A Virtual Wall Following Robot

Jon-Paul Boyd
School of Computer Science and Informatics
De Montfort University
United Kingdom

Abstract— This report presents an approach to addressing a wall-following problem. A virtual robot equipped with ultrasonic sensors is tasked with autonomously navigating an environment interior while maintaining constant distance to the wall. The solution leverages the extensive functionality of the V-REP virtual robot experimentation platform **[1]** to provide the simulated physical environment layer, including wall and mobile robot objects. The control layer, implemented in Python and interfacing with VREP via API, manages environment perception and navigation. A final evaluation is made, comparing single and multi-sensor approaches. The final optimized multi-sensor solution offered best performance, especially at high speed, completing one loop in 155s with a navigation distance error rate of 420.85cm. Recommendations include more refined sensor calibration and further research into proven control methods.

Index – Mobile Robots, Wall Follower, Control Systems, Simulation, VREP, Python

I. Introduction

The goal of this study is to develop a software architecture to achieve the following tasks:

- Initiate robot wander with random bearing.
- On wall encounter proceed to autonomously navigate one environment traversal in random direction, maintaining constant distance.

It can therefore be said this robot is a specialised wall follower, with no other task, and the only uncertainty to deal with is the wall layout. This paper is organized as follows. Section II details the simulated physical environment, while section III provides details on the robot model used. Sections IV and V cover the control cycle and control architecture respectively. Section VI provides some technical detail on the solution implementation. Section VII presents results of testing, with section VIII closing with final evaluation and thoughts.

II. Simulated Environment

Physical and geometric characteristics of the simulated world, including collidable walls and environment dimensions, are modelled as a scene (Fig. 1) in V-REP. The environment, 15m², is partially structured, with no obstacles other than 80cm high walls, that includes one simple L-shaped corner and a more centrally placed, complex configuration. For scale, each smaller square measures 50cm², with the robot present in the centre of the scene. The environment is static, with no planned path for the robot to follow. Floors are flat and smooth. Power consumption, weather conditions, material surfaces and other real-world considerations are not part of this study.

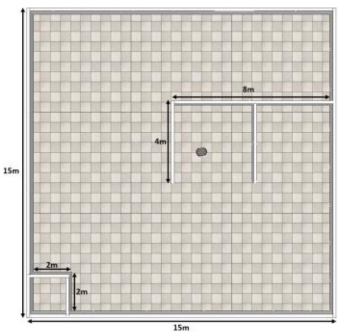

Fig. 1 V-REP scene layout

Fig. 2 Pioneer P3DX robot

III. Pioneer P3DX Robot Model

Modelled in VREP and used here is the well-known Pioneer P3DX robotics research platform [2], in basic configuration without arms or grippers (Fig. 2). Robot body dimensions (Fig. 3) will influence minimum navigable distance between it and walls, therefore needs to be accounted for.

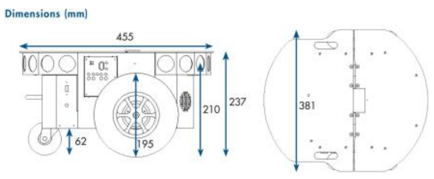

Fig. 3 Pioneer P3DX dimensions

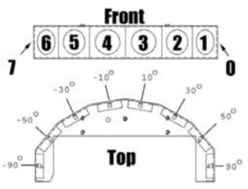

Fig. 4 Sensor array (front)

A. Actuators

Providing mobility are 3 wheels in triangular configuration. The two larger wheels are independently driven by high-speed, high-torque, reversible DC motors, with the 3rd a smaller, passive, stabilizing wheel. This allows for an omnidirectional, extremely manoeuvrable robot able to rotate around its own axis. A positive input value rotates the actuator joint in a forward direction, a negative value for reverse and 0 to stop. Pioneer actuators are revolute joints that accept input values as radians/s. High resolution optical quadrature encoders facilitate odometry on the real model.

B. Sensors

The robot is equipped with 16 ultrasonic sensors, numbered 0-7 and 8-15 for the front and rear arrays respectively (Fig. 4). These active sensors work by emitting sound and detecting return reflection to derive distance information, enabling the robot to measure its proximity to the external environment and thus support navigational positioning relative to walls.

By default, these sensors are a cone type with max range 1m that "*allows for the best and most precise modelling of most proximity sensors*" [3]. Beyond max range a value 4 is returned, otherwise a "*digital*" distance value in metres to a high resolution 17 decimal places. Sensor consistency was tested by graphing proximity to a wall from sensor 3 (front, 10°) for two forward motion runs (Fig. 5), which clearly illustrates the linear and reliable characteristics of the sensor.

Due to its offset position, sensor 3 reads 0.09m when the rover is in physical contact with the wall, also seen in Fig. 5 where proximity flattens off. With an overhanging top plate and side sensors located behind drive wheels, the overall physical design needs to be considered when determining the minimum wall proximity threshold.

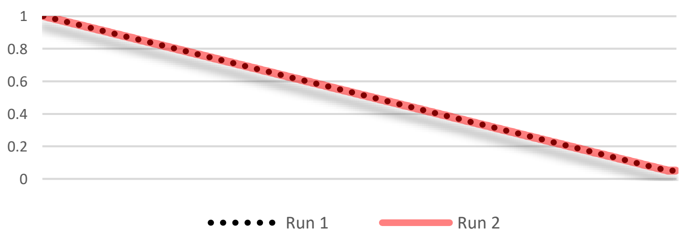

Fig. 5 Ultrasonic Sensor proximity detection distance - 2 run comparison

IV. CONTROL CYCLE

The study problem requires a system capable of executing 4 tasks in sequence: turn random number degrees clockwise (cw) or counter-clockwise (ccw), move a distance in a straight line, stop near first wall encountered, and finally follow the wall. To achieve this, the principles of the "*sense-think-act*" control model have been adopted.

Cycling through a sequence of planned tasks, the robot controller continuously performs the "*sense*" phase to refresh its perception of the environment, updating internal and external state properties. Latest values from the ultrasonic sensor array are transformed into proximity distance to objects. Compass bearing indicates robot heading. Logging the number of turns of the motor revolute joints supports odometry information including distance travelled and average speed. All this is stored in internal state, while external state maintains the robot's absolute position in its world.

With state information updated, the robot system has the best information available on which to "*think*" what next. This may include aborting the current task due to impending collision, comparing actual compass with target bearing or determining if current location is within given distance of target location. Furthermore, internal state is updated with motor velocities calculated to minimise distance to wall error.

The third and final phase in the cycle "*acts*" on internal state, typically updating motor input from values calculated during "*think*", to stop forward locomotion, continuing a rotation or correcting navigational errors to bring the robot closer to minimum wall distance. When a task is complete its status is updated, informing the control cycle to proceed with the next.

V. CONTROL ARCHITECTURE

The robot seeks to execute its 4 tasks; however, the environment is completely unknown beforehand, with no modelled representation of the world or path plan available. A reactive controller will support the *sense-think-act* control cycle, with a set of rules consuming continuously updated state information then directing the robot's behaviour to a given situation.

Having established the planned scenario and creating the simulated robot instance, the system initiates the control cycle. Perception information is first updated before executing the current task (Fig. 6). One example rule commands the robot to stop if wall proximity is below a configured minimum and the current action is not "*wall follow*" (e.g. move), preventing an impending collision.

Fig. 6 shows interaction between the different components, where the first phase of the control cycle (blue) is to update the "*sense*" of the environment (purple), before "*thinking*" on how to "*act*" (red), such as calculating the navigation error and setting motor velocity appropriately.

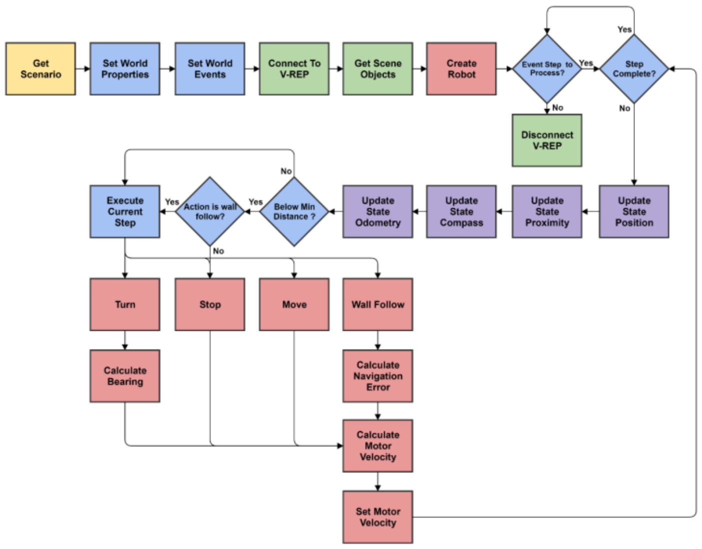

Fig. 6 High-level system diagram

VI. CONTROL IMPLEMENTATION

Replacing an onboard hardware controller managing sensors, motors and the control cycle is a software client architecture implemented in Python, communicating with the V-REP simulation server via an application programming interface (API) (Fig.7).

