DOCUMENTATION OF THE BB-8 BALLBOT CODE

- Vedant Palit

Introduction

Analysing every line of code from a GITHUB repository (BB-8 Repository) which contains code for the controls of a BB-8-based ballot.

The BB-8 structurally consists of two ballbots, i.e the robot itself is a 2 part ballbot. The omniwheel setup inside the head, is part A ballbot while the part B ballbot lies inside the body of the bot.

The code in the repository, is basically for the two ballbots individually. However the major aim of the current project is to combine both the functions into one single omniwheel structure.

Contents of Repository

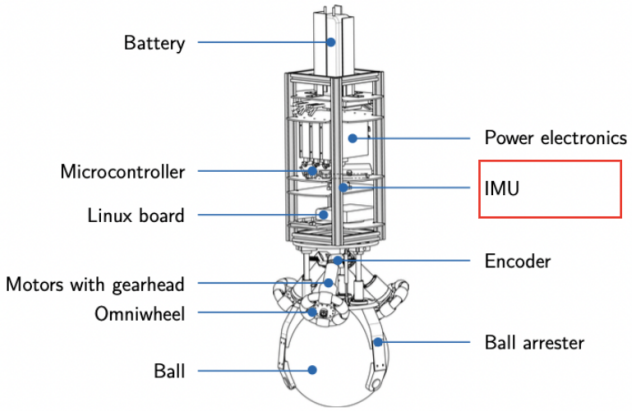


Figure 1.2: System description of Rezero

- **Balance.py** - The upper omniwheel structure's main function is to balance itself on the body as in it has to keep making rapid adjustments when the tilt reaches a threshold value in any particular direction to take itself back to its position of stability.
- **Drive.py** - The lower omniwheel structure's main function is to facilitate movement of the body in all directions, by utilising the roller and wheel function of the omniwheel to move the body under the upper ballbot incase of instability.
- **IMU.py and PID.py** - The code facilitates the function of the tilting head.
- **KalmanFilter.py** - The code utilises the kalman filter to reduce noise in making predictions of the positions of the robot.
- **OmniBot.py** - The code contains the instructions for the movements of the ballbot.

- ➢ **StepperMotor.py** - The code allows the robot to spin around all the axes to facilitate movement within small area.
- ➢ **Twitch.py** - Tilt Cleanup
- ➢ **Hello.py** - The code adds a remote control allowing the user to direct its moves.

Code: Balance.py

```
import Adafruit_BBIO.GPIO as GPIO
from time import sleep
from StepperMotor import *
from termcolor import colored
import math
from IMU import *
from Omnibot import *
from PID import *
```

Import statements -
- **Adafruit_BBIO.GPIO** is basically the library that encodes the instructions to its hardware counterpart known as **BeagleBone Black.** The appropriate replacement we will be using is **Arduino Mega 2560.**

-StepperMotor,IMU,Omnibot & PID is the python file that encodes the stepper motors,IMU and the bot, and all the functions and variables are imported from that file.

StepperMotor Functions -
- **StepperMotor** is a class defined inside the StepperMotor.py, and has four functions inside it along with the **_init_** function file.

> The first 2 arguments correspond to the pins for stepping (**StepPin**) and for direction (**DirectionPin**), assigned in the _init_ function.

> **<Doubt>**[1]

> The three StepperMotor functions are used for the three stepper motors defined as - **Yellow**, **Green** and **Red**.

> For the different motors, the rpm is set for **forward** movements as well as the time for the motors to **sleep**. Following this, the stop function causes the motor to stop.

```
motorYellow = StepperMotor("P9_21", "P9_23", "P9_16", "P9_25", "P9_27", "P9_22", StepperMotor.HALF_STEP)
motorGreen = StepperMotor("P8_8", "P8_10", "P8_19", "P8_12", "P8_14", "P8_13", StepperMotor.HALF_STEP)
motorBlue = StepperMotor("P9_24", "P9_26", "P9_42", "P9_30", "P9_41", "P9_28", StepperMotor.HALF_STEP)

motorBlue.forward(10)
sleep(.25)
motorBlue.stop()

motorGreen.forward(10)
sleep(.25)
motorGreen.stop()

motorYellow.forward(10)
sleep(.25)
motorYellow.stop()

sleep(5)

imu = IMU(1)
imu.zero()

motorBlue.forward(10)
sleep(.25)
motorBlue.stop()

motorGreen.forward(10)
sleep(.25)
motorGreen.stop()

motorYellow.forward(10)
sleep(.25)
motorYellow.stop()
```

IMU Functions -

```
imu = IMU(1)
imu.zero()
```

- **IMU** is an integrated sensor consisting of a 3 axis accelerometer and 3 axis gyroscope.

