# Reaction wheel Inverted Pendulum

(Team 56)

**Authored by**:

| | |
|---|---|
| Mohamed Sabry | 37-18341 |
| Mai Mira | 37-1120 |
| Mohamed Mokhtar | 37-12195 |

**Supervised by**: Prof. Ayman El-Badawy.

**Date**: 6/5/2018

## Table of contents:

**1.** Design summary

i. Summary of the design Page 3

ii. Design components Page 3 & 4

**2.** System details

i. Equation of motion Page 5 & 6

ii. System analysis and performance Page 6, 7 & 8

iii. Brief description of mechanism Page 8 & 9

iv. Components Page 9, 10, 11 & 12

v. Circuit schematic Page 12

vi. A brief flow chart Page 13

**3.** Design Evaluation Page 13

**4.** Partial parts list Page 14, 15 & 16

**5.** Appendix

## Design Summary:

- This project aims to use the concept of closed loop feedback to control a reaction wheel based inverted pendulum, it works depending on the reaction torque produced on the wheel and system from the torque produced by the motor, this torque is used as an input to our system to track a predefined reference (angle 0 in our case) to balance the pendulum and reach equilibrium.

## Design Components:

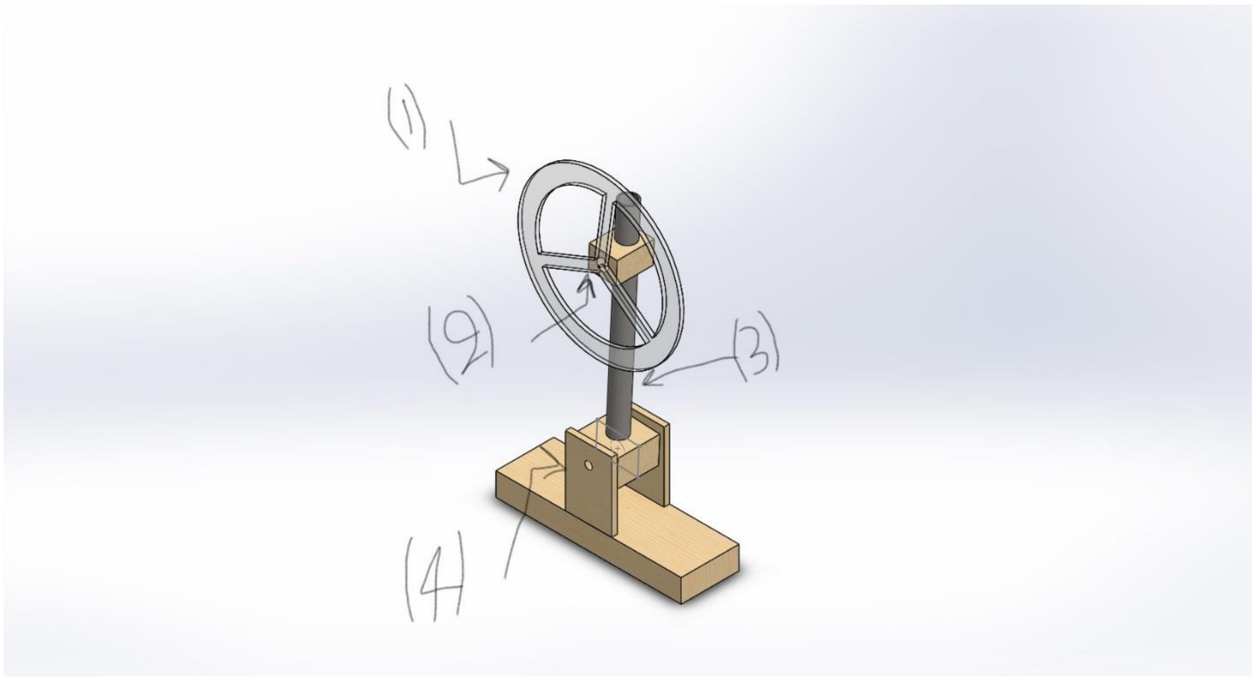


*Figure 1 - Design components and mechanism*

1. 4-mm acrylic reaction wheel
2. 12 volt DC motor and the wheel mounted on it
3. A 450 mm stainless steel tube used as our pendulum
4. A wooden base of a stainless steel rod and bearings used as our hinge of rotation for the motion mechanism

The actual mechanism picture will be placed here

## System Details:

The equation of motion of the system was derived in our analysis using two methods to ensure that our system is verified enough and correctly:

1. Using the Euler-Lagrange equation which is basically a differential equation depending on the kinetic and potential energy of the system, then representing the system in State-Space form to further verify its states and initial conditions (if any), the transfer function calculated was a relation between the input voltage to the DC motor and the actual angle of the system

$$\frac{\theta}{V} = \frac{\dfrac{-Kt}{(mplp^2+4mwlp^2+Ip)Ra} + \dfrac{\dfrac{Kt*Ke}{(mplp^2+4mwlp^2+Ip)Ra}*\dfrac{Kt}{Ra}\left(\dfrac{1}{Iw}+\dfrac{1}{(mplp^2+4mwlp^2+Ip)}\right)*S}{S^2+\dfrac{Kt*Ke}{Ra}*\left(\dfrac{1}{Iw}+\dfrac{1}{(mplp^2+4mwlp^2+Ip)}\right)*S}}{S^2-\dfrac{g*lp(mp+2mw)}{mplp^2+4mwlp^2+Ip}+\dfrac{\dfrac{Kt*Ke}{(mplp^2+4mwlp^2+Ip)Ra}*\dfrac{g*lp(mp+2mw)}{mplp^2+4mwlp^2+Ip}*S}{S^2+\dfrac{Kt*Ke}{Ra}*\left(\dfrac{1}{Iw}+\dfrac{1}{(mplp^2+4mwlp^2+Ip)}\right)*S}}$$