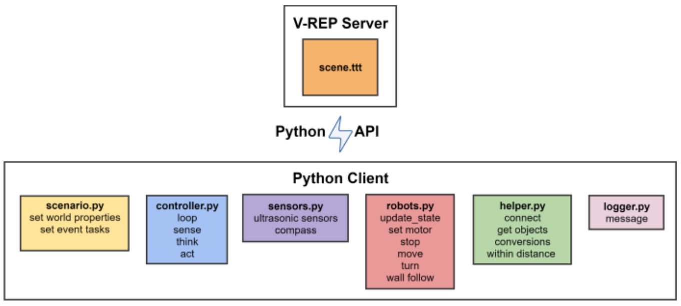

Fig. 7 Implemented software controller architecture

Each feature is associated with a component that encapsulates specific concerns. *Controller.py* manages the "*sense-think-act*" cycle whereas *sensors.py* is responsible for reading the ultrasonic array and *robots.py* executing move and stop directives from the controller.

Note the PioneerP3dx class implementation in *robots.py* provides a datasheet of model specifications, allowing the robot to be aware of characteristics including top speed and wheel circumference (Fig. 8).

```
self.props = {'wheel_rad_m': 0.195,
              'wheel_circ_m': 0.61261,
              'chassis_l_mm': 455,
              'chassis_w_mm': 381,
              'chassis_h_mm': 237,
              'weight_kg': 9,
              'operating_payload_kg': 17,
              'def_ms_v': 0.5,
              'min_speed_ms': 0.12,
              'max_speed_ms': 1.2}
```

Fig. 8 Pioneer P3DX datasheet implementation

Helper.py exposes a collection of generic functions including conversion of metres/s (m/s) to radians/s (rad/s), and calculation of Euclidean distance between points. *Logger.py* outputs status and telemetry data to a log file. Such architecture makes for an easily maintainable, extensible and reusable codebase. The following takes a closer look at component implementation.

A. *Scenario*

A sequence of planned tasks for the wall follow objective are configured as a scenario (Fig. 9) and passed to the controller for execution. A different objective requires minimal change only to the event task list, provided the reactive controller has programmed features to satisfy the task requirements

```
self.world = {'props': {'min_dist_enabled': True, 'min_dist': 0.23, 'max_dist': 0.28},
              'events': [{'task': 'turn', 'degrees': random.randint(-180, 180)},
                         {'task': 'move', 'robot_dir_travel': 1, 'dist': 30, 'velocity': 0.4},
                         {'task': 'wall_follow', 'dir': random.choice([-1, 1]), 'method': 'min', 'coeff': 1.6},
                         {'task': 'stop'}]}
```

Fig. 9 Wall follow scenario configuration

B. *Controller*

The controller first establishes a V-REP server connection then dynamically builds an object tree of the P3DX robot from the V-REP scene hierarchy so that no naming or handles of ultrasonic sensors or motors are hard-coded. The robot model is then created to include instantiation of its modelled sensor array and compass.

The "*sense-think-act*" controller then cycles through planned scenario tasks. Without any sleep timer at the end of each loop, the client code was measured running at ~ 6669Hz. In contrast, the V-REP main server thread runs much slower, between 200Hz and 67Hz. Therefore, in order to prevent unnecessary flooding of the V-REP buffers through excessively high volume data streams, a sleep timer of 0.004ms reduced the client loop to a frequency of ~208Hz.

1) *Sense Phase*

The robot control mechanism and ability to navigate accurately is very closely linked and guided by sensor feed. Therefore, for every control cycle, state is updated to persist latest sensor readings and odometry calculations (Fig. 10). This provides the robot with a latest awareness of its surrounding environment and absolute location within it, on which decisions can be made.

```
self.robot.update_state_position()
self.robot.update_state_proximity(self.world_props)
self.robot.update_state_compass()
self.robot.update_state_odometry()
```

Fig. 10 Sense phase tasks updating state

Updating state proximity sets Boolean flags to true if the front or rear ultrasonic sensor arrays indicate proximity to collidable objects below a scenario defined minimum. This facilitates the decision to stop during any plan task other than "*wall follow*". The implementation of a virtual magnetic compass transforms robot orientation to a bearing heading and used in the "*turn*" task (Fig. 11).

```
res, bearing = vrep.simxGetObjectOrientation(self.client_id, handle, -1, mode)
self.last_read = round(360 * (math.pi + bearing[2]) / (2 * math.pi), 2)
```

Fig. 11 Transforming object orientation to simulated magnetic bearing

The real P3DX model is equipped with quadrature optical encoders to provide odometry, where out-of-phase channels providing pulses per revolution (ppr) enable sensing of motor direction and number of joint turns made. These encoders have not been modelled within V-REP, however odometry is still possible by aggregating the difference in current angle position of a motor joint compared to previous position. With this, total turns can be derived, and when multiplied by wheel circumference gives distance travelled (Fig. 12).

```
def update_joint_pos(self, handle):
    res, current_pos = vrep.simxGetJointPosition(self.h.client_id, handle, vrep.simx_opmode_streaming)
    difference_pos = current_pos - self.state['int']['jpos'][str(self.state['int']['motors'][handle] + '_prev')]

    self.state['int']['jpos'][str(self.state['int']['motors'][handle] + '_prev')] = current_pos

    difference_pos = self.get_pos_ang(difference_pos)
    self.state['int']['jpos'][str(self.state['int']['motors'][handle] + '_total')] += difference_pos

    self.state['int']['jpos'][str(self.state['int']['motors'][handle] + '_turns')] = \
    self.state['int']['jpos'][str(self.state['int']['motors'][handle] + '_total')] / (math.pi * 2)

    self.state['int']['jpos'][str(self.state['int']['motors'][handle] + '_dist')] = \
    self.state['int']['jpos'][str(self.state['int']['motors'][handle] + '_turns')] * self.props['wheel_circ_m']


def get_pos_ang(pos):
    if pos >= 0:
        pos = math.fmod(pos + math.pi, 2 * math.pi) - math.pi
    else:
        pos = math.fmod(pos - math.pi, 2 * math.pi) + math.pi
    return pos
```

Fig. 12 Aggregating joint angle position for odometry

2) *Think*

With updated sensor information available, the controller is supported in making informed decisions during each cycle, for example preventing wall collision with a stop directive (Fig. 13). The "*thinking*" phase also includes calculation of motor joint velocities to correct navigational errors during the "*wall follow*" task, covered later.

```
if self.robot.is_less_min_prox_dir_travel() and step['task'] not in stop_exceptions:
    lg.message(logging.INFO, 'Stop task triggered')
    self.robot.stop(step_status, self.world_props, task_args)
    continue
```

Fig. 13 Control cycle rule for executing stop directive

3) *Act*

This phase executes possible robot tasks: "*turn*", "*move*", "*stop*" and "*wall follow*". A class method within the PioneerP3dx class is named after and implemented for each task and can therefore be called dynamically with command *getattr* (Fig. 14). It will be executed in each cycle loop, allowing sensor information to feed into motor velocity calculations to correct navigational errors. Only when the task is determined complete will the cycle proceed to the next planned task.

```
if step_status['complete'] is None:
   # Execute current route plan step
   task_args = {arg: step[arg] for arg in step if arg not in 'task'}
   lg.message(logging.DEBUG, 'Executing method ' + str(step['task']) + ' args ' + str(task_args))
   getattr(self.robot, step['task'])(step_status, self.world_props, task_args)
   continue

getattr(self.robot, step['task'])(step_status, self.world_props, task_args)
time.sleep(0.004)
```

Fig. 14 Control cycle task execution

C. *Tasks in detail*

1) *Turn*

The first task rotates the robot a random number of degrees in the range -180 to +180 from initial orientation. Target bearing is calculated by adding random degrees to current heading, then subtracting 360 if greater than 360, or adding 360 if less than 0. A "*think*" rule determines whether the turn to achieve the heading goal is a clockwise or counter-clockwise rotation and applies a positive and negative velocity input to the left and right motors as appropriate.