> The _init_ function takes the readings of the accelerometer as well as gyroscope using two functions namely the **accelerometer** and **gyroscope**.

```
class IMU(object):

    DELAY_TIME = .02
    GYRO_NOISE = .001/2
    BIAS_NOISE = .003/2
    ACCEL_NOISE = .01/2

    def __init__(self, i2cBus = 1):
        self.gyro = Gyroscope(i2cBus, Gyroscope.DPS2000)
        self.accel = Accelerometer(i2cBus, Accelerometer.SCALE_A_8G)
        initOrientation = self.accel.getOrientation()
        self.roll = initOrientation.roll
        self.pitch = initOrientation.pitch
        self.yaw = initOrientation.yaw
        self.kalmanFilter = KalmanFilter(IMU.GYRO_NOISE, IMU.BIAS_NOISE, IMU.ACCEL_NOISE)
        self.thread = threading.Thread(target=self.updateOrientation)
        self.thread.daemon = True
        self.thread.start()

    def zero(self):
        self.gyro.zero()
        self.accel.zero()

    def updateOrientation(self):
        self.elapsedTime = IMU.DELAY_TIME
        while True:
            startTime = time.time()
            gyroX, gyroY, gyroZ = self.gyro.getRawData()
            accelOrientation = self.accel.getOrientation()
```

> Gyroscope class : It contains the registers for all the microcontrollers. **<Doubt>**[2]

> Accelerometer class : It contains the registers for all the microcontrollers. **<Doubt>**[3]

> getOrientation function : It utilises the getRawData function to obtained a bit array of the data from the accelerometer to be converted into the integer form. Following this, the Raw Data is processed to calculate the **roll, pitch and yaw**.

Necessary Formulae:

> **roll = tan inverse(y/z) + pi –> if roll > pi –> roll=roll-2*pi**

> **pitch = tan inverse(x/z) + pi –> if pitch > pi –>**

pitch=pitch-2*pi

Finally, the **Orientation Function** will be used to denote the roll and the pitch with zero **yaw** motion.

> KalmanFilter function : The Kalman Filter algorithm to filter out the **noise from the gyroscope**, the **bias that exists in the data** and the **accelerometer noise.**

```
def getOrientation(self):
    x, y, z = self.getRawData()
    roll = math.atan2(y, z) + math.pi
    pitch = math.atan2(x, z) + math.pi
    if roll > math.pi:
        roll = roll - 2 * math.pi
    if pitch > math.pi:
        pitch = pitch - 2 * math.pi

    return Orientation(math.degrees(roll), math.degrees(pitch), 0)
```

> Threading Function : **<Doubt>**[4]

OmniBot Functions -

```
zeroAngle = math.radians(103)
twitch = Omnibot()
```

- Omniwheel class :

> _init_ : It contains the **StepperMotor** function which takes in the **StepPin** and the **DirectionPin** as well as the wheel angle is taken.

```
class Omniwheel(object):

    def __init__ (self, stepPin, directionPin, wheelAngle):
        self.motor = StepperMotor(stepPin, directionPin)
        self.wheelAngle = wheelAngle

    def calculateVelocity (self, drivingAngle, drivingSpeed):
        w = self.wheelAngle
        d = drivingAngle
        wheelSpeed = drivingSpeed * (math.cos(w) * math.cos(d) + math.sin(w) * math.sin(d))
        if wheelSpeed > 0:
            self.motor.forward(wheelSpeed)
        else:
            self.motor.reverse(abs(wheelSpeed))

    def rotate (self, speed):
        if speed >= 0:
            self.motor.forward(speed)
        else:
            self.motor.reverse(abs(speed))
```

> calculateVelocity : The functions takes the **driving angle** as well as the **speed of the robot driving .**

Necessary Formulae:

