A picture containing vector graphics

Description generated with high confidence

# Self-Balancing

# Robot

**Tony Najjar**

**Walid Aboul hosn**

**Table of Contents**

[Introduction 2](#_Toc500176108)

[Background Research 3](#_Toc500176109)

[Technical discussion & Simulation study 6](#_Toc500176110)

[Mathematical Modeling 6](#_Toc500176111)

[CAD Modeling 9](#_Toc500176113)

[Hardware Wiring 11](#_Toc500176114)

[Results 12](#_Toc500176115)

[Balancing Control 12](#_Toc500176116)

[Position Control 13](#_Toc500176117)

[Wireless Control and Camera Streaming 13](#_Toc500176118)

[Conclusion 14](#_Toc500176119)

[References 14](#_Toc500176120)

[Appendix 14](#_Toc500176121)

[Arduino 14](#_Toc500176122)

[Raspberry Pi 23](#_Toc500176123)

[MATLAB 24](#_Toc500176124)

# Introduction

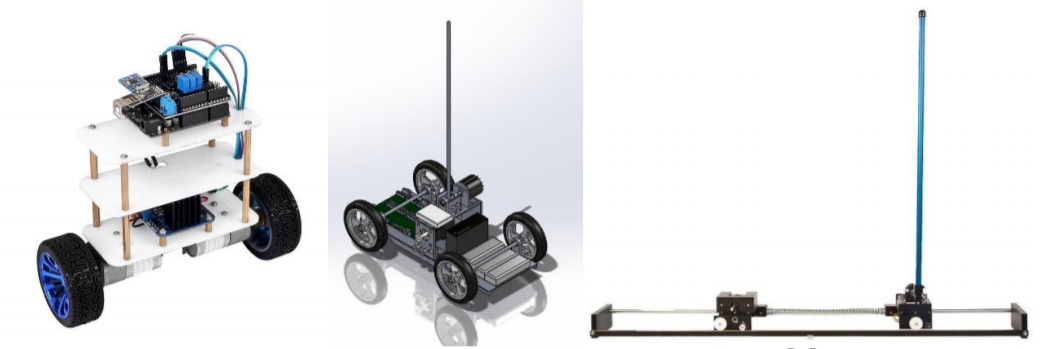
Self-balancing robots have been a topic of interest of many researchers, students and hobbyists worldwide. It is essentially an inverted pendulum on wheels, a derivative of the inverted pendulum on a cart. Unlike traditional robots, which are in a constant state of equilibrium, the self-balancing robot is a naturally unstable system; its design is more complex, as it needs to be actively controlled to maintain its upright position. The primary practical application of a self-balancing robot is human transportation, which was popularized by the release of the Segway.

* This project aims to design, construct and program a self-balancing robot. To achieve the aims of the project, following objectives have been set:
* Design and assemble the chassis of the robot
* Mount all the electronic hardware on the chassis and make all the electrical connections
* Develop the software to read from the sensors and to control the actuators
* Implement a PID controller to enable the robot to stay upright
* Implement a PID controller to enable the robot to return to move
* Develop the software for wireless control and camera streaming

A self-balancing robot has been designed and built from scratch. Mechanically it looks and works as planned. A mathematical model of the robot and a PID control architecture were calculated and simulated to verify the system’s behavior. The PID controllers were successfully implemented on the robot together with a Kalman filter to make it balance, even with disturbances.

# Background Research

The inverted pendulum is a classical problem in control systems, and to explore the unstable dynamics, different platforms have been developed. These platforms are similar in many ways, leading to many of the behaviors being comparable. The most common types are the self-balancing robot, Inverted Pendulum on a cart and an inverted pendulum on a linear track, shown in the figure below:



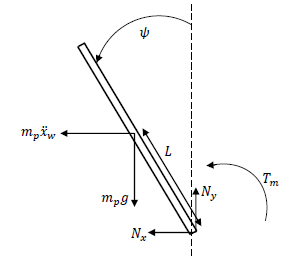
The most renowned practical application of the inverted pendulum is the Segway, a dynamic system used for individual transportation.