Testing determined a slow 0.1m/s velocity achieved best accuracy, preventing overshoot of target, with the goal considered successful when rotation is within 0.5 degrees of calculated target. Note that as the author finds m/s easier to visualise, all velocity speeds are given in m/s. Only within method *set_motor_v* are velocity values converted to rad/s which V-REP expects; a division of m/s by wheel radius as specified in robot datasheet *self.props* (Fig. 8).

2) *Move*

The Pioneer employs differential steering where degree of turning is determined by the difference in input values applied to one motor over the other. Straight line motion over an even surface is achieved with the same values for both motors. This task applies default m/s velocity from datasheet *self.props* (Fig. 8) and continues for the task time specified or close wall proximity triggers a stop action. Every requested motor velocity, regardless of calculation source, is checked against the datasheet top speed and capped should it exceed specification (Fig. 15). This safety measure helps to prevent runaway and unrealistic motor torques experienced during initial testing.

```
def set_motor_v(self):
    if self.state['int']['motor_l_v'] > self.props['max_speed_ms']:  # Cap velocity to top speed per data sheet
       self.state['int']['motor_l_v'] = self.props['max_speed_ms']

    if self.state['int']['motor_r_v'] > self.props['max_speed_ms']:  # Cap velocity to top speed per data sheet
       self.state['int']['motor_r_v'] = self.props['max_speed_ms']

    self.state['int']['motor_l_v'] = self.h.ms_to_radss(self.state['int']['motor_l_v'], self.props['wheel_rad_m'])
    self.state['int']['motor_r_v'] = self.h.ms_to_radss(self.state['int']['motor_r_v'], self.props['wheel_rad_m'])

    t = time.time()
    self.state['int']['motor_all_v'].append((t, self.state['int']['motor_l_v'], self.state['int']['motor_r_v']))

    for m in self.state['int']['motors']:
        if 'left' in self.state['int']['motors'][m]:
           self.set_joint_v(m, self.state['int']['motor_l_v'])
        else:
           self.set_joint_v(m, self.state['int']['motor_r_v'])
```

Fig. 15 Capping, conversion and setting of motor velocity

3) *Stop*

The "*stop*" task simply applies 0 to both motors, bringing any locomotion to a halt. It is executed as a planned task or triggered in close proximity of a wall during a "*move*" task.

4) *Wall Follow*

The aim is to follow the wall once found, maintaining best possible constant minimum distance. This was set at 23cm, approximately half the length of the P3DX and accounting for the main drive wheels being slightly forward of centre. The problem statement does not specify a minimum speed, therefore various are tested to measure the trade-off versus accuracy. Once the task begins starting position is logged and the task considered complete after 1 full loop. The biggest technical test of this study is how best to interpret sensor feedback in regulating speed control of a differential motor so that navigation error is minimised and path is smooth, all while maintaining safe wall distance. There are 3 types of corner to negotiate; 90° inside, 90° outside and 180° outside, and each will present their own challenge.

2 approaches were tested, both using only the front sensor array. In the first (Fig. 16), a baseline velocity is calculated as the difference between the 23cm threshold and minimum proximity distance from any of the 8 sensors, multiplied by a gain coefficient. The square root of the difference is also calculated. When below minimum distance the square root difference is added to baseline and applied to near-side motor, and subtracted from baseline then applied to off-side, in effect steering the robot away from the wall. If above minimum distance the off-side motor is biased with the greater velocity, correcting the navigational error and steering the robot back towards the wall. Continuous calculation of proximity distance and squared difference error as functions of motor velocity adjustment allow for smoother navigation correction.

```
def wall_follow_min(self, min_dist, max_dist, coeff, motor_index):
    distance = self.prox_dist_dir_travel()

    diff = 0
    if distance < min_dist:
       diff = min_dist - distance
    else:
       diff = distance - min_dist

    self.state['int']['err_corr_count'] += diff

    baseline_v = distance * coeff
    diff_map = [[math.sqrt(diff), math.sqrt(diff) * -1], [math.sqrt(diff) * -1, math.sqrt(diff)]]

    if distance < min_dist:
       self.state['int']['motor_l_v'] = baseline_v + diff_map[motor_index][0]
       self.state['int']['motor_r_v'] = baseline_v + diff_map[motor_index][1]
    else:
       self.state['int']['motor_l_v'] = baseline_v + diff_map[motor_index][1]
       self.state['int']['motor_r_v'] = baseline_v + diff_map[motor_index][0]
```
Fig. 16 Minimum distance wall follow approach

The second approach is similar in deriving a baseline velocity from which the squared difference is added or subtracted to bias one motor. However, in this design the full front array of 8 sensors are used with their read values weighted according to the calculations given in Table I, which also shows simulated parallel wall and turn sensor values for illustration. Fusing multiple weighted readings hopes to achieve greater, non-linear navigation by both accounting for multiple close proximity distances within a corner thereby smoothing turns, but also increasing speed when movement is parallel to walls and the path ahead is clear. Note sensors with objects beyond their maximum range return a value 4 and therefore capped at 1.

TABLE I
WEIGHTED SENSOR VELOCITY CALCULATION EXAMPLES

Sensor	Calculation	Weight	Wall	Turn
0	0.75 / 180.0 * π	0.01309	0.23	0.19
1	0.55 / 180.0 * π	0.009599	0.3	0.22
2	0.5 / 180.0 * π	0.01309	0.5	0.24
3	0.25 / 180.0 * π	0.004363	1	0.5
4	0.25 / 180.0 * π	0.004363	1	1
5	0.5 / 180.0 * π	0.01309	1	1
6	0.55 / 180.0 * π	0.009599	1	1
7	0.75 / 180.0 * π	0.01309	1	1
Baseline Velocity		$\sum$ weight * sensor	0.42706	0.375486

After a 5 second delay allowing for initial progress, a check determines if the current location is within 7cm of the starting point, indicating a full wall traversal is complete and task goal met (Fig. 17).

```
def within_dist(self, point_a, point_b, dist_threshold=0.07):
    distance = self.get_euclidean_distance(point_a, point_b)
    lg.message(logging.DEBUG, 'Distance between point A and B is ' + str(distance))
    if distance <= dist_threshold:
       return True
    else:
       return False

def get_euclidean_distance(point_a, point_b):
    return math.sqrt((point_a[0] - point_b[0]) ** 2 + (point_a[1] - point_b[1]) ** 2)
```
Fig. 17 Euclidean distance start to current point checking wall follow done

VII. RESULTS

Initial experimentation using sensor information as the feedback loop into calculation of velocities controlling a differential motor exhibited undesirable effects. Large amplitude oscillations were caused by excessive reaction to navigation error and directly related to sensor weighting, motor bias algorithms and gains used. Similarly, out of control spinning progressively away from the wall was due to error overcompensation. Insufficient differentiation of left and right torque often resulted in the robot getting stuck head-on against a wall or inside corner. In the multi-sensor approach, poor weighting and wheel differentiation resulted in the robot unable to pass inner wall ends as sensors "compete" with each other. All these problems were resolved.

For the second wall follow approach, weights for each sensor were estimated through analysis of parallel wall, approaching wall and turn scenario sensor inputs, and then optimised through many iterations. Indeed this calibration process was exhaustive, often resulting in regression of navigational behaviour, and highlighted the difficulty in tuning a fine, successful balance between several sensorial inputs.

It is clear from Table I that outer sensors 0 and 7 have the largest weighting across the array, with assigned weights decreasing as placement moves towards centre of the robot. This is a deliberate damping mechanism, as higher weights proved the cause of large osciallations as the robot turned and sensor values climbed as the path ahead cleared. Testing higher weights assigned to central sensors 3 and 4, hoping to "encourage" faster straight-line speed given a clear path, again proved contributary to unwanted oscillations. Likewise, many difference error calculation and gain variants were tested to find acceptable balance between speed and successful navigation. Applying an easily configurable gain proved very useful in fine tuning.

Each wall follow approach was tested with 2 different gain values; the first to complete a wall traverse loop in ~190s categorised "*slow*", and the second in ~160s categorised "*fast*". All tests ran clockwise from the same central scene position and bearing. Metrics including distance travelled (m), average speed (m/s) and navigational distance difference error (cm) were captured. Table II presents the results.

TABLE II
MIN AND MULTIPLE SENSOR TEST RESULTS

Method	**Gain**	**Time Taken (s)**	**Average Speed (m/s)**	**Distance (m)**	**Navigation Error (cm)**
min (slow)	1.3	198	0.45	88.54	447.3
multiple (slow)	6.1	187.93	0.47	88.54	400.21
min (fast)	1.6	164.25	0.56	91.49	889.38
Multiple (fast)	7.5	155	0.57	89.07	420.85

Reviewing the "*slow*" category first, the weighted multiple sensor approach navigates the wall 10s faster with 47cm less distance error over the same 88.54m travelled. However, as illustrated by very similar traces (Figs. 18, 19) it is difficult to distinguish between the two tests visually and pinpoint a specific scene hotspot that challenges "*min*". It could be attributed to small error aggregation over the full scene.