> **wheel speed = drivingSpeed * (cos (wheelAngle) * cos (drivingAngle) + sin (wheelAngle) * sin(drivingAngle)) –> if**

wheelSpeed > 0 → forward movement at wheelSpeed | reverse movement at wheelSpeed

> rotate : The function takes a speed as argument
Necessary Formulae:
> **speed -> if speed >=0 -> forward movement at speed | reverse movement at speed**

- Omnibot class :
> _init_ : It uses the **Omniwheel function for the three omniwheels** taking as input the **StepPin** and the **DirectionPin** as well as the **WheelAngle.** Following it the SteppingMode function **<Doubt>**[5]

```
class Omnibot(object):

    def __init__ (self):
        self.blueWheel = Omniwheel("P9_22", "P9_24", math.radians(0))
        self.yellowWheel = Omniwheel("P9_42", "P9_41", math.radians(120))
        self.greenWheel = Omniwheel("P9_16", "P9_15", math.radians(240))
        self.steppingMode = SteppingMode("P9_23", "P9_25", "P9_27")
        self.steppingMode.setMode(5)

    def move (self, angle, speed):
        self.blueWheel.calculateVelocity(angle, speed)
        self.yellowWheel.calculateVelocity(angle, speed)
        self.greenWheel.calculateVelocity(angle, speed)

    def spin (self, speed):
        self.blueWheel.rotate(speed)
        self.yellowWheel.rotate(speed)
        self.greenWheel.rotate(speed)

    def stop (self):
        self.blueWheel.motor.stop()
        self.yellowWheel.motor.stop()
        self.greenWheel.motor.stop()
```

> move : It takes as input the angle(driving angle) and the speed(driving speed). This is used to then **calculateVelocity** of the three wheels - blue, yellow and green.

> spin : It takes as input the speed and instructs forward or reverse movement in each of the three wheels.

> stop : It sets the StepPin to zero, for each wheel causing it to stop.

Newly Defined Functions -

```
def getDriveAngle (x, y, zeroAngle):
        fallingAngle = math.atan2(x, y)
        if fallingAngle < 0:
            fallingAngle = fallingAngle + 2 * math.pi
        drivingAngle = fallingAngle + math.pi + zeroAngle
        if drivingAngle > 2 * math.pi:
            drivingAngle = drivingAngle - 2 * math.pi
        return drivingAngle

def getDriveSpeed (x, y):
    drivingSpeed = 200 * (x ** 2 + y ** 2) ** (.5)
    return drivingSpeed
```

- **getDriveAngle** :
It takes as input x and y values as well as the zeroAngle.
Necessary Formulae:
> **fallingAngle = tan inverse(x/y) –> if fallingAngle<0 –> fallingAngle = fallingAngle + 2*(pi)**

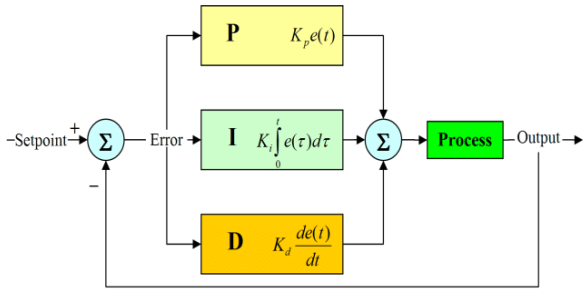
> **drivingAngle = fallingAngle + pi + zeroAngle –> if drivingAngle > 2*pi –> drivingAngle - 2*pi**

- getDriveSpeed :
It takes as input x and y values.
Necessary Formulae:
> **drivingSpeed = 200 * (x **2 + y **2) ** (0.5)**

```
pitch = []
roll = []
pitchPIDController = PIDController()
rollPIDController = PIDController()
count = 0
```

Main -

- **PID Controller** uses a control feedback mechanism to control process variables. The flowchart for the functioning of the controller :



Process Variable = P + I + D

```
class PIDController(object):

  Kcrit = 6.5
  # Pu = 2
  Kp = Kcrit / 1.7 #6
  Ki =  0#1 #  0 # 1
  Kd = .25 # .25 # .25 #.3
  K = .5
  GUARD_GAIN = 10

  def __init__ (self):
    self.totalError = 0
    self.lastError = 0

  def calculateSpeed(self, targetPosition, currentPosition):
    error = targetPosition - currentPosition
    proportionalTerm = PIDController.Kp * error
    self.totalError = self.totalError + error
    integralTerm = PIDController.Ki * max(min(PIDController.GUARD_GAIN, self.totalError), -PIDController.GUARD_GAIN)
    derivativeTerm = PIDController.Kd * (error - self.lastError)
    # self.lastError = error
    return max(min(PIDController.K * (proportionalTerm + integralTerm + derivativeTerm), 50), -50)
```

- PIDController class :

> **Initial value declaration** : Declaration of suitable PID coefficients by **PID tuning**.

