Self-balancing robot

A Project Report

Group:

NING HU

ZEXIN LI

Under the guidance of

VIKRAM KHAPILA

(Professor, Department of Mechanical & Aerospace Engineering)

in partial fulfilment for the course

of

ADVANCED MECHATRONICS (ME – GY 6933)

for

MECHATRONICS AND ROBOTICS ENGINEERING

of

TANDON SCHOOL OF ENGINEERING

11201, 6 MetroTech Center, Brooklyn, NY 11201

contents

1. INTRODUCTION

To make a robot which can balance itself on two wheels. There will be only one axle connecting the two wheels and a platform will be mounted on that .There will be another platform above it. The platform will not remain stable itself. Our job will be to balance the platform using distance sensors and to maintain it horizontal. At first we have decided to just balance the robot on its two wheels .If the platform inclines then microcontroller (in this case it is Arduino) will send signals to motors such that motors would move forward or backward depending on the inclination direction and extent. So if the platform tilts forward then motors will run forward and backward to keep the platform horizontal. For this we will need to code the Arduino in order to perform job according to this.

The motor driver is powered by 12V power supply. The PWM sent to the motor driver has low voltage=0V and high voltage=5V. The motor driver L298N scales it to the range low voltage=0V and high voltage=12V (motor driver is used because motors need 12 V to run. Also they need high current. Microcontroller output ports cannot supply 12 V and high current). Then this PWM is applied across the motor .so ,if the robot tilts such that height of sensor 1 decreases , then accordingly output PWM will be such that (output PWM will be positive) motors will run to accelerate the robot to sensor 1 side. Because of this this acceleration robot will experience a torque (due to pseudo force in the non inertial frame) that will oppose the torque due to gravity and it will bring the robot platform back to the horizontal level .Change in the difference value. The modified part of code measures difference (bet sensor1 value and sensor2 value) regularly after some fixed interval (10 milliseconds). So the change in difference(i.e., difference- previous difference) is considerable. We have not used I (integration) here in the PID control as we found P+D sufficient to balance the robot.

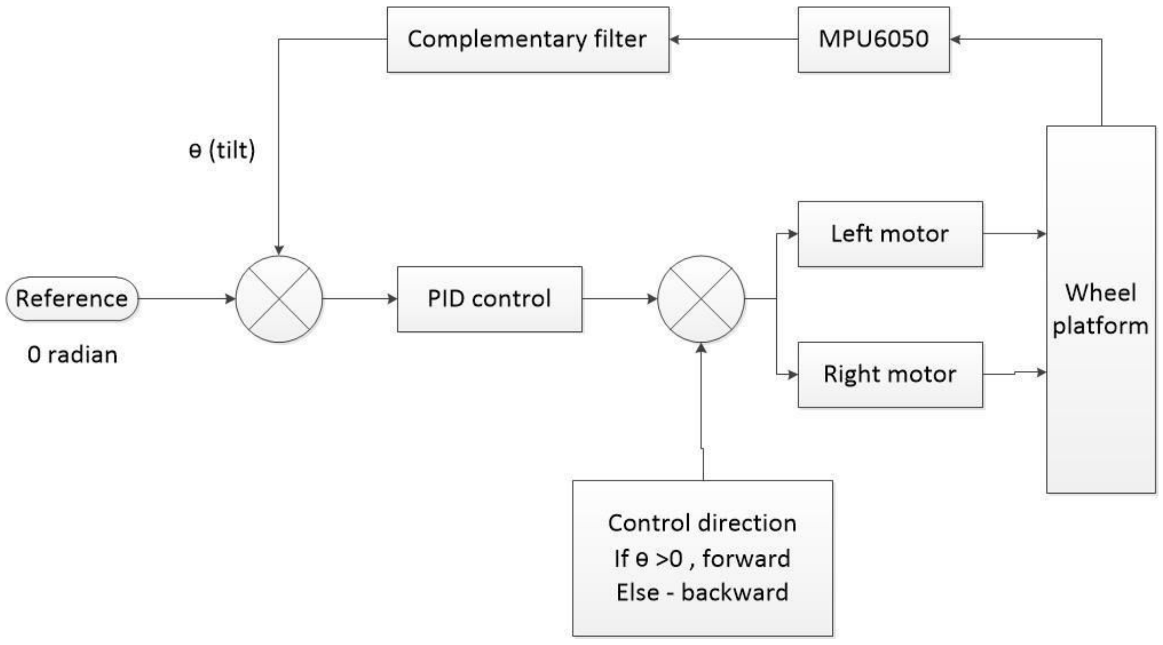
2. IMPLEMENTATION

1 Arduino Uno R3
2 Sensor shield
3 H bridge
4 Dc motor
5 MPU6050
6 Joystick
7 NRF24L01
8 Battery
9 Rasberry PI3
10 Red ball

