IAR Task 2 Report

Angus Pearson – s1311631
Jevgenij Zubovskij – s1346981

October 20, 2016

Abstract

We present an implementation of an algorithm similar to *Bug2*[1] built atop reactive obstacle collision avoidance and edge following behaviour developed for *Task1* with the *Khepera* robot. The existing architecture is extended to include storing and publishing timestamped poses and goal state to a Redis[2] server. Real-time visualisation using a Redis to ROS[3] pipe provides a graphical display of odometry, sensory information and goals in Rviz. After-the-fact plotting is provided with MatPlotLib independent of ROS.

In our testing, the robot was able to navigate to within 10cm of the origin (it's home location) from any location in an environment successfully in 17/20 experiments, where each environment was designed to evoke edge-case behaviour considered hard for the algorithm. The algorithm performs similarly in both a static world and dynamic one in which other actors (e.g. Humans) present a transient obstacle.

1 Introduction

The second IAR assignment entails extending the existing systems from Task 1 with Odometry, to maintain an estimate of the robot's location. The task also calls for 'return to home' ability, "either by retracing its outward route or more directly". As an extention, basic mapping of the world is developed (without localisation correction for odometry dead-reckoning drift).

This new return home behaviour is implemented as extension to the simpler reactive wall-following and collision avoidance code from *Task1*,[9] in a hybrid-subsumtive sense where the higher-level return navigation generally informs the Robot's behaviour, but if proximity thresholds are violated the lower level reaction will peturb the trajectory away from an obstacle until the proximity error clears.

We use Rviz for real-time plotting, a popular graphical tool for ROS to live-render the Robot's pose, odometry trail, obstacles perceived by the IR range sensors and goals. Rviz is plotting standard ROS messages such as Poses, Paths and Point Clouds that we publish to 'ROS topics'.

2 Odometry

We use odometry to find the Pose of the robot (x, y, Θ). The strarting position of the robot as the origin of the coordinate system $(0, 0)$ and the forward-facing direction of the robot as the positive direction of the x axis and all angles are measured in relation to that. Ideally we would like to account for odometry drift over time due to wheel slippage, encoder error and approximate calculations etc.

Moreover, we assume that the wheels have equal diameter and that the data given in the Khepera 2 User Manual[8] is correct and an increment of 12 encoder ticks means a wheel has advanced 1mm in it's rotational direction.

An advantage of the Khepera is that it has very thin wheels, meaning almost a single point of contact to the floor, which make odometry much more accurate.

2.1 Research

Before settling on a method to use, we researched several papers and websites for inspiration or already fully conceptualized solutions or even a fully developed and tested approach. The first one we found used a fourth order *Runge-Kutta* numerical integrator[4] and looked like it would compensate for non-ideal encoders. However, upon implementation, and testing if it correctly detects the robot going in a straight line (equal motor speeds) and said algorithm fialing to significantly reduce error, it was abandoned.

Hence, a simple yet mathematically sound approach[5] was adopted.

2.2 Chosen Method

Pose calculation[5] uses simple trigonometry and differential drive on the wheels to perform its task. The position for (x, y, Θ) are in the global reference pane, relative to initial position.

1. Distance driven by each wheel $\Delta s_{wheel} = \frac{\Delta ticks_{wheel}}{12^{ticks}/_{mm}}$
2. Total distance driven $\Delta s = \frac{\Delta s_r + \Delta s_l}{2}$
3. Change in heading $\Delta\Theta = \frac{\Delta s_r - \Delta s_l}{B}$ where B is the wheel base (distance between points of contact of the two wheels on the floor)
4. x position update $x_{n+1} = x_n + \Delta s \cos(\Theta_n + \frac{\Delta\Theta}{2})$
5. y position update $y_{n+1} = y_n + \Delta s \sin(\Theta_n + \frac{\Delta\Theta}{2})$
6. Lastly update $\Theta_{n+1} = \Theta_n + \Delta\Theta$

The encoders are incremental and are able to detect both forward and backward movement of the wheel, so no software switching is required to differentiate forward and reverse motion.

We do not use Odometry to inform navigation other than by the *§3 Return Algorithm* to ensure the robot is within the desired radius of the goal.

2.3 Calibration

The odometry was tested in a square box to verify it identifies 90°turns correctly (while following the walls of the box) as well as representing accurate distance. We calibrated according to the methods laid out in.[6] We reduced the wheelbase if the angle was shown to be too small and increase if it was too large until the odometry detected the 90°turns correctly. The actual 5 cm wheelbase is calibrated to 5.6 cm in code.

Figure 1 shows a diagram of a simple test to ensure the actual right-angle turn is refelcted correctly in odometry.

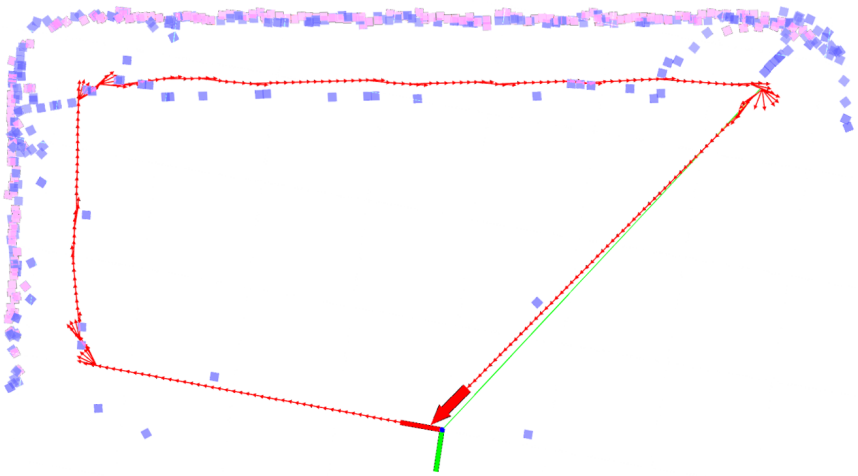


Figure 1: The robot starts in the centre of an empty squre box, heads towards the wall, turns to follow the wall, then turns the box corner which is 90 °, follows the wall for some more time, and heads to its starting position (where it stops).

3 Return Algorithm

Since Task 2 requires odometry, we localise solely with odometry and use the IR sensors for avoiding obstacles as per *Task 1*.[9]

3.1 Research

Choset's so-called *Bug algorithms*,[1] inspired by insect behaviour give simple behaviour capable of navigating to a location suitable for the task. *Bug 2* is robust and in most environments the most efficient out of the three Bug algorithms. Hence, it was chosen to be implemented.

3.2 Method Chosen

The Bug 2 algorithms is conceptually simple. The assumptions it needs to work properly can be simplified to:

1. Known direction to goal and robot can measure distance between points
2. Be in a bounded workspace with a finite number of finite size objects

These assumptions are satisfied by the environment for the task.

Bug 2 Algorithm

1. Head toward goal on the *m-line* – the straight line drawn from the start point to the goal
2. If an obstacle is in the way, follow it until you encounter the *m-line* again closer to the goal
3. Leave the obstacle and continue toward the goal

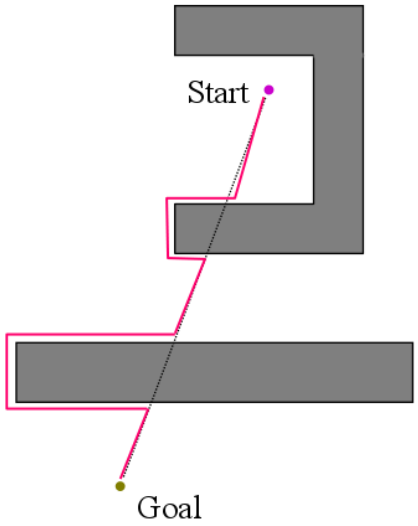


Figure 2: Example Bug 2 operation diagram. The straight line between the Start and Goal is the *m-line*

3.3 Integration

The algorithm is activated after $\approx$ 30 seconds of exploration[9] and after activation is the dominant algorithm unless one of the following is true for that control loop iteration:

1. The reactive control (getting unstuck[9]) activates - reactive avoidance algorithm is in control
2. There is a miscalculation for the new speeds and they robot may try to drive towards the wall it is following - exploration (navigation) algorithm is in control

The second point can occur if *m-line* recomputation occurs as described in *m-line Replanning*

Figure 3 shows how the controling paradigm is chosen from Ractive Avoidance, explorative Navigation and Return at every iteration of the loop. Before the 30 second mark robot performs

exactly as per *Task 1*, after that the "boredom"[9] is disabled. Please note that unlike in the report for *Task 1*, here the collistion voidance is regarded as a separate algorithm from the wall following, forward driving and "boredom" (exploration algorithm) one

The following is a representation of the algorithm in a state diagram form:

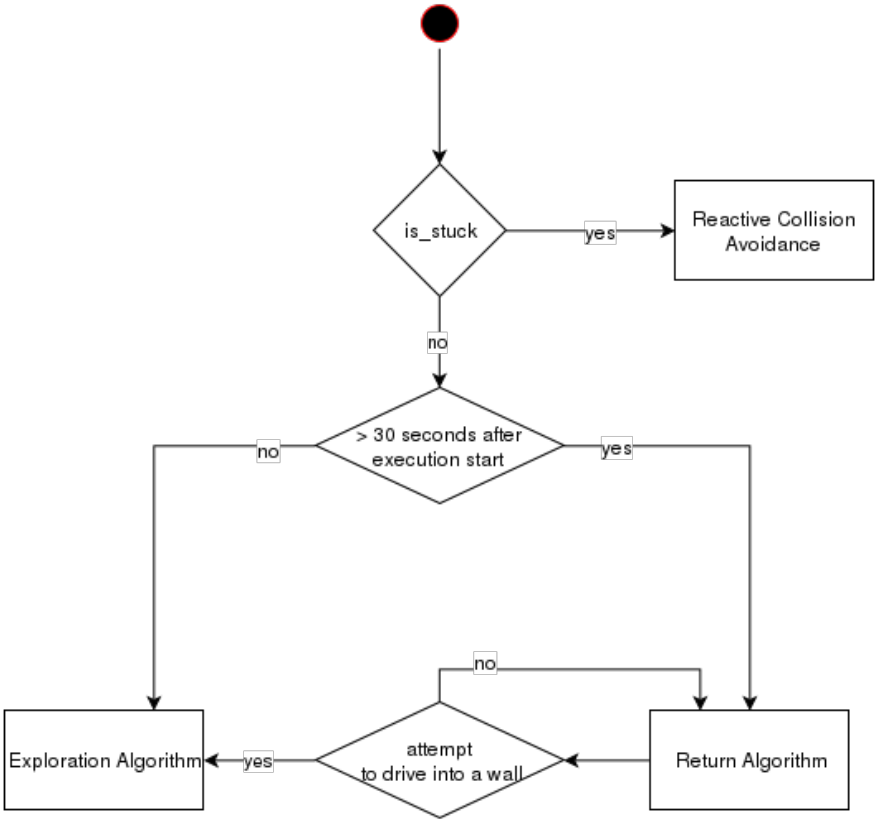


Figure 3: Control algorithm selection in a state diagram form.

