Zaghlol

Self-balancing robot and obstacle detection.

9December, 2017

Abdelrahman Osama ID: 34-17114 Tutorial No.16
Amr Aly ID: 34-04506 Tutorial No.17
Karl Maged ID: 34-484 Tutorial No.12
Mohamed El-Hinamy ID: 34-7712 Tutorial No.16
Salma Youssef ID: 34-0276 Tutorial No.19

youtube: https://www.youtube.com/watch?v=9vASip3uZww&feature=share
Google Drive: https://drive.google.com/open?id=1VxlNDw6J7XIJVwcwYrAh2YhL6JZeUO1U

Problem description

The problem of the balancing robot is essentially solving the classic inverted pendulum problem.

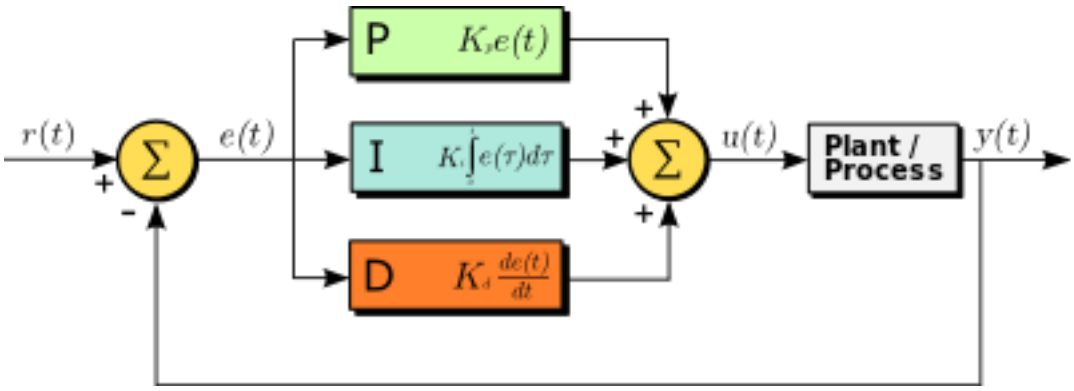
The goal of the control loop is to adjust the wheels' position so that the inclination angle remains stable at a pre-determined value (e.g. the angle when the robot is balanced). When the robot starts to fall in one direction, the wheels should move in the falling direction to correct the inclination angle.

Solving the problem

we divided the problem into two subproblems. First is to balance a two-wheeled inverted pendulum robot it is necessary to have accurate information of the current tilt angle from using a measurement unit. Furthermore a controller needs to be implemented to compensate tilt.

To drive the motors we need some information on the state of the robot. We need to know the direction in which the robot is falling, how much the robot has tilted and the speed with which it is falling. All these information can be deduced from the readings obtained from MPU6050.

To calculate the amount of force is needed to drive the motors to balance the robot. These calculations are done using PID controller.



Implementation

The first challenge was constructing the chassis because we needed a light material therefore we've used an acrylic sheet and designed the parts of the chassis using solid works. The robot is built on three layers of acrylic that are spaced 8 cm. The bottom layer contains a battery, two motors, and L298N driver controller. The middle layer contains the Arduino. The top layer contains MPU-6050, IR proximity sensor, and a buzzer.

The second challenge was calibrating the MPU-6050 and we solved it by

1. Put the MPU6050 in a flat and horizontal surface. Use an inclinometer to check that it is as horizontal as possible.
2. Modify the RAW program to put every offset to 0. ("setXGyroOffset/ setYGyroOffset/setZGyroOffset/setZAccelOffset" =0).
3. - Upload the RAW program to your arduino and open serial monitor so you can see the values it is returning.
4. leaving it operating for a few minutes (5-10) so temperature gets stabilized.
5. Check the values in the serial monitor and write them down.
6. - Now modify your program again updating your offsets and run the sketch, with updated offsets.
7. - Repeat this process until your program is returning 0 for every gyro, 0 for X and Y accel, and +16384 for Z accel.

Finally we've got the accurate values for the gyroscope offset.

