**Part 4**  **Autonomous Rampaging Chariot Software**

The Autonomous Rampaging Chariot Software is written in the Python language, which is generally considered to be the best language for pupils to learn. It is also the language most commonly used by undergraduates at universities. Extensive use is made of the standard python functions provided in the software download provided with the Raspberry Pi.

We have provided a sophisticated software programme that should work ‘straight out of the box’. Every effort has been made to simplify the code and provide extensive notes to explain how it works. For educational reasons we have simplified the code to reduce the performance of the example programme so that pupils can learn to code by experimenting in modifying the code provided to increase the speed, accuracy and sophistication of the code. This is to allow a real challenge to compete against other teams to create the fastest autonomous robot round the Assault Course at the Rampaging Chariot Robotic Games.

Happy coding, and may the Force be with you!

## 4.1 Operating Modes

There are two available modes to use with the autonomous software: Simulated Mode and Real Mode. Simulated mode is used for testing and debugging software code, without the need to be connected to the chariot. Real mode however, requires the Raspberry Pi to be connected to the chariot as it will output to the motors and read from the odometers and other sensors.

### Simulated Mode

When the software is running in simulated mode, the module advSim.py aims to replicate real conditions that might alter the behaviour of the chariot while running. Left-right wheel bias and wheel friction are the two currently applied.

### Real Mode

When the software runs in real mode, the motor commands are output to the chariot motor board via a serial link. In addition the odometer pulses and sensors are read from the chariot and the simulated ones are ignored.

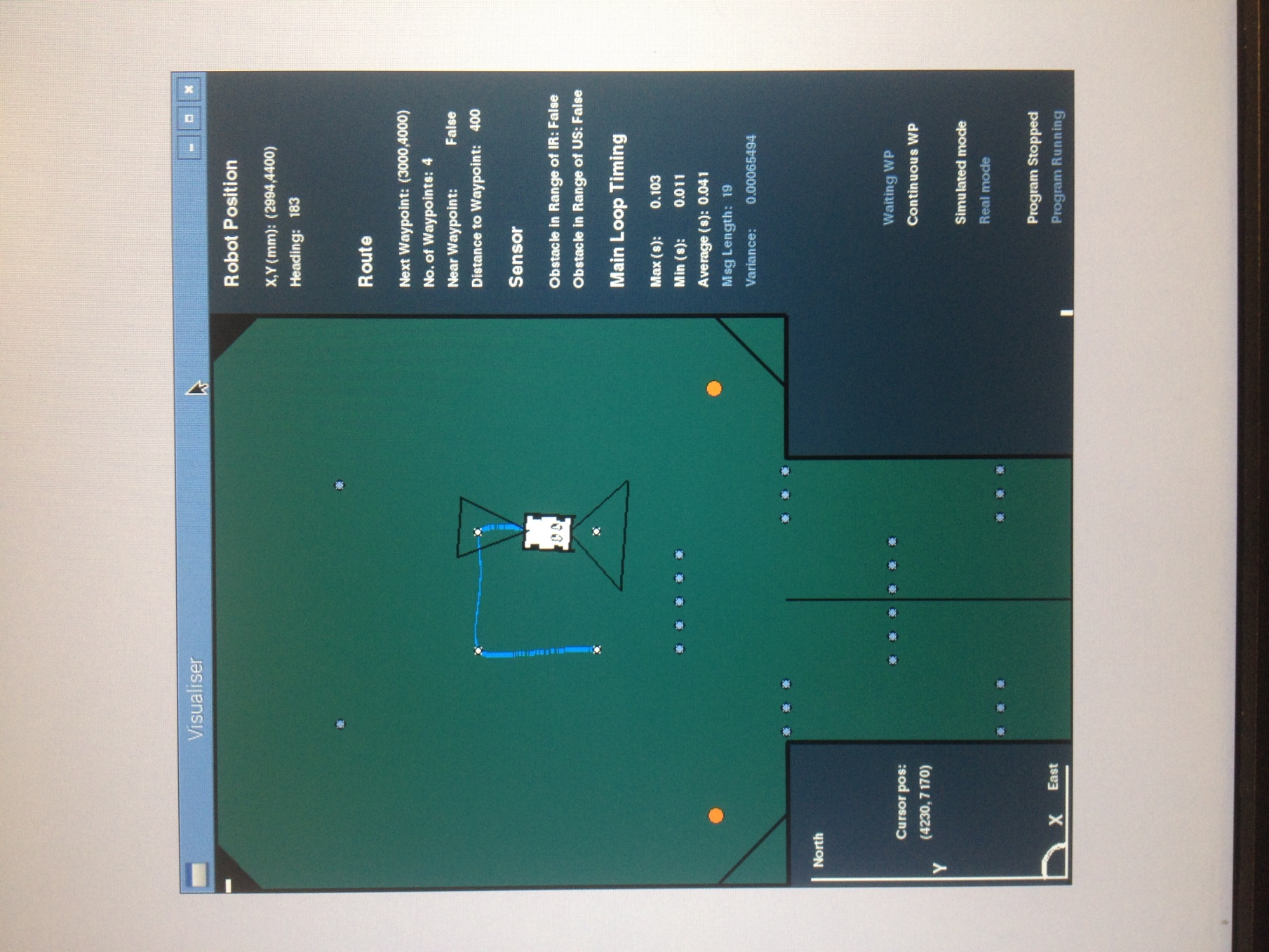
**Before testing the Chariot in Real Mode on the bench it is essential that the user checks that the Rampaging Chariot chassis is supported on a wooden block and the wheels are clear of the bench and free to rotate.**