2. Using simscape tool in Matlab which helped us derive the system EOM directly from the solidworks design without the need of mathematical analysis like the first method “Simscape is a tool used to import solid parts from solidworks, analyses it and creates the proper block diagram with the proper joints, parts and frames of reference” , it resulted in an open loop transfer function where the input is the torque on the system and the output is the actual angle of the system, the motor was not considered as a part of the transfer function as the previous method, so a separate transfer function for the motor was calculated:

$$\frac{T}{V} = \frac{S*Kt*Iw}{S*Ra*Iw+Kt*Ke}$$

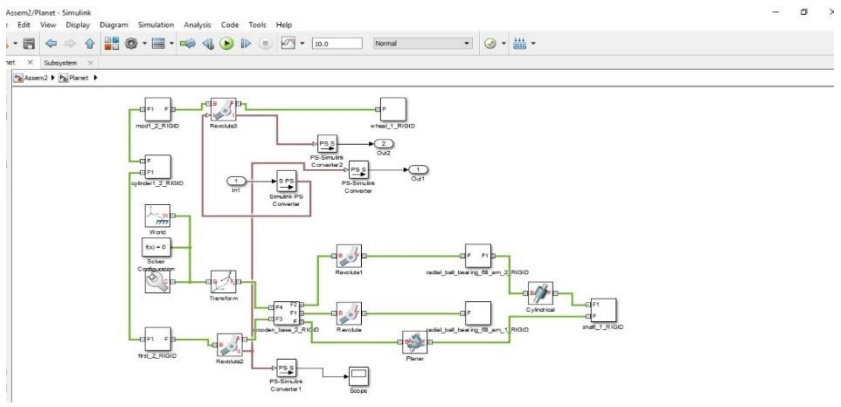
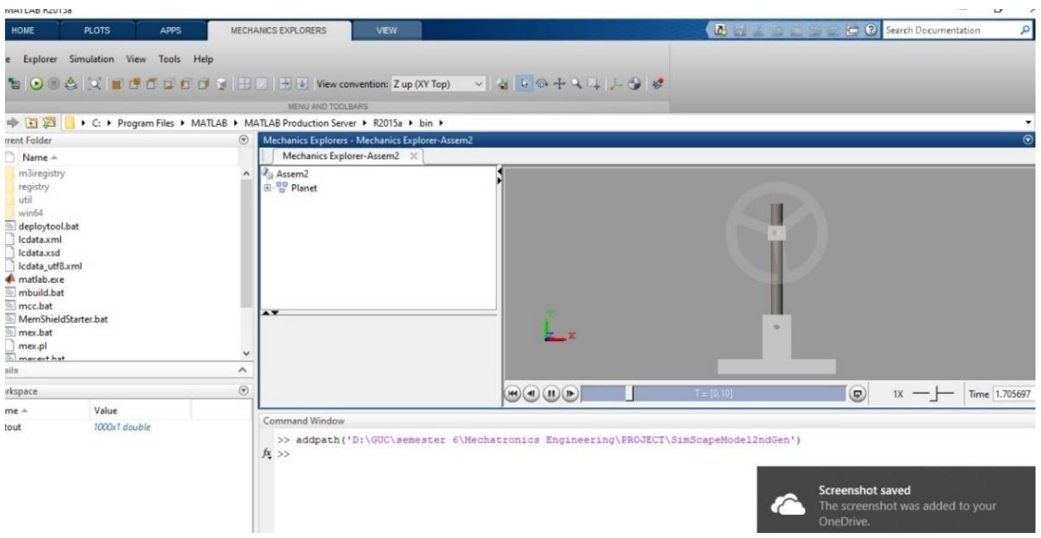


*Figure 2 - Block diagram after importing into simscape*



```
>> addpath('D:\GUC\semester 6\Mechatronics Engineering\PROJECT\SimScapeModel2ndGen')
>>
```

Screenshot saved
The screenshot was added to your OneDrive.

*Figure 3 - The system in matlab simulation tool*

## Simulation and analysis:

After using the previously mentioned methods to get the transfer function, we imported our transfer function into Simulink and created the following block diagram:

- Using the Lagrange equation to get the EOM, we were able to construct the following closed loop system "treating our sensor as a unity feedback" :

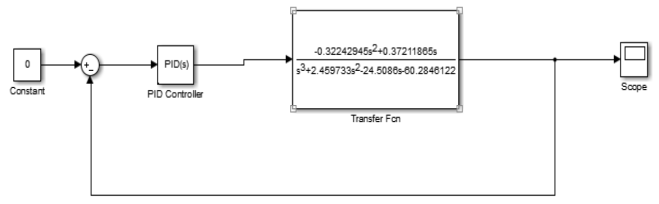


Figure 4 - Closed loop system

- Investigating the system's response to the loop after inserting an initial condition to our system to respond to (an angle of 15 degrees) and using Matlab auto tune tool for the PID block to get the proper gains for a proper performance, we got the following response which satisfied our intentions for the project:

