Chittagong University of Engineering And Technology

Department of Electrical And Electronic Engineering

Project report on

Self Balancing Robot by Using Arduino

COURSE NO.	**: EEE 242**
COURSE TITLE	**: Electronic Shop Practice**

Submitted by

Opy Das (ID-1702081)

Ishraq ahmed(ID-1702084)

Anfaj Islam (ID-1702089)

Dipta Dutta(ID-1702122)

Date of submission: ***27-11-19***

Table of content:

Introduction

That is a very good question, with an even better answer! Self-balancing systems can be seen in many places and they are essential for the smooth running of numerous types of machines. Some of the obvious include Segways, bipedal robots and space rockets (A few rockets have been lost due to faulty system).

But what many people don't realise is that these systems use the same controllers that can be seen in servo-motors, air-conditioning units, and even thermostats. Of course rockets use significantly more complex controllers than air-conditioners, but the underlying principle is still the same: how to adjust the system in order to get as close to the desired outcome as possible. That is why building a self-balancing robot is so educational

Equipments:

Arduino Uno

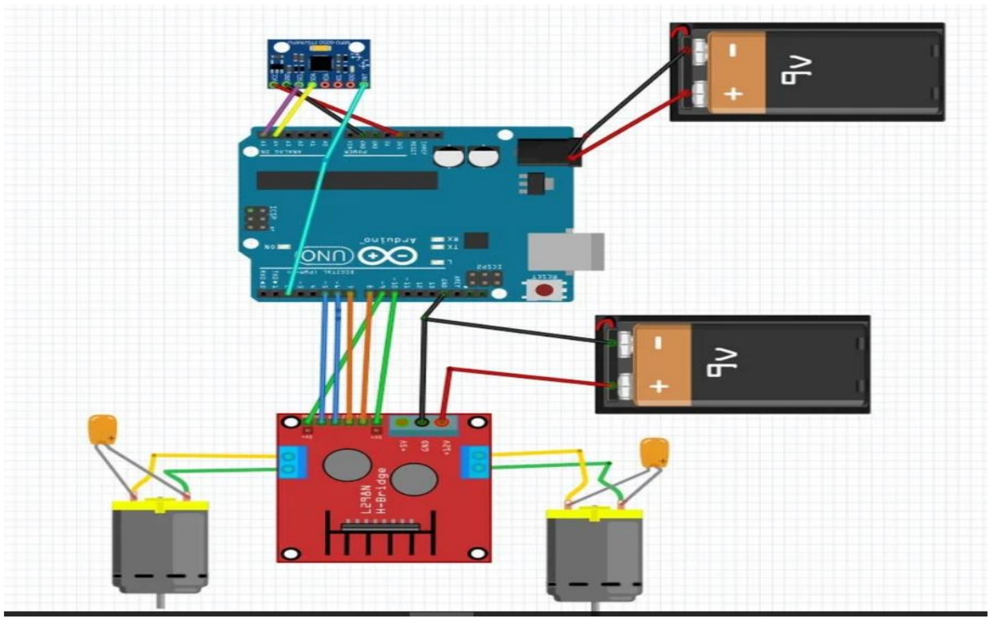
Motor Driver L298N

Gyroscope + Accelerometer Module - MPU6050

Motors, Battery, Jumper Wires and the Chassis.

The Idea is to keep the position of the self-balancing robot upright by countering the forward and backward fall. If the robot starts to fall towards the front we need to get the motors running forward, if it falls backward we need to get the motor running backwards. Something like a Segway bike, when we lean forward bike runs forward.

Setup



Component Description

Motor

One of the most critical and challenging decisions we had to make for this project was selecting the motors. There is a huge variety of motors available for ordering online, but the challenge was finding a set that fit our specific needs for the project. For the robot to properly balance, we knew that we needed motors with high torque and a sufficiently high RPM to allow us to connect the wheels directly to the motor shaft (to simplify the design). A high torque is necessary in order to avoid stalling, and a high enough RPM is required to prevent the motor speed from saturating during normal balancing behavior. Current consumption is also a factor, as we we wanted to be able to power the motors using on-board batteries, and not have to remain tied to a power outlet. As with everything else in this project, we were limited by budget constraints.