**Before testing the Chariot in Real Mode on the floor it is essential that the user checks the area is clear of people and there is a physical barrier between the Autonomous robot and any spectators.**

## 4.2 The Graphical User Interface (GUI)

The Graphical User Interface is an excellent and powerful debugging tool that runs in both Simulated and Real Modes.

* It displays an accurate representation of where the chariot should be, in relation to the course.
* It shows diagnostic information such as loop timings and batch data sizes that can help you if the software doesn’t run as expected.
* It also gives the users a means of interacting with the software during run time, for example; plotting new waypoints around the course, pausing of the loops, deleting and creating obstacle poles and any additional features that you may wish to include.



### The Assault Course Default GUI Provided

### The Course Window

The chariot is displayed in a window alongside the information panel, and shows a visual representation of the robot position X,Y and Heading in relation to the course.

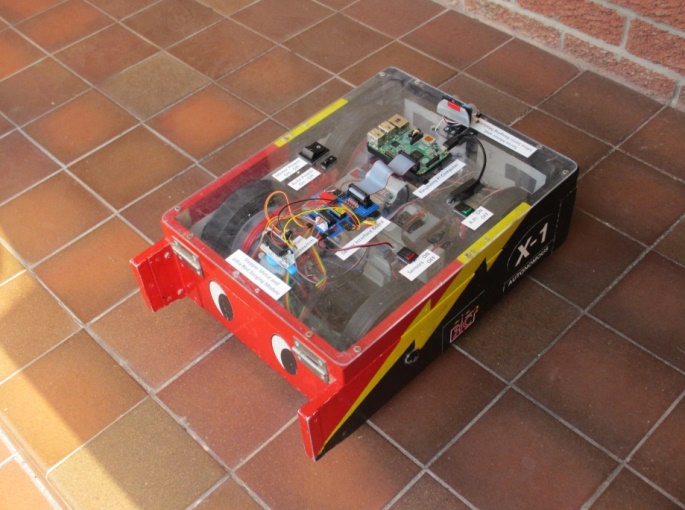
**The Information Panel**

One of the main features of the GUI is the ability to show us useful information while the software is running. This includes, but is not limited to; the positional information of the chariot, information relating to the waypoints, technical information about the loops and runtime information to show whether the loops are paused or running.

All of the course environment can be changed and the sizes adjusted to scale. The default course provided, is the Assault Course at the Rampaging Chariot Robotic Games.

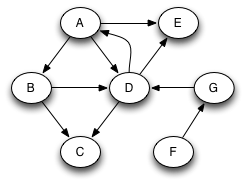
* The blue trail represents where the chariot has already travelled, this originates from its X and Y position as calculated by the Odometers and the Sensors?
* The black cones represents the scan range of the Infra-Red and Ultra-Sonic sensors.
* The orange circles represent the football objects; these are specific to the assault course and can be changed.
* The smaller dark grey circles are pole objects; again the location of these depends of the course, and can be changed.