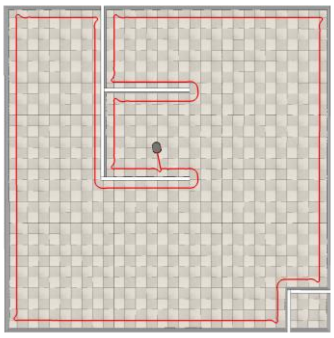
Fig. 18 Min (Slow) wall follow trace

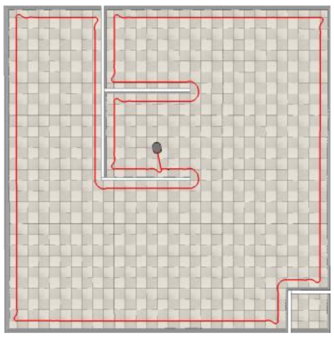
Fig. 19 Multi (slow) wall follow trace

The "*fast*" category again shows the multiple sensor approach the quickest. More significantly, it outperforms single sensor in efficiency with only a minor error increase, in contrast to almost double the error rate with single. In this test the traces (Figs. 20, 21) give a clearer picture. While navigation accuracy of inner corners is similar, the single sensor approach swings wide of both 90° and 180° outer, exposed turns. This is judged a problem with motor control dependant only on 1 minimum sensor reading, which at these scene locations will be sensors 0 and 7 placed laterally. Given the speed increase, the robot is already well beyond the wall before it reacts insufficiently. The multi sensor approach benefits from forward ranging sensors detecting wall ends earlier in the trajectory and adjusting motor bias accordingly.

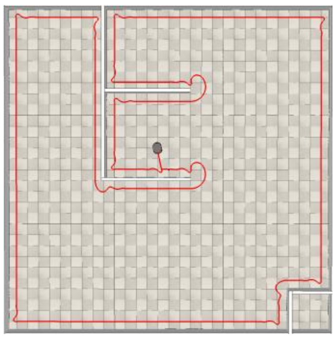
Fig. 20 Min (Fast) wall follow trace

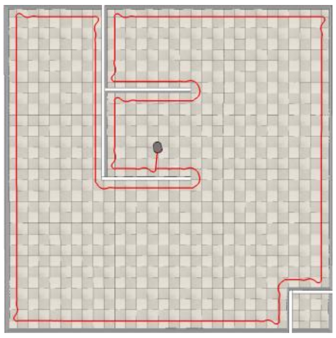
Fig. 21Multi (fast) wall follow trace

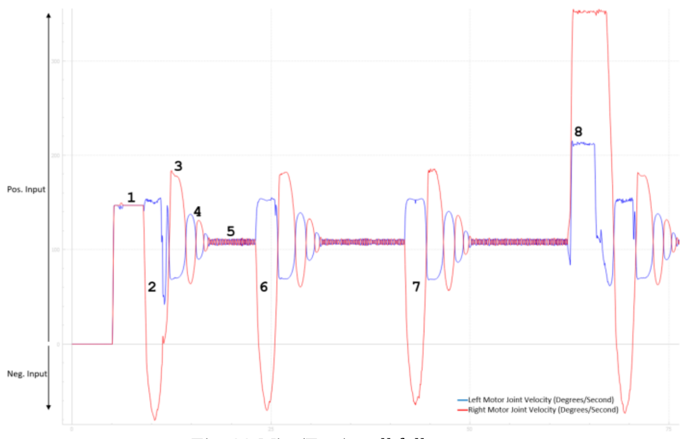

Fig. 22 Min (Fast) wall follow trace

Analysing the graphed left and right motor joint velocity (Fig. 22) can help understand the observed behaviour during the min (fast) test. The robot begins its straight-line wander (1), and first encounters the wall at (2), where it turns with left joint positive and right joint negative input to initiate a clockwise turn. Navigation corrections are made to reduce error (3, 4) before straight-lining (5). The two central, inside corners are navigated (6, 7) with clockwise turns in consistent fashion, with larger negative right joint input pulling the robot back out to minimum distance, as also visualised in the path trace (Fig. 20).

The robot is now straight-lining towards the first outer, 180° turn. In contrast to (2, 6, 7) which exhibit motor joints turning in opposing directions, (8) shows both joints turning in the same direction, with the outside right wheel at approximately double the velocity of the inside wheel for some time which explains the smooth sweeping arc characteristic (Fig. 20). It then appears a sensor picks up the wall to decelerate the inside, left joint velocity moments before the right joint and bring the robot back to the wall where familiar navigation correction patterns are again observed.

For information the experimentation setup is given in Appendix A. Solution source code is given in Appendices B-G.

VIII. Conclusion

Tasked with the problem of implementing a wall following virtual robot solution, this study demonstrated that even a basic single sensor approach proved effective in performing the main task at slow speeds, with error rates (447.3cm) comparable to the more sophisticated multi-sensor alternative presented (400.21cm). However, as locomotive speed is increased, performance of the single sensor is significantly impacted (889.38cm), proving a single sensor solution inadequate whereas multi sensor is more effective and robust (420.85cm). Without doubt there is a trade-off between speed vs accuracy. The availability of more sensors is not significantly better at slower speeds, but if application use-case highly relies on minimal error rate, especially at higher speeds, the multi-sensor approach offers best performance and validates effort in challenging sensor calibration.

There remain many opportunities for improvement. The illustrated sweeping arc problem could be resolved with further calibration of the single sensor approach to prevent high velocity rotation in same direction of both motor joints through tight turns. Explicitly distinguishing parallel walls from corners and handling the required navigation behaviour separately may lead to improved speed and reduced error. A larger study with extensive research would reveal documented alternative and proven approaches including genetic and fuzzy algorithms.

Further work might include more diverse environments that include curved walls for example, testing the expectation the multi sensor approach would again offer best performance. In summary, the V-REP simulation software in combination with the Python API makes for a powerful experimentation platform, exposing an extensive collection of readily available machine and deep-learning frameworks for solving more complex robotics problems. There is an appreciation of differences between simulation and real-world that need to be managed. For example V-REP provides sensor readings to 17 decimal places whereas high performance real world models offer mm accuracy at best (Chirp Microsystems CH-101). Another difference is the presence of quadrature encores on the real model, and no default velocity limits on the virtually modelled Pioneer. However V-REP is a far more accessible platform on which to experiment. The author believes the software control architecture implemented here provides a great base for further study.

References

[1] VREP, “V-REP Virtual Robot Experimentation Platform Home Page,” [Online]. Available: http://www.coppeliarobotics.com/.

[2] P. P3DX, “Pioneer P3DX Rev A Data Sheet,” Adept Mobile Robots, [Online]. Available: https://www.generationrobots.com/media/Pioneer3DX-P3DX-RevA.pdf.

[3] http://www.coppeliarobotics.com, “Proximity sensor types and mode of operation,” [Online]. Available: http://www.coppeliarobotics.com/helpFiles/en/proximitySensorDescription.htm.

[4] M. R. A. Robotics, “Pioneer Manual,” [Online]. Available: https://www.inf.ufrgs.br/~prestes/Courses/Robotics/manual_pioneer.pdf.

APPENDIX A – EXPERIMENTATION SETUP

Software Windows 10 Professional (v18.3), V-REP Pro Edu (v3.6.2), PyCharm Professional 2018.3

Hardware Intel i7-5820K, Nvidia GeForce GTX 1080, 16GB RAM

Main Python Modules vrep (2.2.4)