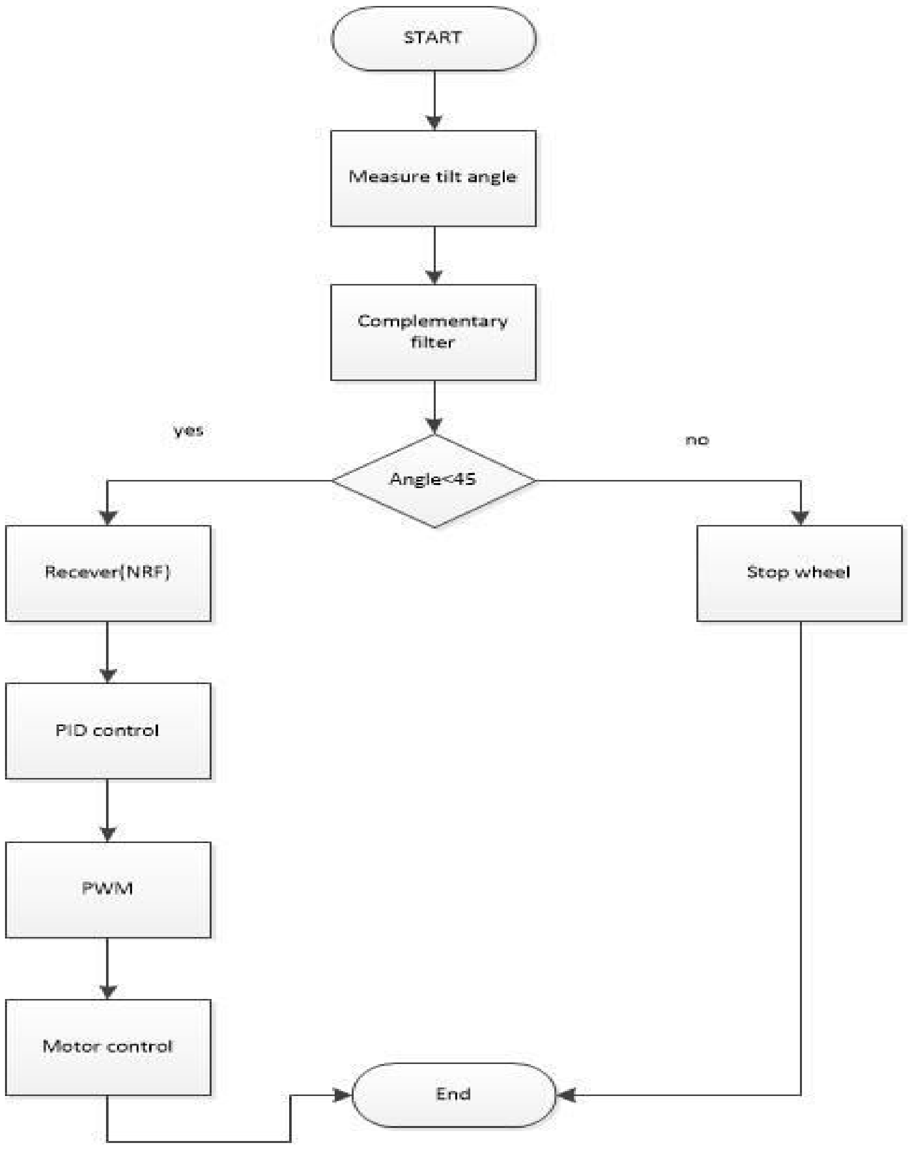
3. Self-balancing robot

3.1 working principle and Software design

1)working principle



2) Software design



3.2 model

1) modeling and PID control

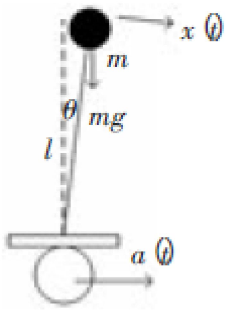
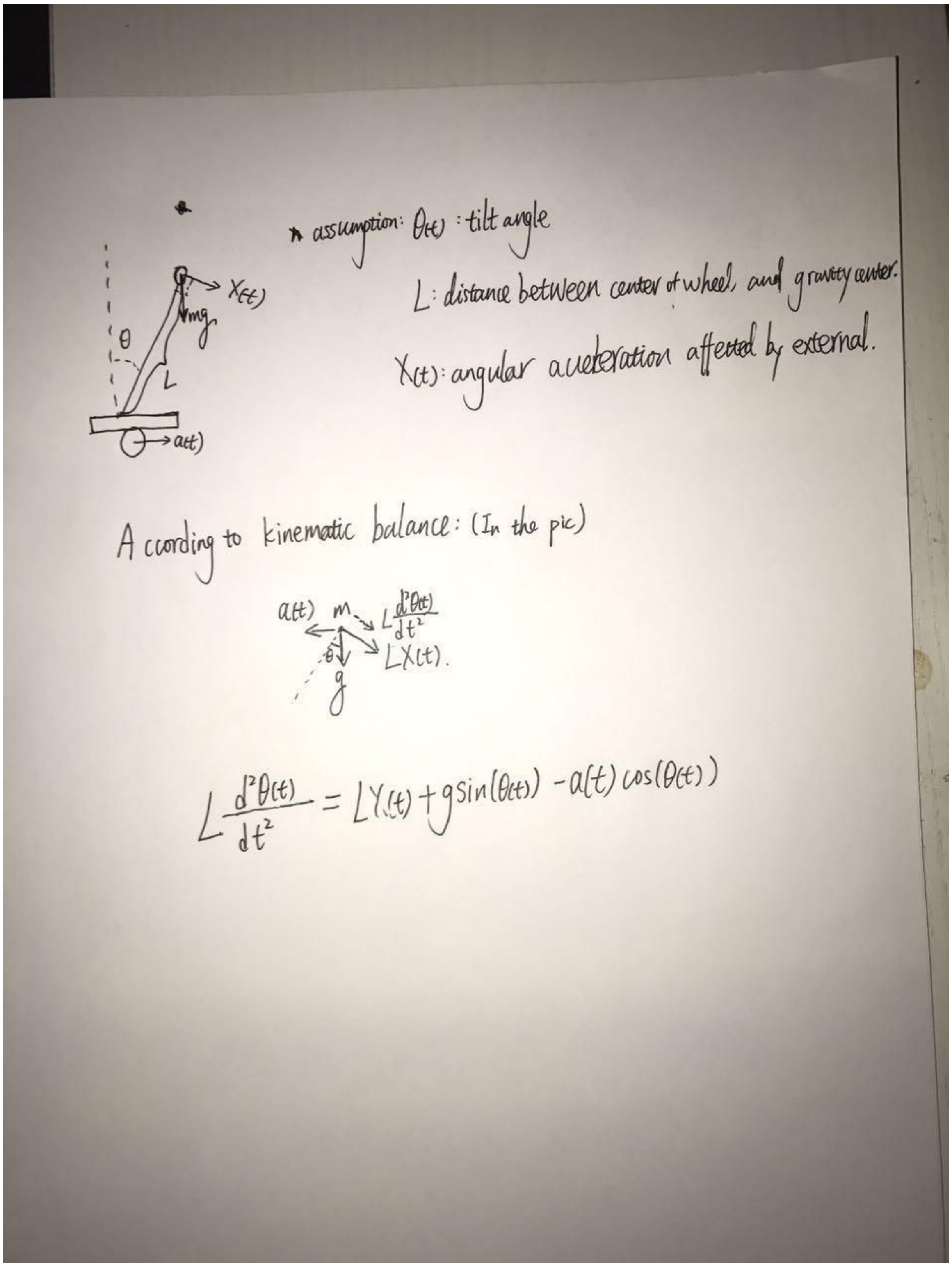


Fig1 force analysis



$X(t)$
mg
θ
L
a(t)
a(t)
m
$L\frac{d^2\theta(t)}{dt^2}$
$LX(t)$.
θ
g

* assumption: $\theta(t)$: tilt angle

L: distance between center of wheel, and gravity center.

$X(t)$: angular accelerationn affected by external.

According to kinematic balance: (In the pic)

$$L\frac{d^2\theta(t)}{dt^2} = LX(t) + g\sin(\theta(t)) - a(t)\cos(\theta(t))$$

According to kinematic balance principle , (a(t) means wheel acceleration)