## 4.3 Advanced Programme Concept

A simple software programme consists of a series of instructions. When you press ‘RUN’ the computer actions each instruction in sequence until it reaches the end of your programme. It then normally loops back to the start and repeats the sequence until you tell it to stop. Larger, more complicated programmes collect a number of instructions relating to the same topic, such as (move an icon on the screen) into blocks of code called Functions. These functions are called from the main programme loop when needed and the programme waits until they are finished before moving on.

This type of programme has the disadvantage that you are never quite sure how long the complete programme is going to take to action. Some actions like reading an Odometer must be done at regular intervals to ensure every full wheel rotation is captured and recorded. An advanced control algorithm that smoothly causes the robot to converge and capture a desired track also relies on being calculated at exact time intervals to prevent the control going unstable and sending the Chariot weaving out of control across the arena.

Figure 1 - Graph with Nodes and Edges



Node

Edge

The Autonomous Rampaging Chariot software is built using a robust design that functions through the use of a network of nodes.

Figure 1 depicts a network of nodes in a diagram which is known as a “graph”.

A node is a fundamental unit of which graphs are formed.

An edge is an ordered pair of nodes, denoting the connection between the nodes as a line.

Each node contains a loop that performs an aspect of the Autonomous Rampaging Chariot software and is known as an ArcNode in the software (Arc = Autonomous Rampaging Chariot).

Each ArcNode is responsible for a particular part of the robot and is run independently on its own “thread” – something that gives us the flexibility to change the timing of each loop, and also means the loops don’t have to wait around for other parts of the system. Each ArcNode is a part of the autonomous chariot system and allows pupils to make alterations that should not cause unforeseen changes to the Chariot performance in other parts of the programme.

The ArcNodes are all launched from the main programme. It will start them all individually and allow the user to pick and choose which ones to run and how that will then affect the Chariot.

Because each ArcNode runs on its own “thread”, they are effectively separate processes and can function in parallel with each other. The main achievement of the design is the ability to have a plug and play scenario where you can pick and choose “modules” that you can insert into the program and also be allowed to leave certain ones out – so long as they aren’t doing anything very important.

The program also has an advanced messaging system that allows messages to be passed between ArcNodes (the “edges” Figure 1). Each ArcNode will have an Initialisation and update function built into them on creation. The initialisation consists of variables and functions that need to be run at the start of the loop one time only. The update function is executed by the ArcNode’s loop, receiving messages from other ArcNodes and acting upon the information it is given.

### Timing

The speed at which the ArcNodes run is variable and depends on a number of factors:

* The time between iterations of an ArcNode’s execution is constrained by settings which dictate a minimum and maximum allowable duration;
* Execution within that time window is further determined by the arrival at the ArcNode of messages queued for processing.

### Translators

Translator functions control the data flow between the ArcNodes. They control both the direction of data flow and supply the ArcNodes with the most up-to-date information. They correspond to the “edges” (as seen in Figure 1)

### Queues

A translator function passes information into its destination ArcNode using a Queue. Queues work using a FIFO (First In First Out) system

## 4.4 Autonomous Rampaging Chariots Software Modules

### main.py

This is not an ArcNode module but is responsible for initializing all of the ArcNode modules, connecting the ArcNodes using suitable translators, and launching them.

### routeControl.py

This ArcNode module is responsible for holding a list of waypoints that the chariot will navigate to.

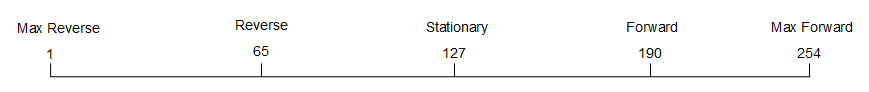
### trackControl.py

This ArcNode module is responsible for calculating the next position that the chariot needs to advance to in order to reach the current waypoint.

### rcChanControl.py (motorControl.py)

This ArcNode module is primarily responsible for sending motor commands to the chariot. The module is sent a demand forward and a demand turn – similar to the commands sent via the standard radio control.