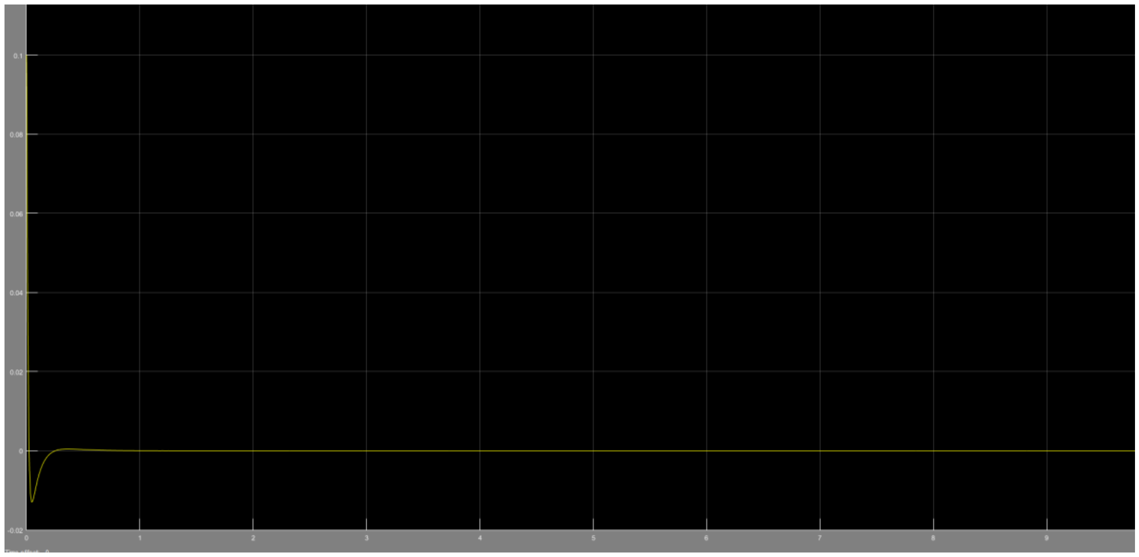
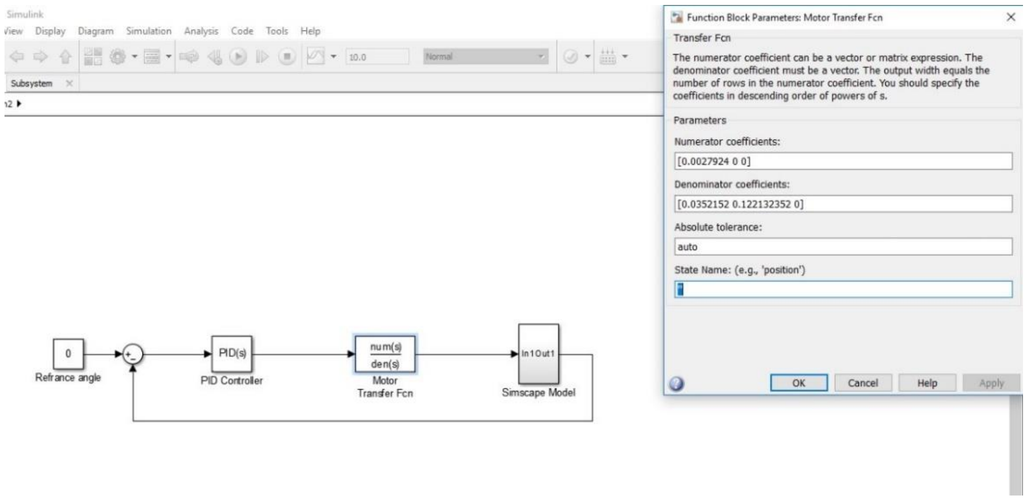


Figure 5 - Closed loop response

- Using simscape to create our transfer function from the solidworks design was no different in the response we got, the loop was created after suppressing our system's block diagram into a subsystem and using it as a black box in our simulation, we used the calculated transfer function of the

DC motor as the transfer function produced by the simscape tool provided the input to be the torque of the motor and the output is the actual angle of the system:



Function Block Parameters: Motor Transfer Fcn

Transfer Fcn

The numerator coefficient can be a vector or matrix expression. The denominator coefficient must be a vector. The output width equals the number of rows in the numerator coefficient. You should specify the coefficients in descending order of powers of s.

Parameters

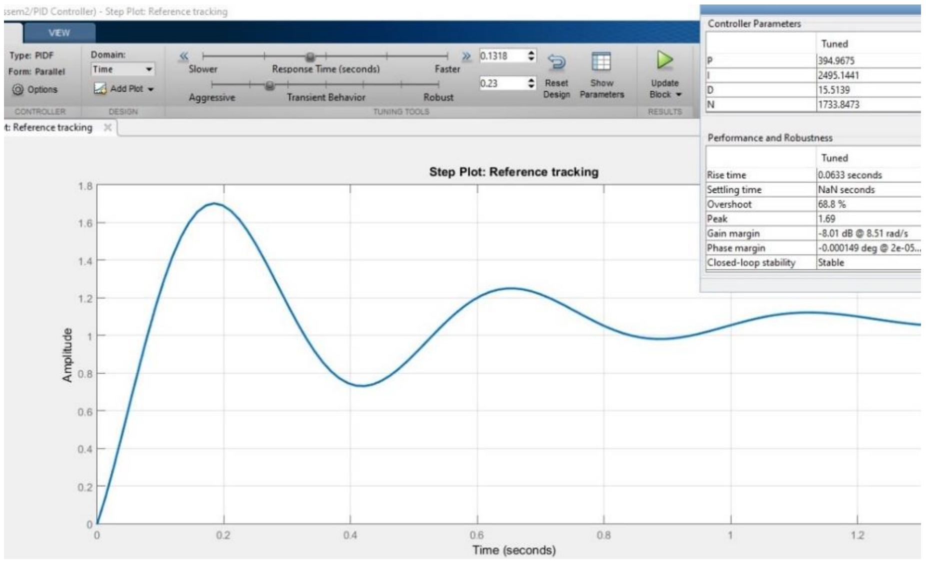
Numerator coefficients: [0.0027924 0 0]