APPENDIX B – SCENARIO.PY

```
import os
import sys
import logging
import logger as lg
from datetime import datetime
import random
from controller import Controller

script_name = os.path.basename(sys.argv[0]).split('.')


class Scenario:
    """
    Scenario objectives: Task 1 - Turn random degrees
                         Task 2 - Move in straight line until within distance of wall
                         Task 3 - Follow perimeter wall for 1 loop
                         Task 4 - Stop
    """
    def __init__(self):
        lg.message(logging.INFO, 'Staring scenario ' + str(script_name[0]))

        # Properties
        # min_dist - stop robot at minimum distance from collidable object - UoM is metres
        # max_dist - max distance from object - UoM is metres
        # velocity - locomotive speed in metres per second
        # robot_dir_travel - 1=Forward, -1=Reverse
        # dir - 1=clockwise, -1=counter clockwise
        # method - algo controlling navigation & motor velocity - min=takes min from 1 sensor, multi=multiple sensors
        #        - multi (fast) - coeff = 7.5 (162s)
        #        - multi (slow) - coeff = 6.1 (206s)
        #        - min   (fast) - coeef = 1.6 (164s)
        #        - min   (slow) - coeef = 1.3 (207s)

        self.world = {'props': {'min_dist_enabled': True, 'min_dist': 0.23, 'max_dist': 0.28},
                      'events': [{'task': 'turn', 'degrees': random.randint(-180, 180)},
                                 {'task': 'move', 'robot_dir_travel': 1, 'dist': 30, 'velocity': 0.4},
                                 {'task': 'wall_follow', 'dir': random.choice([-1, 1]), 'method': 'min', 'coeff': 1.6},
                                 {'task': 'stop'}]}

        lg.message(logging.INFO, 'World props ' + str(self.world['props']))
        lg.message(logging.INFO, 'World events ' + str(self.world['events']))

        Controller(self.world)

        lg.message(logging.INFO, 'Scenario ' + str(script_name[0]) + ' complete')


if __name__ == "__main__":
    log_filename = str(script_name[0] +('_') + datetime.now().strftime("%Y%m%d-%H%M%S") + '.txt')

    logging.basicConfig(filename='logs/' + log_filename, level=logging.DEBUG,
                        format='%(asctime)s - %(levelname)s - %(message)s')

    scenario = Scenario()
```

```
import sys
import time
from helper import Helper
from robots import PioneerP3dx
import logging
import logger as lg
import csv


class Controller:
    """
    Controller managing connection to V-REP simulator, executing the physics loop etc
    """
    def __init__(self, world):
        lg.message(logging.INFO, 'Initialising ' + self.__class__.__name__)
        self.world_props = world['props']
        self.world_events = world['events']
        self.start_time = 0

        # Instantiate helper and connect to VREP
        self.h = Helper()
        if not self.h.connected:
            sys.exit()

        # Get all objects from running VREP scene
        self.full_scene_object_list, self.simple_object_list = self.h.get_objects()

        # Instantiate robot
        self.robot = PioneerP3dx(self.h, self.simple_object_list)

        self.loop()

        self.stats()

        # Disconnect from VREP
        self.h.disconnect_client()

    def loop(self):
        """
        Sequentially process planned events in a sense>think>act control cycle
        """
        lg.message(logging.INFO, 'Starting controller loop')

        self.start_time = time.time()

        stop_exceptions = ['wall_follow']
        step_status = {'start_t': time.time(),
                       'complete': None}

        for step in self.world_events:
            lg.message(logging.INFO, 'Starting event - ' + step['task'])

            task_args = None

            step_status['start_t'] = time.time()  # Log start, as some actions have lifetime defined in route plan
            step_status['complete'] = None  # Step tristate - not started (None), in progress (False), complete (True)
            while not step_status['complete']:
                # Sense
                self.sense()

                # Think
                # Possible impending collision - trigger stop unless excepted actions
                if self.robot.is_less_min_prox_dir_travel() and step['task'] not in stop_exceptions:
                    lg.message(logging.INFO, 'Stop task triggered')
                    self.robot.stop(step_status, self.world_props, task_args)
                    continue

                # Act
                if step_status['complete'] is None:
                    # Execute current route plan step
                    task_args = {arg: step[arg] for arg in step if arg not in 'task'}
                    lg.message(logging.DEBUG, 'Executing method ' + str(step['task']) + ' args ' + str(task_args))
                    getattr(self.robot, step['task'])(step_status, self.world_props, task_args)
                    continue

                getattr(self.robot, step['task'])(step_status, self.world_props, task_args)

                time.sleep(0.004)

    def sense(self):
        """
        Update state on how robot perceives world
        """
        self.robot.update_state_position()
        self.robot.update_state_proximity(self.world_props)
        self.robot.update_state_compass()
        self.robot.update_state_odometry()

    def stats(self):
        """
        Show simulation statistics
        """
        time_taken = round(time.time() - self.start_time, 2)

        avg_joint_dist = 0
        for m in self.robot.state['int']['motors']:
            avg_joint_dist += self.robot.state['int']['jpos'][str(self.robot.state['int']['motors'][m] + '_dist')]

        avg_joint_dist = round(avg_joint_dist / 2, 2)
        avg_speed_ms = round(avg_joint_dist / time_taken, 2)
        lg.message(logging.INFO, 'Distance travelled - {}m'.format(avg_joint_dist))
        lg.message(logging.INFO, 'Average speed - {}m/s'.format(avg_speed_ms))
        lg.message(logging.INFO, 'Nav dist diff error - {}cm'.format(round(self.robot.state['int']['err_corr_count'], 2)))
        lg.message(logging.INFO, 'Controller loop complete - time taken - {}s'.format(round(time.time() -
                                                                                            self.start_time, 2)))

        with open('output/abs_pos_all.csv', 'w', newline='') as out:
            csv_output = csv.writer(out)
            csv_output.writerows(self.robot.state['ext']['abs_pos_all'])

        with open('output/motor_all_v', 'w', newline='') as out:
            csv_output = csv.writer(out)
            csv_output.writerows(self.robot.state['int']['motor_all_v'])
```

```
import logging
import logger as lg
import vrep
import math
import time
from sensors import ProximitySensor, Compass


class Robot:
    """
    Super class implementing robot
    """
    def __init__(self, name, simple_object_list, helper):
        self.h = helper

        # Robot objects
        self.name = name
        self.handle = self.h.get_object_handle_by_name(self.name)
        self.component_tree = self.set_object_tree()

        # Robot facing direction 1=forward, -1=back
        self.robot_dir_travel = 1

    def set_object_tree(self):
        """
        Get object tree for robot
        """
        self.h.get_object_tree(self.handle, 0)
        return self.h.object_tree

    def get_components(self, tag, simple_object_list):
        """
        Get component tree for vrep object
        """
        lg.message(logging.DEBUG, 'Getting V-REP components for ' + tag)
        components = {}
        for comp in self.component_tree:
            for child in self.component_tree[comp].children:
                if tag in simple_object_list[child]:
                    components[child] = simple_object_list[child]
        return components

    def set_joint_v(self, joint, velocity):
        """
        Set joint velocity
        """
        vrep.simxSetJointTargetVelocity(self.h.client_id, joint, velocity, vrep.simx_opmode_streaming)


class PioneerP3dx(Robot):
    """
    Pioneer P3DX robot implementation
    """
    def __init__(self, helper, simple_object_list):
        Robot.__init__(self, 'Pioneer_p3dx', simple_object_list, helper)
        lg.message(logging.INFO, 'Initialising ' + self.__class__.__name__)

        # Pioneer specific properties
        self.props = {'wheel_rad_m': 0.195,
                      'wheel_circ_m': 0.61261,
                      'chassis_l_mm': 455,
                'chassis_w_mm': 381,
                'chassis_h_mm': 237,
                'weight_kg': 9,
                'operating_payload_kg': 17,
                'def_ms_v': 0.5,
                'min_speed_ms': 0.12,
                'max_speed_ms': 1.2,
                'sensor_us_weights': [0.75 / 180.0 * math.pi, 0.55 / 180.0 * math.pi, 0.5 / 180.0 * math.pi,
                                      0.25 / 180.0 * math.pi, 0.25 / 180.0 * math.pi, 0.5 / 180.0 * math.pi,
                                      0.55 / 180.0 * math.pi, 0.75 / 180.0 * math.pi]}

        # Pioneer specific internal and external state
        self.state = {'int': {'motors': [],
                              'motor_l_v': 0.0,
                              'motor_r_v': 0.0,
                              'motor_all_v': [],
                              'jpos': {},
                              'prox_s_arr': [],
                              'prox_s': None,
                              'prox_is_less_min_dist_f': False,
                              'prox_is_less_min_dist_r': False,
                              'prox_min_dist_f': 0,
                              'prox_min_dist_r': 0,
                              'compass': None,
                              'err_corr_count': 0},
                      'ext': {'abs_pos_s': None,
                              'abs_pos_n': None,
                              'abs_pos_all': []}}

        # Initialise internal state for robot components
        self.state['int']['motors'] = self.get_components('Motor', simple_object_list)
        self.state['int']['prox_s_arr'] = self.get_components('ultrasonic', simple_object_list)
        self.state['int']['prox_s'] = ProximitySensor(helper.client_id, sensor_array=self.state['int']['prox_s_arr'])
        self.state['int']['compass'] = Compass(helper.client_id, handle=self.handle)
        self.set_joint_pos()

    def set_state_proximity(self, min_distance):
        """
        Boolean state indicating whether robot within specified minimum distance as detected front or rear array
        """
        self.state['int']['prox_min_dist_f'] = min(self.state['int']['prox_s'].last_read[0:8])
        self.state['int']['prox_min_dist_r'] = min(self.state['int']['prox_s'].last_read[8:16])

        if self.state['int']['prox_min_dist_f'] < min_distance:
            self.state['int']['prox_is_less_min_dist_f'] = True
        else:
            self.state['int']['prox_is_less_min_dist_f'] = False

        if self.state['int']['prox_min_dist_r'] < min_distance:
            self.state['int']['prox_is_less_min_dist_r'] = True
        else:
            self.state['int']['prox_is_less_min_dist_r'] = False

    def is_less_min_prox_dir_travel(self):
        """
        Returns boolean state indicating if robot proximity less than minimum distance for direction of travel
        """
        if self.robot_dir_travel == 1:
            return self.state['int']['prox_is_less_min_dist_f']
        else:
            return self.state['int']['prox_is_less_min_dist_r']

    def prox_dist_dir_travel(self):
        """
        Returns minimum distance between robot and collidable object for direction of travel
        """
        if self.robot_dir_travel:
            return self.state['int']['prox_min_dist_f']
        else:
            return self.state['int']['prox_min_dist_r']

    def update_state_proximity(self, props):
        """
        Read sensor array and update associated proximity states
        """
        self.state['int']['prox_s'].read(vrep.simx_opmode_buffer, sensor_array=self.state['int']['prox_s_arr'])
        if 'min_dist_enabled' in props:
            if props['min_dist_enabled']:
                self.set_state_proximity(props['min_dist'])

    def update_state_compass(self):
        """
        Get current object bearing and update state
        """
        self.state['int']['compass'].read(vrep.simx_opmode_buffer, handle=self.handle)

    def set_motor_v(self):
        """
        Set motor velocity for each motor joint after updating state with conversion m/s to rad/s
        """
        if self.state['int']['motor_l_v'] > self.props['max_speed_ms']:  # Cap velocity to top speed per data sheet
            self.state['int']['motor_l_v'] = self.props['max_speed_ms']
            lg.message(logging.WARNING, 'Max speed left motor limited to ' + str(self.props['max_speed_ms']))

        if self.state['int']['motor_r_v'] > self.props['max_speed_ms']:  # Cap velocity to top speed per data sheet
            self.state['int']['motor_r_v'] = self.props['max_speed_ms']
            lg.message(logging.WARNING, 'Max speed right motor limited to ' + str(self.props['max_speed_ms']))

        lg.message(logging.DEBUG, 'Setting motor left velocity (m/s) - ' + str(self.state['int']['motor_l_v']))
        lg.message(logging.DEBUG, 'Setting motor right velocity (m/s) - ' + str(self.state['int']['motor_r_v']))

        self.state['int']['motor_l_v'] = self.h.ms_to_radss(self.state['int']['motor_l_v'], self.props['wheel_rad_m'])
        self.state['int']['motor_r_v'] = self.h.ms_to_radss(self.state['int']['motor_r_v'], self.props['wheel_rad_m'])

        t = time.time()
        self.state['int']['motor_all_v'].append((t, self.state['int']['motor_l_v'], self.state['int']['motor_r_v']))

        for m in self.state['int']['motors']:
            if 'left' in self.state['int']['motors'][m]:
                self.set_joint_v(m, self.state['int']['motor_l_v'])
            else:
                self.set_joint_v(m, self.state['int']['motor_r_v'])

        lg.message(logging.DEBUG, 'Motor left velocity now set at (rad/s) - ' + str(self.state['int']['motor_l_v']))
        lg.message(logging.DEBUG, 'Motor right velocity now set at (rad/s) - ' + str(self.state['int']['motor_r_v']))

    def set_state_pos_start(self):
        """
        Set starting position as external, absolute position
        """
        res, self.state['ext']['abs_pos_s'] = vrep.simxGetObjectPosition(self.h.client_id, self.handle, -1,
                                                                          vrep.simx_opmode_buffer)
        lg.message(logging.DEBUG, 'Start point set ' + str(self.state['ext']['abs_pos_s']))

    def update_state_position(self):
        """
        Set current (now) position as external, absolute position
        """
        res, self.state['ext']['abs_pos_n'] = vrep.simxGetObjectPosition(self.h.client_id, self.handle, -1,
                                                                          vrep.simx_opmode_streaming)
        self.state['ext']['abs_pos_all'].append(self.state['ext']['abs_pos_n'][0:2])

    def stop(self, step_status, world_props, args):
        """
        Stop robot locomotion
        """
        self.state['int']['motor_l_v'] = 0
        self.state['int']['motor_r_v'] = 0
        self.set_motor_v()
        step_status['complete'] = True

    def move(self, step_status, world_props, args):
        """
        Move robot as locomotion task
        """
        if 'robot_dir_travel' in args:
            self.robot_dir_travel = args['robot_dir_travel']

        if step_status['complete'] is None:
            velocity = self.props['def_ms_v']  # Set default velocity

            # Direction travel - forward = 1, reverse = -1
            self.state['int']['motor_l_v'] = velocity * self.robot_dir_travel
            self.state['int']['motor_r_v'] = velocity * self.robot_dir_travel
            self.set_motor_v()
            step_status['complete'] = False

        # Velocity as a "dist" is applied for specified time. Continue moving for "dist" or stop if proximity detected
        if 'dist' in args:
            if (time.time() - step_status['start_t'] > args['dist']) or self.is_less_min_prox_dir_travel():
                step_status['complete'] = True
                lg.message(logging.INFO, 'Move event complete')

    def turn(self, step_status, world_props, args):
        """
        Turn robot task - calculate new bearing relative to current orientation and turn cw or ccw at constant speed
        """
        if step_status['complete'] is None:
            if 'degrees' in args:

                #
                self.state['int']['compass'].set_to_bearing(args['degrees'])
                lg.message(logging.DEBUG, 'Turn bearing from {} to {}'.format(self.state['int']['compass'].last_read,
                                                                               self.state['int']['compass'].to_bearing))

                # Turn cw or ccw at slow speed to prevent overshooting
                if args['degrees'] > 0:
                    self.state['int']['motor_l_v'] = -0.1
                    self.state['int']['motor_r_v'] = 0.1
                else:
                    self.state['int']['motor_l_v'] = 0.1
                    self.state['int']['motor_r_v'] = -0.1

                self.set_motor_v()
            step_status['complete'] = False

        # Subtract max from min between last compass state and target bearing. Turn threshold is 0.5 degrees
        radius_threshold = 0.5
        diff = max(self.state['int']['compass'].last_read, self.state['int']['compass'].to_bearing) - \
                min(self.state['int']['compass'].last_read, self.state['int']['compass'].to_bearing)
        if diff < radius_threshold:
            step_status['complete'] = True
            lg.message(logging.INFO, 'Turn event complete')

    def wall_follow(self, step_status, world_props, args):
        """
        Wall follow task
        """

        # Set default minimum distance
        min_dist = 0.22
        if 'min_dist' in world_props:
            min_dist = world_props['min_dist']

        max_dist = 0.25
        if 'max_dist' in world_props:
            max_dist = world_props['max_dist']

        coeff = 1
        if 'coeff' in args:
            coeff = args['coeff']

        if step_status['complete'] is None:
            self.set_state_pos_start()  # Set absolute position on start of wall follow task
            step_status['complete'] = False

        # After 5 seconds track euclidean distance between start point and current, to check if task completed (1 loop)
        if self.h.within_dist(self.state['ext']['abs_pos_s'], self.state['ext']['abs_pos_n']) and time.time() - \
                step_status['start_t'] > 10:
            step_status['complete'] = True
            lg.message(logging.INFO, 'Wall Follow event complete')

        # Set motor speed mapping index according to rotation (cw, ccw) and direction of travel (forward, reverse)
        if args['dir'] == 1 and self.robot_dir_travel == 1:
            motor_index = 0
        elif args['dir'] == 1 and self.robot_dir_travel == -1:
            motor_index = 1
        elif args['dir'] == -1 and self.robot_dir_travel == 1:
            motor_index = 1
        else:
            motor_index = 0

        # Use wall follow algorithm specified in scenario
        getattr(self, 'wall_follow_' + args['method'])(min_dist, max_dist, coeff, motor_index)

        # Set motor velocity from updated state
        self.set_motor_v()

    def wall_follow_min(self, min_dist, max_dist, coeff, motor_index):
        """
        This wall follow algorithm uses minimum sensor reading to determine left and right
        motor speeds to orientate and navigate the robot
        """
        distance = self.prox_dist_dir_travel()

        diff = 0
        if distance < min_dist:
            diff = min_dist - distance
        else:
            diff = distance - min_dist

        self.state['int']['err_corr_count'] += diff
        baseline_v = distance * coeff
        diff_map = [[math.sqrt(diff), math.sqrt(diff) * -1], [math.sqrt(diff) * -1, math.sqrt(diff)]]

        if distance < min_dist:
            self.state['int']['motor_l_v'] = baseline_v + diff_map[motor_index][0]
            self.state['int']['motor_r_v'] = baseline_v + diff_map[motor_index][1]
        else:
            self.state['int']['motor_l_v'] = baseline_v + diff_map[motor_index][1]
            self.state['int']['motor_r_v'] = baseline_v + diff_map[motor_index][0]

    def wall_follow_multi(self, min_dist, max_dist, coeff, motor_index):
        """
        This wall follow algorithm uses an approach where each sensor is weighted differently, thereby it's value input
        influencing it's contrubution to motor velocity differently
        """
        sensors = [s if s <= 1 else 1 for s in self.state['int']['prox_s'].last_read[0:8]]
        weighted_sensors_f = sum([s * w for s, w in zip(sensors, self.props['sensor_us_weights'][0:8])])

        distance = self.prox_dist_dir_travel()

        diff = 0
        if distance < min_dist:
            diff = min_dist - distance
        else:
            diff = distance - min_dist

        self.state['int']['err_corr_count'] += diff

        baseline_v = weighted_sensors_f * coeff
        diff_map = [[math.sqrt(diff), math.sqrt(diff) * -1], [math.sqrt(diff) * -1, math.sqrt(diff)]]

        if distance < min_dist:
            self.state['int']['motor_l_v'] = baseline_v + diff_map[motor_index][0]
            self.state['int']['motor_r_v'] = baseline_v + diff_map[motor_index][1]
        else:
            self.state['int']['motor_l_v'] = baseline_v + diff_map[motor_index][1]
            self.state['int']['motor_r_v'] = baseline_v + diff_map[motor_index][0]

    def set_joint_pos(self):
        """
        Set starting positions for revolute joints
        """
        for m in self.state['int']['motors']:
            res, current_pos = vrep.simxGetJointPosition(self.h.client_id, m, vrep.simx_opmode_streaming)
            self.state['int']['jpos'][str(self.state['int']['motors'][m] + '_prev')] = current_pos
            self.state['int']['jpos'][str(self.state['int']['motors'][m] + '_total')] = current_pos

    def update_state_odometry(self):
        """
        Update odometry state by reading the motor revolute joint positions
        """
        for m in self.state['int']['motors']:
            self.update_joint_pos(m)

    def update_joint_pos(self, handle):
        """
        Update motor revolute joint positions. Determine rotation amount since last read
        """

        res, current_pos = vrep.simxGetJointPosition(self.h.client_id, handle, vrep.simx_opmode_streaming)
        difference_pos = current_pos - self.state['int']['jpos'][str(self.state['int']['motors'][handle] + '_prev')]
    self.state['int']['jpos'][str(self.state['int']['motors'][handle] + '_prev')] = current_pos

    difference_pos = self.get_pos_ang(difference_pos)
    self.state['int']['jpos'][str(self.state['int']['motors'][handle] + '_total')] += difference_pos

    self.state['int']['jpos'][str(self.state['int']['motors'][handle] + '_turns')] = \
        self.state['int']['jpos'][str(self.state['int']['motors'][handle] + '_total')] / (math.pi * 2)

    self.state['int']['jpos'][str(self.state['int']['motors'][handle] + '_dist')] = \
        self.state['int']['jpos'][str(self.state['int']['motors'][handle] + '_turns')] * self.props['wheel_circ_m']

@staticmethod
def get_pos_ang(pos):
    """
    Get revolute position as turn angle
    """
    if pos >= 0:
        pos = math.fmod(pos + math.pi, 2 * math.pi) - math.pi
    else:
        pos = math.fmod(pos - math.pi, 2 * math.pi) + math.pi
    return pos
```

