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.

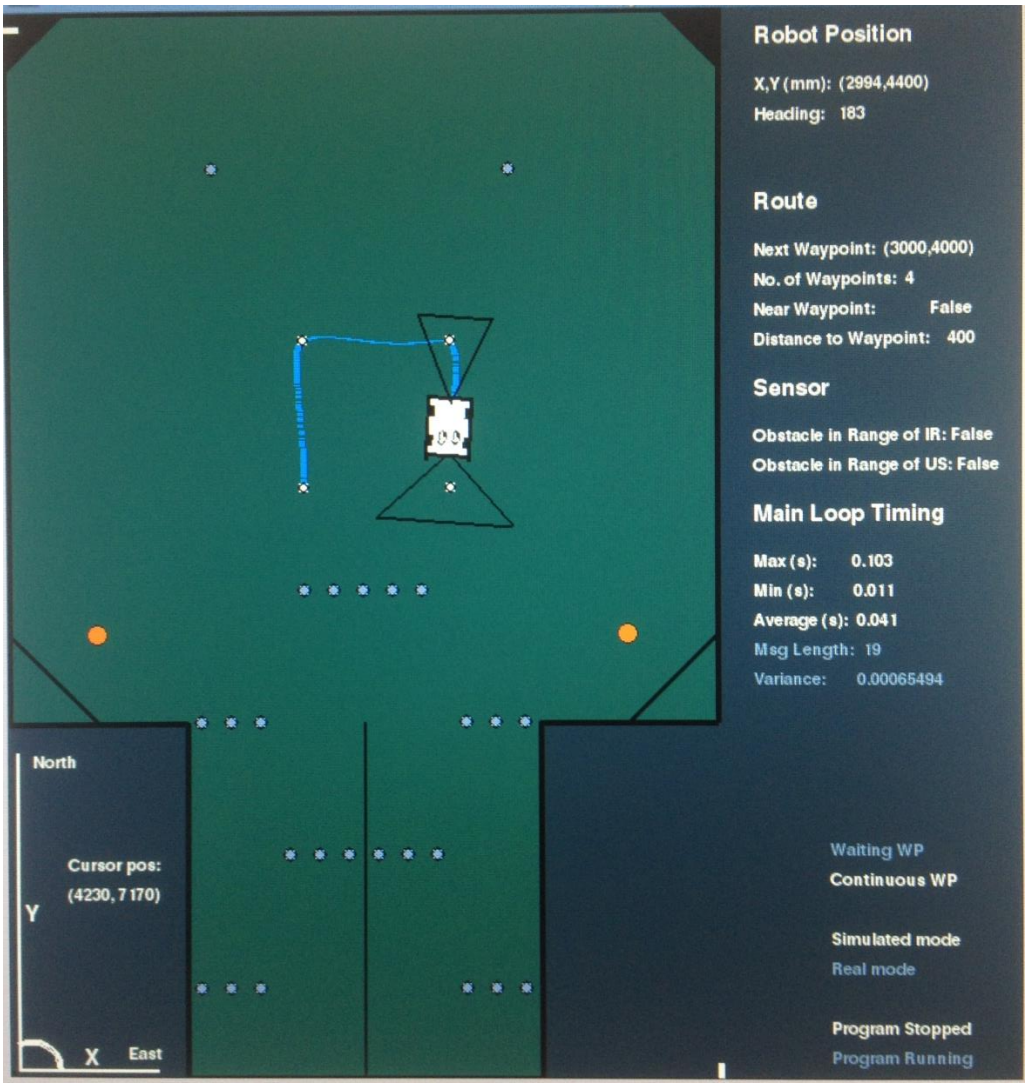
Before testing the Chariot in Real Mode on the bench it is essential that the user checks that the Rampaging Chariot chassis is supported on a wooden block and the wheels are clear of the bench and free to rotate.

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.

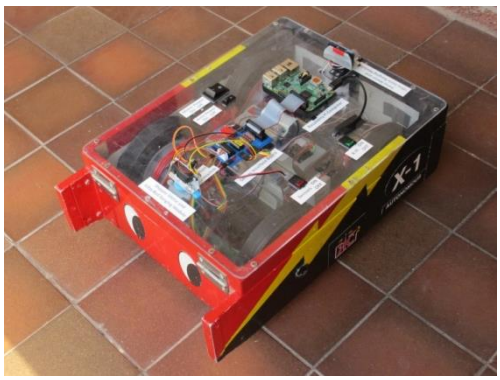
- The blue trail represents where the chariot has already travelled, this originates from its X and Y position as calculated by the Odometers and the Sensors?
- The black cones represents the scan range of the Infra-Red and Ultra-Sonic sensors.
- The orange circles represent the football objects; these 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 Control Loop 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.