Given all of these requirements, we spent a significant amount of time comparing different options for motors before settling for two These motors are rated at between 130 to 150 RPM, with 5.6 lb-in of torque, and requiring about 1.5 A of current at full load. We bought them off of Ebay for about $11 each.

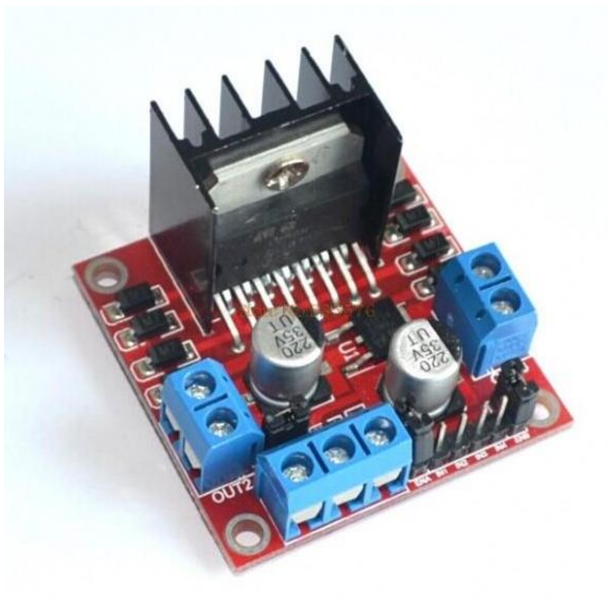
The only problem with these motors was that they did not come with motor brackets, or any way to conveniently attach them to the robot. Because of this, we had to get creative with ways to securely attach them to the chassis and prevent them from sliding around. To solve this problem, we used four from Home Depot, which happened to fit perfectly around the body of the motors, and had two holes that conveniently connected to the bottom platform of the robot with screws and nuts.

Wheel

The wheel diameter was chosen based on the RPM of the motor. We calculated that with 150 RPM, and a wheel diameter of 3", we could get a maximum speed of about 2 ft/s, which is about 0.6 m/s, which seemed like enough, given that we did not entend to drive it around at high speeds. We chose a set of 3" diameter wheels with rubber tires in order to increase the grip on smoother surfaces such as tile floor. We attached the wheels directly to the motor shafts using a set of wheel hubs that we ordered separately

Dual H-Bridge Motor Drivers

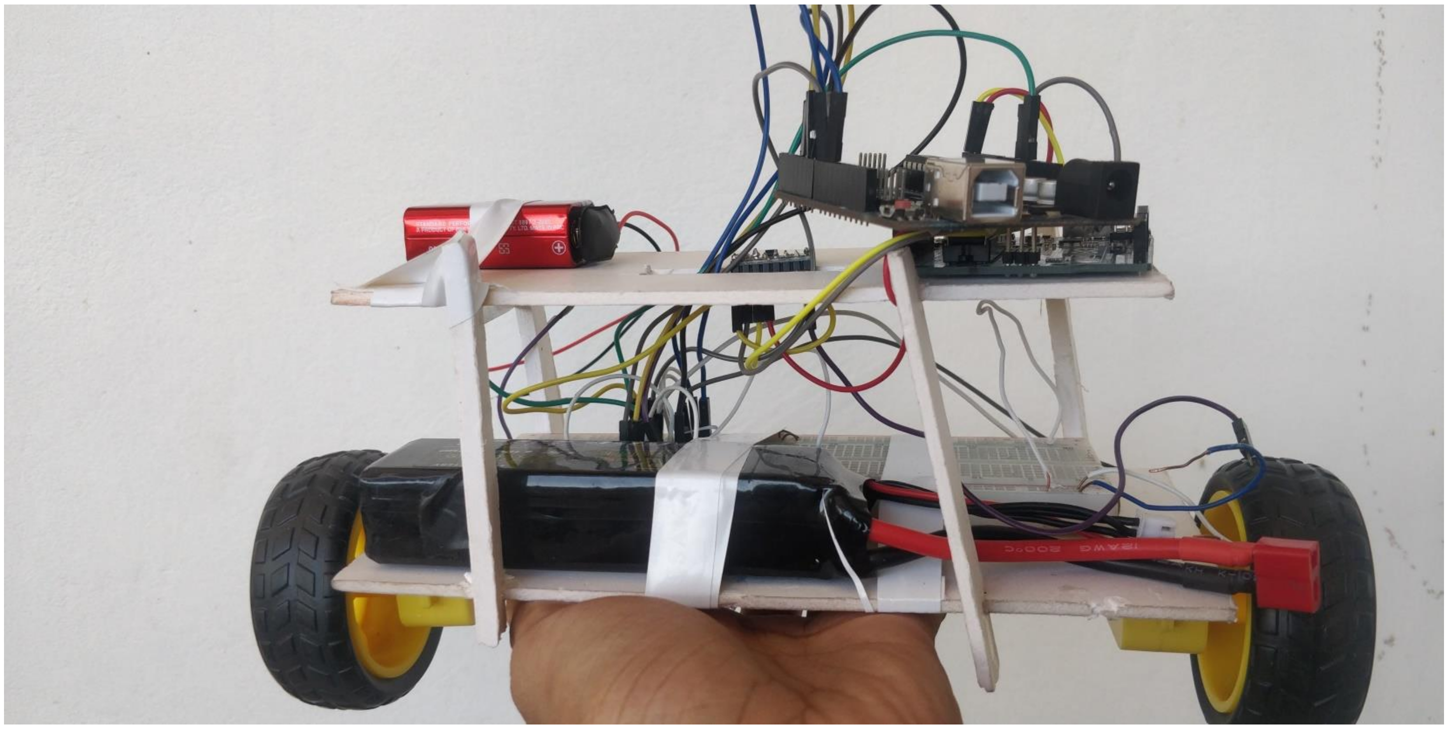
We used two motors that were connected to a dual H-bridge motor driver. We decided to buy this H-bridge device instead of building one because this device is convenient, reliable, and a more compact than what we could have built. The motor driver comes with a heat sink which was very important considering the large current draw of the motors, and the large amount of excess heat.