APPENDIX E – SENSORS.PY

```
import logging
import logger as lg
import vrep
import math
import numpy as np


class Sensor:
    """
    Implements sensors supporting perception of environment
    """
    def __init__(self, client_id, handle=None, sensor_array=None):
        lg.message(logging.INFO, 'Initialising ' + self.__class__.__name__)
        self.client_id = client_id
        self.sensor_array = sensor_array
        self.last_read = None


class ProximitySensor(Sensor):
    """
    Implements and manages a proximity sensor array
    """
    def __init__(self, client_id, handle=None, sensor_array=None):
        Sensor.__init__(self, client_id, sensor_array=sensor_array)
        self.detection_points = []
        self.max_detection_dist = 1
        self.last_read = self.read(vrep.simx_opmode_streaming)

    def read(self, mode, handle=None, sensor_array=None):
        """
        Update value for each sensor in array
        """
        self.last_read = []
        for i, s in enumerate(self.sensor_array):
            res, ds, dp, doh, dsnv = vrep.simxReadProximitySensor(self.client_id, s, mode)
            distance = math.sqrt(sum(i**2 for i in dp))
            if not ds:
                distance = self.max_detection_dist  # If bad read set distance to default max range
            self.last_read.append(distance)
        return self.last_read


class Compass(Sensor):
    """
    Implements a magnetic compass virtually, using V-REP object orientation angles
    """
    def __init__(self, client_id, handle=None, sensor_array=None):
        Sensor.__init__(self, client_id, handle=handle)
        self.to_bearing = None
        self.last_read = self.read(vrep.simx_opmode_streaming, handle)

    def read(self, mode, handle=None):
        """
        Get absolute orientation of object to scene
        """
        res, bearing = vrep.simxGetObjectOrientation(self.client_id, handle, -1, mode)
        self.last_read = round(360 * (math.pi + bearing[2]) / (2 * math.pi), 2)  # Bearing to magnetic degree
        return self.last_read

    def set_to_bearing(self, degrees):
        """
```

```
Calculate target magnetic bearing
"""
self.to_bearing = (self.last_read + degrees)
if self.to_bearing > 360:
    self.to_bearing -= 360
elif self.to_bearing < 0:
    self.to_bearing += 360
```