The Autonomous Rampaging Chariot software is built using a robust design that functions through the use of multiple control loops. Each control loop is responsible for a particular part of the robot and is run independently on its own thread. This 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 control loop 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.



These loops are all run from a central control thread that 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.
Each loop runs on its own thread meaning that they are 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 the loops. Each loop 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 allows the loop to receive messages from other loops and act upon the information it is given.

Timing

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

- Has the control loop been modified by the user whilst the programme was running?
- The minimum time allowed by the user for that loop.
- The maximum time allowed by the user for that loop.

Translators

Translator functions control the data flow between the control loops. They control both the direction of data flow and supply the control loops with the most up-to-date information.

Queues

The translator functions pass information into the destination control loops using Queues. These work using a FIFO (First In First Out) system

4.4 The Control Loops

main.py
This module is responsible for initializing all of the control loops, connecting the suitable translators, and starting all of the loops.

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

trackControl.py
This 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 motor control 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:

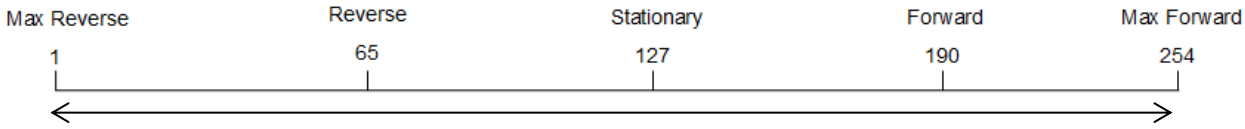


Figure 1 – Motor Operations

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

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 (environmentSimControl.py)
This 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 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 (obstaclesControl.py)
This module stores lists of X-Y coordinates where each of the obstacles on the course are 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 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.

visualControl.py (guiControl.py)
This 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 module is responsible for simulating readings from an IR sensor. TBD

sensorControl.py
This module controls both the simulated sensors or the real sensors. It control 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 a diverted route to be established. TBD
and to allow the robot position to be updated. TBD

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

Lib/navigation.py
This 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.

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 a camera, sonar, 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, arena coordinates.
(X and Y are Cartesian coordinates of the Waypoint

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

a. A turn on the spot
b. A straight line
c. 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'.
.
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 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.

Unfortunately, if either wheel skids during a longitudinal or lateral movement, the odometers will be in error and hence the open loop distance and heading will be in error and the robot will diverge from its planned route. The actual robot position and heading in the arena will be different from the position and heading the autonomous system has calculated.

The Open loop navigation system can be tuned by adjusting specific constants such as wheel diameter to make it reasonably accurate over a short distance, but even small errors caused by skidding can build up over a number of seconds and cause the robot to stray from its intended track and hit an obstacle.

Closed Loop Navigation and Control

The navigation loop is closed by using position and heading fixes from other sensors to determine the actual position of the robot in the arena, or relative to a known obstacle. This allows it to determine its 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.

4.6 Autonomous Software

The Autonomous software programme works by connecting lots of different program files together and then sending information between one another to control the Rampaging Chariot. What happens to the Chariot is then displayed on the GUI screen. The program files themselves are connected together through the use of Control Loops.

The following is an explanation of what blocks of code are contained within each Control Loop file:

4.6.1. Main.py

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

1. First section is importing the program file that you want to place within Control Loops as well as other parameters you might want to use like math and time.

2. Second section is the calibration values to be passed into the program files that have been imported.
   As well as initialisation of the program files themselves with parameters being passed in. This is where most calibration changes will be made for ease of the user so they don't have to dig through files to find out where a variable needs to be changed.

3. Third section is where the imported program files are entered into their own personal Control Loops, ready to be connected to each other.

4. Fourth section is where the program files within their respective Control Loops are connected together to allow information to be pass between them.

5. Fifth section is where each of the Control Loops are started up on their own threads. Run is then called with makes all previously started threads run the code placed on them. In this case it will run all of the Control Loops that have been started up beforehand.

4.6.2. OdometerControl.py

The OdometerControl file is just like the test Odometer program (that you should have been through by now) with just a few changes. 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. While in this file the Odometer just keeps track of the distance travelled and the heading of the Chariot. It, then passes the values to other program files through the Control Loops to have them printed on the screen or used in other calculations.

Methods called Translators are used to pass values between Control Loops. In a translator you specify where it is going and what message is to be sent. For Odometer it is sending a message to trackControl.

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..