$$l\times\frac{d^2\theta(t)}{dt^2} = l\times X(t) + g\times sin\ \theta(t)\ - a(t)\times cos\ \theta(t) \qquad (1)$$

Because the range of $\theta(t)$ is close to 0 , $sin\ \theta(t) \approx \theta\ \ cos\ \theta(t) \approx 1$.

Therefore , the (1) will be

$$l\times\frac{d^2\theta(t)}{dt^2} = l\times X(t) + g\times\theta(t) - a(t) \qquad (2)$$

After Laplace transfer , (because of static , a(t) is equal to 0)

$$l\times s^2\times\theta(s) = l\times X(s) + g\times\theta(s) \qquad (3)$$

$$\frac{\theta(s)}{X(s)} = \frac{l}{l\times s^2 - g}$$

Accoreding to (3) transfer function , its pole points are

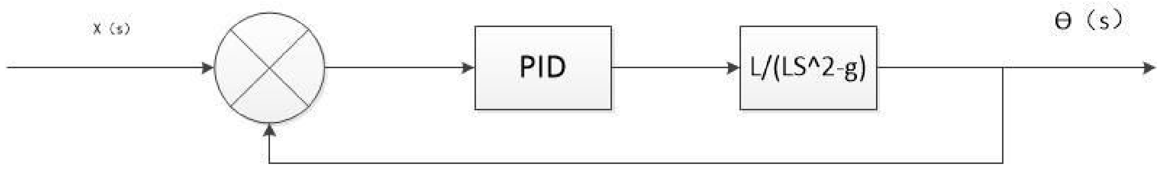
$$S_p = \pm\sqrt{\frac{g}{l}}$$

Because one of poloe points are on the right of s-plane , this system is unstable based on nyquist stability criterion.

