Introduction

The system as I have left it comprises of a number of python modules representing objects in the real-world system as shown in the below diagram, a set of HTML documentation, and a documentation generator.

Just like the rampaging chariot itself, the system requires each of its vital components to be fully operational before it can all come together as a working whole and have any functionality that can be judged.

Because of this, it has not been easy to describe the state of completion of the system as a whole, other than “incomplete,” or “about halfway.”

In order to measure the functionality of what was so far completed, in my penultimate week of working on the project, I assembled a very basic ‘working’ prototype into a separate folder. This was done in under a day, grossly simplifying the structure of the system and including only a few select components.

This folder will be included along with the rest of these materials, but due to technical limitations at the time, it only proved capable of moving the robot in a straight line, in the wrong direction. I had completely overlooked the changes necessary to the method used for turning the chariot.

This set is not suitable for distribution as the final product even if made to work fully, as it does not have the modular, intuitive and extensible aspects of the full model.

It may, however, be convenient to have this set of code for prototyping communication with sensors, especially while still learning to work with it all. I’d suggest making it work there, then applying those ideas into the larger model.

The top-level module is Main. This alone does not represent any real-world object or concept but is the representation of the system itself. It is from here that commands should be issued to the chariot in their most basic, human-readable form.

Main creates an arena, a chariot (using the area) and a visualiser (using the chariot and arena.) It then goes through the commands for the chariot.

A chariot has the properties of a real-world chariot. It can be commanded to go somewhere, or to turn or to go in a straight line.

An arena is a model of an arena, storing all of its properties, contained obstacles, and appropriate waypoints.

A visualiser aims to reproduce the arena and the intended movements of the chariot on-screen.

The two major challenges going forward should be to debug interactions between the system and the components on the i2c bus, and to debug and continue work on the movement and navigation functions.

I would suggest becoming familiar with how the python interacts with the control board in order to drive the wheels, and then attempting to fix the existing problem wherein the chariot moves backwards instead of forwards. Having got an idea of how the methods in Chariot use the control board from this, work on making the turning functional.

This document contains requirements and specifications for the system, as well as the state of completion for each module going forward.

Requirements

* Modular python code that could allow a basic autonomous chariot to be created by a group of 15-19 year olds in under 12 hours.
* Useful and comprehensive guideline comments preceding all classes and methods.
* Provide code capable of making use of two odometers, two rangefinders (of variable type – infrared or ultrasonic) with accompanying servos or stepper motors, all via **i2c**.
* Allow communication between the Raspberry Pi and the chariot’s control board via UART.
* Make provisions for extensibility for other sensors, such as an accelerometer or compass.
* Provide code capable of driving the chariot to waypoints using feedback from the above sensors to keep track of its position.
* Optional visualisation of where the chariot ought to be on the course and how it ought to be moving.
* Create an autonomous chariot using this code that allows a chariot to make its way around a standard assault course unaided.
* Guide detailing the use of the package written with the students in mind.

Main

Working/ almost working methods:

* \_\_init\_\_()
* guiworker()
* getConfig()

Methods with some code: none

Empty methods: none

The getConfig() method retrieves i2c bus addresses from a csv file and stores them in a dictionary which is required as a parameter for several of the following classes. Use your own judgment as to whether or not there will be enough addresses to make this feature necessary.

Visualiser

Working/ Almost Working methods:

* \_\_init\_\_(chariot, arena)
* paint()
* getRangefinderPoints()
* getChariotPoints()

Methods with some code:

Empty methods:

There are many online tutorials for tkinter that will assist you when adding the remaining obstacles to the visualisation of the arena. I wanted to finish the paint method to include everything and give you a nice visualisation myself, but there just wasn’t time. A nice demonstration of what it could look like at the end should be provided in a separate folder called AssaultCoursePrototype, which is where I started on a very early prototype of what my system could look like, before I really started on the system proper.

Run Assault.py and you will at least see how the course should look (not to the proper scale, of course.) Use the modules in this folder, especially the paint() function in Assault.py, to help you complete Visualiser.

Chariot

This is hopefully the module that needs the most work.

Working / Almost Working methods:

* \_\_init\_\_()
* loop()
* getBearing(point)
* getAngle(vector, vector2)
* unitVector(point, point2)
* start()

Methods with some code:

* go(target, obstacles)
* straight(target)
* turn(target)
* updateBearing()

Empty methods:

* arc(target)
* collisionCheck(obstacles, position)

ControlBoard

Explanation:

Working/ Almost Working methods:

* \_\_init\_\_()
* forward(speed)
* reverse(speed)
* right(rspeed)
* left(rspeed)
* center()
* stop()
* transmit()
* loop()
* shutdown()

Methods with some code: none  
Empty methods: none

How it works: every 16 milliseconds, the loop in this class’s thread transmits two bytes over UART to the PIC that mixes the signals on the control board into PWM signals that are output to the drills.

Each of the two bytes should have a value representing a number between 2 and 254. The centre value is 127. Values above this cause the chariot to move forwards. Values below this cause the chariot to move backwards. The greater the difference between the centre value and the value, the greater the motor speed will be.

Values within plus or minus 7 of the centre value will not cause the chariot to move. The code in this module eases the values sent to the control board up to the desired value so that excessive strain is not put on the motors, and also to avoid jumping straight from a low value to a high value. If this happens, there is a safety feature that shuts off the motors and renders the board unresponsive to further signals until it is switched off and on again.

Sight

Please note that as some or all of this functionality may be moved to the rangefinder/servo PIC, these methods may have to be omitted or changed substantially. It may even be necessary for the i2c communication to be moved here, if the servo and rangefinder are to be supplied with code on the same PIC.