Denominator coefficients: [0.0352152 0.122132352 0]

Absolute tolerance: auto

State Name: (e.g., 'position')

*Figure 6 - Closed loop system2*



Controller Parameters

| | Tuned |
|---|---|
| P | 394.9675 |
| I | 2495.1441 |
| D | 15.5139 |
| N | 1733.8473 |

Performance and Robustness

| | Tuned |
|---|---|
| Rise time | 0.0633 seconds |
| Settling time | NaN seconds |
| Overshoot | 68.8 % |
| Peak | 1.69 |
| Gain margin | -8.01 dB @ 8.51 rad/s |
| Phase margin | -0.000149 deg @ 2e-05... |
| Closed-loop stability | Stable |

*Figure 7 - auto tuning for gains and the closed loop response*

## A brief description of mechanism:

The system is a simple inverted pendulum pivoted on a stainless steel rod and two ball bearings to support rotational motion due to reaction torque attained from the motor when it rotates the wheel, the DC motor provides a torque via the reaction wheel mounted on it mechanically and the systems exhibits a reaction from newton's third law (for every action there's a reaction equal in magnitude and opposite in direction) and that reaction is the source of actuation of the system.

## Components:

1. A 12 volts DC motor with 250 rated RPM and 8.8 Kgf.cm rated torque: it's our project's actuator and the source of the torque that balances our project.

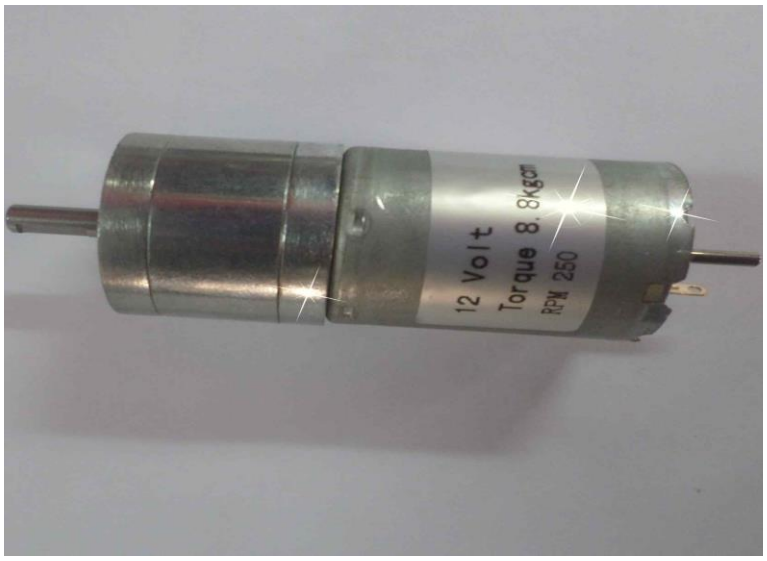


*Figure 8 - DC motor*

2. ***<u>IMU sensor (Accelerometer + Gyroscope + temperature sensor) MPU 6050:</u>*** It's the feedback of our project, it calculates the angle of tilting of the pendulum and sends it to the arduino in order to process the error with respect to our reference angle (0 in our case)

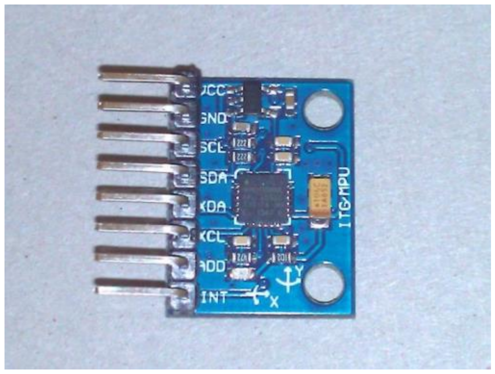


*Figure 9 - MPU 6050 sensor*

3. *__H-Bridge L298N module to drive the DC motor__*: It controls our motor via receiving the PWM signal from the arduino board and produces the proper voltage to rotate the motor with the desired speed.

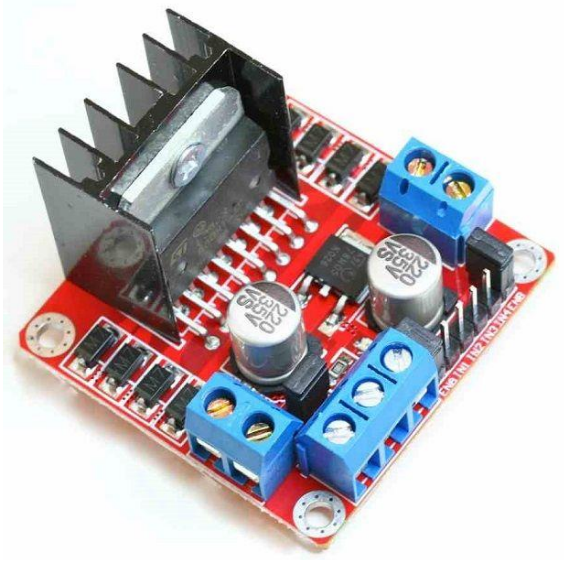


*Figure 10 - DC motor driver*

4. *__Arduino uno R3 board__*: The controller and brain of the project, it receives input from the sensor, evaluates the error and produces the output control signal to our motor.

