NORTHWEST NAZARENE UNIVERSITY

Application of Robotic Turf Mowing

THESIS
Submitted to the Department of Mathematics and Computer Science
in partial fulfillment of the requirements
for the degree of
BACHELOR OF SCIENCE

Aleesha Michai Chavez
2021

THESIS
Submitted to the Department of Mathematics and Computer Science
in partial fulfillment of the requirements
for the degree of
BACHELOR OF SCIENCE

Aleesha Michai Chavez
2021

Application of Robotic Turf Mower

Author: Aleesha Chavez
Aleesha Chavez

Approved: Dale A Hamilton
Dale Hamilton, PhD, Department of Mathematics and Computer Science,
Faculty Advisor

Approved: Cathy Becker
Cathy Becker, PhD , Department of English
Second Reader

Approved: Barry Myers
Barry L. Myers, Ph.D., Chair,
Department of Mathematics & Computer Science

ABSTRACT

Application of a Robotic Turf Mower.

CHAVEZ, ALEESHA (Department of Mathematics and Computer Science), HAMILTON, DR. DALE (Department of Mathematics and Computer Science).

Traditionally turf fields are mowed two times a week, and cost up to 60 dollars an acre a month, which annually ends up costing a turf company that averages 30 acres 43,000 dollars and up. Turf companies have to deal with high labor costs, keeping up with maintenance of machinery, and spending precious time on lawn upkeep. A need exists for more efficient ways for companies to mow. The purpose of the project was to create a robotic turf mower that can cleanly mow acres of grass at a time, with no human help, and then to look into the potential benefits and applications of using a robotic mower for turf fields and other large grass fields, instead of mower operated by man. Initial efforts were focused on programming development on a prototype robot model provided by Green Cut Robotics (GCR). Results show promise in precise lawn mowing capabilities with pre-determined coordinates for accurate and straight-line lawn mowing. Further work will need to be done for cases when the GPS has no signal, and for the robot to have the capability of motion detection. Results strongly support the full functionality of a robotic mower that a turf company money and time by cutting operation costs, and having the capability to move outside the turf field domain.

ACKNOWLEDGEMENTS

This project was done with the support of multiple individuals: I would like to thank GCR (Green Cut Robotics) for the funding to make this project possible. Robert Lawton for employing me and guiding me on the mechanics of the robot, and Jess Tate for guidance in controls and all electrical aspects of this project. Dr. Dale Hamilton for guiding me through my entire academic journey and helping me write this Thesis. Dr. Barry Myers for his advice and edits through this paper.

TABLE OF CONTENTS

LIST OF FIGURES

1. PROJECT BACKGROUND

This project aimed to improve current turf mowing operations. A prototype of a robotic turf mower called the Mark 2.7 was programmed to do autonomous mowing patterns of straight lines and certain degrees of turns. This would provide a cleaner cut, faster mowing times, and less maintenance cost than current mowing done by man. Software development for a working prototype from scratch can often be a difficult process, so the project involved a team of people working together for GCR (Green Cut Robotics), to combine electrical and mechanical engineering with programming for the best results possible.

Traditionally turf fields are mowed two times a week by a hired worker. The typical cost of mowing an acre twice a week of turf is fifty up to sixty dollars per acre, and cost up to 120 dollars an acre a month, which annually ends up costing a turf company of thirty acres 43,200 dollars and up. Labor costs of hiring help to mow can charge a turf company around 10 dollars an hour per worker, in total for the year adding up to 1,500 dollars on top of the cost of mowing. With traditional mowing, a person manning a lawn mower can only mow during the daylight hours, unless massive lighting equipment is used, costing a company even more money. Additionally, according to the Environmental Protection Agency (EPA), traditional lawn mowing emits eighty-nine pounds of CO2 and 34 pounds of other pollutants into the atmosphere from running on gasoline each year per one lawn mower. For turf fields, this more than doubles with the larger size of the mowers.

Implementing a robotic turf mower will save a company labor cost and machine maintenance cost. This is due to not having to hire someone to mow, and selling the

product as a service. The environmental factors of using a robotic turf mower include no greenhouse gas emissions, as the mower is run purely off of battery that will be charged by solar panels on top of a storing shed for the mower.

For this project, all hardware of the Mark2.7, the model of the robot for this project, were background materials, not the focus of this project. The list of electronic hardware was extensive, and for the purpose of this project, I will only be talking about the computing module, the sensors it collected data from and the controllers it sent data to for controlling the mechanical parts of the Mark 2.7. The list of these components is as follows: A computing module called a Raspberry Pi controller where all python code for the Mark2.7 was run, and a Arduino I2C Interface, for sharing data between all sensors, and a Berry Inertial Measurement Unit (IMU) used for Gyroscope (GYRO) sensor and Global Navigation Satellite System (GNSS) sensor purposes. The GYRO allowed the robot to read heading values, also known as Z-axis values, that read what direction the robot is pointed in. A consumer grade GNSS would not be adequate for the scope of this project, as accuracy is not as high. The Real Time Kinematic (RTK) GNSS allowed the robot to read latitude and longitude values. The mechanical hardware involved in the programming process included the following: Optical shaft encoders from Vex Robotics for reading the robot's speed, Victor SPX Motor Controllers for powering both wheels, and a circular disk cutting system on the front of the Mark2.7. This Mark2.7 model is shown below in Figure 1:

Figure 1: Mark2.7 Prototype

2. PROJECT OBJECTIVE

This project aimed to develop, test, and document source code for an agricultural robot that autonomously mows turf fields. This project did not attempt to actually build the robot, as there was a separate mechanical team for GCR. The focus of the project was on the development of the programming of the robot. There were moments of assisting the mechanical team but it was not the main focus of the project. While there are many different languages to choose from for the programming of the agricultural robot, the Python was selected as the main programming language for the Mark2.7. This selection was based off of the preferences of the lead of the electrical engineering team, Jess Tate. Besides program development, testing was a major objective of this project, as the majority of the twelve-week project was spent testing and debugging the code for the Mark2.7. The last three weeks of this project were used for optimization and

improvement of the autonomous driving of the Mark2.7. The project followed a research, development, and testing approach. In the research part of this project, the first step was to familiarize with the GCR's first robotic prototype, the Mark2.0, and perform literary research on controls. The next step was to develop hypotheses on what programming strategies would best work for the second prototype. The developing phase was developing software for the Mark2.7, the second prototype for FBN's robotic mower. The testing phase was testing and debugging written software for the second robotic model. The last phase was improving the current model.

3. METHODS: DEVELOPMENT PART 1

This project was completed within the span of a few months, starting in May of 2020 and ending in August of 2020, corresponding to a summer internship. After the initial research was conducted, it was decided that the best programming strategy for this robot would be an object-oriented program, that utilized the single responsibility principle, where every class in the program has responsibility over a single part of that program's functionality, in order to efficiently manage and organize all sensors and hardware that the Mark2.7 utilizes. Development for the Mark2.7 source code started from the ground up, as no previous code was used from other prototypes. With Object Oriented principles in mind, the first step to development was creating source code that controls basic motor and sensor function. The layout of this program design included hiding the sensor and motor layers, so when new features are added, they cannot break the basic functionality

of the Mark2.7. The base components of the Mark2.7 that control functionality are the motors and GNSS and GYRO sensors. The first phase of development focused on creating python files of basic motor function and sensor function. The files Motor.py, GNSS.py, and GYRO.py were created. As this project was done as part of a team, the focus of the paper will be on motor function. In Motor.py, two main things are controlled, the servo motors and the rotary encoders (see appendix A), that are each assigned to pins on the Arduino board. The servo motors are used for moving and stopping, and the rotary encoders convert motion into some electrical signal and are used for distance tracking on the Mark2.7. Below, Figure 2 shows the abstract program design created before development began.

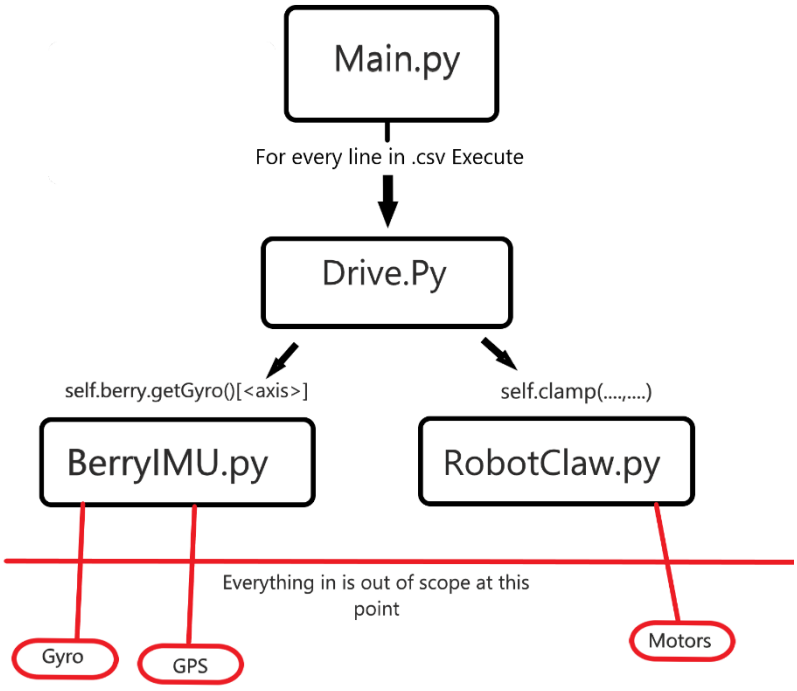

Figure 2: Abstract of Program Design

4. METHODS: TESTING PART 1

Once three source code python files (Gyro.py, Gnss.py, and Motor.py) were written, testing began (see Appendices B through D). The Mark2.7 was put up onto blocks, and the newly developed python code was run from the Mark2.7's raspberry pi. In order to connect to the robot remotely, a laptop would connect to the raspberry pi on the Mark2.7 using the Secure Shell network protocol (SSH). Using MobaXterm, a specialized terminal that provides all remote desktop tools in one executable file, a test program that was developed and run in order to test the functionality of the sensors and the motors. Once the programs were debugged and all errors and issues fixed, higher-level programs were written for the next development phase of the project.

5. METHODS: DEVELOPMENT PART 2

In the next phase of development, the robot was edited to have consistent and even mowing, and the speed of the Mark2.7 controlled. Development of higher-level detailed programs included creating source code that would determine stabilized speed, normalized driving speeds, on course heading values, straight paths, and turns. The higher level code of Drive.py (see appendix E) was based on the lower level stabilized power function code from Motor.py. This included the methods for normalizing speed, speed consistency, setting turn styles, and driving straight. Loop.py is the main program where all robot functionality is called once the first lines of the input text file are read. The Loop.py python file (see appendix F) also contains the information to create a logger that plots the Mark2.7's GPS points onto a digital map. This functionality was added for future implication, where the robot operator can track the movement of the robot on a map, including its current and past speeds on the directed paths driven. The last stage of

development included finalizing Main.py (see appendix G), which read in the robot's input parameters from the terminal, and calls Loop.py accordingly. The input parameters were designed to give the user flexibility on which sensors they want to utilize for the specific driving patterns. The next section elaborates on the input parameters and driving patterns. In order to keep all of the fast pace development organized, a dataflow diagram was created to organize all classes and methods, with arrows describing inheritance. The final code diagram for the Mark2.7 contains the flow of functionality for the robot, and is shown below in Figure 3.