The third challenge we faced was determining the PID values to and we obtained them by using Ziegler Nicholas method. PID stands for Proportional, Integral, and Derivative. Each of these terms provides a unique response to our self-balancing robot.

The **proportional** term, as its name suggests, generates a response that is proportional to the error. For our system, the error is the angle of inclination of the robot.

The **integral** term generates a response based on the accumulated error. This is essentially the sum of all the errors multiplied by the sampling period. This is a response based on the behavior of the system in past.

The **derivative** term is proportional to the derivative of the error. This is the difference between the current error and the previous error divided by the sampling period. This acts as a predictive term that responds to how the robot might behave in the next sampling loop.

Ziegler–Nichols method [edit]

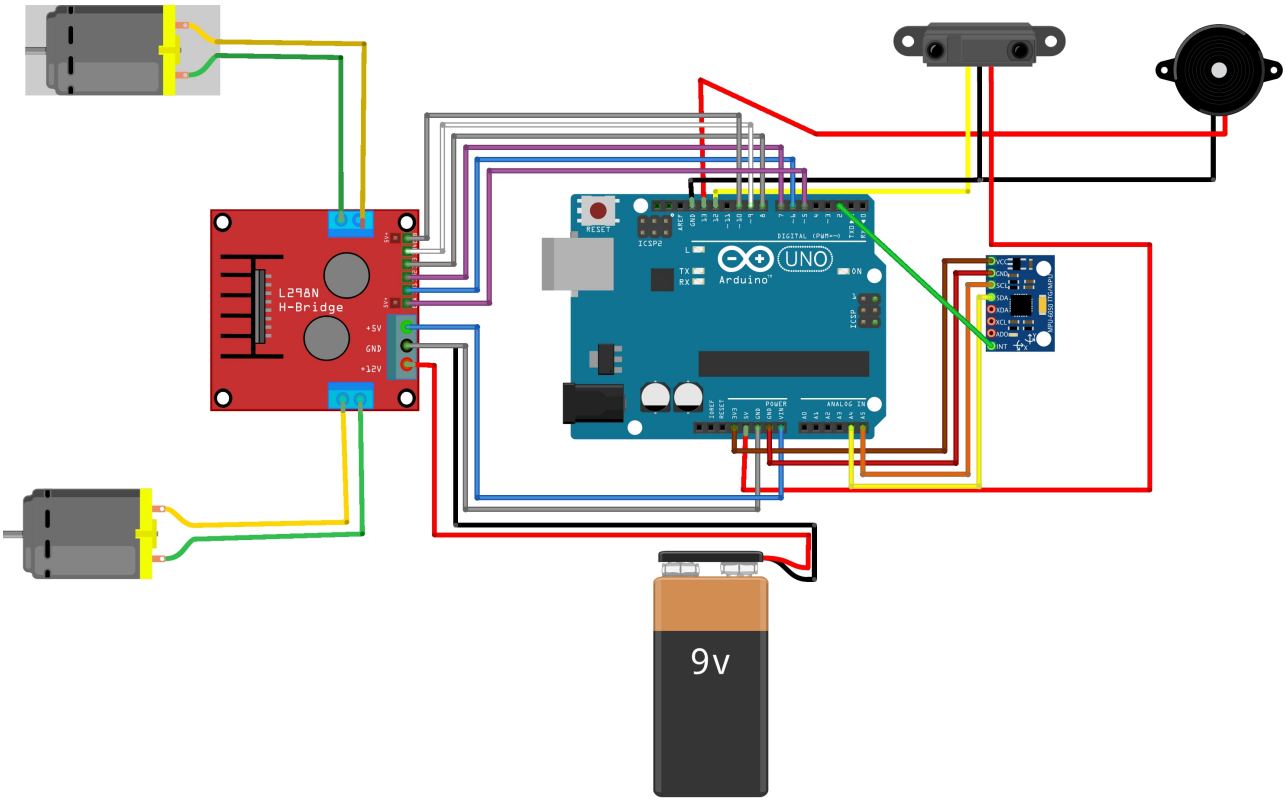
For more details on this topic, see Ziegler–Nichols method.