The Master Motor Control Board on the robot expects to receive an 8bit byte containing a number between 1 and 254 for each channel as follows:



**Figure 2 – Motor Operations**

A value of zero is invalid and causes the robot system to shut down until a valid byte isreceived.

After applying power to the Chariot Motor Control Boards, stationary values of 127, 127 must be transmitted consecutively at least twice from the R-Pi to the motors before the auto system is activated. This safety action is required, because the motors can receive unintended spurious messages from the R-Pi during the R-Pi boot-up process which might cause the chariot to make small unintended movements.

### vsimControl.py (vehicleSimControl.py)

This ArcNode module is responsible for replicating real conditions of the chariot when it is running in simulated mode. To make the simulated performance approximate to the real vehicle, variables for friction and left-right wheel motor bias are added.

### odomControl.py

This ArcNode module is responsible for reading the two wheel rotation odometers via a dual simultaneous serial data bus. If a full rotation (rollover) has occurred, it adds or subtracts a distance equivalent to the circumference of the appropriate wheel. It also calculates the heading of the chassis from the difference in distance travelled by each wheel.

### envSimControl.py (enviromentSimControl.py)

This ArcNode module stores lists of X-Y coordinates where each of the obstacles on the course is located. These lists are then passed to the visualControl module to be displayed on the screen. Some information will also be sent to the sensorControl loop that will deal with simulated collision detection.

### statsControl.py

This ArcNode module is responsible for gathering data from a number of loops in order to give useful diagnostic information to the user. This telemetry is similar to that employed by Formula 1 racing teams and is particularly useful for debugging. Real time graphs of user chosen parameters are also displayed. TBD

### visualControl.py (guiControl.py)

This ArcNode module displays the graphical interface that allows us to see the simulated version of the program. This graphical interface is also used to add new waypoints on the course, and also to display useful diagnostic information.

### scanSimControl.py

This ArcNode module is responsible for simulating readings from an IR sensor. TBD

### sensorControl.py

This ArcNode module controls the mode of scan required and determines whether there is an obstacle within range of the chariot or not. This information is sent to the trackControl module to allow the robot position to be updated.

Files in **Library** are data files that are normally only accessed once when data is required. They are not within ArcNodes graph.

### Lib/navigation.py

This library module contains the sensors to world coordinate conversions as well as a waypoint manager which is used to create waypoints from coordinates and obtain coordinates from waypoints.

### Lib/sensoractions.py

This library module deals with all of the sensor command information. E.g. the different scans to fixed angle or between two angles as well as the calculations for figuring out position and heading differences.

## 4.5 Control and Navigation

In order to navigate round the Assault Course at the Scottish Robotic Games, an autonomous robot needs some sort of on-board sensor to obtain the motion and position of the robot. This can be conducted ‘Open Loop’ which is basically dead reckoning resulting in small errors adding up over time, or ‘Closed Loop’ where the errors are corrected at intervals by using an update of position obtained from some other sensor such as an Infra-red distance sensor, camera, ultra-sonic distance sensor, GPS etc.

The rules for autonomous robots prohibit the use of special transmitter beacons and cameras external to the robot, but we can use inertial systems, range sensors and general transmissions from commercial radio masts, the earth’s magnetic field, GPS or astronomy. Unfortunately as tests are to be conducted indoors the satellite Global Positioning System (GPS), and star sightings are unlikely to work.

### Method of Navigation

The robot will proceed round the course via a series of waypoints defined by X, Y, coordinates.

(X and Y are Cartesian coordinates of the Waypoint measured from the arena origin at bottom left)

Waypoints are numbered consecutively from the start point. Movement between Waypoints is accomplished via:

1. A turn on the spot
2. A straight line
3. A circular arc. (This can be added by students themselves)

### Open Loop Navigation and Control

The Navigation System is based on two odometers measuring the distance travelled by the two drive wheels from the start point. The difference between the distances measured by each drive wheel odometer is used to calculate the current heading relative to the heading at the start point. This technique enables a continuous estimate of raw position and heading to be calculated and is called ‘Open Loop Navigation’ as it relies on ‘dead reckoning’.

Y

X

θ**C**

Open loop Position X,Y

Open loop Heading θ

Waypoint 0

Waypoint 1

Waypoint 2

The distance travelled and the current heading are compared with those expected at the next waypoint and a (near) match is used to trigger a waypoint change.