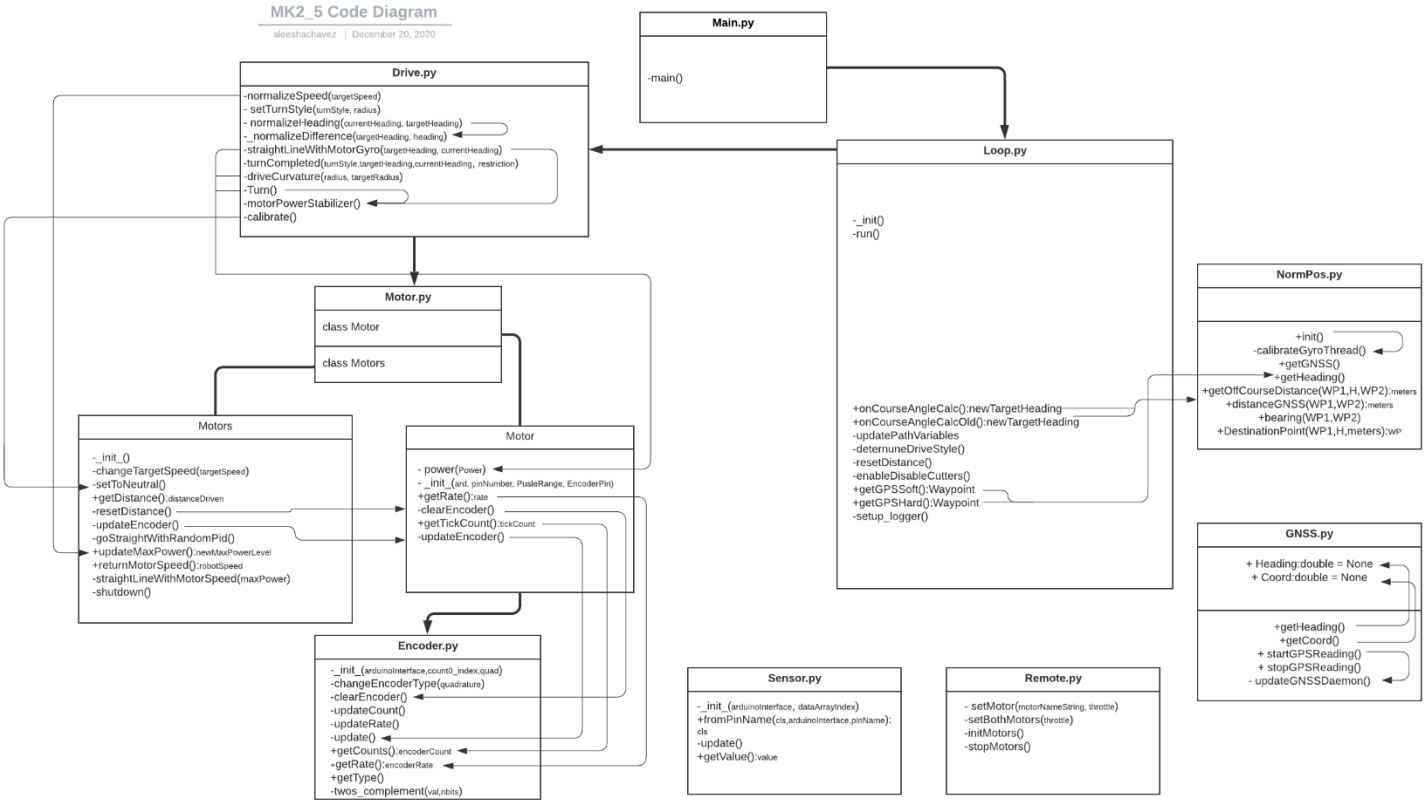

Figure 3: Dataflow Diagram of Mark2.7 Program

6. METHODS: TESTING PART 2

Once initial development was completed, the bulk of the project time was spent testing the autonomous pathways, speeds, and turning functionality on the Mark2.7. In order for the Mark2.7 robot to run, it took several input parameters, and a specific input text file for the driving style. These four parameters include debugging, enabling the encoders, enabling the GPS, and specifying a base point (only if the waypoint input text file is in use). The debugging parameter was specified if the user wanted to see output of the Mark2.7's run on the command prompt for debugging purposes. The encoder and parameter were for enabling the encoders to track distance. The GPS and specifying base point parameter enable the GPS or set a base point location (only when using a waypoint input text file). The sets of input parameters were run and tested, as the Mark2.7 now had three different types of input txt files that could be tested. These input text files contain the designated paths and speeds for a specific mowing pattern. Each input text file created was a specific driving pattern, whether right or left turns, how long each path was, and what square area was covered. The Mark2.7 would read through each line in the text file until completed, signaling the end of the chosen path. The first type was a 'H' Heading text file, containing lines of Heading, Distance, Speed, TurnStyle, and Cutting values. The 'H' Heading file only uses the gyroscope for positioning and follows a straight-line based off its initial position. The 'W' Waypoint text file contains lines of Latitude, Longitude, Speed, TurnStyle, Cutting, and Degree of Turn. The 'W' Waypoint file drives according to specific Latitude and Longitude points, as each text file created is for a specific area. This second technique of driving allows for mowing areas that are not perfect squares, and where lines may not always be straight. The last driving style is the 'G' Geometric text file, which contains lines of X, Y, Speed, TurnStyle, Degree, and

Cutter. The ‘G’ Geometric file drives according to an X, Y coordinate plane, in order for certain shapes to be mowed, where shorter, specific paths can be done. Figures 4, and 5 show examples of these three input text files below.

7. RESULTS

Once final development was finished within the twelve-week timeline, the Mark2.7 was able to do straight-line paths and slightly overlapping turns for up to 90 meters. (in theory it could go farther but 90 was what was tested). It also had the capability of any pattern involving straight lines, 180 and 90 degree turns. All these capabilities were repeatedly tested on a growing turf field with cutting abilities turned off in case of an emergency. Results that were handed off to the employer included edited videos taken from a drone of the Mark2.7 running on a turf field, so future investors and product owners could see its capabilities. While the Mark2.7 is running, two log outputs are created in order to track the robot’s movements and statistics. The final resulted waypoint heading file (Mark2.7 runs off of inputted latitude and longitude points) log output is below in Figure 4. The final resulted speed and distance log file output is below in Figure 5. Full outputs files are included in appendices H and I.

```
1
2
3 >>>>>>>>>>>>>>>>NEW Pathfile Read<<<<<<<<<<<<<<<
4 Initial Waypoint(43.51287479100003, -116.48792405911468)
5 Target Waypoint: (43.5128, -116.48786)
6 currentHeading 74.786122
7 currentHeading 74.7876195546
8 currentHeading 74.790122592
9 currentHeading 74.7911837214
10 currentHeading 74.7959144146
11 Current Waypoint: Lat: 43.5128747934835 Lon: -116.48792404642217
12 TargetHeading142.62660651184842
13 currentHeading 75.0162021298
14 currentHeading 75.6051604032
15 currentHeading 76.558512524
16 currentHeading 77.1552880544
17 currentHeading 79.3329131769
```

Figure 4: Waypoint and Heading Log File

```
5516 Total Distance Driven: 0
5517 Robot Speed: 0.46731405471832427m/s
5518 Total Distance Driven: 0
5519 Robot Speed: 0.4662372942235815m/s
5520 Total Distance Driven: 0
5521 Robot Speed: 0.443759918895828m/s
5522 Total Distance Driven: 0
5523 Robot Speed: 0.4914065707881917m/s
5524 Total Distance Driven: 0
5525 Robot Speed: 0.49019521523160625m/s
5526 Total Distance Driven: 0
5527
5528
5529 >>>>>>>>>>>>>>>>NEW Pathfile Read<<<<<<<<<<<<<<<
5530 TargetDistance20.0
5531 Robot Speed: -1.2720579294766365m/s
5532 Total Distance Driven: 0.003768661731599389
5533 Robot Speed: 0.5452445955253259m/s
5534 Total Distance Driven: 0.10281042680558415
5535 Off course Distance: -0.02745351212762 87m
5536 Robot Speed: 0.5830658079031626m/s
5537 Total Distance Driven: 0.14816896264661966
5538 Robot Speed: 0.588449610376876m/s
```

Figure 5: Speed and Distance Log File

8. SHORTCOMINGS

The project had a few shortcomings from deadline time that kept it from meeting all end goals of the twelve-week project. The list of these shortcomings includes inaccurate GNSS readings that caused the Mark2.7 to go off course slightly if the GPS readings were bad. The way to fix the bad readings was to replace the current RTK GNSS on the Mark2.7 with a newer, more accurate model. Another shortcoming was the inability to

test the cutters on the mower due to the mechanical team and programming team not being able to coordinate a testing time within the deadline because of mechanical issues that were being fixed throughout the process.

9. FUTURE WORK

The goal of this project was to complete the second prototype of a robotic turf mower, with future plans of creating a final product that will be sold as a service to turf companies, and others in need of larger lawn mowing services. For the scope of the Mark2.7 prototype, future work includes replacing the current GNSS with a newer model for better readings, improving the current mechanical design of the robot with three to four times the size of cutter blades, for quicker mowing time on larger areas, and implementing object detection for the robot to avoid items such as pipes used for watering in fields. Another aspect of future work included improvement to the current battery charging system. The current set up was 4 lithium-ion batteries that took an average of 6 to 7 hours to fully charge and had to be plugged in by hand. Future work would include a charging system inside of a shed with solar panels on top for charging power that the Mark2.7 could drive into with a remote control.

10. CONCLUSION

Overall this project was a good initial study into the viability of using an autonomous robot for turf mowing. This project provided the groundwork for future improved models of the Mark2.7 robot and the models following it, despite some mechanical and electrical setbacks. Through the process of many weeks of code revision and improvement, the prototype Mark2.7 was fully functional to run autonomously on even terrain turf fields. It had the capability to run autonomously for up several hours on a fully charged battery.

REFERENCES

Banks, J. L. (n.d.). National emissions from lawn and garden equipment. Retrieved March 27, 2021, from https://www.epa.gov/sites/production/files/2015-09/documents/banks.pdf

Chang, K. (2016). *Introduction to geographic information systems*. New York, NY: McGraw-Hill Education.

APPENDICES

A. Code

A.A. Encoder.py

```
#!/user/bin/python3
# Author: JTATE
# Encoder class using ArduinoI2Cinterface
# To do: add support for using other encoder rate measurements such as RPM (need
ticks per revolution), linear speed (additionally need wheel diameter)


from ArduinoI2Cinterface import ArduinoI2Cinterface
import traceback
import sys
import time
import busio




class Encoder:

    def __init__(self, arduinoInterface, count0_index, quad = True):

        self.ardInt = arduinoInterface  # ArduinoI2Cinterface object that we use to
communicate with arduino

        # Set some indices. If spec changes then these may need changed
        self.countIndex = count0_index
        self.rateIndex = self.countIndex + 4
        self.typeIndex = self.countIndex + 8
        self.clearIndex = self.countIndex + 128
        self.changeTypeIndex = self.clearIndex + 1

        # Let's make sure we are initialized by setting the encoder type and getting
        self.isQuadrature = quad
        self.changeEncoderType(self.isQuadrature)
        self.clearEncoder()     # Clear any encoder ticks on init
        self.encoderCount = 0   # Unit in ticks
```

```
        self.encoderRate = 0    # Unit in ticks/sec
        self.update()


    def changeEncoderType(self, quadrature = True):
        if quadrature:
            #print("Setting 1")
            self.ardInt.writeByte(self.changeTypeIndex, 1)    # Quadrature can be
any non-zero byte. Using 1 because easy
        else:
            #print("Setting 0")
            self.ardInt.writeByte(self.changeTypeIndex, 0)    # Single type must be
zero

    def clearEncoder(self):
        self.ardInt.writeNoData(self.clearIndex)

    def updateCount(self):
        self.encoderCount =
self.twos_complement(self.ardInt.readLong(self.countIndex), 32)

    def updateRate(self):
        self.encoderRate =
self.twos_complement(self.ardInt.readLong(self.rateIndex), 32)

    def update(self):
        self.updateCount()
        self.updateRate()

    def getCounts(self):
        return self.encoderCount

    def getRate(self):
        return self.encoderRate

    def getType(self):
        return self.ardInt.readByte(self.typeIndex)

    def twos_complement(self, val, nbits):
        # Compute the 2's complement of int value val
        # Shamelessly stollen off the internet
        if val < 0:
            val = (1 << nbits) + val
```

```
        else:
            if (val & (1 << (nbits - 1))) != 0:
                # If sign bit is set.
                # compute negative value.
                val = val - (1 << nbits)
        return val

# Just for testing
def main():
    # For testing

    ard = ArduinoI2Cinterface(address=0x9)
    r_enc = Encoder(ard, count0_index = 4, quad = True)
    l_enc = Encoder(ard, count0_index = 16, quad = True)
    while True:
       r_enc.update()
       l_enc.update()
       print("R Counts:", r_enc.getCounts(), "L Counts:", l_enc.getCounts(), "R
Rate:", r_enc.getRate(), "L Rate:", l_enc.getRate())
       time.sleep(1)



if __name__ == "__main__":
    try:
        main()
    except KeyboardInterrupt:
        print("Ctrl-C Detected. Exiting")
        sys.exit(0)
    except:
        #GPIO.cleanup()
        print("Other Exception. Exiting")
        traceback.print_exc()
        sys.exit(0)
```