In order to solve it , PID control is a method .

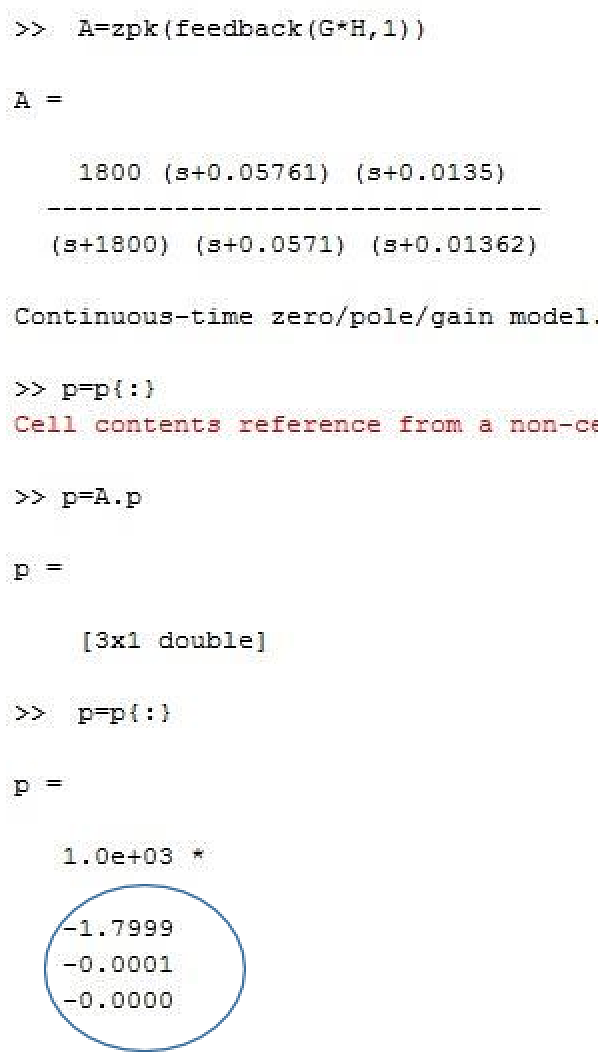
PID controller is $k_p + \frac{K_I}{S} + S * K_D$ in laplace transfer.

The new system model is in the following :



Closed-loop transfer function is $\frac{\theta(s)}{X(s)} = \frac{1}{1+(k_p+\frac{K_I}{S}+S*K_D)\times\frac{l}{l\times s^2-g}}$

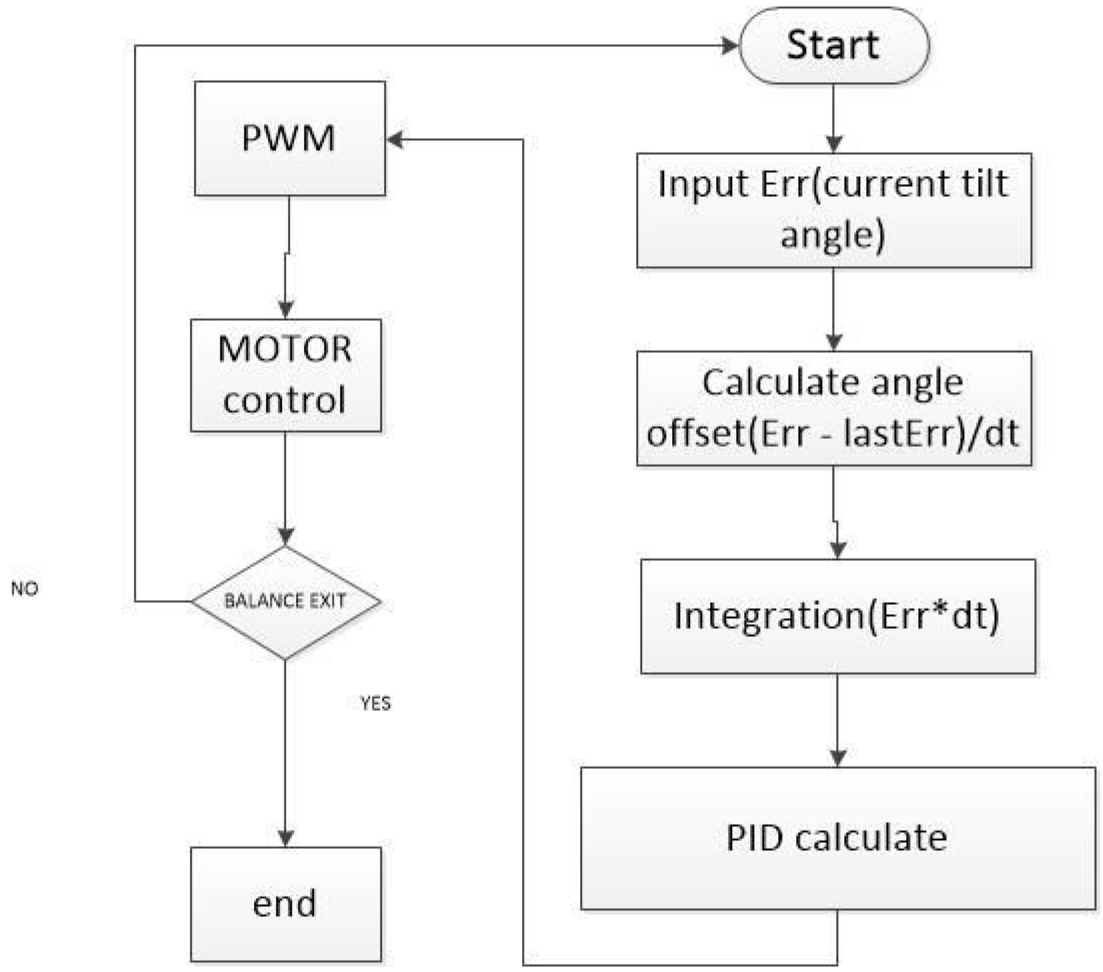
For this closed-loop system , its pole points (A in the pic is the system transfer function , p means pole points ; Kp=112 , Ki=1.4 Kd=1800)