APPENDIX F – HELPER.PY

```
import logging
import logger as lg
import vrep
import math


class Node(object):
    """
    Class for building N-ary tree of V-REP object with descendants
    """
    def __init__(self, data):
        self.data = data
        self.children = []

    def add_child(self, obj):
        """
        Add descendant
        """
        self.children.append(obj)


class Helper:
    """
    Generic helper class supporting simulation in V-REP
    """
    def __init__(self):
        lg.message(logging.INFO, 'Initialising ' + self.__class__.__name__)
        self.object_types = {0: 'shape',
                             1: 'joint',
                             2: 'graph',
                             3: 'camera',
                             4: 'dummy',
                             5: 'proximitysensor',
                             6: 'reserved1',
                             7: 'reserved2',
                             8: 'path_type',
                             9: 'visionsensor',
                             10: 'volume',
                             11: 'mill',
                             12: 'forcesensor',
                             13: 'light',
                             14: 'mirror'}

        self.api_server = {'connectionAddress': '127.0.0.1',
                           'connectionPort': 19999,
                           'waitUntilConnected': True,
                           'doNotReconnectOnceDisconnected': True,
                           'timeOutInMs': 5000,
                           'commThreadCycleInMs': 5}

        self.client_id = -1
        self.connected = self.connect_client()
        self.object_tree = {}

    def connect_client(self):
        """
        Connect this client to V-REP server
        """
        self.client_id = vrep.simxStart(self.api_server['connectionAddress'],
                                        self.api_server['connectionPort'],
                                        self.api_server['waitUntilConnected'],
                                    self.api_server['doNotReconnectOnceDisconnected'],
                                    self.api_server['timeOutInMs'],
                                    self.api_server['commThreadCycleInMs'])

        if self.client_id != -1:  # check if client connection successful
            lg.message(logging.INFO, 'Connected to V-REP')
            return True
        else:
            lg.message(logging.INFO, 'Failed connecting to V-REP')
            return False

    def disconnect_client(self):
        """
        Disconnect this client from the V-REP server
        """
        # Close the connection to VREP
        vrep.simxStopSimulation(self.client_id, vrep.simx_opmode_blocking)
        vrep.simxFinish(self.client_id)
        lg.message(logging.INFO, 'Disconnected from V-REP')

    def get_object_handle_by_name(self, object_name):
        """
        Return V-REP object handle for a given object name
        """
        lg.message(logging.DEBUG, 'Getting V-REP object handle for ' + object_name)
        error_code, handle = vrep.simxGetObjectHandle(self.client_id, object_name, vrep.simx_opmode_oneshot_wait)
        if error_code == 0:
            return handle
        else:
            return -1

    def get_object_tree(self, handle, index):
        """
        Build N-ary tree for V-REP object
       """
        if index == 0:
            self.object_tree[handle] = Node(handle)
        child_handle = self.get_object_child(handle, index)

        if child_handle != -1:
            self.object_tree[handle].add_child(child_handle)
            index += 1
            self.get_object_tree(handle, index)
        else:
            for c in self.object_tree[handle].children:
                self.get_object_tree(c, 0)

    def get_object_child(self, handle, index):
        """
        Get V-REP handle for child object
        """
        error_code, child_handle = vrep.simxGetObjectChild(self.client_id, handle, index, vrep.simx_opmode_blocking)
        return child_handle

    @staticmethod
    def get_object_data(client_id, object_type, data_type):
        """
        Get V-REP object group data by type
        """
        res, handles, int_data, float_data, string_data = vrep.simxGetObjectGroupData(client_id, object_type, data_type,
                                                                                       vrep.simx_opmode_blocking)
        if res == vrep.simx_return_ok:
            return handles, int_data, float_data, string_data
```

```
def get_objects(self):
    """
    Get V-REP object data
    """
    object_names = self.get_object_data(self.client_id, vrep.sim_appobj_object_type, 0)

    # Get object types
    object_types = self.get_object_data(self.client_id, vrep.sim_appobj_object_type, 1)

    # Get parent object handles
    object_parent_handles = self.get_object_data(self.client_id, vrep.sim_appobj_object_type, 2)

    objects = list(zip(object_names[0], object_names[3], object_types[1], object_parent_handles[1]))

    full_scene_object_list = []
    simple_object_list = {}
    for object_handle, object_name, object_type, object_parent_handle in objects:
        full_scene_object_list.append((object_handle, object_name, object_type, self.object_types[object_type],
                                       object_parent_handle))
        simple_object_list[object_handle] = object_name

    return full_scene_object_list, simple_object_list

@staticmethod
def ms_to_radss(metres_per_s, radius):
    """
    Convert metres per second to radius per second
    """
    return metres_per_s / radius
    #return (metres_per_s * 2) / (radius * 2)

def within_dist(self, point_a, point_b, dist_threshold=0.07):
    """
    Determine if point A within distance threshold of point B
    """
    distance = self.get_euclidean_distance(point_a, point_b)
    lg.message(logging.DEBUG, 'Distance between point A and B is ' + str(distance))
    if distance <= dist_threshold:
        return True
    else:
        return False

@staticmethod
def get_euclidean_distance(point_a, point_b):
    """
    Euclidean distance between point A and B
    """
    return math.sqrt((point_a[0] - point_b[0]) ** 2 + (point_a[1] - point_b[1]) ** 2)
```