> **_init_** : This considers the last and the total errors as zero.

> **calculateSpeed** : It takes the target position and the current position as arguments.
Necessary Formulae:

> **error = target position - current position**

> **proportional term = Kp * (error)**

> **total error = error + initial total error**

> **integral term = Ki * (max (min(guard gain coefficient, total error), - guard gain))**

> **derivative term = Kd * (error - last error)**

> **calculateSpeed = max(min(K * (proportional term + integral term + derivative term), 50), -50)**

```
while True:
    startTime = time.time()
    orientation = imu.getOrientation()
    driveAngle = getDriveAngle(math.radians(orientation.roll), math.radians(orientation.pitch), zeroAngle)
    driveSpeedY = pitchPIDController.calculateSpeed(0, orientation.pitch)
    driveSpeedX = rollPIDController.calculateSpeed(0, orientation.roll)
    # print "speedX:", driveSpeedX, "speedY:", driveSpeedY
    driveSpeed = (driveSpeedX ** 2 + driveSpeedY ** 2) ** (.5)
    driveSpeed = min(driveSpeed, 50)
    twitch.move(driveAngle, driveSpeed)

    count = count + 1
    pitch.append(orientation.pitch)
    roll.append(orientation.roll)

    # print colored('----------------------------------','cyan')
    # print colored('roll = {0:.2f}, pitch = {1:.2f}','white') .format(orientation.roll, orientation.pitch)
    # print "direction:", driveAngle, "speed:", driveSpeed
    # print "PID useful time:", (time.time() - startTime)
    sleep(max(.1 - (time.time() - startTime), 0))
    # print "PID controller loop time:", time.time() - startTime
```

- **Encoding the ball bot** :

| Keeping the condition true -

> **startTime** : Obtains system time from the time library.

> **orientation** : Obtains the roll and pitch (yaw = 0), using the **getOrientation** function.

> **driveAngle** : It calculates the **fall angle** based on the roll, pitch and the initial steady state angle.

Theory on DriveSpeedY and DriveSpeedX:

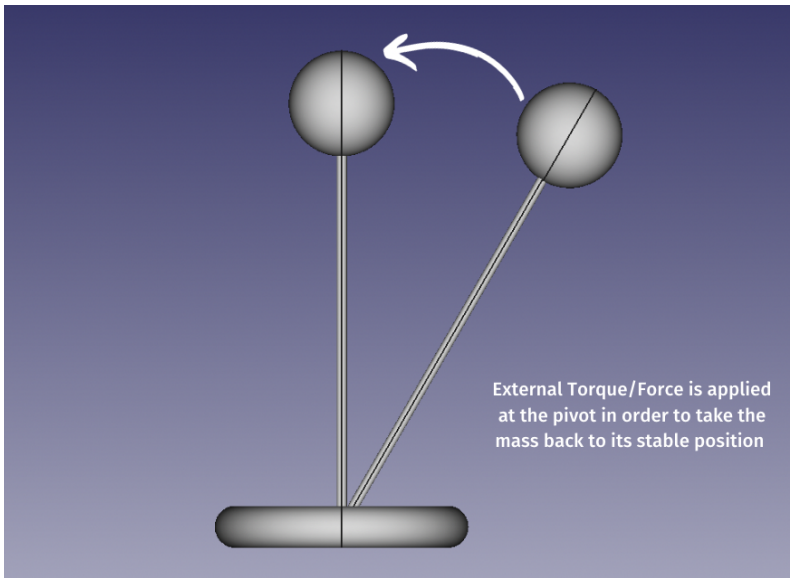
Since the ballbot works on the principle of the inverted pendulum, which means that the centre of mass of the bot, has to be driven towards the central position, as that is the position of stability, that is the ball has to move towards the original just so that it stays balanced.