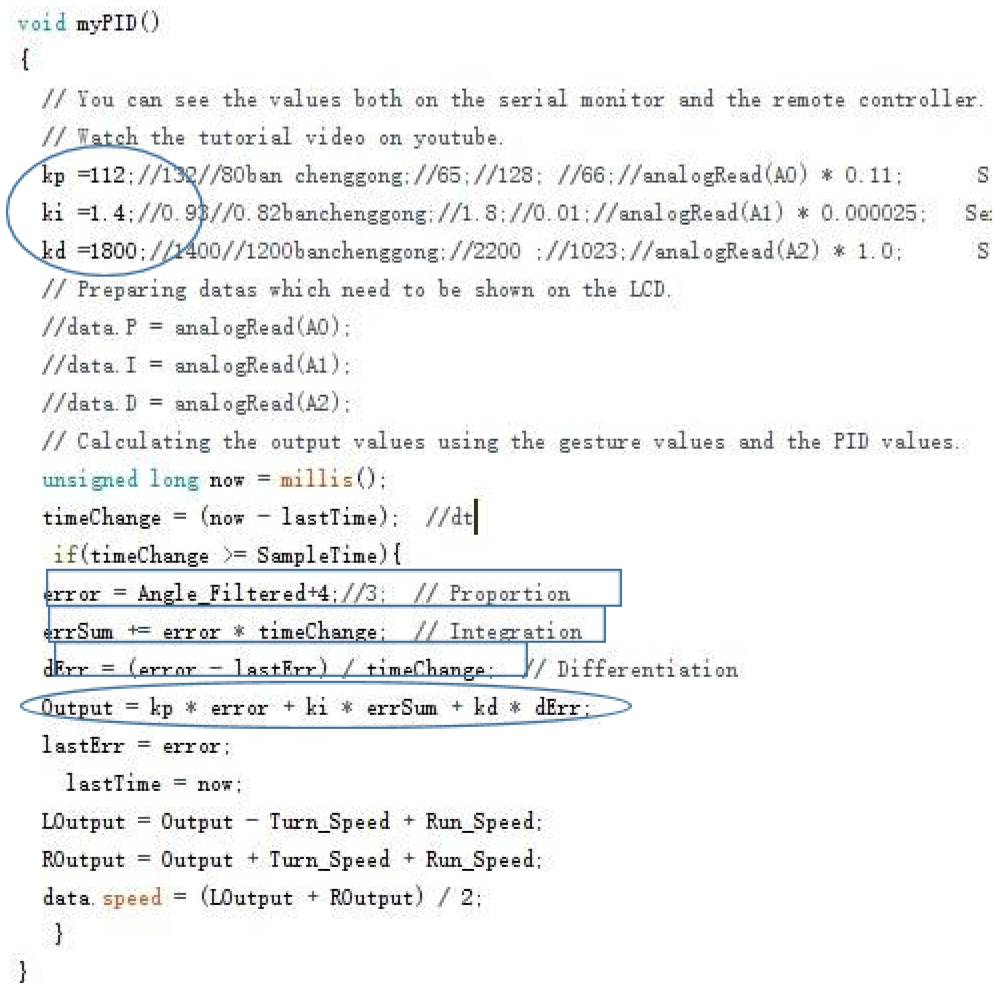


```
>>  A=zpk(feedback(G*H,1))

A =

    1800 (s+0.05761) (s+0.0135)
  -------------------------------
   (s+1800) (s+0.0571) (s+0.01362)

Continuous-time zero/pole/gain model.

>> p=p{:}
Cell contents reference from a non-ce

>> p=A.p

p =

    [3x1 double]

>>  p=p{:}

p =

   1.0e+03 *

   -1.7999
   -0.0001
   -0.0000
```