3.4 Applied Alterations

However, the implementation is much less trivial than the higher level description. It it had to be altered due to problems that became evident during testing and in *§5 Experimental Results*

3.4.1 Thresholding

Due to imperfect encoders, wheel slippage, odometry drift and being unable to sample odometry more often than every 20 ms[8] (due to sequential reads of IR being in the same loop. Moreover, one must remember odometry fomulas have intrinsic systematic error.

Hence, all odometry and position-related calculations can not be strict equalities to compensate for the above error. Hence, all values are considered "on-point" as long as their value fall within the expected range. Some thresholds are several centimeters.

3.4.2 Replanning the *m-line*

Testing revealed that about $1/15$ times the calculations of the approach angle to the goal would fail and be $180°$ more than the real value. This required an alteration to *Bug 2*, so if it is in free space (not following walls or being "unstuck"[9]) and it deviates from the last detected *m-line* segment for more than a threshold value, we recompute the approach vector to the goal (along the new *m-line*) .

Additionally a spike in an IR sensor reading could wrongly present as a departure from the wall, triggering an incorrect recompute of the *m-line*.

This was mitigated by introducing another distance check into the Bug algorithm, to see if we could still theoretically follow the wall, in which case the threshold at which we would recompute the *m-line* was extended. Hence, even if robot gets stuck in a particular scenario, there should be very few edge cases wher it will get trapped and continue reocomputing the *m-line*, getting itself stuck again. Altering the two constants for *m-line* recomputing is what was also done durting testing to minimize said edge cases.

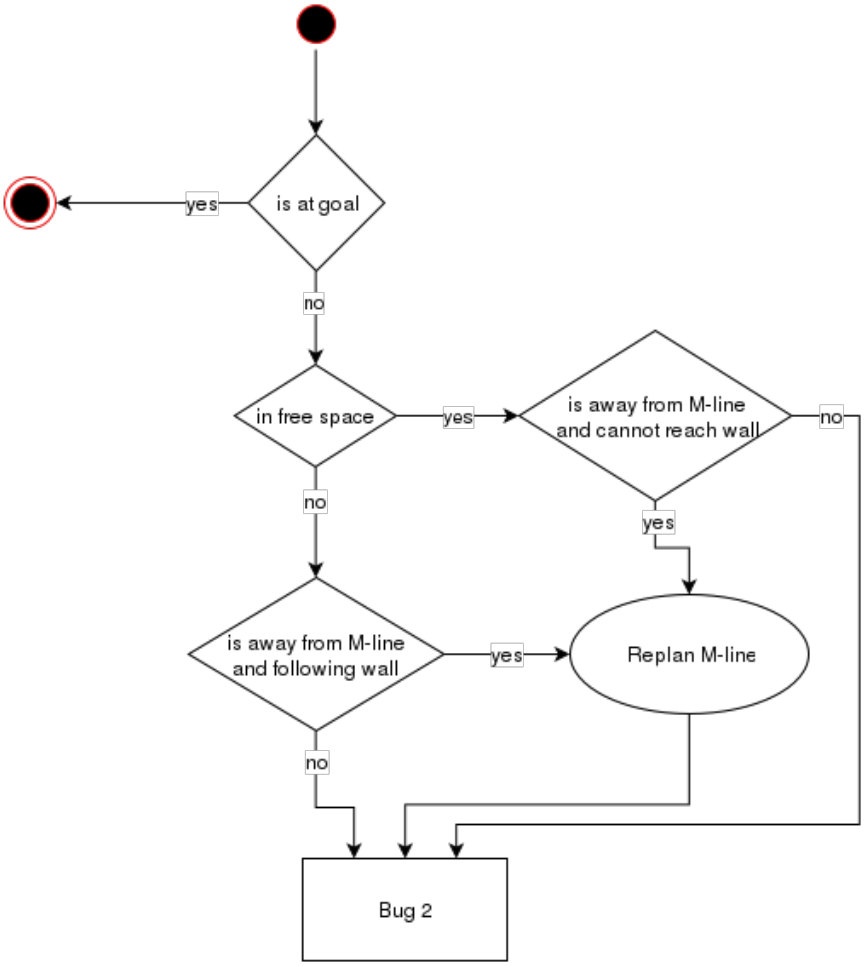


Figure 4: The return algorithm in a state diagram form where the oval shape is the conditional action done before going to *Bug 2* algorithm. That action replans the *m-line*

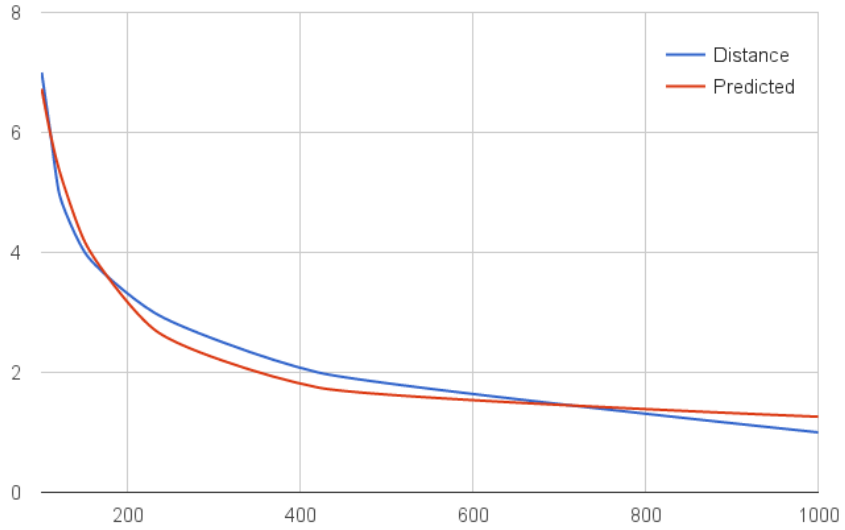
4 Plotting

At each epoch in the control loop, a Pose and sensor-activation array is published to a Redis channel. If the goal has changed, it is also published to a separate channel. In the case of *Bug2* this goal line will be the *m-line*.

As the Redis instance is network-accessible and running as it's own process, this prevents any rendering or plotting activity from stalling or slowing the control loop. This also keeps loop run-time consistent, and allows the rendering to be done on a separate computer.

In the case of using RViz to render our data, `data.py` contains code for subscribing to all the Redis channels the control loop will publish to and exposing ROS Topics for the same data that can be picked up by RViz or any other ROS compatible program.

The data received from the control loop is raw sensor data, that requires both conversion to actual distance data and translation from the Khepera's frame of reference to the global frame. The activation to distance conversion was done using an equation from an equation solver and 10 samples of *distance, activation* pairs.



$$S = 1.074519 + \frac{9.502961}{(1 + \frac{\alpha}{70.42612}^{69.9039})^{0.02119919}} \tag{1}$$

Figure 5: Solved equation translation betweek S, the distance, and α, the raw activation data of the sensor.

The reference pane transformation is performed by a 3×3 matrix, rotating and translating onto the global reference pane.

A simple map is evolved as the Robot explores it's environment. We can derive solids in the environment from the distance sensor readings, by discarding distances above a given threshold but persisting a PointCloud of *voxels* for any readings that are closer than this threshold. These voxels (a 3D pixel) are colorised by proximity to the robot – bluer indicates a farther point, redder indicates closer.

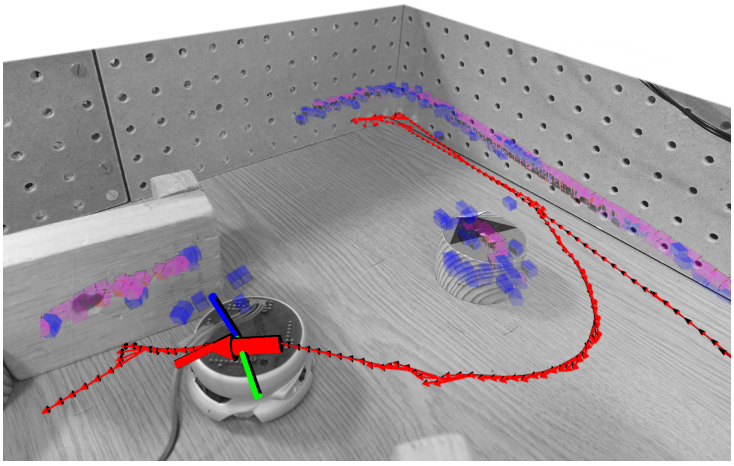


Figure 6: Voxel map, Pose (red arrow), Origin and Odometry trail overlaid onto an image of the *Khepera* and environment taken from the same perspective. Voxels are placed at the Khepera's 'eye level' of 16mm.