*Figure 11 - Arduino uno*

## Circuit schematic:

The circuit schematic was created with “Fritzing”, a software used for creating circuit schematics and PCB layouts.

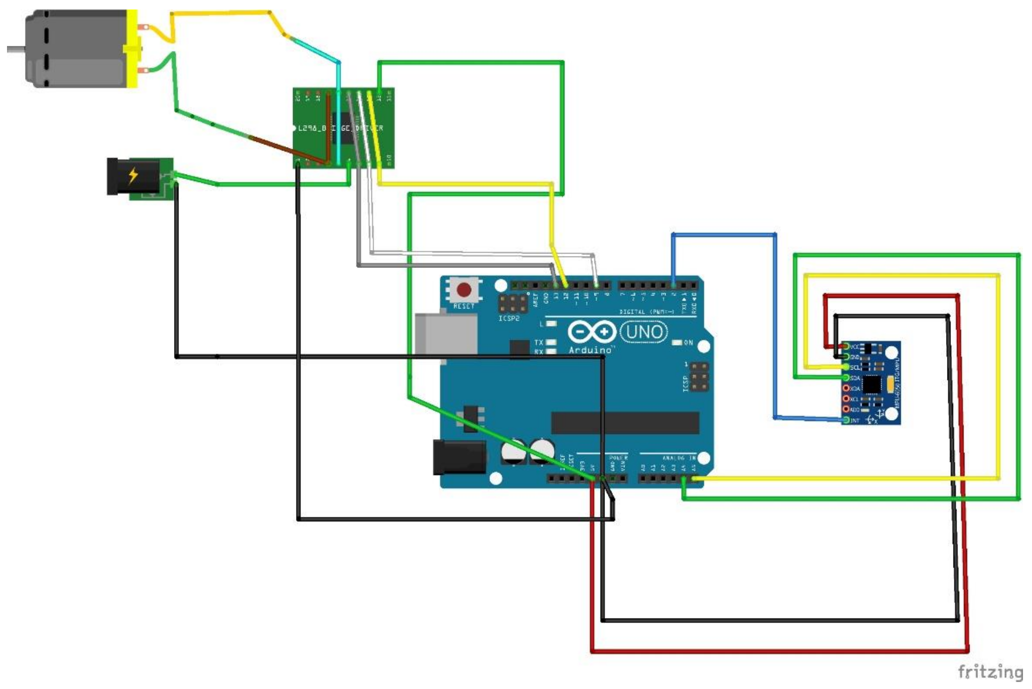


*Figure 12 - Circuit Schematic*

## A brief flowchart describing our control logic:

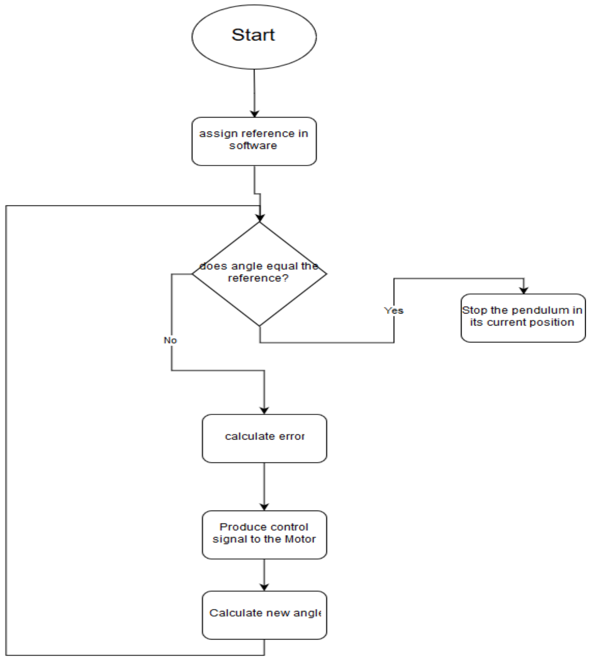

Start
assign reference in software
does angle equal the reference?
Yes
No
Stop the pendulum in its current position
calculate error
Produce control signal to the Motor
Calculate new angle

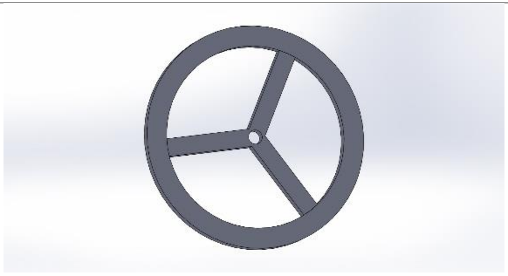
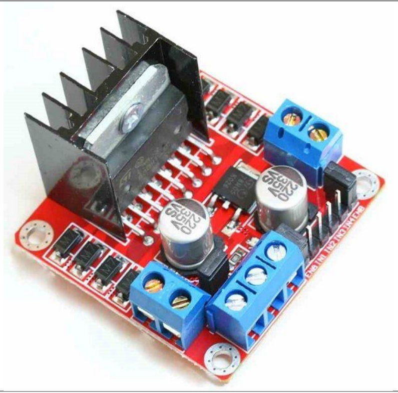
*Figure 13 – Flowchart*