Power

In order to power our robot, we needed a powere source that could supply 12V and sufficient current to power 2 motors, each of which can require as much as 1.5A at maximum load. Therefore, we started with a 12V 3A wall power adapter that we got from Ebay. The problem was that this power adapter seemed to shut down when the motors jittered or suddenly switched direction at maximum speed. This may have been due to current spike above 3A or voltage spikes eminating from the motors when they switch direction. Because of this, we decided to stop using that power adapter, and switch to a variable power supply in the lab, which can supply much higher current. This fixed our problem, and the motors never shut off again, so this is how we demoed our robot, as can be seen in the video above. We also considered running our robot off of 10 rechargeable NiMH AA batteries in series to produce 12V, but we were not able to test that prior to our demo.

Implimentation of Project:



Description of circuit

Put everything together, does not matter what material we use, all we need to do it secure everything together. We can use MDF boards, with some drills in the corner for the standoffs or spacers to make our self-balancing robot. Few holes in between to secure L298N - motor Driver, and UNO board.

we would recommend putting the motor driver at the bottom close to the motors, then battery finally follower by Arduino UNO and the sensor at the top. You essentially do not have to make 3 layers, 2 should do, however, it is very important to install the MPU-6050 on the top so as go get readings even for a small deflection.

Once you got everything together, it's time to get the electronics working

Here are the connections for the Self Balancing Robot.

Motor Driver ---

IN1 - Arduino PIN 7

IN 2- Arduino PIN 6

IN 3- Arduino PIN 9

IN 4- Arduino PIN 8

ENA - Arduino PIN 5

ENB - Arduino PIN 10

Note - Pin Config might change depending upon the connection of motors to the motor Driver board. If the robot is rotating in axis, try swapping the pins with each other.

MPU6050 -----

INT - Arduino PIN 2

SCL - A5 (Serial Clock)

SDA - A4 (Serial Data)

GND & VCC

The first step is to calibrate MPU-6050 Sensor

Yes, Calibration is required before we start the calibration we need to make sure the sensor is securely placed on the robot. Once the sensor is secured we need to calibrate the sensor. Download this MPUcalibration code to get the offsets which you will require in your main code.

You will get the offset by Running the Serial Monitor.

Once you have downloaded the Library, upload the code to Arduino and obtain the offsets. Once you have the offsets, you are good to go ahead into the next step.