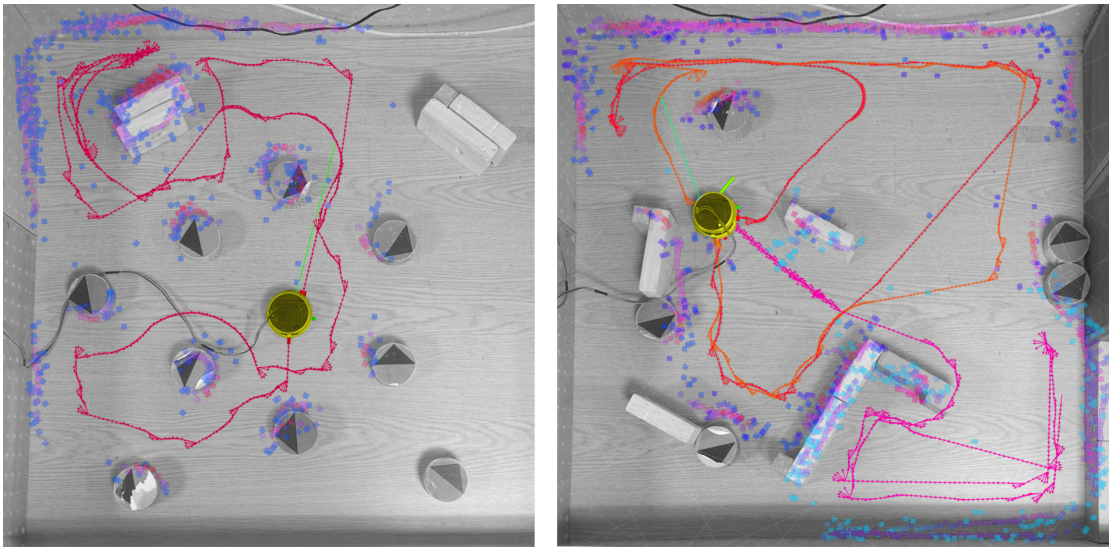


Figure 7: Top-down Rviz plot overlaid on a photograph of the environment, showing **left** a long duration (~1min) exploration, and **right** multiple explorations and returns, where a different route was taken each time. The *Khepera* is highlighted in yellow at it's home location.

5 Experimental Results

In our testing, with two different and complex environments measuring roughly 1m by 2m the robot successfully returned to within 10cm of it's starting position after 30 seconds of unguided exploration in 17/20 tests. The Robot took a different path in each excursion because it was manually guided in the first few seconds by the operator placing a temporary obstacle in it's way. Figure 8 shows the plots of these excursions separated by the two test environments. Both environments contain sparesely populated and dense areas, and 'traps' that require correct Bug behaviour to navigate.

The three failures to return to the home location were due to odometry drift – the robot completed at a location it believed was withing 10cm of home but was in fact further away. One excursion had a near algorithm failure, this eventually cleared and the robot successfully returned home. This incident is visible in the top image depicting the first 10 trials, bottom left where there is an excess of voxels and odometry.

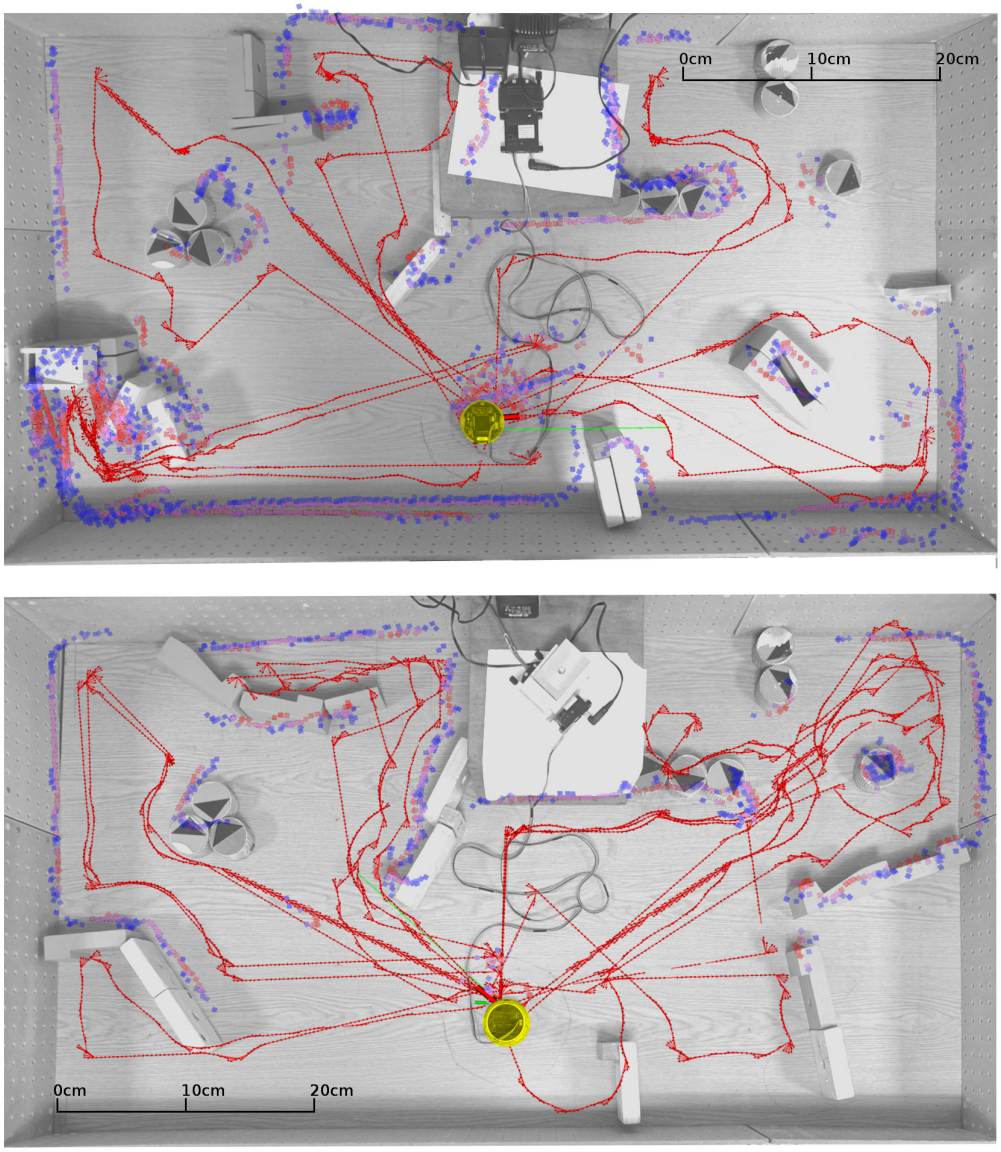


Figure 8: Tracks from the two test sets overlaid on a photograph of the environment. **Top:** The first 10 trials, **Bottom:** the second 10 trials.

6 Discussion & Possible Improvements

Various system implementation methods were researched, best ones chosen, implemented and tested in both sparse and dense environment for both short and long periods of time.

Odometry drift increases with time, and with the number of turns taken. However, the path that was taken by the robot, the angles of the turns and distances were identified very accurately. Even though it worked very well in highly dense environments the above errors were minimal in sparse environments.

The live plotting was highly effective and not only allowed us to properly test our odometry as well as return algorithm, but also visualize the environment in a highly comprehensible and inforamtive way. Moreover, it uses a separate data server, which means if the main program crashes the data is not lost, and data can persist and be played back through Rviz later. Furthermore, we have solved a formula for the distance to the obstacle based on the IR values. Lastly, the current system facilitates an easy move towards SLAM *(hint hint)*.

The subsumptious hybrid architecture that was developed between the three algorithms (as described before) worked effectively and allowed the robot to properly switch between behaviors. The return algorithm's precision depends on positioning errors, so it's error is derived almost entirely from odometry. However, it worked with a very high sucess rate, getting stuck in only edge cases where it would be trapped in a very deep pocket which protruded almost towards the goal and had long walls.

Attempts to correct the algorithmic flaws were made as per above and in *§3.4 Needed Alterations* and could yield better results if wall following is fixed to where the approach that we used would not be needed, leaving the algorithm a faithful implementation of *Bug 2*.

The prime targets for improvement are our trigonometric calculations, and a supplementary localisation technique to counteract the odometry error which is currently our limiting factor. However, we would consider it a sucessful odometric, plotting and returning hybrid subsumptious mobile autonomous system™.

References

[1] Principles of Robot Motion: Theory, Algorithms, and Implementation §2.1 'Bug Algorithms'
Howie Choset

[2] Redis is an open source (BSD licensed), in-memory data structure store, used as database, cache and message broker. It supports data structures such as strings, hashes, lists, sets, sorted sets with range queries, bitmaps, hyperloglogs and geospatial indexes with radius queries.
http://redis.io

[3] The Robot Operating System (ROS) is a flexible framework for writing robot software. It is a collection of tools, libraries, and conventions that aim to simplify the task of creating complex and robust robot behavior across a wide variety of robotic platforms.
http://www.ros.org/

[4] Fourth order Runge - Kutta algorithm for pose estimation
https://www.cs.cmu.edu/afs/cs.cmu.edu/academic/class/16311/www/s07/labs/NXTLabs/Lab%203.html

[5] Princeton University lecture on Autonomous Robot Navigation with odometry formulas and their deriviation from geometry and trigonometry. These formulas are devised specifically for differential drive robots
https://www.cs.princeton.edu/courses/archive/fall11/cos495/COS495-Lecture5-Odometry.pdf

[6] The Technic Gear website article dealign with differential drive robot odometry calculations
http://thetechnicgear.com/2014/06/howto-calibrate-differential-drive-robot/

[7] A paper on Experimental Odometry Calibration of the Mobile Robot Khepera II Based on the Least-Squares Technique from which we took inspiration on calibration and confirmation we are doing odometry in the right fashion
http://ieeexplore.ieee.org/stamp/stamp.jsp?arnumber=1570321