A.B. GYRO.py

```
Import smbus2
bus = smbus2.SMBus(3)
from .LSM9DS0 import *
from .LSM9DS1 import *
import time
import math
import datetime
from numpy import median
from threading import Thread
import traceback
import sys
#from arduinoInterface import arduinoInterface


# JTATE Global Variables
RAD_TO_DEG = 57.29578
M_PI = 3.14159265358979323846
G_GAIN =  0.00875 * 361/360 #0.070 #* 368/360  # [deg/s/LSB]  If you change the dps
for gyro, you need to update this value accordingly. JTATE multiply by empirical
factor to make more accurate
AA =  0.40      # Complementary filter constant


class BerryIMU:

    def __init__(self, usd=1, gps=True, initialHeading = 0.0):
        #usd is upside down. 1 or mincer, 0 on minibot
        # Current angle headings
        self.gyroXangle = 0.0
        self.gyroYangle = 0.0
        self.gyroZangle = 0.0


        # Bias Calibration Values. These default values are from one trial, will be
set when new calibration happens
        self.GYRxBias = -2.74
        self.GYRyBias = -1.05
        self.GYRzBias = -1.68


        self.upsideDown = usd # If upside down is 1, invert the angle output
```

```
        self.previousTime = datetime.datetime.now()
        self.running = True
        self.LSM9DS0 = 0

        self.gpsEnabled = False

        if gps:
            self.gpsEnabled = True
            print("Initializing GPS")
            #self.ardInt = arduinoInterface()
            if True:#self.ardInt.connected == 1:  # Launch Thread
                #self.ardIntThread = Thread(target=self.ardInt.run)
                #self.ardIntThread.daemon = True
                #self.ardIntThread.start()
                self.initGPSHeading = initialHeading#self.ardInt.heading###CHANGE
THIS
                self.hGPSDelta = 0
                self.numRotations = 0


    def terminate(self):
        self.running = False

    def run(self):
        while self.running:
            self.updateGyro()
            time.sleep(0.01)#.01 second sleep should this be our update frequency


    def detectIMU(self):
        #Detect which version of BerryIMU is connected.
        #BerryIMUv1 uses the LSM9DS0
        #BerryIMUv2 uses the LSM9DS1




        try:
            #Check for LSM9DS0
            #If no LSM9DS0 is conencted, there will be an I2C bus error and the
program will exit.
```

```
            #This section of code stops this from happening.
            #print("LSM9DS0 Waiting")
            LSM9DS0_WHO_G_response = (bus.read_byte_data(LSM9DS0_GYR_ADDRESS,
LSM9DS0_WHO_AM_I_G))
            LSM9DS0_WHO_XM_response = (bus.read_byte_data(LSM9DS0_ACC_ADDRESS,
LSM9DS0_WHO_AM_I_XM))
            #print("LSM9DS0 Detected")
        except IOError as e:
            print ('')        #need to do something here, so we just print a space
        else:
            if (LSM9DS0_WHO_G_response == 0xd4) and (LSM9DS0_WHO_XM_response ==
0x49):
                #print ("Found LSM9DS0")
                self.LSM9DS0 = 1

            return True




        try:
            #Check for LSM9DS1
            #If no LSM9DS1 is conencted, there will be an I2C bus error and the
program will exit.
            #This section of code stops this from happening.
            #print("LSM9DS1 Waiting")
            LSM9DS1_WHO_XG_response = (bus.read_byte_data(LSM9DS1_GYR_ADDRESS,
LSM9DS1_WHO_AM_I_XG))
            LSM9DS1_WHO_M_response = (bus.read_byte_data(LSM9DS1_MAG_ADDRESS,
LSM9DS1_WHO_AM_I_M))
            #print("LSM9DS1 Detected")
        except IOError as f:
            print ('')        #need to do something here, so we just print a space
        else:
            if (LSM9DS1_WHO_XG_response == 0x68) and (LSM9DS1_WHO_M_response ==
0x3d):
                #print ("Found LSM9DS1")
                self.LSM9DS0 = 0

            return True


        #time.sleep(1)
```

```
    return False


def writeAG(self, register,value):
        bus.write_byte_data(ACC_ADDRESS , register, value)
        return -1


def writeACC(self, register,value):
        bus.write_byte_data(ACC_ADDRESS , register, value)
        return -1


def writeMAG(self, register,value):
        bus.write_byte_data(MAG_ADDRESS, register, value)
        return -1


def writeGRY(self, register,value):
        bus.write_byte_data(GYR_ADDRESS, register, value)
        return -1


def readACCx(self):
    acc_l = bus.read_byte_data(ACC_ADDRESS, OUT_X_L_XL)
    acc_h = bus.read_byte_data(ACC_ADDRESS, OUT_X_H_XL)
    acc_combined = (acc_l | acc_h <<8)


    return acc_combined  if acc_combined < 32768 else acc_combined - 65536


def readACCy(self):
    acc_l = bus.read_byte_data(ACC_ADDRESS, OUT_Y_L_XL)
    acc_h = bus.read_byte_data(ACC_ADDRESS, OUT_Y_H_XL)
    acc_combined = (acc_l | acc_h <<8)


    return acc_combined  if acc_combined < 32768 else acc_combined - 65536


def readACCz(self):
    acc_l = bus.read_byte_data(ACC_ADDRESS, OUT_Z_L_XL)
```

```
    acc_h = bus.read_byte_data(ACC_ADDRESS, OUT_Z_H_XL)
    acc_combined = (acc_l | acc_h <<8)


    return acc_combined  if acc_combined < 32768 else acc_combined - 65536


def readMAGx(self):
    mag_l = bus.read_byte_data(MAG_ADDRESS, OUT_X_L_M)
    mag_h = bus.read_byte_data(MAG_ADDRESS, OUT_X_H_M)
    mag_combined = (mag_l | mag_h <<8)
    return mag_combined  if mag_combined < 32768 else mag_combined - 65536


def readMAGy(self):
    mag_l = bus.read_byte_data(MAG_ADDRESS, OUT_Y_L_M)
    mag_h = bus.read_byte_data(MAG_ADDRESS, OUT_Y_H_M)
    mag_combined = (mag_l | mag_h <<8)


    return mag_combined  if mag_combined < 32768 else mag_combined - 65536


def readMAGz(self):
    mag_l = bus.read_byte_data(MAG_ADDRESS, OUT_Z_L_M)
    mag_h = bus.read_byte_data(MAG_ADDRESS, OUT_Z_H_M)
    mag_combined = (mag_l | mag_h <<8)


    return mag_combined  if mag_combined < 32768 else mag_combined - 65536


def readGYRx(self):
    try:
        gyr_l = bus.read_byte_data(GYR_ADDRESS, OUT_X_L_G)
        gyr_h = bus.read_byte_data(GYR_ADDRESS, OUT_X_H_G)
    except IOError:
        print ("IO Error")
        return "e"
    gyr_combined = (gyr_l | gyr_h <<8)


    return gyr_combined  if gyr_combined < 32768 else gyr_combined - 65536
```

```
def readGYRy(self):
    try:
        gyr_l = bus.read_byte_data(GYR_ADDRESS, OUT_Y_L_G)
        gyr_h = bus.read_byte_data(GYR_ADDRESS, OUT_Y_H_G)
    except IOError:
        print ("IO Error")
        return "e"
    gyr_combined = (gyr_l | gyr_h <<8)


    return gyr_combined  if gyr_combined < 32768 else gyr_combined - 65536


def readGYRz(self):
    try:
        gyr_l = bus.read_byte_data(GYR_ADDRESS, OUT_Z_L_G)
        gyr_h = bus.read_byte_data(GYR_ADDRESS, OUT_Z_H_G)
    except IOError:
        print ("IO Error")
        return "e"
    gyr_combined = (gyr_l | gyr_h <<8)


    return gyr_combined  if gyr_combined < 32768 else gyr_combined - 65536


def writeACC(self, register,value):
    if(self.LSM9DS0):
        bus.write_byte_data(LSM9DS0_ACC_ADDRESS , register, value)
    else:
        bus.write_byte_data(LSM9DS1_ACC_ADDRESS , register, value)
    return -1


def writeMAG(self, register,value):
    if(self.LSM9DS0):
        bus.write_byte_data(LSM9DS0_MAG_ADDRESS, register, value)
    else:
        bus.write_byte_data(LSM9DS1_MAG_ADDRESS , register, value)
        return -1
```

```
def writeGRY(self, register,value):
    if(self.LSM9DS0):
        bus.write_byte_data(LSM9DS0_GYR_ADDRESS, register, value)
    else:
        bus.write_byte_data(LSM9DS1_GYR_ADDRESS , register, value)
        return -1


def readACCx(self):
    if (self.LSM9DS0):
        acc_l = bus.read_byte_data(LSM9DS0_ACC_ADDRESS, LSM9DS0_OUT_X_L_A)
        acc_h = bus.read_byte_data(LSM9DS0_ACC_ADDRESS, LSM9DS0_OUT_X_H_A)
    else:
        acc_l = bus.read_byte_data(LSM9DS1_ACC_ADDRESS, LSM9DS1_OUT_X_L_XL)
        acc_h = bus.read_byte_data(LSM9DS1_ACC_ADDRESS, LSM9DS1_OUT_X_H_XL)


    acc_combined = (acc_l | acc_h <<8)
    return acc_combined  if acc_combined < 32768 else acc_combined - 65536


def readACCy(self):
    if (LSM9DS0):
        acc_l = bus.read_byte_data(LSM9DS0_ACC_ADDRESS, LSM9DS0_OUT_Y_L_A)
        acc_h = bus.read_byte_data(LSM9DS0_ACC_ADDRESS, LSM9DS0_OUT_Y_H_A)
    else:
        acc_l = bus.read_byte_data(LSM9DS1_ACC_ADDRESS, LSM9DS1_OUT_Y_L_XL)
        acc_h = bus.read_byte_data(LSM9DS1_ACC_ADDRESS, LSM9DS1_OUT_Y_H_XL)


    acc_combined = (acc_l | acc_h <<8)
    return acc_combined  if acc_combined < 32768 else acc_combined - 65536


def readACCz(self):
    if (self.LSM9DS0):
        acc_l = bus.read_byte_data(LSM9DS0_ACC_ADDRESS, LSM9DS0_OUT_Z_L_A)
        acc_h = bus.read_byte_data(LSM9DS0_ACC_ADDRESS, LSM9DS0_OUT_Z_H_A)
    else:
        acc_l = bus.read_byte_data(LSM9DS1_ACC_ADDRESS, LSM9DS1_OUT_Z_L_XL)
        acc_h = bus.read_byte_data(LSM9DS1_ACC_ADDRESS, LSM9DS1_OUT_Z_H_XL)
```

```
    acc_combined = (acc_l | acc_h <<8)
    return acc_combined  if acc_combined < 32768 else acc_combined - 65536


def readMAGx(self):
    if (self.LSM9DS0):
        mag_l = bus.read_byte_data(LSM9DS0_MAG_ADDRESS, LSM9DS0_OUT_X_L_M)
        mag_h = bus.read_byte_data(LSM9DS0_MAG_ADDRESS, LSM9DS0_OUT_X_H_M)
    else:
        mag_l = bus.read_byte_data(LSM9DS1_MAG_ADDRESS, LSM9DS1_OUT_X_L_M)
        mag_h = bus.read_byte_data(LSM9DS1_MAG_ADDRESS, LSM9DS1_OUT_X_H_M)


    mag_combined = (mag_l | mag_h <<8)
    return mag_combined  if mag_combined < 32768 else mag_combined - 65536


def readMAGy(self):
    if (self.LSM9DS0):
        mag_l = bus.read_byte_data(LSM9DS0_MAG_ADDRESS, LSM9DS0_OUT_Y_L_M)
        mag_h = bus.read_byte_data(LSM9DS0_MAG_ADDRESS, LSM9DS0_OUT_Y_H_M)
    else:
        mag_l = bus.read_byte_data(LSM9DS1_MAG_ADDRESS, LSM9DS1_OUT_Y_L_M)
        mag_h = bus.read_byte_data(LSM9DS1_MAG_ADDRESS, LSM9DS1_OUT_Y_H_M)


    mag_combined = (mag_l | mag_h <<8)
    return mag_combined  if mag_combined < 32768 else mag_combined - 65536


def readMAGz(self):
    if (self.LSM9DS0):
        mag_l = bus.read_byte_data(LSM9DS0_MAG_ADDRESS, LSM9DS0_OUT_Z_L_M)
        mag_h = bus.read_byte_data(LSM9DS0_MAG_ADDRESS, LSM9DS0_OUT_Z_H_M)
    else:
        mag_l = bus.read_byte_data(LSM9DS1_MAG_ADDRESS, LSM9DS1_OUT_Z_L_M)
        mag_h = bus.read_byte_data(LSM9DS1_MAG_ADDRESS, LSM9DS1_OUT_Z_H_M)


    mag_combined = (mag_l | mag_h <<8)
    return mag_combined  if mag_combined < 32768 else mag_combined - 65536
```

```
def readGYRx(self):
    if (self.LSM9DS0):
        gyr_l = bus.read_byte_data(LSM9DS0_GYR_ADDRESS, LSM9DS0_OUT_X_L_G)
        gyr_h = bus.read_byte_data(LSM9DS0_GYR_ADDRESS, LSM9DS0_OUT_X_H_G)
    else:
        gyr_l = bus.read_byte_data(LSM9DS1_GYR_ADDRESS, LSM9DS1_OUT_X_L_G)
        gyr_h = bus.read_byte_data(LSM9DS1_GYR_ADDRESS, LSM9DS1_OUT_X_H_G)


    gyr_combined = (gyr_l | gyr_h <<8)
    return gyr_combined  if gyr_combined < 32768 else gyr_combined - 65536


def readGYRy(self):
    if (self.LSM9DS0):
        gyr_l = bus.read_byte_data(LSM9DS0_GYR_ADDRESS, LSM9DS0_OUT_Y_L_G)
        gyr_h = bus.read_byte_data(LSM9DS0_GYR_ADDRESS, LSM9DS0_OUT_Y_H_G)
    else:
        gyr_l = bus.read_byte_data(LSM9DS1_GYR_ADDRESS, LSM9DS1_OUT_Y_L_G)
        gyr_h = bus.read_byte_data(LSM9DS1_GYR_ADDRESS, LSM9DS1_OUT_Y_H_G)


    gyr_combined = (gyr_l | gyr_h <<8)
    return gyr_combined  if gyr_combined < 32768 else gyr_combined - 65536


def readGYRz(self):
    if (self.LSM9DS0):
        gyr_l = bus.read_byte_data(LSM9DS0_GYR_ADDRESS, LSM9DS0_OUT_Z_L_G)
        gyr_h = bus.read_byte_data(LSM9DS0_GYR_ADDRESS, LSM9DS0_OUT_Z_H_G)
    else:
        gyr_l = bus.read_byte_data(LSM9DS1_GYR_ADDRESS, LSM9DS1_OUT_Z_L_G)
        gyr_h = bus.read_byte_data(LSM9DS1_GYR_ADDRESS, LSM9DS1_OUT_Z_H_G)


    gyr_combined = (gyr_l | gyr_h <<8)
    return gyr_combined  if gyr_combined < 32768 else gyr_combined - 65536


def initIMU(self):
```

```
        if (self.LSM9DS0):   #For BerryIMUv1

            #initialise the accelerometer
            self.writeACC(LSM9DS0_CTRL_REG1_XM, 0b01100111)  #z,y,x axis enabled,
continuos update,  100Hz data rate
            self.writeACC(LSM9DS0_CTRL_REG2_XM, 0b00100000)  #+/- 16G full scale

            #initialise the magnetometer
            self.writeMAG(LSM9DS0_CTRL_REG5_XM, 0b11110000)  #Temp enable, M data
rate = 50Hz
            self.writeMAG(LSM9DS0_CTRL_REG6_XM, 0b01100000)  #+/-12gauss
            self.writeMAG(LSM9DS0_CTRL_REG7_XM, 0b00000000)  #Continuous-conversion
mode

            #initialise the gyroscope
            self.writeGRY(LSM9DS0_CTRL_REG1_G, 0b00001111)   #Normal power mode, all
axes enabled
            self.writeGRY(LSM9DS0_CTRL_REG4_G, 0b00110000)   #Continuos update, 2000
dps full scale

        else:       #For BerryIMUv2
            #initialise the gyroscope
            self.writeGRY(LSM9DS1_CTRL_REG4,0b00111000)      #z, y, x axis enabled
for gyro
            #writeGRY(LSM9DS1_CTRL_REG1_G,0b10111000)    #Gyro ODR = 476Hz, 2000 dps
            self.writeGRY(LSM9DS1_CTRL_REG1_G,0b10100000)    #Gyro ODR = 476Hz, 245
dps
            self.writeGRY(LSM9DS1_ORIENT_CFG_G,0b00111000)   #Swap orientation

            #initialise the accelerometer
            self.writeACC(LSM9DS1_CTRL_REG5_XL,0b00111000)   #z, y, x axis enabled
for accelerometer
            self.writeACC(LSM9DS1_CTRL_REG6_XL,0b00101000)   #+/- 16g

            #initialise the magnetometer
            self.writeMAG(LSM9DS1_CTRL_REG1_M, 0b10011100)   #Temp compensation
enabled,Low power mode mode,80Hz ODR
```

```
            self.writeMAG(LSM9DS1_CTRL_REG2_M, 0b01000000)   #+/-12gauss
            self.writeMAG(LSM9DS1_CTRL_REG3_M, 0b00000000)   #continuos update
            self.writeMAG(LSM9DS1_CTRL_REG4_M, 0b00000000)   #lower power mode for Z
axis

        self.previousTime = datetime.datetime.now()  # JTATE Update this


    ### JTATE Added Functions Below ###
    # Helper function to average a list
    def averageList(self, lst):
        return sum(lst) / len(lst)


    # Calibrate Gyro. During this time please keep the IMU still.
    # Seconds is the length of time to take measurements
    # average = 1 means use averages, else
    def calibrateGyro(self, seconds, average=0, heading = 0.0):
        #global G_GAIN, GYRxBias, GYRyBias, GYRzBias


        print ("Calibrating Gyro, please wait")# + str(heading))
        startTime = time.time()
        currentTime = time.time()
        GYRx = []#readGYRx()* G_GAIN]
        GYRy = []#readGYRy()* G_GAIN]
        GYRz = []#readGYRz()* G_GAIN]


        while currentTime - startTime < seconds:
            currentTime = time.time()
            try:
                GYRx.append(self.readGYRx()* G_GAIN)
                GYRy.append(self.readGYRy()* G_GAIN)
                GYRz.append(self.readGYRz()* G_GAIN)
            except:
                continue
            time.sleep(0.01)
```

```
        if average == 1:
            self.GYRxBias = self.averageList(GYRx)
            self.GYRyBias = self.averageList(GYRy)
            self.GYRzBias = self.averageList(GYRz)
        else: # Median
            self.GYRxBias = median(GYRx)
            self.GYRyBias = median(GYRy)
            self.GYRzBias = median(GYRz)
        print (self.GYRxBias, self.GYRyBias, self.GYRzBias)

        if self.gpsEnabled:# and self.ardInt.connected == 1:
            self.initGPSHeading = heading#self.ardInt.heading   # Move initial GPS
heading to end of cal sequence #### <<< change this
            self.runningCountGPS = 0
            self.previousGPSHeading = 0
            self.diffrence = heading
        print ("Gyro Calibration Complete")


    # Set current heading to 0,0,0. Also resets the time
    def resetGyro(self):
        #global gyroXangle, gyroYangle, gyroZangle, previousTime
        self.gyroXangle = 0.0
        self.gyroYangle = 0.0
        self.gyroZangle = 0.0
        self.previousTime = datetime.datetime.now()



    def filterGPS_gyro(self):
        alpha = 0.1 # May need to tune
        self.gyroZangle = alpha * self.gyroZangle + (1-alpha)*self.runningCountGPS

        #print "Straight GPS: %f, Running Count: %f, gyroZangle:
%f"%(self.ardInt.heading, self.runningCountGPS, self.gyroZangle)

    diffrence = 0.0
    def computeGPS(self, heading = 0.0):
        """
        TODO this is where it incorporates GNSS into the GRYO
        we need to filter this by declining and readings wayyyyy off... as GYRO is
nearly perfect as short term
        """
        # Grab headings to work with in local variables
```

```
        hGps = heading#self.ardInt.heading



        #print ("hGPS: %f, gyroGPS: %f, diffrenceGPS: %f"%(hGps, hGyro,hGps-hGyro))


        hGpsNormalized = hGps - self.initGPSHeading # Normalize back to 0 from
whatever angle we started

        deltaGps = hGpsNormalized - self.previousGPSHeading   # Still in 0-360
coordinates
        if deltaGps > 180:
            self.numRotations -= 1
        elif deltaGps < -180:
            self.numRotations += 1

        self.runningCountGPS = hGpsNormalized + self.numRotations * 360
        self.previousGPSHeading = hGpsNormalized
        #print ("Straight GPS: %f, Normalized GPS: %f, Running Count GPS: %f"%(hGps,
hGpsNormalized, self.runningCountGPS))




        #hGps += self.hGPSDelta  # Keeps into account multiple rotations
        #if deltaGps > 180:
        #    hGps-=360
        #    self.hGPSDelta-=360
        #    print "Delta: %f, previous: %f, hGps: %f"%(deltaGps,
self.previousGPSHeading, hGps)
        #elif deltaGps < -180:
        #    hGps+=360
        #    self.hGPSDelta+=360
        #    print "Delta: %f, previous: %f, hGps: %f"%(deltaGps,
self.previousGPSHeading, hGps)
        #self.runningCountGPS += hGps#deltaGps
        #self.previousGPSHeading = self.ardInt.heading - self.initGPSHeading
        #self.ardInt.newData = False


    def combineHeadings(self, heading = 0.0):
        self.computeGPS(heading)
```

```
        self.filterGPS_gyro()


        ## Compute correction factor and apply to base gyro heading
        #correctionFactor = hCombined - hGyro
        #print "GPS angle: %f InitGPSAngle: %f Gyro: %f Combined: %f FinalCorrected:
%f"%(hGPS, self.initGPSHeading, self.gyroZangle, hCombined, self.gyroZangle +
correctionFactor)
        #self.gyroZangle += correctionFactor



    def combineHeadingsOld(self):
        # Grab headings to work with in local variables
        hGyro = self.gyroZangle
        hGPS = self.ardInt.heading ### <<< change this
        alpha = 0.1 # May need to tune
        # Convert Gyro heading to GPS heading coordiantes for apples to apples
comparison
        # First add initial GPS heading
        hGyro += self.initGPSHeading
        # Then need to get gyro heading within 0 <= heading <= 360
        while hGyro < 0:
            hGyro += 360
        while hGyro > 360:
            hGyro -= 360
        # Compute combined heading using complementary filter
        hCombined = alpha * hGyro + (1-alpha)*hGPS
        # Compute correction factor and apply to base gyro heading
        correctionFactor = hCombined - hGyro
        print ("GPS angle: %f InitGPSAngle: %f Gyro: %f Combined: %f FinalCorrected:
%f"%(hGPS, self.initGPSHeading, self.gyroZangle, hCombined, self.gyroZangle +
correctionFactor))
        self.gyroZangle += correctionFactor
        self.ardInt.newData = False




    # Update the angle from the gyro. This function should be called regularly
    def updateGyro(self,heading = None):
```

```
        #global gyroXangle, gyroYangle, gyroZangle
        #global GYRxBias, GYRyBias, GYRzBias
        global G_GAIN, previousTime, upsideDown


        # Read the gyro, apply calibration values to get the angular rate of change

        # Test, delete this
        #rate_gyr_x =  self.readGYRx() * self.G_GAIN - self.GYRxBias
        #rate_gyr_y =  self.readGYRy() * self.G_GAIN - self.GYRyBias
        #rate_gyr_z =  self.readGYRz() * self.G_GAIN - self.GYRzBias


        try:
            rate_gyr_x =  self.readGYRx() * G_GAIN - self.GYRxBias
            rate_gyr_y =  self.readGYRy() * G_GAIN - self.GYRyBias
            rate_gyr_z =  self.readGYRz() * G_GAIN - self.GYRzBias


            if self.upsideDown == 1:
                rate_gyr_z = -1 * rate_gyr_z
        except:
            return ["e","e","e"]


        # Calculate the time since the previous reading
        timeInterval = datetime.datetime.now() - self.previousTime
        self.previousTime = datetime.datetime.now()
        LP = timeInterval.microseconds/(1000000*1.0)  # BerryIMU code did this. Not
totally sure why


        #Integrate the rate to get current angle
        self.gyroXangle+=rate_gyr_x*LP
        self.gyroYangle+=rate_gyr_y*LP
        self.gyroZangle+=rate_gyr_z*LP


        if self.gpsEnabled and heading != None:#self.ardInt.connected == 1:
            #if self.newdata == True
            self.combineHeadings(heading)
```

```
        return [self.gyroXangle, self.gyroYangle, self.gyroZangle] # Returning the
variables might make it easier to use even though they are global


    def getGyro(self):
        return [self.gyroXangle, self.gyroYangle, ((self.gyroZangle +
self.diffrence)%360)]




def main():
    # For testing
    berry = BerryIMU(usd = 1)
    berry.detectIMU()
    berry.initIMU()
    berry.calibrateGyro(10)
    berry.resetGyro()
    print ("starting thread")
    berryThread = Thread(target=berry.run)
    berryThread.daemon = True
    berryThread.start()
    print ("thread started")
    while True:
        gyroResults = berry.getGyro()
        #print "Roll: %f, Pitch: %f, Yaw: %f"%(gyroResults[0], gyroResults[1],
gyroResults[2])
        time.sleep(1)




if __name__ == "__main__":
    try:
        main()
    except KeyboardInterrupt:
        #berry.terminate()
        print("Ctrl-C Detected. Exiting")
        sys.exit(0)
    except:
        #GPIO.cleanup()
        print("Other Exception. Exiting")
```

```
        traceback.print_exc()
        #berry.terminate()
        sys.exit(0)
```