There are three pole points (the numbers in circle) . What's more , two of them are at the left of s-plane and one is at the Virtual axis. According to nyquist stability criterion , this system is stable(system whose all pole points at left of s-plane is stable)

In the coding , PID control is in the following:

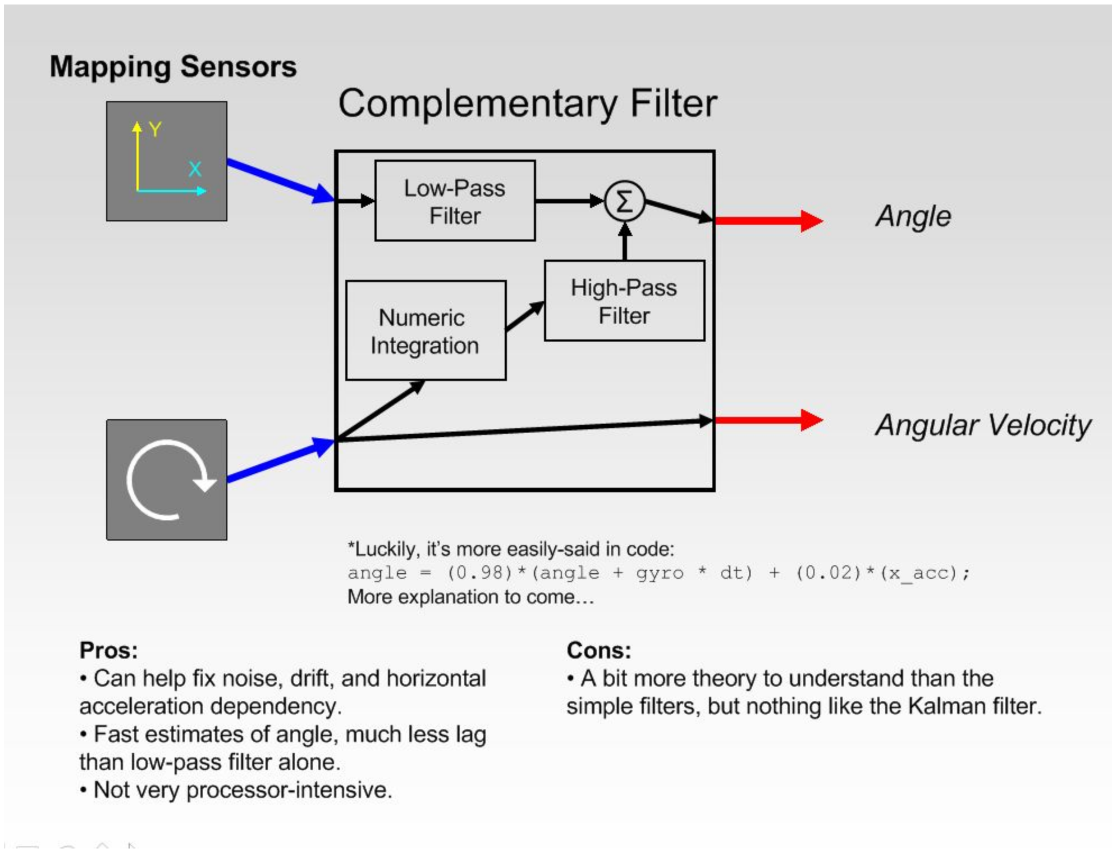

Start
Input Err(current tilt angle)
Calculate angle offset(Err - lastErr)/dt
Integration(Err*dt)
PID calculate
PWM
MOTOR control
BALANCE EXIT
NO
YES
end



```
void myPID()
{
  // You can see the values both on the serial monitor and the remote controller.
  // Watch the tutorial video on youtube.
  kp =112;//132//80ban chenggong;//65;//128; //66;//analogRead(A0) * 0.11;      S
  ki =1.4;//0.93//0.82banchenggong;//1.8;//0.01;//analogRead(A1) * 0.000025;   Se
  kd =1800;//1400//1200banchenggong;//2200 ;//1023;//analogRead(A2) * 1.0;      S
  // Preparing datas which need to be shown on the LCD.
  //data.P = analogRead(A0);
  //data.I = analogRead(A1);
  //data.D = analogRead(A2);
  // Calculating the output values using the gesture values and the PID values.
  unsigned long now = millis();
  timeChange = (now - lastTime);  //dt
   if(timeChange >= SampleTime){
  error = Angle_Filtered+4;//3;  // Proportion
  errSum += error * timeChange;  // Integration
  dErr = (error - lastErr) / timeChange;  // Differentiation
  Output = kp * error + ki * errSum + kd * dErr;
  lastErr = error;
    lastTime = now;
  LOutput = Output - Turn_Speed + Run_Speed;
  ROutput = Output + Turn_Speed + Run_Speed;
  data.speed = (LOutput + ROutput) / 2;
   }
}
```