APPENDIX G – LOGGER.PY

```
import logging


def message(l_type, msg):
    """
    Log message and print to console if classified as info type
    """
    if l_type == logging.INFO:
        logging.info(msg)
        print(msg)
    elif l_type == logging.WARNING:
        logging.warning(msg)
    elif l_type == logging.DEBUG:
        logging.debug(msg)
```

APPENDIX H – EXAMPLE LOG

```
2019-11-06 17:30:30,848 - INFO - Staring scenario scenario3
2019-11-06 17:30:30,849 - INFO - World props {'min_dist_enabled': True, 'min_dist': 0.23, 'max_dist': 0.28}
2019-11-06 17:30:30,849 - INFO - World events [{'task': 'turn', 'degrees': 164}, {'task': 'move', 'robot_dir_travel': 1, 'dist': 30, 'velocity': 0.4}, {'task': 'wall_follow', 'dir': -
1, 'method': 'min', 'coeff': 1.6}, {'task': 'stop'}]
2019-11-06 17:30:30,849 - INFO - Initialising Controller
2019-11-06 17:30:30,849 - INFO - Initialising Helper
2019-11-06 17:30:30,851 - INFO - Connected to V-REP
2019-11-06 17:30:30,900 - DEBUG - Getting V-REP object handle for Pioneer_p3dx
2019-11-06 17:30:32,269 - INFO - Initialising PioneerP3dx
2019-11-06 17:30:32,269 - DEBUG - Getting V-REP components for Motor
2019-11-06 17:30:32,269 - DEBUG - Getting V-REP components for ultrasonic
2019-11-06 17:30:32,269 - INFO - Initialising ProximitySensor
2019-11-06 17:30:32,269 - INFO - Initialising Compass
2019-11-06 17:30:32,269 - INFO - Starting controller loop
2019-11-06 17:30:32,269 - INFO - Starting event - turn
2019-11-06 17:30:32,270 - DEBUG - Executing method turn args {'degrees': 164}
2019-11-06 17:30:32,270 - DEBUG - Turn bearing from 180.0 to 344.0
2019-11-06 17:30:32,270 - DEBUG - Setting motor left velocity (m/s) - -0.1
2019-11-06 17:30:32,270 - DEBUG - Setting motor right velocity (m/s) - 0.1
2019-11-06 17:30:32,270 - DEBUG - Motor left velocity now set at (rad/s) - -0.5128205128205129
2019-11-06 17:30:32,270 - DEBUG - Motor right velocity now set at (rad/s) - 0.5128205128205129
2019-11-06 17:30:35,291 - INFO - Turn event complete
2019-11-06 17:30:35,295 - INFO - Starting event - move
2019-11-06 17:30:35,295 - DEBUG - Executing method move args {'robot_dir_travel': 1, 'dist': 30, 'velocity': 0.4}
2019-11-06 17:30:35,295 - DEBUG - Setting motor left velocity (m/s) - 0.5
2019-11-06 17:30:35,295 - DEBUG - Setting motor right velocity (m/s) - 0.5
2019-11-06 17:30:35,295 - DEBUG - Motor left velocity now set at (rad/s) - 2.564102564102564
2019-11-06 17:30:35,295 - DEBUG - Motor right velocity now set at (rad/s) - 2.564102564102564
2019-11-06 17:30:38,312 - INFO - Stop task triggered
2019-11-06 17:30:38,312 - DEBUG - Setting motor left velocity (m/s) - 0
2019-11-06 17:30:38,312 - DEBUG - Setting motor right velocity (m/s) - 0
2019-11-06 17:30:38,312 - DEBUG - Motor left velocity now set at (rad/s) - 0.0
2019-11-06 17:30:38,312 - DEBUG - Motor right velocity now set at (rad/s) - 0.0
2019-11-06 17:30:38,312 - INFO - Starting event - wall_follow
2019-11-06 17:30:38,312 - DEBUG - Executing method wall_follow args {'dir': -1, 'method': 'min', 'coeff': 1.6}
2019-11-06 17:30:38,312 - DEBUG - Start point set [-3.0630569458007812, -0.0715235248208046, 0.1386939287185669]
2019-11-06 17:30:38,313 - DEBUG - Distance between point A and B is 0.0
2019-11-06 17:30:38,313 - DEBUG - Setting motor left velocity (m/s) - 0.30479008800707197
…
…
2019-11-06 17:33:27,468 - INFO - Wall Follow event complete
2019-11-06 17:33:27,468 - DEBUG - Setting motor left velocity (m/s) - 0.36370101709391994
2019-11-06 17:33:27,468 - DEBUG - Setting motor right velocity (m/s) - 0.3722406427411246
2019-11-06 17:33:27,468 - DEBUG - Motor left velocity now set at (rad/s) - 1.8651334209944612
2019-11-06 17:33:27,468 - DEBUG - Motor right velocity now set at (rad/s) - 1.9089263730314083
2019-11-06 17:33:27,472 - INFO - Starting event - stop
2019-11-06 17:33:27,472 - INFO - Stop task triggered
2019-11-06 17:33:27,472 - DEBUG - Setting motor left velocity (m/s) - 0
2019-11-06 17:33:27,472 - DEBUG - Setting motor right velocity (m/s) - 0
2019-11-06 17:33:27,472 - DEBUG - Motor left velocity now set at (rad/s) - 0.0
2019-11-06 17:33:27,472 - DEBUG - Motor right velocity now set at (rad/s) - 0.0
2019-11-06 17:33:27,472 - INFO - Distance travelled - 92.81m
2019-11-06 17:33:27,472 - INFO - Average speed - 0.53m/s
2019-11-06 17:33:27,472 - INFO - Nav dist diff error - 952.69cm
2019-11-06 17:33:27,472 - INFO - Controller loop complete - time taken - 175.2s
2019-11-06 17:33:28,250 - INFO - Disconnected from V-REP
2019-11-06 17:33:28,255 - INFO - Scenario scenario3 complete
```