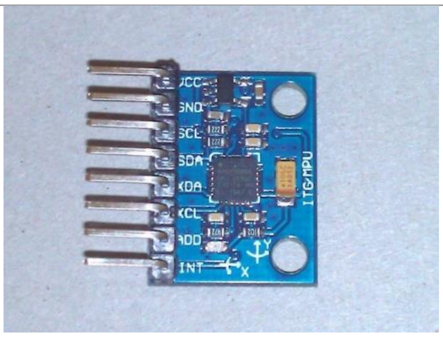
## Design evaluation:

- After tuning the PID parameters and due to the fluctuations in the reading of the sensor, we had to go with the trial and error to adjust our gains a bit more or less of what we got from modelling and simulation on matlab, the system responded aggressively but in an accepted manner, it balanced itself for a while reaching the zero position and suffering from fluctuations about the zero position as the performance produced by the matlab simulations showed, so the performance was a success in overall

## Partial parts list:

| Part Name | Image | Price | Brief description |
|---|---|---|---|
| *Ardunio UNO R3* | | 150 L.E. | The arduino board is the brain and controller of our project, the software is uploaded on it, it processes the input signal, calculates he error and produces the proper control signal to control the system |

| | | | |
|---|---|---|---|
| ***<u>DC Motor</u>*** | | 250 L.E. | It’s the actuator that moves our project, it’s a 12V rating DC motor with a 34:1 gear box and produces a rated torque of 8.8 kgff.cm and 250 RPM at its rated operating point, it receives varying voltage depending on the error and moves with a specific speed to produce the intended torque via the reaction wheel attached to it. |
| ***<u>Reaction wheel</u>*** |  | 100 L.E. | It’s the source of the reaction torque that will move the system when the motor exerts a specific torque on it. |
| ***<u>H-Bridge driver</u>*** |  | 35 L.E. | It receives the control signal from the arduino and produce the proper voltage via the output port that the motor is |

| | | | |
|---|---|---|---|
| | | | connected to and cause it to rotate, it also receives the direction commands from the arduino via the input port to control the direction of rotation of the motor |
| *<u>MPU-6050 sensor</u>* |  | 70 L.E. | It's the sensor responsible for the feedback in our project and is the one calculating the current tilt angle and inputs it to the arduino to process as previously mentioned |