[8] Khepera 2 user Manual containing all the fundamental information about the Khepera 2 robot, including information abotu encoders and their value meanings
http://ftp.k-team.com/khepera/documentation/Kh2UserManual.pdf

[9] IAR 2016 Task1 Report
Angus Pearson, Jevgenij Zubovskij

Appendix

.1 Code Listings

../../main.py

```
#!/usr/bin/env python

#
#     - - - -  I A R  - - - -
#
#  s1311631         Angus Pearson
#  s1346981    Jevgenij Zubovskij
#

from __future__          import print_function
from comms           import Comms

from odometry_algorithm     import Odometry_Algorithm
from odometry_state         import Odometry_State

from navigation_state       import Navigation_State
from navigation_algorithm   import Navigation_Algorithm

from bug_algorithm      import Bug_Algorithm
from bug_state          import Bug_State



import constants
import sys
import getopt      # CLI Option Parsing
import whiptail    # Simplest kinda-GUI thing
import time
import math
import matplotlib.pyplot as plt
from data import DataStore

namebadge = " -- IAR C&C -- "
helptext = str(sys.argv[0]) + ' -p <serial port> -b <baud rate> -t <timeout> -s <server hostname>
    '

wt = whiptail.Whiptail(title=namebadge)



def main():


    try:
        #flashy to see if robot works
        comms.blinkyblink()

        odo = Odometry_Algorithm()
        odo_state = Odometry_State()
    nav_state = Navigation_State()
    nav = Navigation_Algorithm()
    bug = Bug_Algorithm()
    bug_state = Bug_State()


        # varaibles to not resend speeds during wall following
```

```
        speed_l = 0
        speed_r = 0

        # List of dict for Redis -> ROS goal line render
        mline_hm = [
            {'x': bug_state.m_line_start[0],
             'y': bug_state.m_line_start[1]},
            {'x': bug_state.m_line_end[0],
             'y': bug_state.m_line_end[1]}
            ]

        # reset odometry for this robot run
        comms.reset_odo()


    #begin control loop
        while True:


        odo_state = odo.new_state(odo_state, comms.get_odo())
        nav_state.dist = comms.get_ir()

        #check reactive first, then bug
        nav_state = nav.new_state(nav_state, odo_state, bug_state, comms, bug)
        #if have free movement, use the bug algorithm
        if bug_state.algorithm_activated and bug_state.in_control == True:
        nav_state = bug.new_state(nav_state, odo_state, bug_state)
        #if we are done break the control loop, stop the robot and exit
        if bug_state.done:
            print("DONE")
            comms.drive(0, 0)
            comms.blinkyblink()
            break

        #only send stuff over serial if new values
        if not( speed_l == nav_state.speed_l and speed_r == nav_state.speed_r):
            comms.drive(nav_state.speed_l, nav_state.speed_r)

            speed_l = nav_state.speed_l
            speed_r = nav_state.speed_r

            ds.push(odo_state, nav_state.dist)

            if (mline_hm[1]['x'] != bug_state.m_line_end[0] or
                mline_hm[1]['y'] != bug_state.m_line_end[1]):
                # Update and push
                mline_hm[1]['x'] = bug_state.m_line_end[0]
                mline_hm[1]['y'] = bug_state.m_line_end[1]
                ds.push_goal(mline_hm)

        # do not attempt to instantly read sensors again
            time.sleep(constants.MEASUREMENT_PERIOD_S)

    except TypeError as e:
        comms.drive(0,0)
        raise(e)

# ######################
# Init & CLI gubbins...
# ######################

if __name__ == "__main__":
    # Ignore 1st member, which is the name
```

```
    # the program was invoked with
    args = sys.argv[1:]

    # Read & Parse command line options
    try:
        optlist, args = getopt.getopt(args, 'hp:t:b:s:',
                                      ['help','port=','server=','baud=','timeout='])
    except getopt.GetoptError:
        print("Invalid Option, correct usage:")
        print(helptext)
        sys.exit(2)

    # Our defaults, may be different
    # from the ones built into the class'
    # __init__(self) constructor
    port = "/dev/ttyUSB0"
    timeout = 1
    baud = 9600

    server = "localhost"

    for opt, arg in optlist:
        if opt in ('-h', '--help'):
            print(namebadge)
            print(helptext)
            sys.exit(0)
        elif opt in ('-p', '--port'):
            # change serial port to use
            port = str(arg)

        elif opt in ('-t','--timeout'):
            # change blocking timeout for reading
            timeout = float(arg)

        elif opt in ('-b', '--baud'):
            # change baud rate
            baud = int(arg)

        elif opt in ('-s', '--server'):
            server = str(arg)
            print("Connecting to Redis server at " + str(server))

    # Initialise a serial class, or
    try:
        comms = Comms(port, baud, timeout)
    except Exception as e:
        if wt.confirm("Can't initialise serial, exit?\n\n"+str(e)):
            sys.exit(1)
        raise(e)

    print(namebadge)

    ds = DataStore(host=server)

    try:
        main()
    except KeyboardInterrupt as e:
        comms.drive(0,0)
        ds.save()
        print("Stopping and Quitting...")
        raise e

else:
```

```
    # if *not* running as __main__
    # invoke the class with defaults
    comms = Comms()
    ds = DataStore()
```