The robot motor control parameters are demanded velocity ‘V’ and demanded heading rate ‘ω’. This is the same as the control parameters used in manual operation when using the standard radio control.

Slight differences in the performance of the two wheel motors may cause the robot to stray from its intended heading. This can be corrected by comparing the calculated heading regularly with the planned heading and applying a minor lateral correction to motor power (Demanded heading rate ‘ω’) to correct any heading discrepancy.

The odometers can be calibrated by adjusting specific constants such as wheel diameter to make it reasonably accurate over a short distance. Unfortunately, if either wheel skids during a manoeuvre, the odometers will be in error which causes the calculated open-loop distance and also the heading to be in error. This will cause the robot to diverge from its planned route and probably hit an obstacle.

A small error in X or Y caused by a wheel skid will cause a position error that is **constant** until another wheel skid occurs. Unfortunately a wheel skid will also cause a heading error, and this will cause **an increasing error in both the X and Y** **positions** throughout subsequent manoeuvres.

### Closed Loop Navigation and Control

The navigation loop is closed by using position and heading fixes from other sensors to determine the actual position and heading of the robot in the arena, or relative to a known obstacle. This allows it to determine its current position and heading errors, update its actual position and then calculate a new path to regain the planned route.

Our autonomous robot is provided with an IR distance sensor attached to a stepper motor which enables it to scan over a wide arc. This sensor is mounted on the front of the detachable lid and is used to detect and track specific obstacles.

An ultra-sonic distance measurer mounted on a servo motor is also provided and can also be used to detect and track obstacles. These two sensors and actuators have different characteristics which can be measured and compared.

By measuring the distance and associated angle of two obstacles, or the side of the arena, a geometry method called ‘triangulation’, can be used to calculate the ‘actual’ position of the robot in the arena and ‘actual’ robot heading.

The ‘actual position’ is then used to correct or update the estimated ‘open loop’ position and allow the robot to regain the planned track.

Obstacle B

Radius A

Radius B

θ**C**

Y

X

θ**C**

Obstacle A

RA, ϕA

Waypoint 0

RB, ϕB

Obstacle A

RA, ϕA

RB, ϕB

Y

X

ϕ**B**

Waypoint 0

ϕ**A**

**θC**

Obstacle B

**Actual Sensor Derived Position**

**Open Loop Odometer Position**

**Open Loop Odometer Position**

**Triangulation to Determine Actual Position & Heading of robot in arena.**

(Triangulation using the known coordinates of the obstacles is shown in red)

**Measurement of Distance and Angle**

**of two known obstacles A and B**.

(The expected position of these obstacles using open-loop odometer navigation is shown in blue)

Some waypoints are designated as ‘update waypoints’ and at least two known obstacles (or the arena side) are designated by the programmer for a particular sensor (or sensors) to scan. The choice of obstacles to obtain an accurate update position and heading is very important.

Consider:

Uniqueness,

Scan sector angle required,

Field of view of sensor,

Angle between sensor and obstacles near to 90 degrees.

The robot stops at the update waypoint position calculated by the odometers (shown in blue) and the Infra-Red and/or Ultra-Sonic sensor scans over an arc where it expects the known Obstacle A to be located and records the minimum distance of any obstacle in that arc and the associated scan angle. The Infra-Red and/or Ultra-Sonic sensor then rotates to scan over an arc where it expects the known Obstacle B to be located and records the minimum distance of any obstacle in that arc and the associated scan angle.

By using the maths technique of ‘triangulation’ (Described in Appendix ?) the software calculates the actual position of the robot in the arena and the actual heading (**θC**) of the robot chassis. It then calculates a new route to intercept the original planned route or to go direct to the next waypoint.

## 4.6 Autonomous Software

As described in [**4.3**](#_4.3__Advanced) and [**4.4**](#_4.4_Autonomous_Rampaging)**,** the Autonomous software programme works by connecting lots of different program files together and then sending information between one another to control the Chariot. What happens to the Chariot is then displayed on the GUI screen. The program files define ArcNodes (Figure 1) that are joined together by communication channels.