## Appendix:

- First trial code:

```
#include <Wire.h>
#include "Kalman.h" // Source:
https://github.com/TKJElectronics/KalmanFilter
//#include "MPU6050_6Axis_MotionApps20.h"
#define RESTRICT_PITCH // Comment out to restrict roll to ±90deg instead -
please read:
http://www.freescale.com/files/sensors/doc/app_note/AN3461.pdf
//MPU6050 mpu;
Kalman kalmanX; // Create the Kalman instances
Kalman kalmanY;

/* IMU Data */
double accX, accY, accZ;
double gyroX, gyroY, gyroZ;
int16_t tempRaw;

double gyroXangle, gyroYangle; // Angle calculate using the gyro only
double compAngleX, compAngleY; // Calculated angle using a
complementary filter
double kalAngleX, kalAngleY; // Calculated angle using a Kalman filter

uint32_t timer;
uint8_t i2cData[14]; // Buffer for I2C data

/* PID data */
//The motor parameters

int Pwm = 9;
int int1 = 12;
int int2 = 13;

//Define Variables we'll be connecting to
```

```
double Setpoint;

//Specify the links and initial tuning parameters
double Kp=-5795.6724490643, Ki=-22671.6412364561 , Kd=-
363.798837982102;

float delta_t = 0;                                          //initialize sampling time
float time_old = micros() ;
float error_old = 0;
float diff_term = 0;                                         //derivartive term
float int_term = 0;                                          //inegrator term
float agresssive;                                           //set agressive input limit
int guard;

void setup() {
  Serial.begin(115200);
  Wire.begin();
  TWBR = ((F_CPU / 400000L) - 16) / 2; // Set I2C frequency to 400kHz

  i2cData[0] = 7; // Set the sample rate to 1000Hz - 8kHz/(7+1) = 1000Hz
  i2cData[1] = 0x00; // Disable FSYNC and set 260 Hz Acc filtering, 256 Hz
Gyro filtering, 8 KHz sampling
  i2cData[2] = 0x00; // Set Gyro Full Scale Range to ±250deg/s
  i2cData[3] = 0x00; // Set Accelerometer Full Scale Range to ±2g
  while (i2cWrite(0x19, i2cData, 4, false)); // Write to all four registers at
once
  while (i2cWrite(0x6B, 0x01, true)); // PLL with X axis gyroscope reference
and disable sleep mode

  while (i2cRead(0x75, i2cData, 1));
  if (i2cData[0] != 0x68) { // Read "WHO_AM_I" register
    Serial.print(F("Error reading sensor"));
    while (1);
  }
  delay(100); // Wait for sensor to stabilize

  /* Set kalman and gyro starting angle */
  while (i2cRead(0x3B, i2cData, 6));
  accX = (i2cData[0] << 8) | i2cData[1];
  accY = (i2cData[2] << 8) | i2cData[3];
  accZ = (i2cData[4] << 8) | i2cData[5];

 accX=-3671;
 accY=1518;
 accZ=4130;
 gyroX=145;
 gyroY=2;
 gyroZ=36;


  // Source:
http://www.freescale.com/files/sensors/doc/app_note/AN3461.pdf eq. 25
and eq. 26
  // atan2 outputs the value of -π to π (radians) - see
http://en.wikipedia.org/wiki/Atan2
  // It is then converted from radians to degrees
#ifdef RESTRICT_PITCH // Eq. 25 and 26
  double roll  = atan2(accY, accZ); //the output of this function is radians
  double pitch = atan(-accX / sqrt(accY * accY + accZ * accZ)); //the output
of this function is radians
#else // Eq. 28 and 29
  double roll  = atan(accY / sqrt(accX * accX + accZ * accZ)); //the output of
this function is radians
  double pitch = atan2(-accX, accZ) * RAD_TO_DEG;
#endif

  kalmanX.setAngle(roll); // Set starting angle
  kalmanY.setAngle(pitch);
  gyroXangle = roll;
  gyroYangle = pitch;
  compAngleX = roll;
  compAngleY = pitch;


  //Turn PID on and ser your reference angle in radians

  Setpoint = 0;

  timer = micros();
}

void loop() {
  /* Update all the values */
  while (i2cRead(0x3B, i2cData, 14));
  accX = ((i2cData[0] << 8) | i2cData[1]);
  accY = ((i2cData[2] << 8) | i2cData[3]);
  accZ = ((i2cData[4] << 8) | i2cData[5]);
  tempRaw = (i2cData[6] << 8) | i2cData[7];
  gyroX = (i2cData[8] << 8) | i2cData[9];
  gyroY = (i2cData[10] << 8) | i2cData[11];
  gyroZ = (i2cData[12] << 8) | i2cData[13];

  double dt = (double)(micros() - timer) / 1000000; // Calculate delta time
  timer = micros();

  // Source:
http://www.freescale.com/files/sensors/doc/app_note/AN3461.pdf eq. 25
and eq. 26
  // atan2 outputs the value of -π to π (radians) - see
http://en.wikipedia.org/wiki/Atan2
  // It is then converted from radians to degrees
#ifdef RESTRICT_PITCH // Eq. 25 and 26
  double roll  = atan2(accY, accZ) * RAD_TO_DEG;
  double pitch = atan(-accX / sqrt(accY * accY + accZ * accZ)) *
RAD_TO_DEG;
#else // Eq. 28 and 29
  double roll  = atan(accY / sqrt(accX * accX + accZ * accZ)) * RAD_TO_DEG;
  double pitch = atan2(-accX, accZ) * RAD_TO_DEG;
#endif

  double gyroXrate = gyroX / 131.0; // Convert to deg/s
  double gyroYrate = gyroY / 131.0; // Convert to deg/s

#ifdef RESTRICT_PITCH
  // This fixes the transition problem when the accelerometer angle jumps
between -180 and 180 degrees
  if ((roll < -90 && kalAngleX > 90) || (roll > 90 && kalAngleX < -90)) {
    kalmanX.setAngle(roll);
    compAngleX = roll;
    kalAngleX = roll;
    gyroXangle = roll;
  } else
    kalAngleX = kalmanX.getAngle(roll, gyroXrate, dt); // Calculate the angle
using a Kalman filter

  if (abs(kalAngleX) > 90)
    gyroYrate = -gyroYrate; // Invert rate, so it fits the restriced
accelerometer reading
  kalAngleY = kalmanY.getAngle(pitch, gyroYrate, dt);
#else
  // This fixes the transition problem when the accelerometer angle jumps
between -180 and 180 degrees
  if ((pitch < -90 && kalAngleY > 90) || (pitch > 90 && kalAngleY < -90)) {
    kalmanY.setAngle(pitch);
    compAngleY = pitch;
    kalAngleY = pitch;
    gyroYangle = pitch;
  } else
```

```
    kalAngleY = kalmanY.getAngle(pitch, gyroYrate, dt); // Calculate the angle
using a Kalman filter

  if (abs(kalAngleY) > 90)
    gyroXrate = -gyroXrate; // Invert rate, so it fits the restriced
accelerometer reading
  kalAngleX = kalmanX.getAngle(roll, gyroXrate, dt); // Calculate the angle
using a Kalman filter
#endif

  gyroXangle += gyroXrate * dt; // Calculate gyro angle without any filter
  gyroYangle += gyroYrate * dt;
  gyroXangle += kalmanX.getRate() * dt; // Calculate gyro angle using the
unbiased rate
  gyroYangle += kalmanY.getRate() * dt;

  compAngleX = 0.93 * (compAngleX + gyroXrate * dt) + 0.07 * roll; //
Calculate the angle using a Complimentary filter
  compAngleY = 0.93 * (compAngleY + gyroYrate * dt) + 0.07 * pitch;

  // Reset the gyro angle when it has drifted too much
  if (gyroXangle < -180 || gyroXangle > 180)
    gyroXangle = kalAngleX;
  if (gyroYangle < -180 || gyroYangle > 180)
    gyroYangle = kalAngleY;



  //Arduino PID without the Library
  delta_t = micros() - time_old;                        //calculate sampling
time
  sensor_read = kalAngleX*DEG_TO_RAD;


  //The PID code without the library
  float error = reference - sensor_read ;                        //calc. error "e"
  float E = E + (error * delta_t);                           //calc. error integration
  float e_dot = error - error_old;                            //calc. error
diffrentiation

  // ----------Two tricks---------------
  if (delta_t != 0){
    diff_term = e_dot / delta_t ;                            //calc. derivative term
  }else { diff_term = 0;}

  if (e_dot < agressive){
    guard = 1;
  }else { guard = 0 ;}
  // ------------------------------------

  int_term = E ;                                    //calc. itegrator term

  float u = kp* error + guard * ki * int_term + kd * diff_term ;

   if(sensor_read > 0){
      sensor_read = -1 * sensor_read;
      digitalWrite(int1, LOW);
      digitalWrite(int2, HIGH);
   }
   else if (sensor_read == 0){
      digitalWrite(int1, LOW);
      digitalWrite(int2, LOW);
   }
   else{
      digitalWrite(int1, HIGH);
      digitalWrite(int2, LOW);
   }

  analogWrite(Pwm, u);
```

```
 Serial.print(" | Angle  = ");
 Serial.print(Input);
 Serial.print("    ");
 Serial.println(Output);


 delay(2);
}
```