A.C. GNSS.py

```
from __future__ import print_function
#import qwiic_i2c
import time
from .qwiic_ublox_gps_FBN import *
#import qwiic_ublox_gps
import sys
import threading
import math
import random
from smbus2 import SMBus




class GPS:
    """
    ########################################
    This will start a GPS Daemon
    Note GPS Updates Every Second
    Usage:

    GPS1 = GPS(0x42)#connect to GPS
    GPS1.startGPSReading()#start Daemon

    #do whatever
    coord = (0.0,0.0)
    coord = GPS1.getCoord()
    print("Lat: " + coord[0] + "\nLon: " + coord[1]")
    #repeate the 2 lines above whenever
    #....
    GPS1.stopGPSReading() #kill thread
    ########################################
    """
       ### values ###
```

```
_UBX = True
latitude = 0.0
longitude = 0.0
heading = 0.0
headingPVT = (0.0,0.0) #heading,speed
freshData = False
qwiicGPS = QwiicUbloxGps()
lock = threading.Lock()
GPS_thread = threading.Thread()
kill = False
validCheck = 0.0
lastCheck = 0.0
averageCheck = 0.0
checks = 0
RelPosValid = 0.0




   ###Constructor###

def __init__(self, address = 0x42):
    self.qwiicGPS = QwiicUbloxGps(address)
    self.address = address


   ###Functions###


def startGPSReading(self):
    global kill
    kill = False
    GPS_thread = threading.Thread(target=self.updateGNSS)
    GPS_thread.daemon = True
    GPS_thread.start()



def stopGPSReading(self):
    global kill
    kill = True
```

```
    def updateGNSS(self):
        global kill
        global latitude
        global longitude
        seconds = -1
        if self.qwiicGPS.connected == False:
            print("Could not connect to GPS")
            return

        self.qwiicGPS.begin()

        #self.qwiicGPS.set_navigation_frequency(1)


        i = 0
        while kill == False :



            self.qwiicGPS.check_ublox()#parse and read data from
GPS module
            time.sleep(0.06)
            #if self.address == 0x43:
            #    time.sleep(0.03)

            if self.qwiicGPS.newData == True:
                #self.lock.acquire()

                if self.address == 0x42:
                    latitude = self.qwiicGPS.high_res_latitude
                    longitude = self.qwiicGPS.high_res_longitude
                    #print(latitude,longitude)
                else:

                    self.heading =
self.qwiicGPS.relative_pos_info['rel_pos_heading']
                    self.RelPosValid =
self.qwiicGPS.relative_pos_info['rel_pos_length']

#print(self.qwiicGPS.relative_pos_info['acc_D'])
                    self.validCheck =
self.qwiicGPS.relative_pos_info['acc_D']
```

```
            if self.qwiicGPS.newDataPVT == True:
                #self.lock.acquire()

                speed = self.qwiicGPS.ground_speed
                headingPVT = self.qwiicGPS.heading_motion
                self.headingPVT = (headingPVT,speed)



    def getCoord(self):
        """get lat and long when its not under write lock in
updateGNSS"""
        #f.qwiicGPS.newData)


        #print(self.qwiicGPS.newData)
        if self.qwiicGPS.newData == True:

            lat = self.qwiicGPS.high_res_latitude
            lon = self.qwiicGPS.high_res_longitude
            #print("SetFalse Coord")
            self.qwiicGPS.newData = False
            #print((lat,lon))
            return (lat,lon)


        else:
            return None



    def getHeading(self):
        #print(self.qwiicGPS.newData)
        if self.qwiicGPS.newData == True:
            #timeNow = time.clock()
            hed = self.heading
            #print("SetFalse Heading")
            self.qwiicGPS.newData = False
```

```
            return hed#uncomment this line to filter out bad
messages.... but you might have moments (10-20 second) of bad
readings

            if(self.RelPosValid > 46 and self.RelPosValid < 54):
#between .47 and .53 meters
                return hed
            else:
                print (self.RelPosValid)
                return None

            #self.averageCheck = float(int((timeNow -
self.lastCheck)*1000))/100
            #print("lastCheck 0x43 " + str(self.averageCheck)+ "
seconds" )
            #self.checks+=1

            #self.lastCheck = timeNow
        return None

    def getHeadingPVT(self):
        #print(self.qwiicGPS.newData)
        if self.qwiicGPS.newDataPVT == True:
            #timeNow = time.clock()
            hed = self.headingPVT
            #print("SetFalse PVT")
            self.qwiicGPS.newDataPVT = False

            return hed


        return None




#print ("done")
if __name__ == "__main__":
    newGPS = GPS(0x42)
    newGPS.startGPSReading()
    newGPS2 = GPS(0x43)
    newGPS2.startGPSReading()
    #command = input()
```

```
    i = 0
    while i < 860:
        time.sleep(.2)

        newGPS2.getHeading()
        #print("<<<<<<<<<<<<<<<<<<<<<<<")
        print(newGPS.getCoord())

        print(newGPS.getHeadingPVT())
        i+=1
    newGPS.stopGPSReading()
    newGPS2.stopGPSReading()

if False:

    qwiicGPS = QwiicUbloxGps(0x43)
    qwiicGPS.begin()
    i = 0
    data_status = False
    #qwiicGPS.set_auto_pvt(True)
    while True:
        time.sleep(.005)
        #data_status = qwiicGPS.get_parsed_nmea()


        dataStatus = qwiicGPS.check_ublox()
        #print(qwiicGPS.newData)
        #if qwiicGPS.newData == True:
        #print("longitude" + str(qwiicGPS.longitude))
        if qwiicGPS.newRelData == True:
            print(qwiicGPS.relative_pos_info['rel_pos_heading'])
            qwiicGPS.newRelData = False

        if qwiicGPS.newPosData == True:
            print (qwiicGPS.latitude)
            print (qwiicGPS.longitude)
            qwiicGPS.newPosData = False
```