The Segway is anchored to a base platform that has a wheel mounted on each side. In this case, a motor drives each wheel independently. The torque from the motors makes the base move to balance pitch angle of the pendulum. It can move along curved paths by driving the motors at different speeds. The two-wheeled inverted pendulum has been proposed as a portable transporter due to its high maneuverability

The Segway is a device that transports one person at relatively low speeds. The low-speed (limited to approximately 12 mph) operation combined with its electric propulsion system makes the Segway a candidate for providing short-distance transportation on city streets, sidewalks, and inside buildings. When a Segway is in use, the device is driven by two wheels that are placed side-by-side, rather than the standard in-line configuration of a bicycle or a motorcycle. When the operator leans forward, the wheels turn in unison in the same direction to provide forward motion. In order to stop, the wheels must accelerate forward to get out in front of the system's center of mass and then apply a deceleration torque to slow the system down without causing the operator to fall forward of the device. These operating principles are reversed to allow the system to move backward. In order to turn, the wheels rotate at unequal speeds causing the system to travel in an arc. If the system is not translating forward or backward, then the wheels can rotate in opposite directions to turn the machine in place. Given the side-by-side wheel configuration, and the elevated center of mass, the mechanical design of the transporter is unstable. It will fall over if the computerized control system is not continuously turning the wheels. This constant adjusting of the device is similar to a person balancing an inverted broom in their hand. In order to keep the broom upright, the person must continually move their hand in the direction that the broom is falling. The broom's center of mass to generate a torque that will cause the broom to start rotating in the opposite direction. As a result, the broom is always falling, but the hand motion keeps changing the direction of the fall. Just like the inverted broom, the Segway and rider are always falling. However, it is not possible for the human operator to balance the device, as they can with a human-powered inverted pendulum such as a unicycle. The sensors in the device must constantly be measuring the state of the machine and feeding this information to the computer controller. The controller then uses this feedback signal to adjust the wheel speed so that the forward/backward (pitch) falling motion is maintained within an acceptable envelope so that device and rider do not fall over. Note that under many operating conditions, the system is mechanically stable in the side-to-side (roll) direction. Therefore, the computer does not attempt to control the roll motion. Assuming wheel-ground rolling stiction, the system is also stable in the yaw direction.

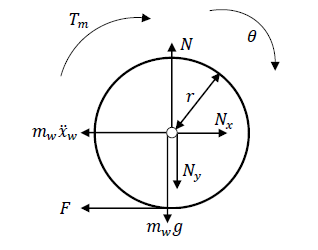
# Technical discussion & Simulation study

## Mathematical Modeling



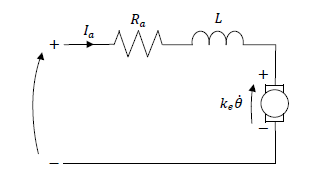
**Pendulum**

1. *Tm + mpgLsinψ + mpwLcosψ = Jpψ*
2. *−Nx − mpw = mpp*
3. *Ny − mpg = mpp*
4. *xp = −Lsinψ*
5. *yp = −Lcosψ*
6. *p = −Lcosψ + L 2 sinψ*
7. *p = −Lsinψ + L 2 cosψ*



**Wheels**

1. *Tm + Fr = Jw*
2. *Nx − mww − F = 0*
3. *N − Ny − mwg = mww*
4. *w = r*
5. *w = 0*



**Motors**

1. *U = RaIa + keθ*
2. *Tm = nktIa*
3. *Ia = U − keRa*
4. *Tm = nktU Ra − nkekt Ra*

**Nonlinear Result**

1. *=*
2. *= Lrmpcosψ (gLRampsinψ + 2nkt (U − ke)) + Jp (2nkt (U − ke) − LrRamp 2 sinψ (Ra (Jp (Jw + r 2 (mp + mw)) − L2r 2m2 p cos2ψ))*