The following gives greater detail regarding the software modules introduced in [**4.4**](#_4.4_Autonomous_Rampaging):

### 4.6.1. Main.py

The Main.py file is the one that contains the setup for the ArcNodes. It is broken up into 5 sections.

1. First section imports the modules that must be configured and assembled to build the system, as well as other libraries you might want to use like math and time.
2. Second section is the calibration values to be used as initialisation parameters for the modules that have been imported.   
   This is where most calibration changes will be made for user convenience, avoiding the need to dig through files to find out where a variable needs to be changed.
3. Third section is where the imported modules and their functions are used to construct individual ArcNodes, ready to be connected to each other.
4. Fourth section is where the program files within their respective ArcNodes are connected together to allow information to be pass between them (i.e. the “edges” are formed). Information that flows between ArcNodes is processed through Translator functions. Translators are responsible for constructing messages from source information (or “state”), converting source ArcNode terms into terms that a destination ArcNode will understand.
5. Fifth section is where the ArcNodes are launched as parallel threads.

### 4.6.2. OdometerControl.py

The odometerControl module is just like the test Odometer program (that you should have been through by now) with just a few changes this is labelled by the start of Section 1. In the test program you have to print the values to the screen and handle all the information about the RC in just that program file, however in this module the Odometer solely keeps track of the distance travelled and the heading of the Chariot.

Section 2 is the only difference since it has the **Translators** in it. There are 2 translators: 1 from odo to trackControl and 1from odo to visualControl. Inside both of the translator they contain the items to be sent inside a python feature known as a ‘dictionary’. A dictionary is like an array but instead of being indexed by integers it is instead indexed by readable names (known as dictionary “keys”). For example in the odoToTrackTranslator the readable name “sensedAngle” is used to store an angle value in the dictionary in a manner that is easily retrievable.

Also in the odoToTrackTranslator translator extra calculations are done to get the angle of the chariot within world space.

### 4.6.3. RouteControl.py

The routeControl module has been broken up into sections like main. There are 5 sections.

1. The first section imports the ArcNode structure so that some functions can be accessed as well as importing libraries you might want to use like math and time.
2. Second section initialises parameters that will be need for use later on. These parameters include creating the initial waypoints, so this would be were you could pre-program a route, setting the near value which is the detection radius to reaching a waypoint.
3. In section three there is a loop that checks if any messages have been sent to routeControl from another module. I.e. a new waypoint has been placed using the visualControl and so this new waypoint needs to be appended to the list of waypoints. Also runActions is reset here since it needs to be outside of the messaging loop.
4. Section four deals with checking to see if the distance to go before reaching the waypoint has entered within the “near” radius. The “near” radius is in place so that the robot does not get finicky about when it has reached the waypoint otherwise the robot could end up circling round the waypoint for a while never bang on hitting the waypoint but being 1mm or so out constantly. After entering the “near” to waypoint radius the target waypoint will normal be updated to the next one in the waypoint list but if a waitPeriod is set then the module will wait till the period of time set has elapsed before updating to the next waypoint. Inside the wait period the action to be run by the sensors is also set from the waypoint information.
5. Section five contains routeControls **Translators**. There are 3 translators: 1 to track, 1 to sensor and 1 to visual. The one to trackControl contains information about the waypoint the chariot is on and the one it needs to go towards, while to sensorControl any actions that need to be performed by the sensors at a waypoint are sent. The one to visualControl just contains the list of all the waypoints that way they can be projected onto the gui.

### 4.6.4. SensorControl.py

The sensorControl module has been broken up into sections like main. There are 5 sections.

1. The first section imports the ArcNode structure so that some functions can be accessed as well as importing libraries you might want to use like math and time.
2. Second section initialisation parameters that will be need for use later on. These parameters include robot position and heading, the objects that are wanted scanned, sensor offsets and waiting condition.
3. Section three there is a loop that checks if any messages have been sent to sensorControl form another module. E.g. a “sensedRobot” message which will contain updated robot position and heading. Also actions is reset here since it needs to be outside of the messaging loop, otherwise when an action is received from routeControl the action would be reset before a message can be sent with it to sensoractions.
4. Section four contains two “if” statements the first “if” is to check if a request to reinitialise the sensor has been sent. The other “if” is to check to if an action has been given by routeControl to run on the sensors.
5. Section five contains sensorControls **Translators**. There is only a translator to trackControl which will contain a message with the angle and position differences after a scan action has been completed.