A.D. Motor.py

```
#!/usr/bin/python3
from adafruit_servokit import ServoKit
from Encoder import Encoder
import time
import math
import traceback
import sys
import busio
import RPi.GPIO as GPIO
from hardwareSpex.parser import HSparser
from simple_pid import PID
from ArduinoI2Cinterface import ArduinoI2Cinterface
from Encoder import Encoder



kit = ServoKit(channels=16) #for servo motors


#motor class for controlling motors
class Motor:
	#initialzing
	pinNumber = 0 #assigns pin number
	global kit #for servo motors




	def __init__(self, ard, pinNumber,PulseRange = (1000, 2000), EncoderPin =
0): #had ard passed in for the self.enc
		global kit
		kit._pca.frequency = 100 #setting frequency for motors
		self.encoderPin = EncoderPin #for encoder pins on robot
		self.pinNumber = pinNumber #for the motor pin number
		self.pulseRange = PulseRange #for the set_pulse_width_range
		self.enc = Encoder(ard, count0_index = self.encoderPin, quad = True)
#for Encoders

		#set up to pass in different frequency for left and right motor
frequencys
```

```
        kit.continuous_servo[self.pinNumber].set_pulse_width_range(PulseRange[0],
PulseRange[1]) #microseconds


        #function to run motor
        def power(self, Power):
                global kit
                kit.continuous_servo[self.pinNumber].throttle = Power



        ### these function should not be called if Encoder is set to off ###
        def getRate(self):
                rate = self.enc.getRate() #getting the rate
                return int(rate)

        def clearEncoder(self):
                self.enc.clearEncoder()


        def getTickCount(self):
                tickCount = self.enc.getCounts()
                return int(tickCount)

        def updateEncoder(self):
                self.enc.update()







#class for initializing two motors
class Motors:

        ### Variables ###
        wheelRadius = HSparser().Search("WheelRadius")
        ticksPerRev = HSparser().Search("ticksPerRev")
```

```
        speedPID = PID(2.0,5.0, 0.1, setpoint=1.0,
sample_time=None,output_limits=(0.32, 0.95), auto_mode = True,
proportional_on_measurement=False) #when using encoders


        def __init__(self):
                ard = ArduinoI2Cinterface(address=0x9)
                #left motor getting 0 pin and pulseRange (1000,2000) and Encoder pin
(4)
                self.leftMotor =
Motor(ard,HSparser().Search("LMotor"),(1000,2000),HSparser().Search("ECLmotor"))
                #rightMotor getting 1 pin and pulseRange (1000,2000) and Encoder pin
(16)
                self.rightMotor =
Motor(ard,HSparser().Search("RMotor"),(1000,2000),HSparser().Search("ECRmotor"))
    #self.cutterMotor =
Motor(ard,HSparser().Search("CMotor"),(1000,2000),HSparser().Search("ECCmotor"))
        ### Functions ###
        def changeTargetSpeed(self, targetSpeed): #only with encoders
                print("Target Speed: " + str(targetSpeed))
                self.speedPID = PID(2.0,5.0, 0.1, setpoint=targetSpeed,
sample_time=None,output_limits=(0.32, 0.95), auto_mode = True,
proportional_on_measurement=False)

        def setToNeutral(self): #called in Loop to set motors to neutral before
running
                self.leftMotor.power(0.0)
                self.rightMotor.power(0.0)

        #distanceDriven calculated from encoders when enabled
        def getDistance(self):
                leftTicks = self.leftMotor.getTickCount()
                rightTicks = self.rightMotor.getTickCount()
                totalTicks = (leftTicks + rightTicks)/ 2 #average of left and right
motor total ticks
                distanceDriven = self.wheelRadius *2*math.pi * totalTicks /
self.ticksPerRev #use math to convert ticks to distance driven
                return distanceDriven


        #I think this is straightForward...
        def resetDistance(self):
                self.leftMotor.clearEncoder()
```

```
                self.rightMotor.clearEncoder()

        #updating left and right motor encoders
        def updateEncoders(self):
                self.leftMotor.updateEncoder()
                self.rightMotor.updateEncoder()




        #not currently in use bu for future reference
        def goStraightWithRandomPid(self, targetValue, CurrentValue, MaxPower):
                """We can add the go straight with Heading here?? just a thought """
                pass




        #power adjustment given from PID, based off of robots current speed
        def updateMaxPower(self):
                lSpeed = self.leftMotor.getRate()
                rSpeed = self.rightMotor.getRate() #will return a double, ticks per
second
                averageSpeed = (lSpeed + rSpeed)/2
                robotSpeed = self.wheelRadius *2* math.pi * averageSpeed /
self.ticksPerRev #calculating speed of robot
                newMaxPowerLevel = self.speedPID(robotSpeed) #NewMaxPowerLevel is
adjustment speed given from PID
                return newMaxPowerLevel


        def returnMotorSpeeds(self):
                """
                Return Motor speeds in m/s, for logging purposes in Loop
                """
                lSpeed = self.leftMotor.getRate()
                rSpeed = self.rightMotor.getRate() #will return a double, ticks per
second
                averageSpeed = (lSpeed + rSpeed)/2
                robotSpeed = self.wheelRadius *2* math.pi * averageSpeed /
self.ticksPerRev #calculating speed of robot based off of calculated speed above,
and hardware specs of robot
                return robotSpeed #returning robotSpeed for logging purposes
```

```
        ### GO STRAIGHT WITH WHEEL ENCODER, NO LONGER IN USE ###
        def straightLineWithMotorSpeed(self,maxPower):
                """############################################################"'
         Process
         1. determine if one side is going to slow
         2. come up with a number into which the robot heading to lead in to
(adjustment)
         3. adjust the motors so that the it is not putting more power then the max
power
         '""############################################################"""
                lSpeed = leftMotor.getRate()
                rSpeed = rightMotor.getRate() #will return a double, ticks per second
                currentSpeedRatio = lSpeed/rSpeed #ratio of left and right speed
                adjustment = self.speedPID(currentSpeedRatio)#1.1/1.0, pid gives
adjustment of speed


                if adjustment > 0: #if going right
                        self.rightMotor.power(maxPower) #we dont want to add more then
max power
                        self.leftMotor.power(maxPower - adjustment) #left motor is too
fast


                if adjustment < 0: #if adjustment Heading will be less than 0
                        self.rightMotor.power(maxPower + adjustment) #right motor is
too fast
                        self.leftMotor.power(maxPower) # we dont want to add more then
max power




        #called in Loop,
        def shutdown(self):
                """Shutdown all 16 Pins"""
                ## to prevent any bugs we shutdown them the hard way, through a loop
                i = 0
```

```
            while i < 2:
                    kit.continuous_servo[i].throttle = 0   # First set it to
neutral
                    kit._pca.channels[i].duty_cycle = 0x0  # This is how you
totally turn off output signal
                    i+=1
```

A.E. Drive.py

```
import math
from Motor import * #imports the motor file
from simple_pid import PID #imports the PID file
from hardwareSpex.parser import HSparser #imports hardware spec's

#imports for cutters
#import adafruit_mcp4725
#import time
#import sys
#import traceback


class Drive:


    #initializing
    maxPower = 0.90 #given motor power, can adjust
    motors = Motors() #from motors.py
    targetRatio = 1.0 # leftwheel/Rightwheel
    turningRight = False
    #used in driving straight with gyro
    headingPID = PID(0.5,0.5, 0.01, setpoint=0.0, sample_time=None,output_limits=(-
0.35, 0.28), auto_mode = True, proportional_on_measurement=False)
    #used in drive curvature
    turnPID = PID(2.0,2.0, 0.1, setpoint=1.0, sample_time=None,output_limits=(-0.35,
0.35), auto_mode = True, proportional_on_measurement=False)



    ### Get Raw Instances ###
    def __init__(self):
```

```
        pass
    def EStop(self):
        pass
    def shutdown(self):
        self.motors.shutdown()#calling shutdown function for motors in motor


    ##functions###


    def normalizeSpeed(self,targetSpeed): #only with encoders
        """Use Speed PID to make sure the average of the wheel is doing what is
supposed to do"""
        newMaxPower = self.motors.updateMaxPower() #referencing updateMaxPower()
function in motors

        if newMaxPower != None: #if maxPower has value
            self.maxPower = newMaxPower



    def setTurnStyle(self, turnStyle = 0, radius = 0):
        """##########################################
        Purpose is to calculate the distribution of power between both motors
        vv Output vv
        1 == go straight
       -1 == 0 deg turn
        if leftwheel/rightwheel < 1 it will inverse it... this is necessary to make
the ratios consistent
        """##########################################


        #print("Turn Style Set to" + str(turnStyle))
        if turnStyle == 0: #going straight
            self.targetRatio = 1.0#ratio between both wheel should be the same


        elif abs(turnStyle) == 1:
            self.targetRatio = -1.0 #Means 0 Deg turn
            if turnStyle > 0:
                self.turningRight = True
            else:
                self.turningRight = False
```

```
        elif abs(turnStyle) == 2:#  10cm overlap
            #variables
            fullWidth = HSparser().Search("FullWidth") #fullWidth = 0.62865
            trackWidth = HSparser().Search("TrackWidth") #trackWidth = 0.5461
            overLap = HSparser().Search("Overlap") #overLap = 0.1


            #calculating ratios
            leftwheel = (fullWidth/2 - overLap/2) + trackWidth/2 * (turnStyle/2) *
1.0 #-1 means when style/2 == 1 left leftwheel strengh should be weaker
            rightwheel = (fullWidth/2 - overLap/2) + trackWidth/2 * (turnStyle/2) *
-1.0
            self.targetRatio = leftwheel/rightwheel #
            if turnStyle > 0: #if turn style is bigger than 0, a right turn
                self.turningRight = True
            else: #if turn style is negative, a left turn
                self.turningRight = False
        else: #abs(self.turnstyle == 3):
            trackWidth = HSparser().Search("TrackWidth") #trackWidth = 0.5461
            #calculating ratios
            leftwheel = (radius) + trackWidth/2 * (turnStyle/3) * 1.0 #-1 means when
style/2 == 1 left leftwheel strengh should be weaker
            rightwheel = (radius) + trackWidth/2 * (turnStyle/3) * -1.0
            self.targetRatio = leftwheel/rightwheel
            if turnStyle > 0: #if turn style is bigger than 0, a right turn
                self.turningRight = True
            else: #if turn style is negative, a left turn
                self.turningRight = False
    def normalizeHeading(self, currentHeading, targetHeading):
        """
        -Uses current heading (probably obtained from gyro) to adjust the wheel
power to always match the heading
        -Waypoint Heading changer should constitinly change the targetHeading so
that it stays on the path
        """
        #use modules to make sure 359deg -> 2deg is a +3 deg diffrence not -357
        headingDifference = self.__normalizedDifference(targetHeading,
currentHeading)
        pass
```

```
    def __normalizedDifference(self, targetHeading = 0.0 , heading = 0.0):#,
turnStyle):


        """Checked and Works
        if targetheading is to right of heading then negative
        if targetheading is the left of heading then positive
        """
        headDifference = heading - targetHeading #difference between robots actual
heading and target heading
        if turnStyle < 0: #turning Right
            if headDifference < -45:
                headDifference -= -360

        #else if turnStyle > 0
        #    if headDifference > 45:#means targetHeading falls some where around 340
-> 360 and should be converted to -20 -> - 0
        #        headDifference -= 360

        return targetHeading - heading




    ### GO STRAIGHT WITH GYRO  ###
    oldPowerLevel =0.0 #initial power of motors
    def straightLineWithMotorGyro(self,targetHeading = 0.0, currentHeading = 0.0):
        """############################################################"'
        1. Determine which way its leaning
        2. recommend and "adjustment" with PID
        3. apply the adjustmenst to the two motors
        '"""############################################################"""

        headingDifference = (((currentHeading - targetHeading) + 540) % 360) - 180
#difference between current heading and target heading
        adjustment = self.headingPID(headingDifference) #PID will give adjusted
heading

        stabalizedMaxPowerLevel = float(self.motorPowerStabilizer(self.maxPower,
self.oldPowerLevel)) #giving an adjusted power level to gradually reach max power.

        #print(adjustment)
        if adjustment >= 0.0: #if adjustment Heading will be greater than 0
```

```
            self.motors.rightMotor.power(stabalizedMaxPowerLevel - adjustment)
#power adjustment on rightMotor to ensure not going over maxPower
            self.motors.leftMotor.power(stabalizedMaxPowerLevel)


        if adjustment < 0.0: #if adjustment Heading will be less than 0
            self.motors.rightMotor.power(stabalizedMaxPowerLevel) # + and - based
off of calculated angle
            self.motors.leftMotor.power(stabalizedMaxPowerLevel + adjustment) #power
adjustment on leftMotor to ensure not going over maxPower




    def turnCompleted(self, turnStyle = 0, targetHeading = 0.0, currentHeading =
0.0, restriction = False):
        """
        if restriction = False -> +-15 degrees
        if restriciton = True -> 0 - 15 degrees or -15 - 0 degrees

        """
        headingDifference = (((currentHeading - targetHeading) + 540) % 360) - 180
#difference between current and target heading
        restrictionAngle = 15 #15 degrees of angle to meet finished turn

        if restriction == False:
            if abs(headingDifference) < restrictionAngle: #there is a 30 degree
window in which we call it a complete turn 15+ 15 = 30
                return True
            else:
                return False
        else:#restriction == True
            if self.turningRight:
                if headingDifference > 0 and headingDifference < restrictionAngle:
                    return True
                else:
                    return False
            else:# self.turningRight == False
                if headingDifference < 0 and headingDifference > (0 -
restrictionAngle):
                    return True
                else:
                    return False
```

```
    #used for turnstyle =3, for arc turning
    def driveCurvature(self,radius, targetRadius):

        adjustment = self.turnPID(radius/targetRadius)#if radius is too large it
will return a negative number
        print ("adjustment" + str(adjustment))


        newTargetRatio = self.targetRatio #initializing here
        if abs(self.targetRatio) >= 1:# if leftmotor/rightmotor > 1
            newTargetRatio = 1.0/self.targetRatio


        fasterWheel = self.maxPower
        slowerWheel = self.maxPower * newTargetRatio

        if self.turningRight:#turning Right
            if adjustment > 0:#Radius is to small
                self.motors.rightMotor.power(slowerWheel) #no power adjustment on
rightMotor to ensure not going over 0.25 speed
                self.motors.leftMotor.power(fasterWheel - adjustment)
            else:#adjustment < 0#Radius is too large
                self.motors.rightMotor.power(slowerWheel + adjustment) #no power
adjustment on rightMotor to ensure not going over 0.25 speed
                self.motors.leftMotor.power(fasterWheel)
        else:# adjustment < 0:#turning left
            if adjustment > 0:#Radius too small
                self.motors.rightMotor.power(fasterWheel - adjustment) #no power
adjustment on rightMotor to ensure not going over 0.25 speed
                self.motors.leftMotor.power(slowerWheel)
            else:#adjustment < 0#Radius too large
                self.motors.rightMotor.power(fasterWheel) #no power adjustment on
rightMotor to ensure not going over 0.25 speed
                self.motors.leftMotor.power(slowerWheel + adjustment)


    ### turn function, adjusted to no longer use a PID###
    oldPowerLevel =0.0 #used in for slowly increasing wheel power when turning
    def Turn(self):
        """
        SOME GOALS OF THIS FUNCTION
```

```
        What do you need for a successful turn adjustment
        1. Target turn style
        2. Current Turn Status
        3. Determine how we should adjust the slower wheel
        4. a way to change your slower wheel
        ##
        """
        newTargetRatio = self.targetRatio #initializing here
        if abs(self.targetRatio) >= 1:# if leftmotor/rightmotor > 1
            newTargetRatio = 1.0/self.targetRatio



        #fasterWheelPWR = self.maxPower
        stabalizedFasterWheelPWR = float((self.motorPowerStabilizer(self.maxPower,
self.oldPowerLevel)) - 0.05) #giving an adjusted power level to reach max power.
        slowerWheelPWR = (newTargetRatio) * (self.maxPower - 0.05) #0.05 less than
maxpower so slower wheel doesnt cap at the highest power, needed for less wheel
slippage

        if self.targetRatio == -1:
            stabalizedFasterWheelPWR /= 2
            slowerWheelPWR = 0 - stabalizedFasterWheelPWR


        if self.turningRight:#turning Right
            self.motors.rightMotor.power(slowerWheelPWR) #
            self.motors.leftMotor.power(stabalizedFasterWheelPWR)
        else:#leftturn


            self.motors.rightMotor.power(stabalizedFasterWheelPWR) #
            self.motors.leftMotor.power(slowerWheelPWR)


    #function for calibrating motors to neutral
    def calibrate(self):
        self.motors.setToNeutral() #calls setToNeutral in Motor.py


    #makes power adjustments based off current power level and maxpower level
    #called 100 times a second
    def motorPowerStabilizer(self, maxPower, oldPower):
```

```
        oldPower = self.oldPowerLevel #initial starting power
        sumAdjustment = 0.02 #motor power adjustment
        if maxPower < oldPower: #adjust power down
            self.oldPowerLevel = oldPower - sumAdjustment
            return oldPower - sumAdjustment
        if maxPower > oldPower: #adjust power up
            self.oldPowerLevel = oldPower + sumAdjustment
            return oldPower + sumAdjustment
        else:
            self.oldPowerLevel = oldPower#they were the same
            return maxPower
if __name__ == "__main__":
    try:
        drive = ToWayPoint(.25,0)
        print(drive.originWaypoint)# + " " + drive.destinationWaypoint)


    except KeyboardInterrupt:
        #cutter.normalized_value = 0 #for cutters
        print("Ctrl-C Detected. Exiting")
        sys.exit(0)
    except:
        print("Other Exception. Exiting")
        traceback.print_exc()
        #stop(ramp=True)
        #GPIO.cleanup()
        #cutter.normalized_value = 0 #for cutters
        #motor shutdown loop, but should not even be called here
        i = 0
        while i < 2:
            kit.continuous_servo[i].throttle = 0   # First set it to neutral
            kit._pca.channels[i].duty_cycle = 0x0  # This is how you totally turn
off output signal
            i+=1
        sys.exit(0)
```