Notice(Turn_speed and Run_speed are controlled by joystick)

3.3 FILTER



Mapping Sensors

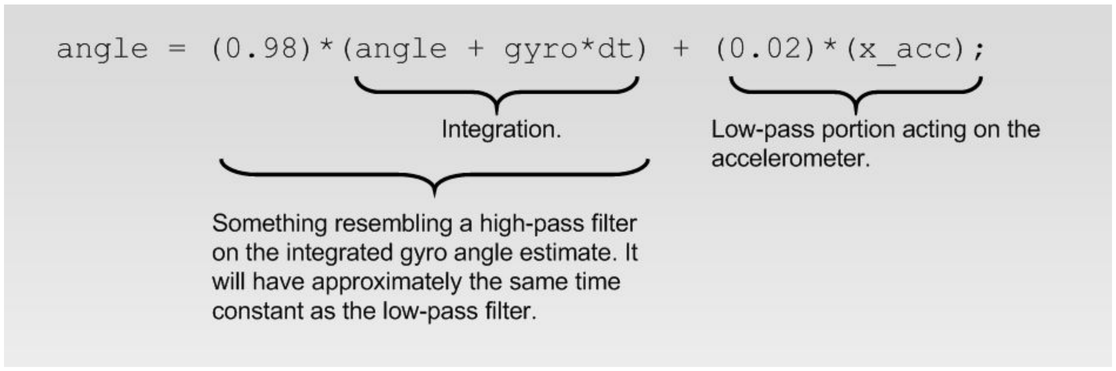
*Luckily, it's more easily-said in code:

```
angle = (0.98)*(angle + gyro * dt) + (0.02)*(x_acc);
```

More explanation to come…

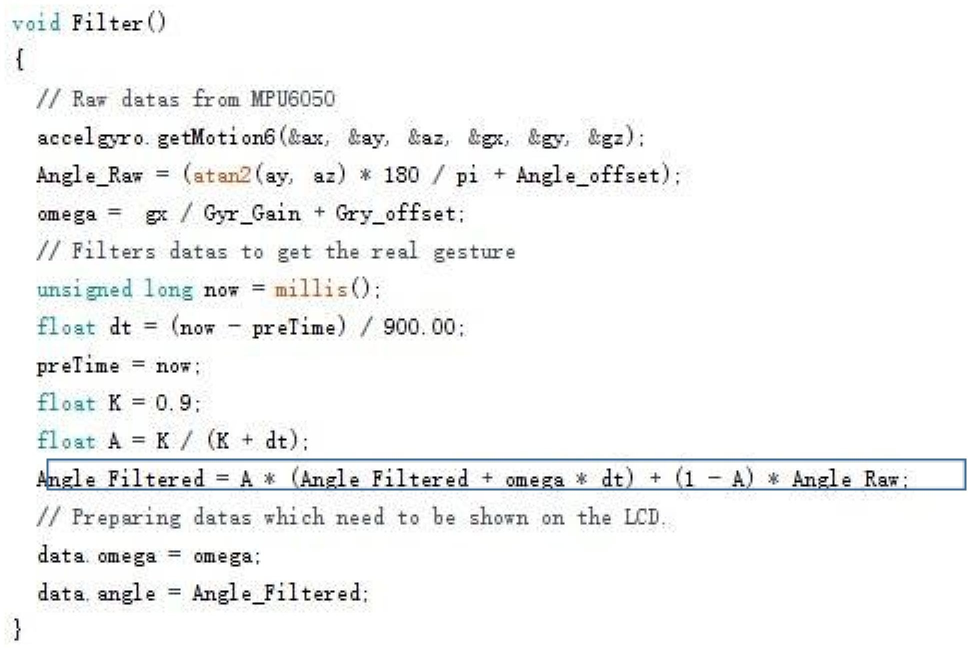
Pros:
- Can help fix noise, drift, and horizontal acceleration dependency.
- Fast estimates of angle, much less lag than low-pass filter alone.
- Not very processor-intensive.

Cons:
- A bit more theory to understand than the simple filters, but nothing like the Kalman filter.



$$\texttt{angle} = \underbrace{(0.98)*\underbrace{(\texttt{angle} + \texttt{gyro*dt})}_{\text{Integration.}}}_{\text{Something resembling a high-pass filter on the integrated gyro angle estimate. It will have approximately the same time constant as the low-pass filter.}} + \underbrace{(0.02)*(\texttt{x\_acc})}_{\text{Low-pass portion acting on the accelerometer.}};$$

In the code , the filter is in the following:



```
void Filter()
{
  // Raw datas from MPU6050
  accelgyro.getMotion6(&ax, &ay, &az, &gx, &gy, &gz);
  Angle_Raw = (atan2(ay, az) * 180 / pi + Angle_offset);
  omega =  gx / Gyr_Gain + Gry_offset;
  // Filters datas to get the real gesture
  unsigned long now = millis();
  float dt = (now - preTime) / 900.00;
  preTime = now;
  float K = 0.9;
  float A = K / (K + dt);
  Angle_Filtered = A * (Angle_Filtered + omega * dt) + (1 - A) * Angle_Raw;
  // Preparing datas which need to be shown on the LCD.
  data.omega = omega;
  data.angle = Angle_Filtered;
}
```

the formula in the square is complementary filter.