../../data.py

```
#!/usr/bin/env/python

#
#     - - - -  I A R  - - - -
#
#   s1311631         Angus Pearson
#   s1346981    Jevgenij Zubovskij
#

from __future__ import print_function
import redis
import whiptail
from state import *
import sys
import getopt
import matplotlib.pyplot as plt
import math
import time
import yaml
import pprint

import constants


# Check for ROS (http://ros.org/) support
try:
    import rospy
    ros = True
    from std_msgs.msg import String
    from geometry_msgs.msg import PoseWithCovariance, PoseStamped, Point32
    from nav_msgs.msg import Odometry, Path
    from sensor_msgs.msg import PointCloud, ChannelFloat32
    import tf
except ImportError as e:
    print(e)
    print("Continuing without ROS integration")
    ros = False
    pass

# ROS Import check wrapper
def requireros(func):
    def check_and_call(*args, **kwargs):
        if not ros:
            raise ImportError("ROS (rospy) Not supported/found")
        else:
            return func(*args, **kwargs)
    return check_and_call


#       ^v^v^v^v^v^v^v^v^v^v^v^v^v^v^v^v^v^v^v^v^v^v^v^v^v^v^v^v^v^v^v^v^v^v^v^v^



class DataStore:

    def __init__(self, host='localhost', db=0, port=6379):
        self.r = redis.StrictRedis(host=host, port=port, db=db)
```

```
        self.wt = whiptail.Whiptail()
        # Redis List and Channel name
        self.listname = "statestream"
        self.goallist = "goalstream"
        # ROS Topics
        self.posetopic = self.listname + "pose"
        self.odomtopic = self.listname + "odom"
        self.disttopic = self.listname + "dist"
        self.goaltopic = self.listname + "goal"

        # Test Redis connection
        self.r.ping()

        self.pp = pprint.PrettyPrinter(indent=4)


    def __del__(self):
        self.save()


    def keys(self):
        return self.r.keys()


    def push(self, pose, ranges=None):
        # Type duck between GenericState and a dict
        if isinstance(pose, GenericState):
            hmap = {
                't'    : pose.time,
                'x'    : pose.x,
                'y'    : pose.y,
                'theta': pose.theta
            }
        else:
            hmap = {
                't'    : pose['t'],
                'x'    : pose['x'],
                'y'    : pose['y'],
                'theta': pose['theta']
            }

        # Name we'll give this pose in Redis
        mapname = "pose-" + str(hmap['t'])

        # Build hashmap from given state class
        if ranges is not None and len(ranges) == 8:
            hmap.update({
                'r0' : float(ranges[0]),
                'r1' : float(ranges[1]),
                'r2' : float(ranges[2]),
                'r3' : float(ranges[3]),
                'r4' : float(ranges[4]),
                'r5' : float(ranges[5]),
                'r6' : float(ranges[6]),
                'r7' : float(ranges[7])
                })

        # For each part of the _python_ dict we
        # create a Redis hashmap using the time as index
        self.r.hmset(mapname, hmap)

        # We push a reference to this new hashmap onto
        # the statestream list
```

```
    self.r.lpush(self.listname, mapname)

    # Also publish onto a channel
    self.r.publish(self.listname, mapname)


def push_goal(self, path):
    # Expects a list of dicts
    # e.g. [{'x': 0.0, 'y':100.0},{'x': 1502.5, 'y': 1337.0}]

    # Clear out old points
    for key in self.r.lrange(self.goallist, 0, -1):
        self.r.delete(key)
    # Delete old list
    self.r.delete(self.goallist)

    pointnum = 0
    for point in path:
        pointname = "goal" + str(pointnum)
        pointnum += 1
        self.r.hmset(pointname, point)
        self.r.lpush(self.goallist, pointname)

    self.r.publish(self.goallist, self.goallist)


def sub(self):
    # Mostly a test method, subscribe and print
    sub = self.r.pubsub()
    sub.subscribe([self.listname, self.goallist])

    # Loop until stopped plotting the path
    try:
        for item in sub.listen():
            if self.r.exists(item['data']):
                data = self.r.hgetall(item['data'])
                item['data'] = data
            self.pp.pprint(item)
    except KeyboardInterrupt as e:
        print("Stopping...")
        pass


def get(self, start=0, stop=-1):
    # Returns a list in-order over the range
    keys = self.r.lrange(self.listname, int(start), int(stop))
    ret = list()

    # Pull all the hashmaps out of the store
    for key in keys:
        ret.append(self.r.hgetall(key))

    return ret


def get_dict(self, start=0, stop=-1):
    # Return a dictionary instead of a list
    lst = self.get(start, stop)
    ret = dict()
    for item in lst:
        # time as key, dict as value
        ret[int(item['t'])] = item
```

```
    return ret


def delete_before(self, time):
    # Remove data with a key from earlier than specified

    if time < 0:
        raise ValueError("Time must be positive you crazy person!")

    keys = self.r.lrange(self.listname, 0, -1) # All keys in our stream

    for key in keys:
        if int(key) <= int(time):
            # Remove from list
            self.r.lrem(self.listname, count=0, value=key)
            # delete the key (hashmap)
            self.r.delete(key)


def plot(self, start=0, stop=-1):
    self.static_plot(start, stop)


def live_plot(self):
    print("Deprecated, rospipe + rviz will work better")
    try:
        pubsub = self.r.pubsub()
        pubsub.subscribe([self.listname])
        plt.axis([-500, 500, -500, 500])
        plt.ion()
        while True:
            # Loop until stopped plotting the path
            for item in pubsub.listen():
                #print(item)
                if self.r.hexists(item['data'], 'x'):
                    data = self.r.hgetall(item['data'])
                    plt.scatter(float(data['x']), float(data['y']))
                    #print(str(data['x']) + " " + str(data['y']))
                    plt.show()
                    plt.pause(0.0001)

    except KeyboardInterrupt as e:
        print(e)


def static_plot(self, start=0, stop=-1):
    # Plot all existing data after a run
    data = self.get(start, stop)

    xs = []
    ys = []

    for point in data:
        xs.append(point['x'])
        ys.append(point['y'])

    plt.axis([-600, 600, -600, 600])
    plt.ion()
    plt.plot(xs, ys)
    plt.show()


@requireros
```

```
    def rospipe(self):

        print('''
 __    __    __      _______     _____     _____
/  \  /  \  /  \    |  ____ \   / ___ \   / ____|
\__/  \__/  \__/    | |    \ | | /   \ | | |
 __    __    __     | |    | | | |   | | | \____
/  \  /  \  /  \    | \___/ /  | |   | | \_____ \
\__/  \__/  \__/    |  __   \  | |   | |       \ |
 __    __    __     | |   \  | | |   | |       | |
/  \  /  \  /  \    | |    | | | \___/ |  ____/  |
\__/  \__/  \__/    |_|    |_|  \_____/  |______/
''')

        print("Piping Redis ---> ROS")
        # Pipe data out of Redis into ROS

        rg = ROSGenerator()

        pose_pub = rospy.Publisher(self.posetopic, PoseStamped, queue_size=100)
        odom_pub = rospy.Publisher(self.odomtopic, Odometry, queue_size=100)
        dist_pub = rospy.Publisher(self.disttopic, PointCloud, queue_size=100)
        goal_pub = rospy.Publisher(self.goaltopic, Path, queue_size=10)
        tbr = tf.TransformBroadcaster()

        rospy.init_node('talker', anonymous=True)

        sub = self.r.pubsub()
        sub.subscribe([self.listname, self.goallist])

        # Loop until stopped plotting the path
        for item in sub.listen():

            if rospy.is_shutdown():
                print("\nStopping rospipe...")
                break

            if item['type'] == "subscribe":
                rospy.loginfo("Subscribed successfully to " + item['channel'])
                continue

            try:
                if item['channel'] == self.listname:
                    # Stream of poses and distance data

                    # Pull from redis:
                    data = self.r.hgetall(item['data'])

                    required_keys = ['x','y','theta','t']
                    for key in required_keys:
                        if key not in data.keys():
                            raise KeyError("Missing key " + str(key) + " from hashmap " +
                                           str(data) + " on channel " + str(item['channel']))

                    # Generate a new Quaternion based on the robot's pose
                    quat = tf.transformations.quaternion_from_euler(0.0, 0.0,
                                                                    float(data['theta']))

                    # Generate new pose
                    pose = rg.gen_pose(data, quat)
                    pose_pub.publish(pose)

                    # Publish Khepera transform
```

```
                    tbr.sendTransform((pose.pose.position.x, pose.pose.position.y, 0),
                                      quat, rospy.Time.now(), "khepera", "map")

                    # Generate odometry data
                    odom = rg.gen_odom(data, quat)
                    odom_pub.publish(odom)

                    # Generate pointcloud of distances
                    dist = rg.gen_dist(data)
                    dist_pub.publish(dist)

                    rospy.loginfo(" Redis " + str(item['channel']) + " --> ROS")


                elif item['channel'] == self.goallist:
                    # Less frequent planning channel

                    # Pull co-ords from Redis
                    data = self.r.lrange(self.goallist, 0, -1)
                    path = rg.gen_path()
                    quat = tf.transformations.quaternion_from_euler(0.0, 0.0, 0.0)

                    for datapoint in data:
                        raw_pose = self.r.hgetall(datapoint)

                        required_keys = ['x','y']
                        for key in required_keys:
                            if key not in raw_pose.keys():
                                raise KeyError("Missing key " + str(key) + " from point " +
                                               str(raw_pose) + " on channel " +
                                               str(item['channel']))


                        this_pose = rg.gen_pose(raw_pose, quat)
                        path.poses.append(this_pose)

                    goal_pub.publish(path)
                    rospy.loginfo(" Redis " + str(item['channel']) + " --> ROS")


                else:
                    # If we get an unexpected channel, complain loudly
                    raise ValueError("Encountered unknown channel '" +
                                     str(item['channel']) + "'")


            except (KeyError, ValueError) as e:
                rospy.logwarn(str(e))
                pass


    def replay(self, limit=-1):
        # Replay data already stored, by re-publishing to the Redis channel
        print('''
  ____________________________
 /|..........................|
| |:      KHEPERA REWIND      :|
| |:       "Redis & Co."      :|
| |:     ,-.   _____   ,-.    :|
| |:    ( ')) [_____] ( '))   :|
|v|:     '-'   ' ' '   '-'    :|
|||:     ,_______________.     :|
|||...../::::o::::::o::::\.....|
```

```
 |^|..../:::O:::::::::O:::\....|
 |/'---/--------------------'---|
 '.___/ /====/ /=//=/ /====/____/
      '--------------------'

  <<<------------------------###
'''')
        data = self.r.lrange(self.listname, 0, limit)
        data.reverse()
        # ... this might take a while
        try:
            for epoch in data:
                print("epoch " + str(epoch))
                self.r.publish(self.listname, epoch)
                time.sleep(constants.MEASUREMENT_PERIOD_S)

        except KeyboardInterrupt as e:
            print(e)

        print("Replay done.")


    def _purge(self):
        # Clear out all (literally all) data held in Redis
        if self.wt.confirm("Really destroy all data in Redis store?\n\nThis is not undoable!\n(
     run FLUSHDB)",
                           default='no'):
            self.r.flushdb()
            print("!!! Flushed Redis Data Store !!!")
        else:
            print("Did not purge Redis Store")


    def save(self):
        # Copy DB to disk
        return self.r.save()


#       ^v^v^v^v^v^v^v^v^v^v^v^v^v^v^v^v^v^v^v^v^v^v^v^v^v^v^v^v^v^v^v^v^v^v^v^v^v^v^v^v^v^v^v^v^


# Assistant class, generates ROS Classes from data hashmap
@requireros
class ROSGenerator:

    def gen_pose(self, data, quat):
        pose = PoseStamped()

        pose.pose.orientation.x = quat[0]
        pose.pose.orientation.y = quat[1]
        pose.pose.orientation.z = quat[2]
        pose.pose.orientation.w = quat[3]

        pose.pose.position.x = float(data['x'])
        pose.pose.position.y = float(data['y'])
        pose.pose.position.z = 0.0

        pose.header.frame_id = "map"
        pose.header.stamp = rospy.Time.now()

        return pose
```

```
def gen_odom(self, data, quat):
    odom = Odometry()
    opose = PoseWithCovariance()

    opose.pose.orientation.x = quat[0]
    opose.pose.orientation.y = quat[1]
    opose.pose.orientation.z = quat[2]
    opose.pose.orientation.w = quat[3]

    opose.pose.position.x = float(data['x'])
    opose.pose.position.y = float(data['y'])
    opose.pose.position.z = 0.0

    odom.header.frame_id = "map"
    odom.header.stamp = rospy.Time.now()
    odom.pose = opose

    return odom


def gen_dist(self, data):
    # Publish a point cloud derived from the IR sensor activations
    dist = PointCloud()

    dist.header.frame_id = "khepera" # Tie to Khepera's frame of reference
    dist.header.stamp = rospy.Time.now()

    # Angles of the sensor from the X axis (in rad)
    sensor_angles = [
        0.5 * math.pi,   # Left perpendicular
        0.25 * math.pi,  # Left angled
        0.0,             # Left forward
        0.0,             # Right forward
        1.75 * math.pi,  # Right angled
        1.5 * math.pi,   # Right perpendicular
        math.pi,         # Back right
        math.pi          # Back left
    ]

    # Physical (x,y) offset of the sensor from the center of the bot in mm
    # Where x is forward, y is left lateral
    sensor_offsets = [
        ( 15.0,  25.0),  # Left perpendicular
        ( 20.0,  20.0),  # Left angled
        ( 27.0,   8.0),  # Left forward
        ( 27.0,  -8.0),  # Right forward
        ( 20.0, -20.0),  # Right angled
        ( 15.0, -25.0),  # Right perpendicular
        (-26.0, -10.0),  # Back right
        (-26.0,  10.0)   # Back left
    ]

    keys = ['r0','r1','r2','r3','r4','r5','r6','r7']
    for key in keys:
        if key not in data.keys():
            raise KeyError("No range data -- Missing key " +
                           str(key) + " " + str(data))

    pre_points = zip(keys, sensor_angles, sensor_offsets)
    points = []
    intensities = []
```

```
        # Basic trig, converts ranges to points relative to robot
        for point in pre_points:
            reading = float(data[point[0]])
            distance = self._ir_to_dist(reading)

            #print(str(point[0]) + ” at ” + str(distance))

            # Don’t render ’infinite’ distance
            if distance > 60.0:
                continue

            pt = Point32()

            # point[2] is the sensor’s coords relative to the robot
            # point[1] is the angle the sensor takes relative to the robot’s x axis

            pt.x = (distance * math.cos(point[1])) + point[2][0]
            pt.y = (distance * math.sin(point[1])) + point[2][1]
            pt.z = 0.0

            intensities.append(distance)
            points.append(pt)

        dist.points = points
        intensities_chan = ChannelFloat32()
        intensity_chan = ChannelFloat32()

        intensities_chan.name = ”intensities”
        intensities_chan.values = intensities
        intensity_chan.name = ”intensity”
        intensity_chan.values = [
            30.0,   # min intensity
            1.0,    # max intensity
            0.0,    # min color
            1.0     # max color
        ]

        dist.channels = [intensities_chan, intensity_chan]

        return dist


    def gen_path(self):
        path = Path()
        path.header.stamp = rospy.Time.now()
        path.header.frame_id = ”map”
        return path

    def gen_path_pose(self, data):
        pass


    def _ir_to_dist(self, reading):
        ’’’
        From solved equation:
        y = 1.074519 + (10.57748 - 1.074519)/(1 + ( x /70.42612)^69.9039)^0.02119919
        ’’’
        return 10.0 * ( 1.074519 + (10.57748 - 1.074519)
                        /
                        math.pow(1 + (math.pow((reading / 70.42612),69.9039)), 0.02119919 ))


#       ^v^v^v^v^v^v^v^v^v^v^v^v^v^v^v^v^v^v^v^v^v^v^v^v^v^v^v^v^v^v^v^v^v^v^v^
```

```
# Only run if we're invoked directly:
if __name__ == "__main__":

    args = sys.argv[1:]

    try:
        optlist, args = getopt.getopt(args,
                                      'ds:pre',
                                      ['delete','server=', 'plot', 'rospipe', 'replay'])
    except getopt.GetoptError:
        print("Invalid Option, correct usage:")
        print("-d or --delete   : Destroy all data held in Redis")
        print("-s or --server   : Hostname of redis server to use. Default localhost")
        print("-p or --plot     : Live plot of published data")
        print("-r or --rospipe  : Pipe redis messages into ROS topics")
        print("-e or --replay   : Take historical data from Redis and re-publish to channel")
        sys.exit(2)

    server = "localhost"

    # Pre-instantiation options
    for opt, arg in optlist:
        if opt in ('-s', '--server'):
            print("Using Redis server at '" + str(arg) + "'")
            server = str(arg)

    ds = DataStore(host=server)

    # Post-instantioation options
    for opt, arg in optlist:
        if opt in ('-d', '--delete'):
            ds._purge()

        elif opt in ('-p', '--plot'):
            ds.plot()

        elif opt in ('-r', '--rospipe'):
            ds.rospipe()

        elif opt in ('-e', '--replay'):
            ds.replay()
```