A.F. Loop.py

```
from Drive import *
from Positioning import NormPositioning
from pathFile.parser import PFparser #for reading in H_HashTag.txt
import math
import time
import board
import busio
import logging
#import adafruit_mcp4725
import traceback
#imports for GPXroute, visible waypoint map
import gpxpy
import gpxpy.gpx
from simple_pid import PID


i2c = busio.I2C(board.SCL,board.SDA)
#cutter= adafruit_mcp4725.MCP4725(i2c)


class Loop:
    """

    """

    freqency = .02#100 times per second
    #cutter= adafruit_mcp4725.MCP4725(i2c)
    DriveController = Drive()


    #config
    encodersEnabled = True


    #disabled
    metersAhead = 2 #heading will match path 3 meters ahead, when robot gets off
course
    #initializing variables


    _debug = False
```

```
    onObsticalAvoidence = False
    GPSEnabled = False
    straight = 0
    geometricWaypoint = 2
    heading = 1
    waypoint = 0

    #varibles
    #initializing cutter
    driveStyle = 0 #waypoint or geometricHeading
    targetSpeed = 1.5#m/s
    robotSpeed = 0.0
    targetHeading = 0.0
    newTargetHeading = 0.0
    targetDistance = 0.0
    turnStyle = 0 #straight
    #cutterOn = False
    globalDistanceDriven = 0.0
    pathTimeStart = 0.0
    initialWayPoint = (0.0,0.0)
    currentWaypoint = (0.0,0.0)
    freshCurrentWayPoint = False
    lastEncoderDistance = 0.0 #if waypoint and encoders... this help sync gps and
gyro
    targetWayPoint = (0.0,0.0)
    circleCenter = (0.0,0.0)
    degree = 0.0
    currentRadius = 0.0
    targetRadius = 0.0
    offCourseDistanceNum = 0.0



    x_y_Coord = (0.0,0.0)
    Prev_x_y_Cord = (0.0,0.0)
    baseCoord = (0.0,0.0)
    baseHeading = 0


    #for viewable waypoint file
    gpx = gpxpy.gpx.GPX()#to create a viewable wayfile
    GPXroute = gpxpy.gpx.GPXRoute()
    totalPathsInFile = 1
```

```
    def __init__(self, GPSEnabled = True, debug = False, encoders = True,
basePosition=(0.0,0.0,0.0)):
        """
        anything that can be in init should be here
        """
        #set Global Varibles
        self.GPSEnabled = GPSEnabled
        self._debug = debug
        self.encodersEnabled = encoders


        #Start up the GPS
        self.Positioning = NormPositioning(GPSEnabled)


        #for geometric waypoint file only
        self.baseCoord = (basePosition[0],basePosition[1])
        self.baseHeading = basePosition[2]


        ### LOGGING and stuff ####
        setup_logger('log1', 'log_waypoint.txt')
        setup_logger('log2', 'log_IMU.txt')
        self.logger_Waypoint = logging.getLogger('log1')
        self.logger_IMU = logging.getLogger('log2')
        #logger_Waypoint.info('111messasage 1')
        #self.logger_IMU.info('222ersaror foo')

        #GPX FILE INIT
        self.gpx.routes.append(self.GPXroute)

        #for logging


    #function for runnnig the robot
    def run(self, fileName):
        """
        Current Loop structure
        1. read sensors
        2. power on motors
        3. check if complete, if not rerun loop
        """
```

```
        try:
            PathFileArray = PFparser().ReadLines(fileName) #for parsing inputted
file
        except Exception as e:
            traceback.print_exc()
            print("Unable to Find File")
            return


        totalPathsInFile = len(PathFileArray)



        ### Heading or Waypoint
        if self.determineDriveStyle(fileName[0]) == None:# see headingOrWaypoint
Definition
            return #unknown file type... don't go into loop


        if self.driveStyle == self.geometricWaypoint and self.baseCoord ==
(0.0,0.0):
            self.baseCoord = self.getGPSHard()
            print("No baseCoord detected setting current waypoint as baseCoord")

        #to set motors to neutral before running
        self.DriveController.calibrate()


        distanceDriven = 0.0 #initial distance is 0
        i = 0
        GNSSCounter = 0#helps to not check GNSS too often


        completion = False
        pathCompletion = True # first one set to true


        while completion == False:


            ### Anything that needs to be called once per pathLine should be called
in this If statement
            if pathCompletion == True:
```

```
                ### save any old pathFile values ###


                #The file only provides TargetWaypoint... if be have old
targetWaypoint then we have initial waypoint
                previousCoord = self.targetWayPoint


                wasTurning = self.turnStyle
                if self.driveStyle == self.geometricWaypoint:
                    self.Prev_x_y_Cord = self.x_y_Coord



                #read in next line
                self.updatePathVariables(PathFileArray, i)


                #change the SpeedPID target value to self.targetSpeed
                self.DriveController.motors.changeTargetSpeed(self.targetSpeed)


                ### turn on/off cutter ###
                self.enableDisableCutters(self.cutterOn)


                #Reset Local distance driven, add on to global Distance Driven
(including encoders
                self.resetDistance(distanceDriven)
                distanceDriven = 0.0#local varible



                #at the end of the driveStraight this value is set to .4.... so slow
it down, after that we need to pick it backup
                self.DriveController.maxPower = .95


                #pathCompletion is set to true if distanceDriven > targetDistance OR
turncompleted = True
                pathCompletion = False
```

```
                #incriment File Number

                #wait until we get a fresh GPS heading...
                    #if waypoint mode
                    #or if geometricPoint and on its first run.... this is becuase
we do not know the waypoint at the beginning of the drivestule
                if (self.GPSEnabled):
                    if i == 0: #we don't have the orignal waypoint... so the very
first run, it will not move unitil it gets its current waypoint
                        self.initialWayPoint = self.getGPSHard()
                    elif i == 1:# or self.driveStyle == self.waypoint:# we don't
have orignal coord, but we dont't want it to go on a wait loop... so attempt to get
new waypoint, else set to last known waypoint
                        self.initialWayPoint = self.getGPSSoft(self.currentWaypoint)
                    else:
                        self.initialWayPoint = previousCoord



                if abs(self.turnStyle) == 3:
                    #all the waypoint math for a turnstyle 3

                    self.circleCenter =
self.Positioning.destinationPoint(self.baseCoord,(self.baseHeading+(math.atan2(self.
x_y_Coord[0],self.x_y_Coord[1])*180/math.pi))%360,baseToCoordDist)

                    self.targetHeading = self.Positioning.bearing(self.circleCenter,
self.initialWayPoint) + (self.degree * (self.turnStyle/abs(self.turnStyle)))

                    print(self.circleCenter)
                    print(self.initialWayPoint)
                    self.targetRadius =
self.Positioning.distanceGNSS(self.circleCenter, self.initialWayPoint)
                    print(self.targetRadius)
                    self.targetWayPoint =
self.Positioning.destinationPoint(self.circleCenter,self.heading,self.targetRadius)


                elif self.driveStyle == self.waypoint:# and self.turnStyle != 0:
                    #all the waypoint math for the waypoint drivestyle
                    print(str(self.turnStyle) + "Check TH" +
str(self.targetHeading))
```

```
                    #set targetHeading to the bearing between current waypoint and
targetwaypoint, if it turned incorrectly an error may occure, to prevent this do
math to make the intended waypoint
                    if self.turnStyle == 0 or i == 0 or i == 1:
                        self.targetHeading =
self.Positioning.bearing(self.initialWayPoint,self.targetWayPoint)
                    else:
                        self.targetHeading = self.degree


                    #voids if turnstyle != 0
                    self.targetDistance =
self.Positioning.distanceGNSS(self.initialWayPoint, self.targetWayPoint)


                    self.currentWaypoint = self.getGPSSoft(self.initialWayPoint)
                    print("Check TH" + str(self.targetHeading))
                elif self.driveStyle == self.geometricWaypoint:
                    #all the waypoint math for the geometric driveStyle

                    self.currentWaypoint = self.initialWayPoint


                    if i == 0 or i == 1:#first run we do not have the x and y
position of the robot

                        self.targetHeading =
self.Positioning.bearing(self.initialWayPoint, self.baseCoord)

                        self.targetDistance =
self.Positioning.distanceGNSS(self.initialWayPoint, self.baseCoord)


                        self.targetWayPoint = self.baseCoord

                        print("Target Heading " + str(self.targetHeading))
                        print("targetDistance " + str(self.targetDistance))

                    else:

                        if self.turnStyle == 0:
                            self.targetHeading = math.atan2(self.x_y_Coord[0]-
self.Prev_x_y_Cord[0],self.x_y_Coord[1]-self.Prev_x_y_Cord[1])*180/math.pi
```

```
                            print(self.targetHeading)
                            self.targetHeading = (self.targetHeading +
self.baseHeading)%360
                        else:
                            self.targetHeading = (self.degree +
self.baseHeading)%360


                        self.targetDistance = math.sqrt(math.pow(self.x_y_Coord[0]-
self.Prev_x_y_Cord[0],2) + math.pow(self.x_y_Coord[1]-self.Prev_x_y_Cord[1],2))

                        baseToCoordDist = math.sqrt(math.pow(self.x_y_Coord[0],2) +
math.pow(self.x_y_Coord[1],2))
                        self.targetWayPoint =
self.Positioning.destinationPoint(self.baseCoord,(self.baseHeading+(math.atan2(self.
x_y_Coord[0],self.x_y_Coord[1])*180/math.pi))%360,baseToCoordDist)###<<<<  double


                #set Turn style to help drive.py functions
                #sets the wheel ratios inside Drive.py straight = 1.0, 0 point turn
= -1.0, and etc
                self.DriveController.setTurnStyle(self.turnStyle, self.targetRadius)
                if self.turnStyle != 0:
                    self.Positioning.caliberationEnabled = False
                else:
                    self.Positioning.caliberationEnabled = True

                ### this is still part of the if statement to read in a new pathfile




                ### LOGGING INITIAL STATS, called once per new path ###
                #FOR WAYPOINT FILE
                self.logger_Waypoint.info("\n\n>>>>>>>>>>>>>>>>NEW Pathfile
Read<<<<<<<<<<<<<")
                self.logger_Waypoint.info("Initial Waypoint" +
str(self.initialWayPoint))
                self.logger_Waypoint.info("Target Waypoint: " +
str(self.targetWayPoint))


                NewDescription = "Target Heading: " + str(self.targetHeading)
```

```
self.gpx.waypoints.append(gpxpy.gpx.GPXWaypoint(self.initialWayPoint[0],self.initial
WayPoint[1], description = NewDescription))

self.gpx.waypoints.append(gpxpy.gpx.GPXWaypoint(self.targetWayPoint[0],self.targetWa
yPoint[1]))#, description = NewDescription)

self.GPXroute.points.append(gpxpy.gpx.GPXRoutePoint(self.initialWayPoint[0],self.ini
tialWayPoint[1],name="Path", description = NewDescription))

self.GPXroute.points.append(gpxpy.gpx.GPXRoutePoint(self.targetWayPoint[0],self.targ
etWayPoint[1],name="Path"))#, description = NewDescription))


                #FOR IMU FILE
                self.logger_IMU.info("\n\n>>>>>>>>>>>>>>>>>NEW Pathfile
Read<<<<<<<<<<<<<")
                self.logger_IMU.info("TargetDistance" + str(self.targetDistance))


                i+=1


            ### READING SENSORS ###

            #read Gyro
            self.currentHeading = self.Positioning.getHeading() #Z axis only  ...
removed [2]

            #read Speeds
            self.robotSpeed = self.DriveController.motors.returnMotorSpeeds()


            #read GNSS
            if self.GPSEnabled:
                #check every tenth of a second
                GNSSCounter+=1
                if (GNSSCounter%((1/self.freqency)/10)) == 1:
                    newCurrentWaypoint = self.getGPSSoft()
                    #print(newCurrentWaypoint)
```

```
                    if newCurrentWaypoint != None:
                        self.freshCurrentWayPoint = True
                        self.currentWaypoint = newCurrentWaypoint
                    else:
                        self.freshCurrentWayPoint = False




            ### PROCESS DATA ###


            #calculate the new target heading so that the robot is on course
            #if GPS is is disabled then no need to do the calculaiton
            if self.turnStyle == self.straight and self.GPSEnabled == True: #if
driving straight and using waypoint
                if self.freshCurrentWayPoint == True:#if new data then calcualte off
course distance
                    self.newTargetHeading = self.onCourseAngleCalcOld()

                    #enable this if the robot is having too much zigzags from bad
GNSS readings
                else:#This will make the heading slowely go back to original target
heading.... this way if the GNSS freezes it will eliminate the zigzaging
                #    self._debugPrint("before target heading :"
+str(self.newTargetHeading))
                    self.newTargetHeading =
(self.newTargetHeading*((1/self.freqency)*2) +
self.targetHeading)/(((1/self.freqency)*2) + 1)
                    #print("After target heading:" +str(self.newTargetHeading))
            else:#drivestyle = self.heading
                self.newTargetHeading = self.targetHeading


            if self.encodersEnabled:
                #print("got to when encoders are enabled") #for debugging
                self.DriveController.motors.updateEncoders()

                #decrease/incrase power to make the robot
```

```
                self.DriveController.normalizeSpeed(self.targetSpeed) #updating
maxPower from Drive.py, from motors, adjusting robot speed
                #print("normalized speed with encoders")

            #here is where we can put a is stuck detection then drive backwards
            #attempt to get distance driven, first from GPS, then from encoders, if
both failed then set it to none
            if self.GPSEnabled and self.turnStyle == 0: #if waypoints and going
straight
                if self.freshCurrentWayPoint == True:
                    distanceDriven =
self.Positioning.distanceGNSS(self.initialWayPoint, self.currentWaypoint)#GNSS
distance from NormPos
                elif self.encodersEnabled:
                    #this will sync encoders with GPS distance( it will get the
distance driven between last GNSS reading from encoders and add it on to the
distanceDriven
                    encoderDistance = self.DriveController.motors.getDistance()#get
current distance
                    #print("distanceDriven with Encoders" + str(encoderDistance))
                    #print("distanceDriven with Encoders Last Check" +
str(self.lastEncoderDistance))
                    distanceDrivenSinceLastTick = (encoderDistance -
self.lastEncoderDistance)

                    if distanceDrivenSinceLastTick < 2.0:# it should be around .1-.3
every time... their is a bug where it reads 15,000 distance drives... we want to
ignore that
                        distanceDriven = distanceDriven +
distanceDrivenSinceLastTick#get distance driven diffrence and add to GNSS distance
                        self.lastEncoderDistance = encoderDistance#save distance
driven from encoders
                    #else:
                        #do nothing as we have a bad encoder reading... take the
last distanceDriven Value.. maby add .05 or some approximate number
            elif self.encodersEnabled and self.turnStyle == 0: #if encoders are
enabled and going straight
                distanceDriven = self.DriveController.motors.getDistance()#Encoder
Distance
            elif self.turnStyle == 0: #if driving straight with no encoders or
waypoints
                distanceDriven = time.time() - self.pathTimeStart #distance as
seconds driven
```

```
            else:
                distanceDriven = 0



            #nothing works here as it supposed to... I will work on this
            if abs(self.turnStyle) == 3:#TODO add freshdata checker and code here
                if self.encodersEnabled:
                    self.currentRadius =
self.Positioning.distanceGNSS(self.circleCenter, self.currentWaypoint)
                    self.currentHeading =
self.Positioning.bearing(self.circleCenter, self.currentWaypoint)
                else:
                    self.currentRadius = self.targetRadius
                    self.currentHeading = self.distanceDriven*360/(2*math.pi *
self.targetRadius)#this will output how much degrees of arc it has driven, this is
only neccessary if we do not have GPSenabled




            powerSlowDown = 0 #initializing powerSlowDown

            if self.turnStyle == 0 and self.targetDistance - distanceDriven < 2: #if
robot only has 1 meter left to drive and going strai
                self.DriveController.maxPower = .8 #may need to adjust with new
motors



            ### DONE PROCESSING DATA ###


            ### LOGGING BEGIN####
            ####
            #waypoint debug, aka info
            if self.freshCurrentWayPoint:
                self.logger_Waypoint.info("Current Waypoint: Lat: " +
str(self.currentWaypoint[0]) + " Lon: " + str(self.currentWaypoint[1]))
                #self.logger_Mapper.info(str(self.currentWaypoint[0]) + "," +
str(self.currentWaypoint[1]))
```

```
                self.logger_Waypoint.info("TargetHeading" +
str(self.newTargetHeading))

                NewDescription = "-robot_speed:"  + str(self.robotSpeed) + "m/s\n "
                NewDescription += "-Off_Course_Distance:" +
str(self.offCourseDistanceNum) + "m \n"
                NewDescription += "-DistanceDriven:" + str(distanceDriven) + "m\n "
                NewDescription += "-targetDistance:" + str(self.targetDistance) +
"m\n "
                NewDescription += "-Target_Heading:" + str(self.targetHeading) +
"`\n "
                NewDescription += "-currentHeading:" + str(self.currentHeading) +
"`\n "
                NewDescription += "-New_Target_Heading:" +
str(self.newTargetHeading) + "`\n "
                NewDescription += "-initialWaypoint: " +str(self.initialWayPoint) +
"\n "
                NewDescription += "-targetWaypoint: " + str(self.targetWayPoint)

self.gpx.waypoints.append(gpxpy.gpx.GPXWaypoint(self.currentWaypoint[0],self.current
Waypoint[1], description = NewDescription))
                print(NewDescription)

                #if the robot crashes there is now way to save to file... except if
we save it every .3 seconds


            self.logger_Waypoint.info("currentHeading " + str(self.currentHeading))


            #IMU debug, aka info
            self.logger_IMU.info("Robot Speed: " + str(self.robotSpeed) + "m/s")
            self.logger_IMU.info("Total Distance Driven: " + str(distanceDriven))
            self.logger_IMU.info("target Distance: " + str(self.targetDistance))
            if self.freshCurrentWayPoint:
                self.logger_IMU.info("Off course Distance: " +
str(self.offCourseDistanceNum) + "m")

            #### END OF LOGGING  ###
```

```
            ### RUN DRIVE FUNCTIONS ###


            #Adjusting/Powering Motors
            if self.turnStyle == 0: #if straight

self.DriveController.straightLineWithMotorGyro(self.newTargetHeading,self.currentHea
ding) #calling driving straight function in Drive
            elif abs(self.turnStyle) == 3: #if arc turning

self.DriveController.driveCurvature(self.currentRadius,self.targetRadius) #calls
driveCurvature function in drive
            elif self.turnStyle != 0: #if turning
                self.DriveController.Turn() #we need to rethink how to intigradte
turning into waypoint




            #setfreshdata to false
            self.freshCurrentWayPoint = False#every ~tenth of a second if a new GNSS
data comes in... it will mark the GNSS data as fresh and adjust the robot.. then it
will mark it false again


            #Check End Condition
            if self.turnStyle == 0: #if straight
                if self.targetDistance < distanceDriven:#changed
                    print ("distanceDriven before path completion called: " +
str(distanceDriven))#I found a bug here were the robot would randomly mark it
complete when I have like 10 meters left... if it ever occries this is the right
place to check
                    print(self.targetDistance)
                    print(self.initialWayPoint)
                    print(self.targetWayPoint)
                    pathCompletion = True
            elif self.turnStyle != 0: #if turning
                restrictionEnabled = False


                if abs(self.turnStyle) == 3:
```

```
                    restrictionEnabled = True# target heading is not not gyro target
heading, but rather angle of turn target heading... This is turn complete function
is true if currentHeading is +-15degrees off the targetHeading, but in a turnstyle
we need 0-15degree on the angle
                #print(self.currentHeading)
                if
(self.DriveController.turnCompleted(self.turnStyle,self.newTargetHeading,self.curren
tHeading, restrictionEnabled) == True):
                    pathCompletion = True
            else:
                pathCompletion = False



            #Check Nested End Conditiong (robot will turn off if True)
            if pathCompletion == True and i >= totalPathsInFile:
                completion = True
                print("ready to shutdown")



            #self._debugPrint("\n\n")
            time.sleep(self.freqency) #adjust accordingly, adjust even lower now
that encoders are working


        self.DriveController.shutdown()
        if self.GPSEnabled:
            print("Print XML")
            f = open("GPX.gpx", "w+")
            print(self.gpx.to_xml())
            f.write(self.gpx.to_xml())
            f.close()
            self.Positioning.abort()
        print("ShutDown")



    driveStraightPID = PID(10.0,15.0, 10.0, setpoint=0.0,
sample_time=None,output_limits=(-18.0, 18.0), auto_mode = True,
proportional_on_measurement=False)
```

```
    def onCourseAngleCalc(self): #when robot gets off course, it'll adjust the
target heading
        """
        PID version of the onCourseAngleCalcOLD
        """

        offCourseDistanceNum =
self.Positioning.getOffcourseDistance(self.targetWayPoint,self.targetHeading,self.cu
rrentWaypoint)
        self.offCourseDistanceNum = offCourseDistanceNum

        if (offCourseDistanceNum == 0.0):
            self.newTargetHeading = self.targetHeading
            return self.targetHeading


        angle = self.driveStraightPID(offCourseDistanceNum)
        print("OCDN" + str(self.offCourseDistanceNum) + " -> " + str (angle))


        self.newTargetHeading = self.targetHeading + angle


        return self.newTargetHeading



    def onCourseAngleCalcOld(self):
        """
        when robot gets off course, it'll adjust the target heading


        lets say it needs to drive 10 meters at 0 degrees. However at the 5th meters
it is .3 meters offCourse
        it will calcualte what angle does the robot need to face to get back on
course (self.metersAhead) meters ahead of it, it will store that value in
newTargetHeading
        """
        offCourseDistanceNum = 0.0
```

```
        offCourseDistanceNum =
self.Positioning.getOffcourseDistance(self.targetWayPoint,self.targetHeading,self.cu
rrentWaypoint)
        self.offCourseDistanceNum = offCourseDistanceNum#for printing out in debug


        print("OCDN: " + str(offCourseDistanceNum))
        #self._debugPrint("target heading: " + str(self.targetHeading))

        if (offCourseDistanceNum == 0.0):
            self.newTargetHeading = self.targetHeading
            return self.targetHeading



        angle = (180/math.pi)* math.atan(self.metersAhead /
abs(offCourseDistanceNum))
        angle = 90 - angle


        # -offCourseDistance means its to the left
        # +offCourseDistance means its to the right


        if offCourseDistanceNum < 0:
            self.newTargetHeading = (self.targetHeading + angle)%360.0

        if offCourseDistanceNum > 0:
            self.newTargetHeading = (self.targetHeading - angle)%360.0


        #self._debugPrint("Adjusted Heading: " + str(self.newTargetHeading))


        return self.newTargetHeading






    def updatePathVariables(self, PathFileArray, i):
        """
```

```
        Just simply reads in process al the value inthe path file... i is the row
number
        """

        self.targetSpeed = PathFileArray[i]["Speed"]
        self.turnStyle =  PathFileArray[i]["TurnStyle"]
        print(PathFileArray[i]["Cutter"])
        self.cutterOn = PathFileArray[i]["Cutter"]




        #self._debugPrint("Speed: " + str(self.targetSpeed))
        #self._debugPrint("TurnStyle: " + str(self.turnStyle))
        #self._debugPrint("Cutter: " + str(self.cutterOn))




        if self.driveStyle == self.heading:
            self.targetDistance = PathFileArray[i]["Distance"]
            self.targetHeading = PathFileArray[i]["Heading"]
            print("Check TH" + str(self.targetHeading) +
"<<<<<<<<<<<<<<<<<<<<<<<<<<<")
            #self._debugPrint("TargetDistance: " + str(self.targetDistance))
            #self._debugPrint("TargetHeading" + str(self.targetHeading))


        elif self.driveStyle == self.waypoint:
            self.targetWayPoint = PathFileArray[i]["WayPoint"]
            self.degree = PathFileArray[i]["Degree"]


            #self._debugPrint("TargetWayPoint" + str(self.targetWayPoint))
            #self._debugPrint("Degree" + str(self.degree))
        elif self.driveStyle == self.geometricWaypoint:
            self.x_y_Coord = PathFileArray[i]["Coord"]
            self.degree = PathFileArray[i]["Degree"]


            #self._debugPrint("Coord" + str(self.x_y_Coord))
            #self._debugPrint("Degree" + str(self.degree))
```

```
    def determineDriveStyle(self, character):
        """
        simply lets the robot know which driveStyle it is
        """
        print (character)
        ### Heading ###
        if character.upper() == 'H':
            print (character)


            #self._debugPrint("ToHeading File Found")#this only print if debug is
True
            self.driveStyle = self.heading #drive straight with no waypoint
            self.DriveController = ToHeading()
            return 1


        ### Waypoint ###
        elif character.upper() == 'W':
            print (character)
            #self._debugPrint("ToWaypoint File Found")
            self.driveStyle = self.waypoint
            return 0


        ##GeometricWaypoint
        elif character.upper() == 'G':
            print (character)
            #self._debugPrint("ToWaypoint File Found")
            self.driveStyle = self.geometricWaypoint
            return 2
        else:
            #self._debugPrint("Unkown File Type")
            return None


    def resetDistance(self, distanceDriven = 0.0):
        """
        resets encoders and stores global distance driven
        """
        self.globalDistanceDriven += distanceDriven
```

```
        self.distanceDriven = 0
        if self.encodersEnabled: #if distance is calculated from encoders
            self.lastEncoderDistance = 0.0
            self.pathTimeStart = time.time()
            self.DriveController.motors.resetDistance()
            print("Encoders Distance Reset")


        elif self.driveStyle == self.heading:
            self.pathTimeStart = time.time()
            #self._debugPrint("Virtual Distance Reset")


    def enableDisableCutters(self, cuttersEnabled):
        if self.cutterOn == True:
            #cutter.normalized_value = 0.5 #turn on on cutter here.... We dont want
to be turning it on 200 times a second
            print("cutters Enabled")
        else:
            #cutter.normalized_value = 0.0
            print("cutters Disabled")



    def getGPSSoft(self, alterntiveGPS = None):
        """
        this will get GNSS wether it has to wait or not... returns None or
alterntiveGPS if was not able to get fresh data
        """
        #print("softGPS Called")
        if self.GPSEnabled:
            WayPoint = self.Positioning.getGNSS() #getting initial waypoint for
distance
            if WayPoint == (0.0,0.0) or WayPoint == None:#This will only be true the
first run and having trouble GPS data collecting data (in a loop but the robot
should not be moving here)
                WayPoint = alterntiveGPS
            #print(WayPoint)
            #self._debugPrint("waypoint inside getGPSSoft" + str(WayPoint))
            return WayPoint
```

```
    def getGPSHard(self):
        """
        this will get GNSS wether it has to wait or not
        use only if absolutly necessary.. hint let the robot stop before calling
this function
        """
        if self.driveStyle == self.waypoint or self.driveStyle ==
self.geometricWaypoint:
            WayPoint = self.Positioning.getGNSS() #getting initial waypoint for
distance
            while WayPoint == (0.0,0.0) or WayPoint == None:#This will only be true
the first run and having trouble GPS data collecting data (in a loop but the robot
should not be moving here)
                WayPoint = self.Positioning.getGNSS()
                time.sleep(.01)
            #self._debugPrint("waypoint inside getGPSHard" + str(WayPoint))
            return WayPoint




def setup_logger(logger_name, log_file, level=logging.INFO):
    l = logging.getLogger(logger_name)
    formatter = logging.Formatter('%(message)s')
    fileHandler = logging.FileHandler(log_file, mode='w')
    fileHandler.setFormatter(formatter)
    streamHandler = logging.StreamHandler()
    streamHandler.setFormatter(formatter)


    l.setLevel(level)
    l.addHandler(fileHandler)

    #l.addHandler(streamHandler)
```