3.4 Result and conclusion

A self-balancing robot was designed and manufactured as desired with limited resources possible. It was able to balance smoothly with a maximum tilt error of 5 degrees. Range of payload and its height for which it balances was quantified. Maximum angle of tilt for balancing was also determined through various experiments. However there are some limitations. Such technology is suitable only for flat ground. In order to make it work on slant surface, the angle of slant needs to be fed into the system, either manually or using some intelligence

4. Red ball tracking

4.1 General principle

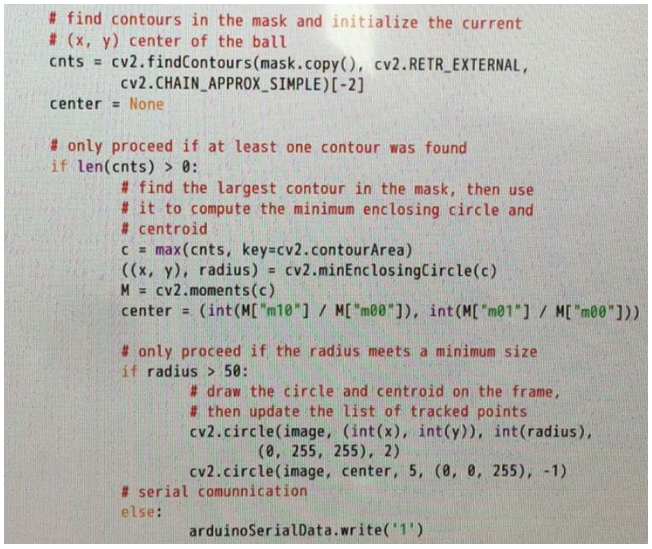
In order to achieve the performance that the red ball can be followed by the robot but stop when the distance is close enough. Some computer vision work should be carried out.

Red ball tracking based on Open CV platform. By using the HSV image we caputure, Open CV can find the contour of red ball. Then we can aquire the radius of the contour using cv2.minEnclosingCircle() command.

Now the radius has been calculated,then:

if radius > a(red ball is close enough to the camera), do nothing
if radius<=a(robot need move forward), transmit a signal from rpi to arduino.

4.2 Core code

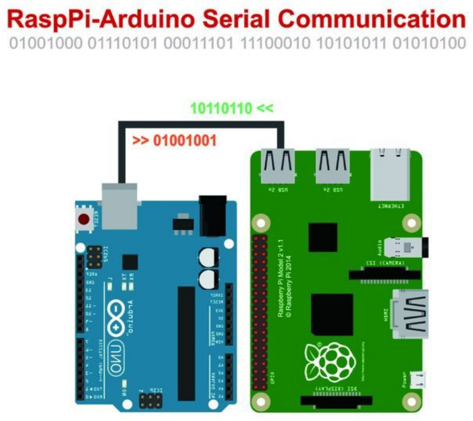


```
# find contours in the mask and initialize the current
# (x, y) center of the ball
cnts = cv2.findContours(mask.copy(), cv2.RETR_EXTERNAL,
        cv2.CHAIN_APPROX_SIMPLE)[-2]
center = None

# only proceed if at least one contour was found
if len(cnts) > 0:
        # find the largest contour in the mask, then use
        # it to compute the minimum enclosing circle and
        # centroid
        c = max(cnts, key=cv2.contourArea)
        ((x, y), radius) = cv2.minEnclosingCircle(c)
        M = cv2.moments(c)
        center = (int(M["m10"] / M["m00"]), int(M["m01"] / M["m00"]))

        # only proceed if the radius meets a minimum size
        if radius > 50:
                # draw the circle and centroid on the frame,
                # then update the list of tracked points
                cv2.circle(image, (int(x), int(y)), int(radius),
                        (0, 255, 255), 2)
                cv2.circle(image, center, 5, (0, 0, 255), -1)
        # serial comunnication
        else:
                arduinoSerialData.write('1')
```

Appendix:

cv2.findContours(): find every red area in the image;
max(): find the red area with maximum space;
minEnclosingCircle(): find a circle that can cover the given area with minmum area, and return the radius of the circle;
cv2.moment: calculate the image moment and find the center of the image.

4.3 Communication with robot



In order to combine the robot with computer vision, the communication is essential. Because we don't need a heavy job based on the communication, our project apply serial communication between the Arduino borad(Robot part) and Raspberry PI(Open CV part).