> **driveSpeedY (V_y)** : The target position being the origin, it uses the PID controller to obtain the driving speed needed to change the pitch to zero.

> **driveSpeedX (V_x)** : The target position being the origin, it uses the PID controller to obtain the driving speed needed to change the roll to zero.

> **driveSpeed** : The net velocity is calculated by

[(V_x)^2 + (V_y)^2]^(0.5)



> **twitch** : The variable is defined as **Omnibot( )**, from which the **move** function is used with the **driveAngle** and **driveSpeed** as arguments.

- Finally, the pitch and the roll values are appended to the empty lists of pitch and roll.

> **sleep** causes the bot to sleep for a certain amount of time.

Code: Drive.py

```
import time
import math
from Omnibot import *
```

Import statements -

- Omnibot is the python file that encodes the bot, and all the functions and variables are imported from that file.

Main -

> **twitch** : The variable is defined as **Omnibot( ).** From here **move** function is used with angle 77 degrees and speed of 2 units

> Following the robot sleeps for 2 units of times.

```
twitch = Omnibot()
twitch.move(math.radians(77), 2)
time.sleep(2)

while True:
    #twitch.move(math.radians(77), 2)
    time.sleep(60.0 / 400 / 50)
```

Code: KalmanFilter.py -

```
def filter (self, accelerometerAngle, gyroRate, dt, filteredAngle, bias, P):

        # predict
        predictedAngle = filteredAngle + dt * (gyroRate - bias)
        P[0][0] = P[0][0] - dt * (P[1][0] + P[0][1]) + self.gyroNoise * dt
        P[0][1] = P[0][1] - dt * P[1][1]
        P[1][0] = P[1][0] - dt * P[1][1]
        P[1][1] = P[1][1] + self.biasNoise * dt

        # correct
        innovation = accelerometerAngle - predictedAngle
        noise = P[0][0] + self.accelNoise
        kalmanGainAngle = P[0][0] / noise
        kalmanGainBias = P[1][0] / noise

        filteredAngle = predictedAngle + kalmanGainAngle * innovation
        bias = bias + kalmanGainBias * innovation
        P[0][0] = P[0][0] - kalmanGainAngle * P[0][0]
        P[0][1] = P[0][1] - kalmanGainAngle * P[0][1]
        P[1][0] = P[1][0] - kalmanGainBias * P[0][0]
        P[1][1] = P[1][1] - kalmanGainBias * P[0][1]

        return filteredAngle, bias, P
```

Logic and Working : Kalman Filter Algorithm

Code:StepperMotor.py -

```
class SteppingMode:

    class __SteppingMode:
        def __init__(self, m0, m1, m2):
            GPIO.setup(m0, GPIO.OUT)
            GPIO.setup(m1, GPIO.OUT)
            GPIO.setup(m2, GPIO.OUT)
            self.m0 = m0
            self.m1 = m1
            self.m2 = m2

    instance = None

    def __init__(self, m0, m1, m2):
        if not SteppingMode.instance:
            SteppingMode.instance = SteppingMode.__SteppingMode(m0, m1, m2)

    def setMode(self, mode):
        SteppingMode.instance.mode = mode
        GPIO.output(SteppingMode.instance.m0, mode & 1 == 1)
        GPIO.output(SteppingMode.instance.m1, mode & 2 == 2)
        GPIO.output(SteppingMode.instance.m2, mode & 4 == 4)

    @staticmethod
    def getFrequency(rpm):
        if SteppingMode.instance.mode > 5:
            mode = 5
        else:
            mode = SteppingMode.instance.mode
        return rpm * 200 * (2 ** mode) / 60
```

- **_SteppingMode** class : The _init_ function takes input of 3 output pins, and initialising the instance as **None.**

> _init_ function : The first function checks that if the instance is not None, then it sets it as SteppingMode._SteppingMode(m0,m1,m2) value.
> setMode : It takes the mode as argument and basically alters the instance in each port to give a **high or low output.**
> getFrequency : It takes in the rpm as input as well a condition that is the mode is greater than 5, then it sets the mode as 5. If the condition is false, it keeps the mode as mode itself. It finally returns the frequency.

Necessary Formulae:
frequency = rpm*200 + (2^mode)/60

---

end