../../comms.py

```
#
#     - - - -  I A R  - - - -
#
#  s1311631         Angus Pearson
#  s1346981    Jevgenij Zubovskij
#

from __future__ import print_function
import serial # Documentation: http://pyserial.readthedocs.io/en/latest/index.html
import sys
from serial.tools import list_ports as list_ports
import time

class CommsReadException(Exception):
    pass
```

```
class Comms:

    port = serial.Serial()

    # Initialise the class, trying to open the Serial Port
    # with multiple levels of graceful failure.
    def __init__(self, port="/dev/ttyUSB0", baud=9600, timeout=1):

        self.port.baud = baud
        self.port.timeout = timeout

        try:
            self.port.port = port
            self.port.open()

        except serial.serialutil.SerialException as e1:
            print(e1)
            print("Trying alternate port...")

            try:
                self.port.port = "/dev/ttyUSB1"
                self.port.open()
                pass

            except serial.serialutil.SerialException as e2:
                print(e2)
                print("\n!!! CAN'T OPEN PORT !!!")

                # We can't open a port, enumerate them for the user
                print("\nAvailable Serial Ports:")
                for possible in list_ports.comports():
                    print(possible)
                print("")

            raise(e1)
        self.clear_port()


    def __del__(self):
        if self.port.is_open:
            self.port.close()

    def _parse_sensor(self, string):
        data = string.strip('\n\r').split(",")
        data_ints = [int(d) for d in data[1:]]
        return data_ints

    def clear_port(self):
        self.port.reset_input_buffer()

    # directly control motor speeds
    def drive(self, lspeed, rspeed):
        cmd = "D," + str(int(lspeed)) + "," + str(int(rspeed)) + "\n"
        print(cmd, end="")
        self.port.write(cmd)
        #print(self.port.readline(), end="")
        self.port.readline()

    # self-explanatory
    def stop(self):
        self.drive(0,0)

    # Return odometry (wheel rotation) data
```

```
def get_odo(self):
    self.port.write("H\n")
    odo = self._parse_sensor(self.port.readline())

    if len(odo) != 2:
        raise CommsReadException("Odometry wrong length")

    return odo

# Reset the robot's wheel counts to 0
def reset_odo(self):
    self.port.write("G,0,0\n")
    self.port.readline()

# Return IR Distance measurements
def get_ir(self, sensor_no=None):
    self.port.write("N\n")
    dist = self._parse_sensor(self.port.readline())

if len(dist) is not 8:
    raise CommsReadException("IR wrong length")

    if sensor_no is None:
        return dist
    else:
        return dist[sensor_no]


# Return IR Ambient Light Measurements
def get_ambient(self, sensor_no=None):
    self.port.write("O\n")
    amb = self._parse_sensor(self.port.readline())

if len(amb) is not 8:
    raise CommsReadException("IR wrong length")
return amb

# control status LEDs on the robot
def led(self, led_num=None, state=1):
    if state not in [0,1]:
        state = 0

    if led_num is None or led_num == 0:
        self.port.write("L,0," + str(state) + "\n")
    else:
        self.port.write("L,1," + str(state) + "\n")
self.port.readline()


def blinkyblink(self):
    self.led(0,1)
    self.led(1,0)
    time.sleep(0.1)
    self.led(0,0)
    self.led(1,1)
    time.sleep(0.1)
    self.led(0,1)
    self.led(1,0)
    time.sleep(0.1)
    self.led(0,0)
    self.led(1,1)
    time.sleep(0.1)
    self.led(0,1)
```

```
        self.led(1,0)
        time.sleep(0.1)
        self.led(0,0)
        self.led(1,1)
        time.sleep(0.1)
        self.led(0,0)
        self.led(1,0)
        time.sleep(0.1)
        self.clear_port()
```

../../state.py

```
#
#     - - - -  I A R  - - - -
#
#  s1311631         Angus Pearson
#  s1346981    Jevgenij Zubovskij
#

# Class to define state to not have arrays... cause arrays are ugly
class GenericState(object):

  def __init__(self):
    self.time = None
    # x location relative to initial placament at time of recording
    self.x = None
    # y location relative to initial placament at time of recording
    self.y = None
    # y location relative to initial placament at time of recording
    self.theta = None

# Subclass
class OtherState(GenericState):
  def __init__(self):
    GenericState.__init__(self)
```

../../bug_state.py

```
#
#     - - - -  I A R  - - - -
#
#  s1311631         Angus Pearson
#  s1346981    Jevgenij Zubovskij
#


import constants

class Bug_State:

    #absolute angle that we have before starting return algorithm
    theta_start = 0
    #number of control loop epochs that we have been exploring for
    exploration_cycle = 0
    #is the return algorithm activated
    algorithm_activated = False
    #is the point fro mwhich we need to head back towards (0,0) recorded
    algorithm_point = False

    #last closest recorded position on mline, closest in regards to distance to goals
    last_m_x = 0
    last_m_y = 0
    #have we reached the goal
```

```
    done = False

    #does the return algorithm have control
    in_control = True

    #two end points of the M-line
    m_line_start = [0,0] # 0,0 always, here as a variable for completeness
    m_line_end   = [0,0]
```