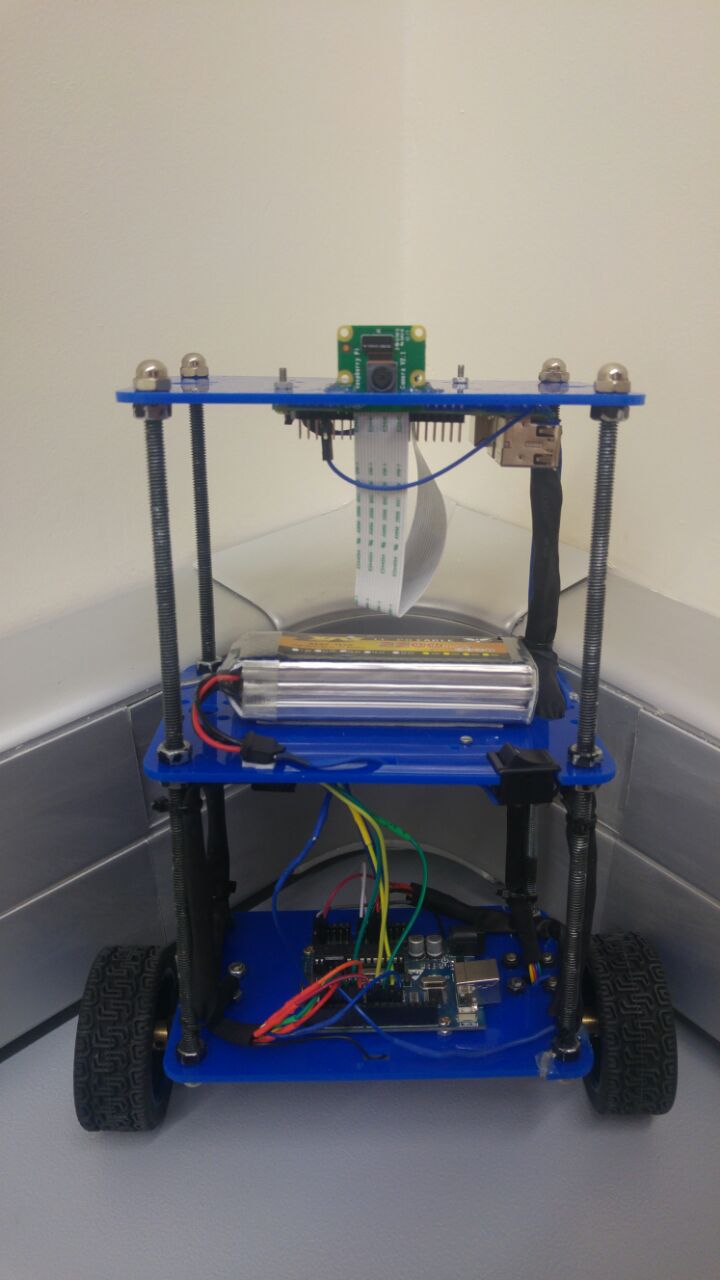
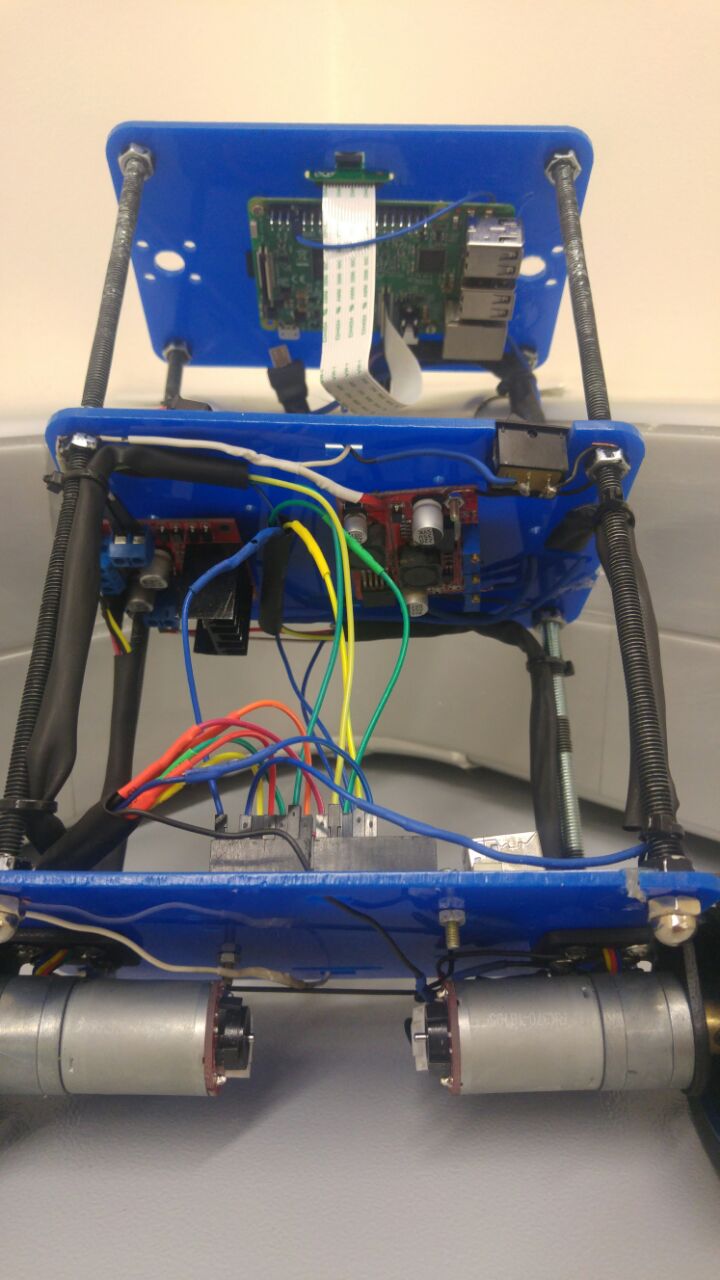
**Linear Result**