### 4.6.5. TrackControl.py

TBD

### 4.6.6. Lib/Navigation.py

The navigation lib file has been broken up into sections like main. There are 3 sections.

1. The first section imports needed functions from math. It also has the waypoint class that defines the structure of a waypoint. Do not be misled by “actions” though it looks like a single parameter there is actually a lot of components that will be stored within “actions” when a scanning waypoint is made.
2. Section two contains the waypoint manager and waypoint type enum. The waypoint manager is used when a waypoint is placed using the GUI after the program has been run from main this allow for easy construction of a waypoint without the user having to enter or the data manually. Waypoint type enum is used when swapping between a continuous and waiting waypoint.
3. Section three contains two important functions for the chariot. The two functions are sensor to world and world to sensor. These functions are used in multiple places throughout the program files. They are used for changing the perspectives that the program is working with in difference areas. Axis rotation is just a support function for the other two.

### 4.6.7. Lib/SensorActions.py

The sensoractions lib file has been broken up into sections like main. There are 10 sections.

1. Section one contains the required imports for the other functions to work as well as the setup for a named tuple that will be used within a function further in the file. A named tuple sort of what is says it is the data storage type of a tuple but instead of indexing through the elements by numeric values, you index by names assigned to the stored values when the tuple is created.
2. Section two contains functions to do with the setting up connection to the I2C as well as reading and writing messages to it. readData just sorts out the bytes that are received back from the I2C and gives puts them in variable with names so we know what we have got back. Inside the I2C class the init just sets up default variables to be used with the I2C class context (i.e. so that the functions within I2C know what the variables are). Write sends a packaged message to the I2C containing the address, control and txBytes. Address is so the message is sent to the right part of the I2C, control contains a value that equates to what type of command to request of the sensors and txBytes contains the actual data being sent i.e. an angle for the sensor to turn to.
3. Section three contains parents/bases for different classes within the sensoractions file to inherit from. What is referred to here is the concept of inheritance. In inheritance there are different types of class, one type is the parent/base class and another is the child/derived class. The child/derived class inherit most fields and functions from the parent class it has been linked to. This is done so that code that is common to many classes can be placed in a parent/base class and reused. For more information on Inheritance search “inheritance programming” most resources found on search give adequate detail.
4. Section four contains the Stepper Motor class with the functions to call the I2C class functions with the correct control message according to the current situation. Be it a write message to get the motor to turn to a set angle or a read to get what the current angle at that time is. The Stepper Motor class is also a child of the parent class Motor in section 3.
5. Section five contains the IR Class which is pretty much the same as the Stepper Motor class but it is a child of the Sensor Class not Motor with messages being sent relating to commanding the sensors not the motor.
6. Section six contains the different setup functions for the triangulate function. A triangulate action is made of 2 scan actions and a scan action contains the chosen sensor, coordinates of targeted object, width of the scan to be performed, number of scan passes to occur and the speed at which the sensor is to move. The makeTriangulator function calls the makeTriangulate function so that after the scanActions are filled they can be made into one triangulate action while the makeScan function is only called if you want to make a scanAction manual instead of having makeTriangulator deal with it
7. Section seven contains the triangulate function which has the math for figuring out the actual position of the RC from objects real positions and where they have been detected at. It has been sub divided into 3 major parts. They calculate the angleDifference, positionDifference and the point on the object that it is detected closest.
8. Section eight has the class that deals with the triangulate actions. So within this it calls the scan actions to occur and get results from them. Then calls triangulate with the results to get the difference in the actual position and where the RC though it was.
9. Section nine has the class that deals with scan actions. It calculates where the object should be then scans around that point using the scan width given in the creation of the scan action. At the end it has a range and angle of the object from current position which can be used by the triangulate function.
10. Section ten contains a “façade design pattern” for the scanning sensor class. Normally used to hide larger complex systems behind simpler interfaces, in this case instead of having to figure out which class to call for which function you just call scanning sensor which will call the associated function.

This manual is a work in progress and an update of this section will be issued before you start modifying code in the main programme.

**Appendix ?**

**Programs, Processes, Threads, Functions, Loops and Pipes**