Step 2

Understanding the concept is very important before we try to implement coding for robotics. The idea is to the keep the robot upright by countering the fall and backward fall of the robot by moving the motors forward and backward. The greater the fall is faster the motors should move. The angle of inclination is fed by the sensor and the aim is to maintain the robot verticle.

So now we have a set point maintain all we have to do is move the robot forward and backward to counter the fall. Looks simple but when it comes to implementation it seems almost impossible as the motors will keep moving forward and backward without giving the desired result.

To implement this we have to use something known as PID controller.

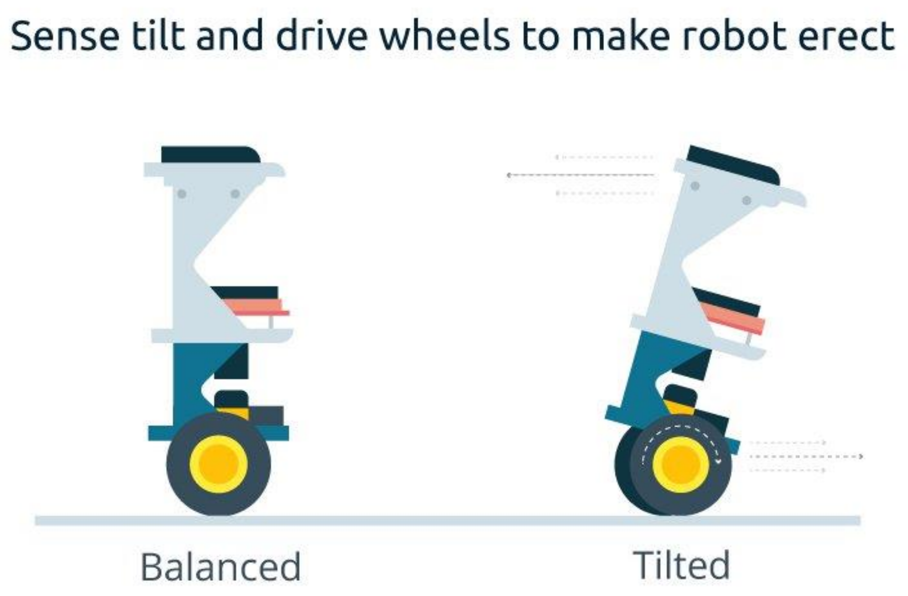
PID Control

```
#include <PID_v1.h>

#include <LMotorController.h>

#include "I2Cdev.h"

#include "MPU6050_6Axis_MotionApps20.h"

#if I2CDEV_IMPLEMENTATION == I2CDEV_ARDUINO_WIRE #include "Wire.h"

#end

if

#define MIN_ABS_SPEED 20 MPU6050 mpu;

// MPU control/status vars bool dmpReady = false;

// set true if DMP init was successful uint8_t mpuIntStatus;

// holds actual interrupt status byte from MPU uint8_t devStatus;

// return status after each device operation (0 = success, !0 = error)

uint16_t packetSize; // expected DMP packet size (default is 42 bytes) uint16_t fifoCount; //

count of all bytes currently in FIFO uint8_t fifoBuffer[64];

// FIFO storage buffer // orientation/motion vars Quaternion q;

// [w, x, y, z] quaternion container VectorFloat gravity;

// [x, y, z] gravity vector float ypr[3];

 // [yaw, pitch, roll] yaw/pitch/roll container and gravity vector //PID double originalSetpoint = 173;

double setpoint = originalSetpoint; double movingAngleOffset = 0.1;

double input, output;

//adjust these values to fit your own design double Kp = 50;
```

```
double Kd = 1.4;
double Ki = 60;
PID pid(&input, &output, &setpoint, Kp, Ki, Kd, DIRECT);
double motorSpeedFactorLeft = 0.6;
double motorSpeedFactorRight = 0.5;
//MOTOR CONTROLLER int ENA = 5;
int IN1 = 6;
int IN2 = 7;
int IN3 = 8; int IN4 = 9; int ENB = 10;
LMotorController motorController(ENA, IN1, IN2, ENB, IN3, IN4,
motorSpeedFactorLeft, motorSpeedFactorRight);
volatile bool mpuInterrupt = false;
// indicates whether MPU interrupt pin has gone high void dmpDataReady()
{
mpuInterrupt = true; }
void setup()
{
// join I2C bus (I2Cdev library doesn't do this automatically)
#if I2CDEV_IMPLEMENTATION == I2CDEV_ARDUINO_WIRE Wire.begin();
TWBR = 24;
// 400kHz I2C clock (200kHz if CPU is 8MHz)
#elif I2CDEV_IMPLEMENTATION == I2CDEV_BUILTIN_FASTWIRE Fastwire::setup(400, true);
#endif mpu.initialize();
devStatus = mpu.dmpInitialize();
// supply your own gyro offsets here, scaled for min sensitivity mpu.setXGyroOffset(220); mpu.setYGyroOffset(76);
mpu.setZGyroOffset(-85);
mpu.setZAccelOffset(1788);
// 1688 factory default for my test chip
// make sure it worked (returns 0 if so) if (devStatus == 0)
{
// turn on the DMP, now that it's ready mpu.setDMPEnabled(true);
// enable Arduino interrupt detection attachInterrupt(0, dmpDataReady, RISING);

mpuIntStatus = mpu.getIntStatus();

// set our DMP Ready flag so the main loop() function knows it's okay to use it dmpReady = true;

// get expected DMP packet size for later comparison packetSize = mpu.dmpGetFIFOPacketSize();

//setup PID pid.SetMode(AUTOMATIC); pid.SetSampleTime(10); pid.SetOutputLimits(-255, 255); }

Else

 { // ERROR! // 1 = initial memory load failed // 2 = DMP configuration updates failed

// (if it's going to break, usually the code will be 1) Serial.print(F("DMP Initialization failed (code "));
Serial.print(devStatus);

Serial.println(F(")")); } } void loop() { // if programming failed, don't try to do anything if (!dmpReady) return;

// wait for MPU interrupt or extra packet(s) available while (!mpuInterrupt && fifoCount < packetSize) { //no mpu data -
performing PID calculations and output to motors pid.Compute();

motorController.move(output, MIN_ABS_SPEED);

 }

// reset interrupt flag and get INT_STATUS byte mpuInterrupt = false; mpuIntStatus = mpu.getIntStatus();

// get current FIFO count fifoCount = mpu.getFIFOCount();

// check for overflow (this should never happen unless our code is too inefficient) if ((mpuIntStatus & 0x10) || fifoCount
== 1024)

{

// reset so we can continue cleanly mpu.resetFIFO();

Serial.println(F("FIFO overflow!"));

// otherwise, check for DMP data ready interrupt (this should happen frequently) \

}

else if

(mpuIntStatus & 0x02)

{

// wait for correct available data length, should be a VERY short wait while (fifoCount < packetSize) fifoCount =
mpu.getFIFOCount(); // read a packet from FIFO mpu.getFIFOBytes(fifoBuffer, packetSize);

// track FIFO count here in case there is > 1 packet available

// (this lets us immediately read more without waiting for an interrupt) fifoCount -= packetSize;
mpu.dmpGetQuaternion(&q, fifoBuffer);

mpu.dmpGetGravity(&gravity, &q);
mpu.dmpGetYawPitchRoll(ypr, &q, &gravity); input = ypr[1] * 180/M_PI + 180;
}
}
```