../../bug_algorithm.py

```
#!/usr/bin/env python

#
#     - - - -  I A R  - - - -
#
#  s1311631         Angus Pearson
#  s1346981    Jevgenij Zubovskij
#

from navigation_state import Navigation_State

import constants
import sys

import time
import math

class Bug_Algorithm:


    def lineMagnitude (self, x1, y1, x2, y2):
        lineMagnitude = math.sqrt(math.pow((x2 - x1), 2)+ math.pow((y2 - y1), 2))
        return lineMagnitude

    #Calc minimum distance from a point and a line segment (so if too far to reach it, we start a
      new heading, so we do not go in a full loop around the box)
    def DistancePointLine (self, px, py, x1, y1, x2, y2):

        LineMag = self.lineMagnitude(x1, y1, x2, y2)

        if LineMag < 0.00000001:
        DistancePointLine = 9999
        return DistancePointLine

        u1 = (((px - x1) * (x2 - x1)) + ((py - y1) * (y2 - y1)))
        u = u1 / (LineMag * LineMag)

        if (u < 0.00001) or (u > 1):
        #// closest point does not fall within the line segment, take the shorter distance
        #// to an endpoint
        ix = self.lineMagnitude(px, py, x1, y1)
        iy = self.lineMagnitude(px, py, x2, y2)

        if ix > iy:
            DistancePointLine = iy
        else:
            DistancePointLine = ix
        else:
        # Intersecting point is on the line, use the formula
        ix = x1 + u * (x2 - x1)
        iy = y1 + u * (y2 - y1)
        DistancePointLine = self.lineMagnitude(px, py, ix, iy)
```

```
    return DistancePointLine


#check if c is between a and b
def is_on_mline(self,a, b, c):

    #thresholded cross product to account for non-ideal odometry
    crossproduct = (c[1] - a[1]) * (b[0] - a[0]) - (c[0] - a[0]) * (b[1] - a[1])
    if abs(crossproduct) > 20 : return False

    #thresholded dot product to account for non-ideal odometry
    dotproduct = (c[0] - a[0]) * (b[0] - a[0]) + (c[1] - a[1])*(b[1] - a[1])
    if dotproduct < -20 : return False


    #thresholded to account for non-ideal odometry
    squaredlengthba = (b[0] - a[0])*(b[0] - a[0]) + (b[1] - a[1])*(b[1] - a[1])
    if dotproduct > squaredlengthba + 20: return False

    return True


#vector magnitude calculator
def vector_magnitude(self, vector):
    return math.sqrt( math.pow(vector[0],2) + math.pow(vector[1],2))


#normalizes angle in degrees to -180 : 180 degrees
    def normalize_angle(self, angle):
    if angle < 0:
        angle = angle % -360
        if angle < -180:
        angle = 360 + angle
    else:
        angle = angle % 360
        if angle > 180:
        angle = -(360 - angle)
    return angle

#vector difference calculator
def vector_diff(self, vector_1, vector_2):
    dx = vector_1[0] - vector_2[0]*1.0
    dy = vector_1[1] - vector_2[1]*1.0
    return [dx,dy]


#get angle while ON the M-line
def get_angle_on_m(self, odo_state, bug_state):
    direction = self.vector_diff( bug_state.m_line_start,  bug_state.m_line_end)
    direction_angle = math.atan2(direction[1] , direction[0])
    #if no difference, well then we never left the spot
    vector_magnitude = self.vector_magnitude(direction)
    if vector_magnitude == 0:
        return 0

    direction_angle = math.degrees(direction_angle) - self.normalize_angle(math.degrees(
 odo_state.theta))
    return self.normalize_angle(direction_angle)

#return new state
    def new_state(self, nav_state, odo_state, bug_state):
```

```
            speed_l = nav_state.speed_l
        speed_r = nav_state.speed_r

    #do not turn if not needed

    turn_less = constants.CONST_SPEED
    turn_more = constants.CONST_SPEED


    #set up more convenient variables for checking distances, use integers to account for
rounding errors
    current_pos = [int(odo_state.x), int(odo_state.y)]
    previous_pos = [int(bug_state.last_m_x) , int(bug_state.last_m_y)]
    line_start = [ int(bug_state.m_line_start[0]) , int(bug_state.m_line_start[1]) ]

    #if we are on the mline again
    on_mline = self.is_on_mline(line_start, previous_pos, current_pos)
    closer_on_mline = False

    #new distance to the origin
    new_distance = self.vector_magnitude(current_pos)
    old_distance =  self.vector_magnitude(previous_pos)


    #distance to M-line (more accurately)
    dist_to_mline = self.DistancePointLine(odo_state.x , odo_state.y, bug_state.m_line_start
[0] , bug_state.m_line_start[1] , bug_state.m_line_end[0], bug_state.m_line_end[1])

    on_mline =  dist_to_mline <= constants.ON_MLINE or on_mline


    #check if we reached the destination
    if new_distance < 40:
        bug_state.done = True

    #if far away from M-line, different constants for free space and wall following
    far_in_open_space = dist_to_mline > constants.M_DISTANCE * 4.0 and nav_state.system_state
 == constants.STATE_DRIVE_FORWARD
    far_on_wall = dist_to_mline > constants.M_DISTANCE*10.0
    #recalculate M-line if such a need arises
    if far_in_open_space or far_on_wall:

        #renew m-line record
        bug_state.m_line_end  = [odo_state.x , odo_state.y]
        bug_state.m_line_start= [0,0]

        #update last position on line
        bug_state.last_m_x = odo_state.x
        bug_state.last_m_y = odo_state.y

        #turn aggressively
        turn_less = -constants.CONST_SPEED
        turn_more = constants.CONST_SPEED

    #Try keeping the robot on the M-line (be it old one or updated one)
    if True:
        #initially assume we are not following a wall
        is_wall_following = False
        #then assign it a value, for clarity in this case
        is_wall_following = nav_state.system_state == constants.STATE_RIGHT_FOLLOW and
nav_state.system_state == constants.STATE_LEFT_FOLLOW
        #If are in free space
        if nav_state.system_state == constants.STATE_DRIVE_FORWARD:
```

```
                turn_less = -constants.CONST_SPEED
                turn_more = constants.CONST_SPEED

                #record new closer position on m-line if we are getting closer along it
                if on_mline and new_distance <= old_distance:
                    bug_state.last_m_x = odo_state.x
                    bug_state.last_m_y = odo_state.y


            # if can leave a wall closer on m-line
            elif new_distance <= old_distance and is_wall_following and on_mline:

                #turn aggressively
                turn_less = -constants.CONST_SPEED
                turn_more = constants.CONST_SPEED

                #make this the new closest we have gotten along m-line
                bug_state.last_m_x = odo_state.x
                bug_state.last_m_y = odo_state.y
                closer_on_mline = True


            angle_to_m = self.get_angle_on_m(odo_state, bug_state)

            #OUR angle too small
            if angle_to_m > constants.M_N_ANGLE:

                #if are following a wall and are closer along m-line to goal
                #or are not wall following (are in free space)
                if closer_on_mline or not is_wall_following:
                    # check if there is a wall on the left
                    if nav_state.system_state is not constants.STATE_LEFT_FOLLOW:
                        # if there is none, turn left
                        speed_r = turn_more
                        speed_l = turn_less

            #OUR angle too big
            elif angle_to_m < -constants.M_N_ANGLE:


                #if are following a wall and are closer along m-line to goal
                #or are not wall following (are in free space)
                if closer_on_mline or not is_wall_following:
                    # check if there is a wall on the right
                    if nav_state.system_state is not constants.STATE_RIGHT_FOLLOW:
                        # if there is none, turn right
                        speed_r = turn_less
                        speed_l = turn_more

        #send out the new speed controls
        nav_state.speed_l = speed_l
        nav_state.speed_r = speed_r
        return nav_state
```

../../navigation_state.py

```
#
#     - - - -  I A R  - - - -
#
#  s1311631         Angus Pearson
#  s1346981    Jevgenij Zubovskij
#
```

```
import constants

class Navigation_State:

    #control loop epochs that the robot was geting ”bored” for
    boredom_counter = 0
    #is the robot done turning from boredom causing wall / object
    boredom_turn_counter = 0
    #constants speed initla conditions
    speed_l = constants.CONST_SPEED
    speed_r = constants.CONST_SPEED
    #initially drive forward
    system_state = constants.STATE_DRIVE_FORWARD
    #IR sensor readings
    dist = [0]*8
```