We can linearize the system by approximating cosθ= 1 and sinθ= θ since θ is considered very small

1. *=*
2. *= Lrmp (gLRampψ + 2nkt (U − ke)) + Jp (2nkt (U − ke) − LrRamp 2ψ (Ra (Jp (Jw + r2(mp + mw)) − L2r 2m2 p))*

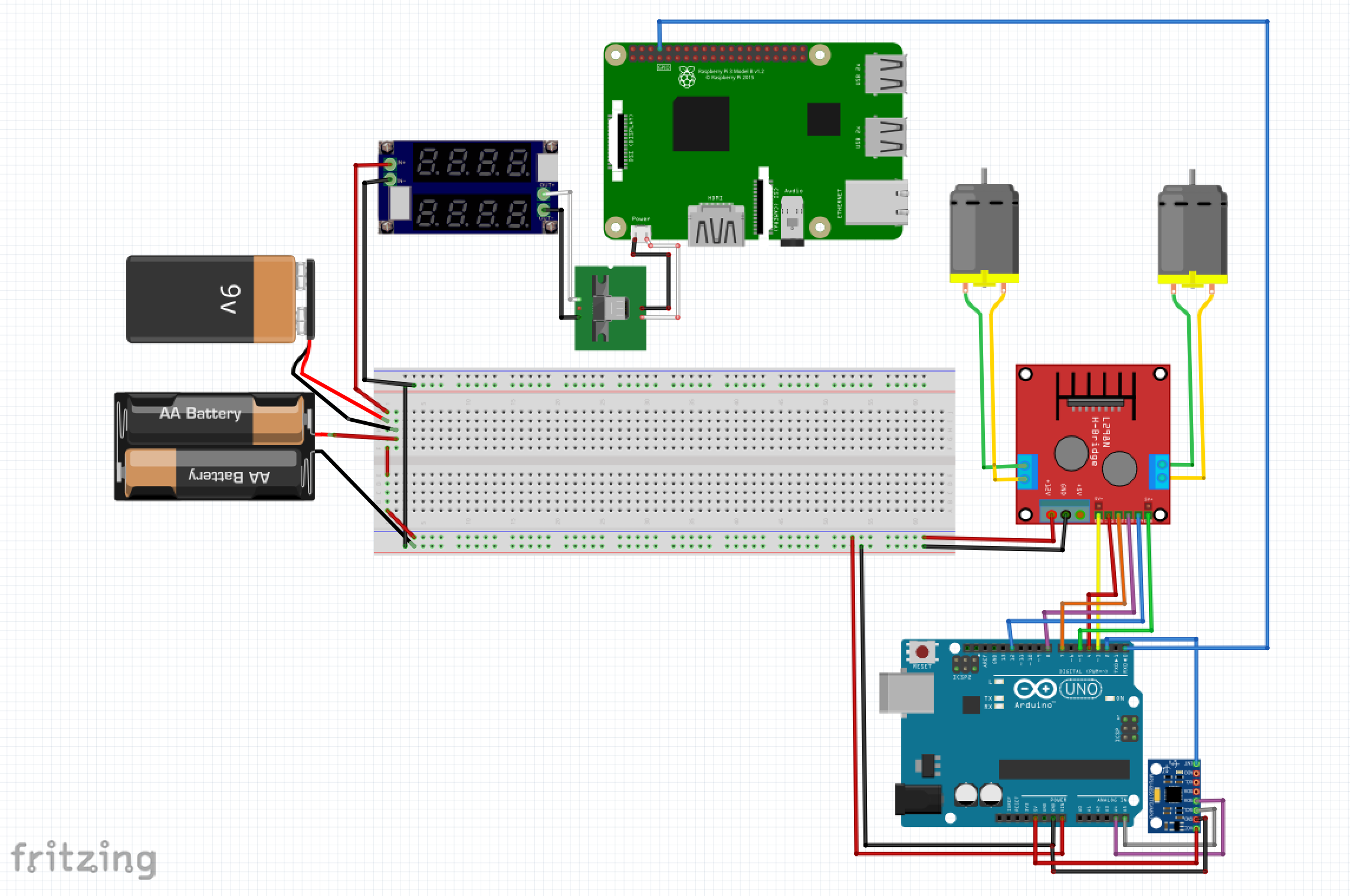
## CAD Modeling





|  |  |  |
| --- | --- | --- |
|  | **Item** | **Quantity** |
| **Mechanical** | Threaded Rods | 4 |
| Plexiglass Board | 3 |
| Motor Brackets | 2 |
| **Electrical** | Arduino | 1 |
| Raspberry Pi | 1 |
| Raspberry Pi camera | 1 |
| IMU MPU6050 | 1 |
| Motor Driver L298N | 1 |
| Step-down DC Transformer | 1 |
| Battery | 1 |
| Geared Motors with Encoders | 2 |

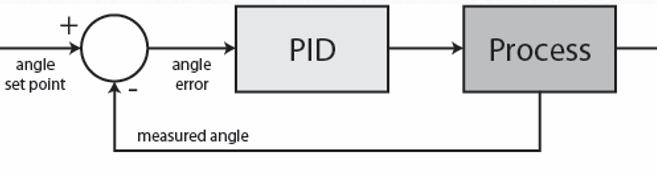
## Hardware Wiring



# Results

## Balancing Control

The control algorithm that is used to maintain the balance on the autonomous self-balancing robot is the PID controller. The proportional, integral, and derivative (PID) controller is well known as a three-term controller. The input to the controller is the error from the system. The Kp, Ki, and Kd are referred as the proportional, integral, and derivative constants (the three terms get multiplied by these constants) respectively. In the PID controller the error gets managed in three ways. The error will be used on the PID controller to execute the proportional term, integral term for reduction of steady state errors, and the derivative term to handle overshoots



It is worthwhile mentioning that the LQR controller is also vastly used in inverted pendulum applications. LQR is a form of optimal control that aims to minimize the performance index whist taking into account the control effort, as often, higher input effort would imply higher energy consumption. LQR control requires derivation of the state-space model of the system, thus it is more challenging to implement.