Method of Balancing

To keep the robot balanced, the motors must counteract the robot falling. This action requires feedback and correcting elements. The feedback element is the MPU6050 gyroscope + accelerometer, which gives both acceleration and rotation in all three axes. The Arduino uses this to know the current orientation of the robot. The correcting element is the motor and wheel combination.



Cost analysis

Name of the Component	Price(TK)
Arduino	450
Motor	500
Motor driver	350
Battery	2000
MPU6050	350
Jumper wire	100
Bread board	90
Wheel	200
Total	3690

Advantages

Having manual control of the PID coefficients via the potentiometer turned out to be an enormous help, because it saved us an immeasureable amount of time while tuning the PID controller. By gradually tuning the PID coefficients, we were able to improve the stability of the system. It turns out that the robot is most stable for settings with very low D coefficients in PID, so our controller is effectively PI. To find the reason why this is the case will require a deeper analysis. After a lot of trial and error, we successfully got our robot to balance. If the gyroscope is properly calibrated, the tilt is zeroed, and the robot is placed gently in an upright position, it stays upright despite disturbances that would normally make it fall over. It is even able to carry a significant load and remain perfectly balanced:

In our project, the robot balanced for about 2 minutes before we turned it off, although it is capable of balancing for significantly longer periods. On a surfaces with high resistance, like carpets, the robot is able to balance more steadily than it does on hard surfaces like floors. We also noticed that raising the center of gravity improves the stability of the robot. That is the reason why we put the batteries (the heaviest part other than the motors) at the very top.