Working/ Almost Working methods:

* \_\_init\_\_(chariot, configVariables, arena)
* begin()
* nextEdge(threshold, maxdist, timeout)
* nextObstacle(threshold, maxdist, timeout)
* checkObject(obj, posA, posB)
* trackNextPole()
* closestObject()

Methods with some code: none

Empty methods: none

This code is adapted from my prototype code that was able to identify a pole, detect that it had been moved and find it again. This, and the video of it working, should be included.

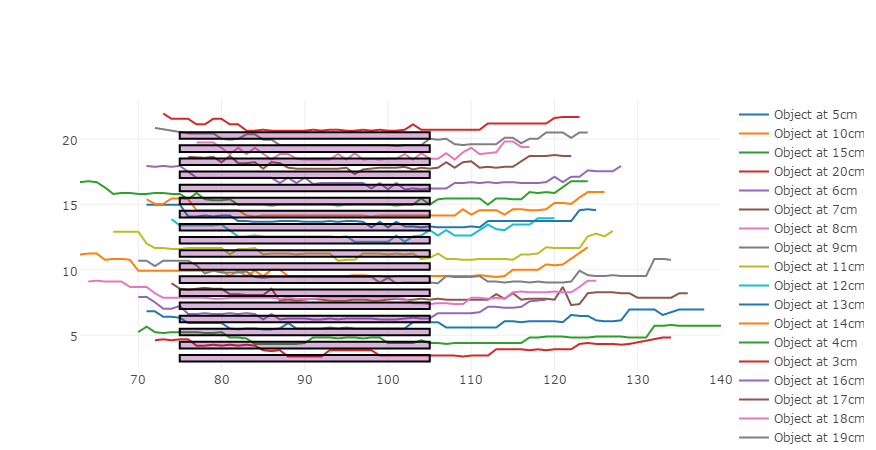
Rangefinder

Working/ Almost Working methods:

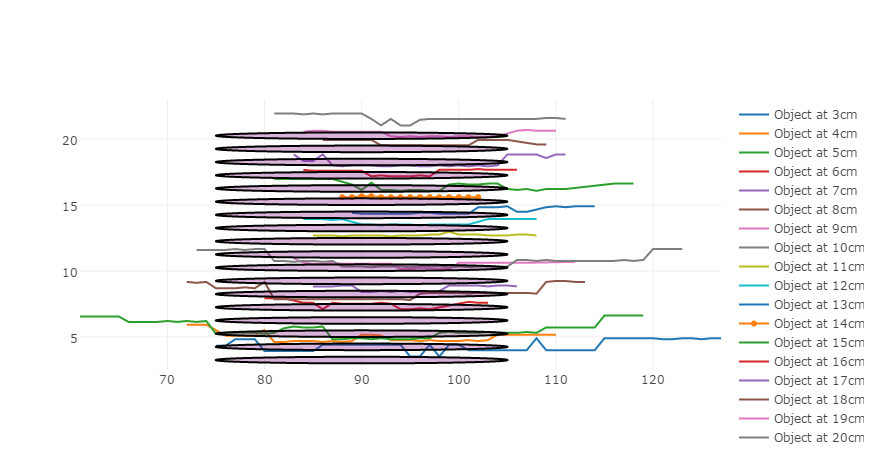
* \_\_init\_\_(configVariables)
* read()

Methods with some code: None

Empty Methods: None  
  
As one of the requirements is to have support for either infrared or ultrasonic rangefinders, the input received from this may need to be different based on the sensor. The ultrasonic distance sensor I have been using for prototyping is an HC-SR04 with an operational range between +2cm and 4m. Distance is reported in mm and is accurate to within 1cm.



HC-SR04 Reading Accuracy on Flat 5cm Plane



HC-SR04 Reading Accuracy on 5cm Diameter Pole

I moved the setup by hand and had little means of accurately measuring the true distance, so much of the random error is most likely my own.

How it works (HC-SR04): pulses of sound are emitted from one transducer and received through the other. The trig line is briefly brought high by the master. Once it has been emitted, the rangefinder brings the echo line high. The master detects this and waits for it to drop again. It is brought low when the rangefinder receives the echo of the sound it emitted. The length of time that passes while the line is high is then used by the master to calculate the distance from the closest surface (that which the sound bounced off.)

StepMotor

Working/ Almost Working methods:

* \_\_init\_\_()
* scan()
* restrict()
* setSpeed()
* reverse()
* stop()
* move()
* transmit()

Methods with some code:

* read()

Empty Methods: None

I believe one of our aims is to have separate module available for handling either a stepper motor or a servo.

The servo I have been using for prototype is a Futaba S3003.

<http://www.servodatabase.com/servo/futaba/s3003>

The code here, too, may become completely obsolete.

Arena

This is important! Here is a refactoring task for you: the python module that refers to an arena is actually still called Course.py, and the class within is called Course. you need to change both of these to arena, and then find every reference to Course and change it to Arena. These will be in imports and in the \_\_init\_\_.py files.

This module is hidden inside the course folder and actually contains quite a few classes. Most importantly other than Arena (Course..) there is Obstacle, which, if you know what this is, would be an abstract class with the basic attributes of an obstacle. The other classes below it are types of Obstacle; they inherit Obstacle so that they start out with the same kinds of attributes, but have more added to them. For example, a net is passable by default, where an Obstacle is not.

The only notable method in Arena is readCsv(), which needs to be completed so that all the different types of obstacles will be loaded in correctly when kept in the csv file (also in that folder.)