A.F. Main.py

```
import math
from Loop import Loop
import sys
import argparse #importing the argparse library
import board
import busio
import adafruit_mcp4725
from adafruit_servokit import ServoKit
import traceback
#kit = ServoKit(channels=16)
#cutter= adafruit_mcp4725.MCP4725(i2c)
#paramters
# pathfile = "pathFile/FILENAME.txt"
# debug  set to true
# encoders Enabled  -ec
#feed in run
#python3 Main.py "H_HashTag.txt" -d -ec
def main():

    #creating parser
    parser = argparse.ArgumentParser()

       #adding arguments
    parser.add_argument('filename', action='store', help='type in file located
inside Mk2_5/pathFile') #
    parser.add_argument('-d' , action='store_true', required = False, default =
False) #specify true or false in the actual parameter
    parser.add_argument('-ec', action='store_true', required = False, default =
False)
    parser.add_argument('-gps',action='store_true', required = False, default =
False) #inable GPS
    parser.add_argument('-bp',action='store', required = False, default =
(0.0,0.0,0.0) )#base position


       #executing the parse_args() method
    args = parser.parse_args()
    if args.bp != (0.0,0.0,0.0):
           split = str(args.bp).split(",")
           x = float(split[0])
           y = float(split[1])
```

```
            heading = float(split[2])
            args.bp = (x,y,heading)


        #now using
    loop = Loop(args.gps, args.d, args.ec, args.bp) #line to run added arguments
              #loop
    loop.run(args.filename) #call run here, parameters passed in


if __name__ == "__main__":
    try:
        main()


    except KeyboardInterrupt:
        print("Ctrl-C Detected. Exiting")
        kit = ServoKit(channels=16)
        i2c = busio.I2C(board.SCL,board.SDA)
        #cutter= adafruit_mcp4725.MCP4725(i2c)
        #cutter.normalized_value=0.0 #shutting down cutter

        #loops through 1 -2 pins where motors are
        i = 0
        while i < 2:
            kit.continuous_servo[i].throttle = 0   # First set it to neutral
            kit._pca.channels[i].duty_cycle = 0x0  # This is how you totally turn
off output signal
            i+=1
        sys.exit(0)

    except:
        print("Other Exception. Exiting")
        traceback.print_exc()


        kit = ServoKit(channels=16)
        i2c = busio.I2C(board.SCL,board.SDA)
        #cutter= adafruit_mcp4725.MCP4725(i2c)
        #cutter.normalized_value=0.0 #shutting down cutters


        #loops through 1 -2 pins where motors are
```

```
        i = 0
        while i < 2:
            kit.continuous_servo[i].throttle = 0   # First set it to neutral
            kit._pca.channels[i].duty_cycle = 0x0  # This is how you totally turn
off output signal
            i+=1
        sys.exit(0)
```