It also became very clear that the type of filturing used on the IMU data has an enormous effect on the stability of the robot. Due to incremental design, we initially had it such that the tilt measurement came from the accelerometer and the rotation rate came from the gyroscope. Both values were low-passed using an IIR filter. The significant latency and laggy response that occured as a result made it impossible for the robot to balance. But the complementary filter we then implemented solved all of the latency issues, and that is how we eventually got the robot to balance.

While the robot performed at least as well, if not better, than we expected, we did notice a few limitations that affected its performance. For example, regardless of how the PID coefficients are tuned, the robot always exhibits a minor wobble. We think this is due to the motors having a minimum PWM duty cycle threshold below which they do not move. That can possibly be compensated for in software to improve stability.

Second of all, while we supplied exactly the same power to both of the wheels simultaneously, one wheel tends to spin noticeably faster than the other at low speeds. This causes the robot to move in somewhat of an arc. The difference seems less noticeable at higher speeds. The robot still balances, so we did not spend too much time looking into this issue.

A third issue has to do with a design choice that we made. Due to budget limitations, we decided to not add motor encoders to our robot. Encoders are not inherently necessary for self-balancing, but they would would have allowed us to directly measure and control the speed of the robot. When testing our robot, we noticed that, for various reasons, it sometimes begins to accumulate speed. It remains balanced until it reaches a point where the motor speed saturates, the wheels cannot keep up with the body of the robot, and it falls over. Having faster motors could have helped, but the issue here is the lack of encoders. With encoders, we could have prevented the robot from approaching speeds that it cannot handle, and even controlled the robot such that it stayed in place rather than roaming around with no speed control.

Usability

We made a significant effort to increase the useability of this robot, both for ourselves to aid in debugging, and for the general user. The potentiometer PID coefficient control and the LCD display are two examples of ways to aid in debugging. By controlling the PID coefficients via potentiometer, we were able to make rapid iterations and test them immediately, without having to worry about reprogramming the robot. The LCD display, which showed us the coefficient values for PID, allowed us to record the ones that worked so that we could replicate them in the future.

In terms of general useability, we tried to make operating this robot as simple as possible. The user simply has to connect the robot to power, and then flip two switches to the ON position - one to power up the MCU and the other to power the motors. It is recommended that the robot lay down on a flat surface as the MCU is powered on, because that allows it to successfully complete the gyroscope calibration procedure before it begins operation. To calibrate the robot for an uneven load, simply press the tilt-zero button in order to zero the desired tilt to the current tilt of the robot, to compensate for uneven loads. While the LCD and potentiometers are very useful for development purposes, they can be completely removed from the product when given to an everyday user. Once the robot is tuned properly, there is simply no use for these parts anymore.

Safety

Because of the way the chassis is designed, if the robot falls over, the wheels are raised off the ground, and the robot becomes immobile until it is again placed upright. As an additional precaution, we could have also had an automatic motor shutoff past a certain tilt angle, but we never took the time to implement it. But regardless, this device is not a toy and has powerful motors and batteries. It should be used by kids only under adult supervision. Always use in a dry environment, and never expose the robot to water or moisture. Always remove the batteries before storing for extended periods of time.

Future Improvement

1) Better motors can be used
2) Structure should be improved
3) Surface should be better to run it properly
4) Length should be set in that way so it can balance perfectly
5) Mechanical instruments should be more efficient
6) The program we used can be made for effective

Conclusion

This project needs a lot of improvement. Though we tried best, there were errors in building this self-balancing robot. It needs little clarification that this project will be a great use for us if we can utilize it properly. To recapitulate, we can make this project more successful if we can reduce the errors.

Acknowledgement

The authors would like to thank Department of Electrical and Electronic Engineering (EEE) of Chittagong University of Engineering and Technology (CUET), Chattogram – 4349, Bangladesh