## Position Control

One issue with the balancing control loop is that the robot has no insight on its position and could stay upright while constantly moving. This is usually undesirable due to space constraints. To solve this issue we used parallel compensation; an extra encoder PID controller was added to the scheme to control the robot’s position. The encoder PID measures the error in position and outputs the setpoint for the angle PID loop which in turn will control the PWM of the motors. So if there's an offset in position the angle setpoint will change to, let say -2 degrees (from upright 0 degrees). So now the angle PID will make the robot move in order to correct the angle offset and therefore also correcting the position error.

## Wireless Control and Camera Streaming

Wireless control and camera streaming was implemented on the robot using a Raspberry Pi.

The small computer receives keyboard commands via Wifi and communicates them serially to the Arduino. The only factor we control is the position setpoint; we set it to be slightly positive or negative to make the robot lean forward or backward and consequently accelerate.

As for the on-board camera, the robot has the ability to wirelessly stream with minimum lag the camera’s output over the network.

# Conclusion

Overall, this self-balancing robot project was a big success, meeting all the objectives set at the beginning. The robot’s performance compares very well with other self-balancing robots found on the internet in terms of smooth balancing and position control.

We look forward to enhancing the robot in many ways such as smoothening the controllers for better transient response and adding: computer vision (to follow a ball for example), ultrasonic sensors for table/stair edge detection and side arms for fall cushioning and automatic stand up.

# References

<https://www.coursehero.com/file/p1e3kuv/Other-studies-considered-changing-the-relative-position-of-the-global-center-of/>

<http://ieeexplore.ieee.org/iel7/6550152/6564841/06565146.pdf>

http://www.ijcert.org/V2I1249.pdf

# Appendix

## Arduino

#include <PID\_v1.h>

#include <LMotorController.h>

#include "I2Cdev.h"

#include <SoftwareSerial.h>

#include <Encoder.h>

#include "MPU6050\_6Axis\_MotionApps20.h"

#if I2CDEV\_IMPLEMENTATION == I2CDEV\_ARDUINO\_WIRE

#include "Wire.h"

#endif

#define MOVE\_BACK\_FORTH 0

#define MIN\_ABS\_SPEED 80

Encoder left(9, 6);

Encoder right(10, 11);

long oldPosition = -999;

int encCount = 0;

//MPU

unsigned long counter;

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 = 174.29;

double setpoint = originalSetpoint;

double movingAngleOffset = 1.5;

double input = originalSetpoint, output;

int moveState = 0; //0 = balance; 1 = back; 2 = forth

double posInput, posOutput, posSetpoint = 0;

PID pid(&input, &output, &setpoint, 25, 240, 2, DIRECT);

PID posPID(&posInput, &posOutput, &posSetpoint, 0.8, 0, 1, REVERSE);

//MOTOR CONTROLLER

int motorSpeed;

double motorSpeedFactorLeft = 0.8;

double motorSpeedFactorRight = 1;

//MOTOR CONTROLLER

const int ENA = 3;

const int IN1 = 4;

const int IN2 = 7;

const int IN3 = 8;

const int IN4 = 12;

const int ENB = 5;

LMotorController motorController(ENA, IN1, IN2, ENB, IN3, IN4, motorSpeedFactorLeft, motorSpeedFactorRight);

//timers

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

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(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();

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(100);

pid.SetOutputLimits(-255, 255);

posPID.SetMode(AUTOMATIC);

posPID.SetSampleTime(100);

posPID.SetOutputLimits(-movingAngleOffset, movingAngleOffset);

