
We will do so by importing the library header, constructing an instance of the Monster class, and using a Timer Interrupt (from the TimerOne.h library) to call the ComputeBalance() function at a regular interval.

The implementation code is as follows:

```
#include "monstro.h"
#include <TimerOne.h>

Monstro meuMonstro(13, 12, 11, 3, 5, 6, 13, 12, 8, 7);

void setup()
{
  // COM
  Serial.begin(9600);
  // Timer Interrupt
  Timer1.initialize(5000);
  Timer1.attachInterrupt(BalanceRobot);
  meuMonstro.Initialize();
}

void loop()
{
  meuMonstro.Update();
}

void BalanceRobot() {
  meuMonstro.ComputeBalance();
}
```

Upload this to your Arduino, plug in the male barrel jack into the robots female barrel jack for power, and the robot should begin balancing itself.

Step 11: Future Development & References

There are a few things which we would like to have done differently, or would encourage to explore further.

These are:

1. Integrate the power supply into the chassis of the robot as a battery, so that it no longer needs to be tethered and can roam freely.
2. Spend more time tuning the PID constants, this time with a graphing tool in order to have a more informed tuning approach.
3. Develop a method to manually control the robot with a wireless controlle.
4. Replace the BNO055 with a MPU6050 in order to leverage the higher sample rate.

Also, we would like to recommend a few other resources which were very helpful in the development of this project:

1. Developing PID algorithms: http://brettbeauregard.com/blog/2011/04/improving...

2. Visually tuning PID controllers: https://tinyurl.com/y8c89mxc
3. IMU Information: https://learn.adafruit.com/adafruit-bno055-absolu...

And some image references:

Theory Section:
Zhang, J., 2016. Design of a two-wheeled self-balance personal transportation robot. 2016 IEEE 11th Conference on Industrial Electronics and Applications (ICIEA), [Online]. 11, N/A. Available at: https://tinyurl.com/y5ryp7rl [Accessed 5 May 2019].

Very fun little rolling guy! I like watching the video of it :)