Another heuristic tuning method is formally known as the Ziegler–Nichols method, introduced by John G. Ziegler and Nathaniel B. Nichols in the 1940s. As in the method above, the K_i and K_d gains are first set to zero. The proportional gain is increased until it reaches the ultimate gain, K_u, at which the output of the loop starts to oscillate. K_u and the oscillation period T_u are used to set the gains as shown:

Ziegler–Nichols method

Control Type	K_p	K_i	K_d
P	$0.50K_u$	-	-
PI	$0.45K_u$	$1.2K_p/T_u$	-
PID	$0.60K_u$	$2K_p/T_u$	$K_pT_u/8$

Circuit diagram



Project

The chassis was made by Salma Youssef.
L298N motor module: Karl Maged.
Calibrating the MPU-6050 sensor: Abdelrahman Osama.
Controlling and balancing: Amr Aly.
Obstacle detection System: Mohamed Amr.

Source Code

```
#include <PID_v1.h>
#include <LMotorController.h>
#include "I2Cdev.h"
#include "MPU6050_6Axis_MotionApps20.h"

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

#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 = 175.8;
double originalSetpoint = 182.5 ;
double setpoint = originalSetpoint;
double movingAngleOffset = 0.1;
double input, output;
int moveState=0; //0 = balance; 1 = back; 2 = forth


double Kp = 330;
double Kd = 15;
// kd = 11 || 15; // is the right value
double Ki = 140;
PID pid(&input, &output, &setpoint, Kp, Ki, Kd, DIRECT);

double motorSpeedFactorLeft = 0.55;
```

```
double motorSpeedFactorRight = 0.4;
//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);

// IR
int irOut = 12;
//Buzzer
int buzzerIn = 13;

//timers
long time1Hz = 0;
long time5Hz = 0;

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

   // initialize serial communication
   // (115200 chosen because it is required for Teapot Demo output, but it's
   // really up to you depending on your project)
   Serial.begin(115200);
   while (!Serial); // wait for Leonardo enumeration, others continue immediately
```

```
    // initialize device
    Serial.println(F("Initializing I2C devices..."));
    mpu.initialize();

    // verify connection
    Serial.println(F("Testing device connections..."));
    Serial.println(mpu.testConnection() ? F("MPU6050 connection successful") :
F("MPU6050 connection failed"));

    // load and configure the DMP
    Serial.println(F("Initializing DMP..."));
    devStatus = mpu.dmpInitialize();

    // supply your own gyro offsets here, scaled for min sensitivity
    mpu.setXGyroOffset(100);
    mpu.setYGyroOffset(20);
    mpu.setZGyroOffset(10);
    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
        Serial.println(F("Enabling DMP..."));
        mpu.setDMPEnabled(true);

        // enable Arduino interrupt detection
        Serial.println(F("Enabling interrupt detection (Arduino external interrupt 0)..."));
        attachInterrupt(0, dmpDataReady, RISING);
        mpuIntStatus = mpu.getIntStatus();

        // set our DMP Ready flag so the main loop() function knows it's okay to use it
        Serial.println(F("DMP ready! Waiting for first interrupt..."));
        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(")"));
    }

    //IR
    pinMode(irOut,INPUT);
    //Buzzer
    pinMode(buzzerIn,OUTPUT);

}


void loop()
{

  if(digitalRead(irOut)==HIGH)
  {
    digitalWrite(buzzerIn,HIGH);
  }
  else
  {
    digitalWrite(buzzerIn,LOW);
  }

    // 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();
//        Serial.print("K:");
//        Serial.println(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);
        #if LOG_INPUT
            Serial.print("ypr\t");
            Serial.print(ypr[0] * 180/M_PI);
            Serial.print("\t");
            Serial.print(ypr[1] * 180/M_PI);
            Serial.print("\t");
            Serial.println(ypr[2] * 180/M_PI);
        #endif
        input = ypr[1] * 180/M_PI + 180;
    }
}
// END OF THE CODE
```