}

else{

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){

//Serial.println(abs(input - originalSetpoint));

if (abs(input - originalSetpoint) > 20) {

motorController.move(0);

break;

}

double newPosition = (right.read() + left.read()) / 2;

if (newPosition != oldPosition && abs(newPosition - oldPosition) > 5) {

oldPosition = newPosition;

posInput = newPosition / 100;

//Serial.println(posInput);

}

// Serial.println(posInput);

//no mpu data - performing PID calculations and output to motors

posPID.Compute();

calculateSetpoint();

pid.Compute();

if (input > setpoint + 0.3 || input < setpoint - 0.3) {

motorController.move(output, MIN\_ABS\_SPEED);

}

else {

motorController.move(0);

}

unsigned long currentMillis = millis();

if (currentMillis - time5Hz >= 5000)

{

//moveBackForth();

time5Hz = currentMillis;

}

}

// 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 & 0x11) || 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[2] \* 180 / M\_PI + 180;

}

}

void calculateSetpoint() {

if (Serial.available()) {

counter = millis();

char c = Serial.read();

//Serial.println(c);

switch (c) {

case 'w':

motorController.setMotorConst(1, 1);

setpoint = originalSetpoint + movingAngleOffset;

break;

case 's':

motorController.setMotorConst(1, 1);

setpoint = originalSetpoint - movingAngleOffset;

break;

case 'a':

motorController.setMotorConst(0.5, 1);

setpoint = originalSetpoint + movingAngleOffset;

break;

case 'd':

motorController.setMotorConst(1, 0.5);

setpoint = originalSetpoint + movingAngleOffset;

break;

}

}

else if (millis() - counter > 500) {

posInput=0;

left.write(0);

right.write(0);

posPID.Compute();

setpoint = originalSetpoint + posOutput;

motorController.setMotorConst(0.75, 1);

}

}

## Raspberry Pi

import sys, tty, termios

import serial

ser = serial.Serial(port='/dev/serial0', baudrate=115200,

parity=serial.PARITY\_NONE,

stopbits=serial.STOPBITS\_ONE,

bytesize=serial.EIGHTBITS,

timeout=1)

def getch():

fd = sys.stdin.fileno()

old\_settings = termios.tcgetattr(fd)

try:

tty.setraw(sys.stdin.fileno())

ch = sys.stdin.read(1)

finally:

termios.tcsetattr(fd, termios.TCSADRAIN, old\_settings)

return ch

while True:

char = getch()

print char

if char == 'p':

break

ser.write(char)

## MATLAB

%Uncomment one set of PID tunings at a time and run the code

clear

M = 0.5;

m = 0.2;

b = 0.1;

I = 0.006;

g = 9.8;

l = 0.3;

q = (M+m)\*(I+m\*l^2)-(m\*l)^2;

s = tf('s');

P\_pend = (m\*l\*s/q)/(s^3 + (b\*(I + m\*l^2))\*s^2/q - ((M + m)\*m\*g\*l)\*s/q - b\*m\*g\*l/q);

%Unstable

% Kp = 10;

% Ki = 0;

% Kd = 0;

%Underdamped

Kp = 100;

Ki = 1;

Kd = 1;

%Overdamped

% Kp = 100;

% Ki = 1;

% Kd = 20;

%axis([0, 2.5, -0.2, 0.2]);

C = pid(Kp,Ki,Kd);

T = feedback(series(P\_pend,C),1);

t=0:0.01:10;

[Y, T]=impulse(T,t);

h=figure;

subplot(2,1,1)

plot(T,Y)

subplot(2,1,2)

axis([-1 1 0 1.5])

for i=1:1:size(Y)

if ~ishghandle(h)

break

end

x= cosd(Y(i,1));

y= sind(Y(i,1));

link=line([0 y], [0 x], 'LineWidth' , 4);

pause(0.01);

delete(link)

end