- Second trial code:

```
/*******************************************************
 * PID Basic Example for the RWIP
 *******************************************************/

#include <PID_v1.h>
#include "I2Cdev.h"
#include "MPU6050_6Axis_MotionApps20.h"
#if I2CDEV_IMPLEMENTATION == I2CDEV_ARDUINO_WIRE
   #include "Wire.h"
#endif

MPU6050 mpu;

//The motor parameters

#define pwm 3
const int in1 = 12;
const int in2 = 13;

//the sensor parameters

#define OUTPUT_READABLE_YAWPITCHROLL
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
double time1,time2;

// orientation/motion vars
Quaternion q;           // [w, x, y, z]         quaternion container
VectorInt16 aa;         // [x, y, z]            accel sensor measurements
```

```
VectorInt16 aaReal;     // [x, y, z]            gravity-free accel sensor
measurements
VectorInt16 aaWorld;    // [x, y, z]            world-frame accel sensor
measurements
VectorFloat gravity;    // [x, y, z]            gravity vector
float euler[3];         // [psi, theta, phi]    Euler angle container
float ypr[3];           // [yaw, pitch, roll]   yaw/pitch/roll container and gravity
vector

// packet structure for InvenSense teapot demo
uint8_t teapotPacket[14] = { '$', 0x02, 0,0, 0,0, 0,0, 0,0, 0x00, 0x00, '\r', '\n'
};

//
================================================================
===
// ===               INTERRUPT DETECTION ROUTINE                ===
//
================================================================
===

volatile bool mpuInterrupt = false;     // indicates whether MPU interrupt
pin has gone high
void dmpDataReady() {
   mpuInterrupt = true;
}


//Define Variables we'll be connecting to
double Setpoint, Input, Output;

//Specify the links and initial tuning parameters
double Kp=1612.62764385817, Ki=1760.60255185303,
Kd=241.395867072866;
PID myPID(&Input, &Output, &Setpoint, Kp, Ki, Kd, DIRECT);
```

```
//-----------------------------------------------------------------------------------

void setup()
{
  //initialize the variables we're linked to
  pinMode(pwm,OUTPUT);
  pinMode(in1,OUTPUT);
  pinMode(in2,OUTPUT);

//MPU configuration


// 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

    // NOTE: 8MHz or slower host processors, like the Teensy @ 3.3v or
Ardunio
    // Pro Mini running at 3.3v, cannot handle this baud rate reliably due to
    // the baud timing being too misaligned with processor ticks. You must
use
    // 38400 or slower in these cases, or use some kind of external separate
    // crystal solution for the UART timer.
    // 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"));

    // wait for ready
    Serial.println(F("\nSend any character to begin DMP programming and
demo: "));
    while (Serial.available() && Serial.read()); // empty buffer
    while (!Serial.available());                 // wait for data
    while (Serial.available() && Serial.read()); // empty buffer again

    // 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(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
        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();
    } 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(")"));
    }


  Setpoint = 0.00;

  //turn the PID on
  myPID.SetMode(AUTOMATIC);
  myPID.SetSampleTime(20);

}

//--------------------------------------------------------------------------------

void loop()
{
```

```
// if programming failed, don't try to do anything
   //time1= millis();
   if (!dmpReady) return;

   // wait for MPU interrupt or extra packet(s) available
   while (!mpuInterrupt && fifoCount < packetSize) {
      // other program behavior stuff here
      // .
      // .
      // .
      // if you are really paranoid you can frequently test in between other
      // stuff to see if mpuInterrupt is true, and if so, "break;" from the
      // while() loop to immediately process the MPU data
      // .
      // .
      // .
   }

   // 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;


    #ifdef OUTPUT_READABLE_YAWPITCHROLL
       // display Euler angles in degrees
       mpu.dmpGetQuaternion(&q, fifoBuffer);
       mpu.dmpGetGravity(&gravity, &q);
       mpu.dmpGetYawPitchRoll(ypr, &q, &gravity);
      // time2 = millis();
     //  Serial.print(time2-time1);
       //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]);
      // delay(10000);
    #endif

 }

Input = ypr[2];
myPID.Compute();

if(Output < 0){
 digitalWrite(in1,HIGH);
```

```
    digitalWrite(in2,LOW);
  }else if(Output > 0){
    digitalWrite(in1,LOW);
    digitalWrite(in2,HIGH);
  }else{
    digitalWrite(in1,LOW);
    digitalWrite(in2,LOW);
  }
  Serial.println(Output);
  analogWrite(pwm, abs(Output));
  delay(20);
}
```

|  |  |
|---|---|
| Kt | motor torque constant |
| Ke | motor back emf constant |
| Iw | moment of inertia of the wheel about its center of mass |
| Ip | moment of inertia of the pendulum about the pivot |
| Ra | motor armature resistance |
| Mw | mass of the wheel |
| Mp | mass of the pendulum |
| T | reaction torque actuating the system |
| V | armature voltage of the motor |
| Lp | half the length of the pendulum |