A.G. Waypoint and Heading Log File

```
>>>>>>>>>>>>>>>>NEW Pathfile Read<<<<<<<<<<<<<<<
Initial Waypoint(43.51287479100003, -116.48792405911468)
Target Waypoint: (43.5128, -116.48786)
currentHeading 74.786122
currentHeading 74.7876195546
currentHeading 74.790122592
currentHeading 74.7911837214
currentHeading 74.7959144146
Current Waypoint: Lat: 43.5128747934835 Lon: -116.48792404642217
TargetHeading142.62660651184842
currentHeading 75.0162021298
currentHeading 75.6051604032
currentHeading 76.558512524
currentHeading 77.1552880544
currentHeading 79.3329131769
Current Waypoint: Lat: 43.51287484439131 Lon: -116.48792371721873
TargetHeading142.62660651184842
currentHeading 80.9098522981
currentHeading 82.7486941654
currentHeading 84.8539908885
currentHeading 86.9601680781
currentHeading 89.2622098526
Current Waypoint: Lat: 43.512874472670845 Lon: -116.48792388217458
TargetHeading142.62660651184842
currentHeading 92.0625567412
currentHeading 94.2596269006
currentHeading 96.6193975251
currentHeading 99.1091338566
currentHeading 101.723588641
Current Waypoint: Lat: 43.51287359345408 Lon: -116.48792505342635
TargetHeading142.62660651184842
currentHeading 104.986543338
currentHeading 107.872387157
currentHeading 110.796109029
currentHeading 113.521063855
currentHeading 116.301462799
currentHeading 119.090460896
currentHeading 122.048859771
currentHeading 125.187918949
currentHeading 128.310637359


>>>>>>>>>>>>>>>>NEW Pathfile Read<<<<<<<<<<<<<<<
Initial Waypoint(43.51287278003096, -116.48792603362068)
Target Waypoint: (43.5128, -116.48786)
currentHeading 131.432448902
currentHeading 135.487034353
currentHeading 138.794795497
currentHeading 141.582723646
currentHeading 143.941710822
currentHeading 145.257165424
Current Waypoint: Lat: 43.51287242246693 Lon: -116.48792808111574
TargetHeading132.30366871592082
currentHeading 146.150474086
currentHeading 146.273064208
currentHeading 145.910936773
currentHeading 145.389026302
currentHeading 144.71609316
```

A.I. Speed and Distance Log File

```
5516 Total Distance Driven: 0
5517 Robot Speed: 0.46731405471832427m/s
5518 Total Distance Driven: 0
5519 Robot Speed: 0.4662372942235815m/s
5520 Total Distance Driven: 0
5521 Robot Speed: 0.443759918895828m/s
5522 Total Distance Driven: 0
5523 Robot Speed: 0.4914065707881917m/s
5524 Total Distance Driven: 0
5525 Robot Speed: 0.49019521523160625m/s
5526 Total Distance Driven: 0
5527
5528
5529 >>>>>>>>>>>>>>>NEW Pathfile Read<<<<<<<<<<<<<<
5530 TargetDistance20.0
5531 Robot Speed: -1.2720579294766365m/s
5532 Total Distance Driven: 0.003768661731599389
5533 Robot Speed: 0.5452445955253259m/s
5534 Total Distance Driven: 0.10281042680558415
5535 Off course Distance: -0.0274535121276287m
5536 Robot Speed: 0.5830658079031626m/s
5537 Total Distance Driven: 0.14816896264661966
5538 Robot Speed: 0.5884496103768 76m/s
5539 Total Distance Driven: 0.16297441944933155
5540 Robot Speed: 0.6117345560756865m/s
5541 Total Distance Driven: 0.1788566367467861
5542 Robot Speed: 0.6202140449717851m/s
5543 Total Distance Driven: 0.1937966886113408
5544 Robot Speed: 0.6708217882246912m/s
5545 Total Distance Driven: 0.21183242689828075
5546 Robot Speed: 0.6957218746656157m/s
5547 Total Distance Driven: 0.23054114049443486
5548 Robot Speed: 0.6966640400985156m/s
5549 Total Distance Driven: 0.24951904421427465
5550 Robot Speed: 0.7210257462920688m/s
5551 Total Distance Driven: 0.2652666664498864
5552 Robot Speed: 0.7316587561776527m/s
5553 Total Distance Driven: 0.28518673560262603
5554 Robot Speed: 0.7459258327329933m/s
5555 Total Distance Driven: 0.30726032574485107
5556 Robot Speed: 0.7605966944738624m/s
5557 Total Distance Driven: 0.32556525415547666
5558 Robot Speed: 0.7095851660354278m/s
5559 Total Distance Driven: 0.3456199183700591
5560 Robot Speed: 0.7426955512487654m/s
5561 Total Distance Driven: 0.36325187147147053
5562 Robot Speed: 0.7604620994120196m/s
5563 Total Distance Driven: 0.3837103208715815
5564 Robot Speed: 0.7356966080329379m/s
5565 Total Distance Driven: 0.4065914813848635
5566 Robot Speed: 0.7689415883081182m/s
5567 Total Distance Driven: 0.4236850542389036
5568 Robot Speed: 0.791957343883243m/s
5569 Total Distance Driven: 0.4436051233916432
5570 Robot Speed: 0.7735178204107745m/s
5571 Total Distance Driven: 0.46527492834833967
5572 Robot Speed: 0.747002593227736m/s
5573 Total Distance Driven: 0.486271757995822
5574 Robot Speed: 0.724929003085511m/s
```