../../navigation_algorithm.py

```
#!/usr/bin/env python

#
#     - - - -  I A R  - - - -
#
#  s1311631         Angus Pearson
#  s1346981    Jevgenij Zubovskij
#

from navigation_state import Navigation_State

import constants
import sys

import time
import math

class Navigation_Algorithm:


    # check if we are stuck
    def is_stuck(self, dist):
        #check if we are scraping on the sides
        # multiple of 1.2 as 1.0 is handled by following
        stuck_cone_left  = dist[1] > constants.CONST_WALL_DIST
        # multiple of 1.2 as 1.0 is handled by following
        stuck_cone_right = dist[4] > constants.CONST_WALL_DIST
        #check if we are about to be stuck in the front
        stuck_cone_front = dist[2] > constants.CONST_WALL_DIST*0.5 or dist[3]  > constants.
     CONST_WALL_DIST*0.5

        return stuck_cone_left or stuck_cone_right or stuck_cone_front


    # check if we ”see” the left wall and it is closer than the one on the right
    def should_follow_left_wall(self, dist, system_state):
        within_range =  dist[0] > constants.CONST_WALL_DIST * 0.5
        #to not switch wall if two walls nearby
        followed_right = system_state == constants.STATE_RIGHT_FOLLOW
        return within_range and dist[0] > dist[5] and not followed_right


    # check if we ”see” the right wall and it is closer than the one on the left
    def should_follow_right_wall(self, dist, system_state):
        within_range =  dist[5] > constants.CONST_WALL_DIST * 0.5
```

```
    followed_left = system_state == constants.STATE_LEFT_FOLLOW
    return within_range and not(dist[0] > dist[5]) and not followed_left

# check if we are over the distance threshold w.r.t. object on the left
def too_close_to_left(self, dist):

    distance_close   = dist[0] > constants.CONST_WALL_DIST
    return distance_close


# check if we are over the distance threshold w.r.t. object on the right
def too_close_to_right(self, dist):

    distance_close   = dist[5] > constants.CONST_WALL_DIST
    return distance_close



# check if we are under the distance threshold w.r.t. object on the right
def is_away_from_right(self, dist):

    wall_in_range = dist[5] < constants.CONST_WALL_DIST * 0.8
    return wall_in_range


# check if we are under the distance threshold w.r.t. object on the left
def is_away_from_left(self, dist):

    wall_in_range = dist[0] < constants.CONST_WALL_DIST * 0.8
    return wall_in_range


# check if there is more space on the right of the robot than on the left
def is_more_space_on_right(self, dist):

    values_on_right = dist[3] + dist[4] + dist[5]
    values_on_left  = dist[1] + dist[2] + dist[0]

    return (values_on_left > values_on_right)


# check if the system is being unstuck
def is_being_unstuck(self, state):

    return (state is constants.STATE_STUCK_LEFT) or (state is constants.STATE_STUCK_RIGHT)


# check if it is more beneficial to unstuck by turning to the right
def should_unstuck_right(self, dist, system_state):
    stuck_on_left = system_state == constants.STATE_LEFT_FOLLOW
    return stuck_on_left or self.is_more_space_on_right(dist)

# check if robot is ”bored”
def bored(self, boredom_counter):
    return boredom_counter >= constants.CONST_WALL_BORED_MAX

# check if we are handling boredom
def is_boredom_handled(self, state):
    return state == constants.STATE_BOREDOM_ROTATE or state == constants.STATE_BOREDOM_DRIVE


    def new_state(self, nav_state, odo_state, bug_state, comms, bug):
```

```
    result = Navigation_State()
    result = nav_state


    #variable to indicate if reactive avoidance is in control of the robot
    bug_state.in_control = False


       ########################
       #HANDLE THE BUG OPERATIONS
       ########################

    if bug_state.exploration_cycle < constants.EXPLORATION_CYCLES:
    bug_state.exploration_cycle += 1

    #now that we have explored enough, can record our position from where we need to return
    elif (not bug_state.algorithm_point):

    comms.led(0,1)
    print("beginning turn")
    #record the line parameters
    bug_state.m_line_end = [odo_state.x, odo_state.y]
    bug_state.m_line_start = [0,0]

    #record last position on the M-line
    bug_state.last_m_x = odo_state.x
    bug_state.last_m_y = odo_state.y
    #record that we have already recorded said values

    bug_state.algorithm_point = True
    #do the turn towards the goal, called 180 because we were going away from the goal and
now are going towards it
    nav_state.system_state = constants.STATE_BUG_180

    #now, if have not yet turned back towards the goal
    elif not bug_state.algorithm_activated:
    #check if that is the case now
    angle = bug.get_angle_on_m(odo_state, bug_state)

    #thresholded value due to imperfect sensor reading timings
    pointing_at_origin = abs(angle) < 10
    #if done turning
    if nav_state.system_state == constants.STATE_BUG_180 and pointing_at_origin:
        #if done turning, then can proceed with the return algorithm
        nav_state.system_state = constants.STATE_DRIVE_FORWARD
        #reset boredom
            result.boredom_counter = 0
        bug_state.algorithm_activated = True
    else:
       #if not done turing, check which way we need to turn to reduce the angle difference
       #between the M-line (directed towards (0,0)) and the pose of the robot
       if angle > 10:
        result.speed_l = -constants.CONST_SPEED
                result.speed_r = constants.CONST_SPEED
       elif angle < -10:
        result.speed_r = -constants.CONST_SPEED
                result.speed_l = constants.CONST_SPEED
       #continue turning
       nav_state.system_state = constants.STATE_BUG_180
       return result

        ########################
        #HANDLE BOREDOM COUNTER
```

```
        ########################

    #make sure boredom never activates durin bug algorithm
    if not(bug_state.algorithm_activated):

        #record that we are still moving along a wall and not "exploring"
        if result.system_state == constants.STATE_LEFT_FOLLOW or result.system_state ==
constants.STATE_RIGHT_FOLLOW:
            result.boredom_counter +=1


        #check if robot is "bored" and it is not being handled
        if self.bored(result.boredom_counter) and not self.is_boredom_handled(result.
system_state):

            #turn away from the wall that was last followed
            if result.system_state == constants.STATE_LEFT_FOLLOW:
            result.speed_l = constants.CONST_SPEED
                result.speed_r = -constants.CONST_SPEED
            elif system_state == constants.STATE_RIGHT_FOLLOW:
                result.speed_l = -constants.CONST_SPEED
                result.speed_r = constants.CONST_SPEED

            # reset state
            result.boredom_turn_counter = 0
            result.boredom_counter = 0

            # set state
            result.system_state = constants.STATE_BOREDOM_ROTATE

        # if we are rotating on the spot
        elif result.system_state == constants.STATE_BOREDOM_ROTATE:
            #check if we are done rotating
            if result.boredom_turn_counter >= constants.CONST_BORED_TURN_MAX:
                #different from normal forward driving as we try to get very far from the
wall
                result.system_state = constants.STATE_BOREDOM_DRIVE

            result.speed_l = constants.CONST_SPEED
            result.speed_r = constants.CONST_SPEED

            # otherwise, continue rotationg for this loop iteration
            else:
                result.boredom_turn_counter += 1
            #turn until done turning
            return result


        ############################
        # IF STUCK
        ############################

        if self.is_stuck(result.dist):

             # do not interrupt if already handle
             if self.is_being_unstuck(result.system_state):
        return result

             # determine direction of where better to turn to unstuck
             if self.should_unstuck_right(result.dist, result.system_state):

                  result.system_state = constants.STATE_STUCK_RIGHT
          result.speed_l = constants.CONST_SPEED
```

```
            result.speed_r = -constants.CONST_SPEED

               else:

                     result.system_state = constants.STATE_STUCK_LEFT
            result.speed_l = -constants.CONST_SPEED
            result.speed_r = constants.CONST_SPEED


               # stop following the wall (as it could ahve potentially led to being stuck)
               result.boredom_counter = 0
        return result


           ########################
           #HANDLE BOREDOM DRIVE
           ########################

           # if robot not stuck and we are driving away from a "boring" wall, continue doing so
           if result.system_state == constants.STATE_BOREDOM_DRIVE and not (bug_state.
algorithm_activated):
               return result

           ####################
           ##WALL FOLLOWING LEFT
           ####################

           if self.should_follow_left_wall(result.dist, result.system_state):

       #reactive control not activated, can use the return algorithm
       bug_state.in_control = True

       turn_least = constants.CONST_SPEED * constants.TURN_LESS
       turn_most  = constants.CONST_SPEED * constants.TURN_MORE
       no_turn    = constants.CONST_SPEED
       speed_r = result.speed_r
       speed_l = result.speed_l

               # set state accordingly
               result.system_state = constants.STATE_LEFT_FOLLOW

               # keep the distance within the threshold range
               if self.too_close_to_left(result.dist) and not (speed_l == turn_most and speed_r
== turn_least):

                   speed_l = turn_most
                   speed_r = turn_least

               elif self.is_away_from_left(result.dist) and not (speed_l == turn_least and
speed_r == turn_most):

                   speed_l = turn_least
                   speed_r = turn_most

               elif not (speed_l == no_turn and speed_r == no_turn):

                   speed_l = no_turn
                   speed_r = no_turn

       result.speed_r = speed_r
               result.speed_l = speed_l

           ####################
```

```
            ##WALL FOLLOWING RIGHT
            ######################
            elif self.should_follow_right_wall(result.dist, result.system_state):

        #reactive control not activated, can use the return algorithm
        bug_state.in_control = True

                #print("following right")

        turn_least = constants.CONST_SPEED * constants.TURN_LESS
        turn_most  = constants.CONST_SPEED * constants.TURN_MORE
        no_turn    = constants.CONST_SPEED
        speed_r = result.speed_r
        speed_l = result.speed_l

                # set state accordingly
                result.system_state = constants.STATE_RIGHT_FOLLOW

                # keep the distance within the threshold range
                if self.too_close_to_right(result.dist) and not (speed_l == turn_least and
    speed_r == turn_most):

                    speed_l = turn_least
                    speed_r = turn_most

                elif self.is_away_from_right(result.dist) and not (speed_l == turn_most and
    speed_r == turn_least):

                    speed_l = turn_most
                    speed_r = turn_least

                elif not (speed_l == no_turn and speed_r == no_turn):

                    speed_l = no_turn
                    speed_r = no_turn

        result.speed_r = speed_r
                result.speed_l = speed_l

            ######################
            # IF NONE OF THE ABOVE
            ######################
            else:
            #reactive control not activated, can use the return algorithm
            bug_state.in_control = True

                    # reset variables as not doing anything
                    result.boredom_counter = 0

                    # set state accordingly
                    result.system_state = constants.STATE_DRIVE_FORWARD
                result.speed_l = constants.CONST_SPEED
            result.speed_r = constants.CONST_SPEED



        return result
```

../../odometry_algorithm.py

```
#!/usr/bin/env python

#
#     - - - -  I A R  - - - -
```

```
#
#  s1311631         Angus Pearson
#  s1346981    Jevgenij Zubovskij
#


from odometry_state import Odometry_State
import constants


import math

import sys
import time

class Odometry_Algorithm:
        def __init__(self):
                pass

    #calculate differences in distance driven by different wheels
    def delta_s(self, delta_odo):

        result      = 0
        result      = ((delta_odo[0] + delta_odo[1]) / float(2) ) / constants.TICKS_PER_MM

        return result # mm

    #calculate the orientation angle
    def delta_theta(self, delta_odo):

        result      = 0
        result      = ((delta_odo[1] - delta_odo[0]) / constants.TICKS_PER_MM) / constants.
     WHEEL_BASE_MM


        return result #radians

    #calculate the change in angle, X and Y
    def delta_x_y_angle(self, curr_theta, delta_odo):
        result      = [0]*3

        delta_dist  = self.delta_s(delta_odo)
        delta_angle = self.delta_theta(delta_odo)


        new_angle = curr_theta + delta_angle / float(2) # this is the alternative


        delta_x     = delta_dist*math.cos(new_angle) # in mm
            delta_y     = delta_dist*math.sin(new_angle) # in mm


        result[0] = delta_x
        result[1] = delta_y
        result[2] = delta_angle


        return result


    #get the new state
    def new_state(self, prev_state, new_odo):
```

```
        delta_odo = [new_odo[0] - prev_state.odo[0], new_odo[1] - prev_state.odo[1] ]
        state_change = self.delta_x_y_angle(prev_state.theta, delta_odo)

        #update the variables
        prev_x = prev_state.x
        prev_y = prev_state.y
        prev_theta = prev_state.theta
        t = constants.MEASUREMENT_PERIOD_S

        #calculate the new values
        x_n     = prev_x     + state_change[0]
        y_n     = prev_y     + state_change[1]
        theta_n = prev_theta + state_change[2]



        #return new state
        result       = Odometry_State()
        result.time  = prev_state.time + t
        result.x     = x_n
        result.y     = y_n
        result.odo   = new_odo
        result.theta = theta_n

        return result
```

../../odometry_state.py

```
#
#     - - - -  I A R  - - - -
#
#  s1311631         Angus Pearson
#  s1346981    Jevgenij Zubovskij
#


from state import GenericState

class Odometry_State(GenericState):
  def __init__(self):
    GenericState.__init__(self)
    # time when state recorded
    self.time   = 0
    # x location relative to initial placement at time of recording
    self.x  = 0
    # y location relative to initial placement at time of recording
    self.y  = 0
    # y location relative to initial placement at time of recording
    self.theta  = 0

    self.odo = [0]*2
    #ir values
    self.nav = [0]*8
```

../../constants.py

```
#
#     - - - -  I A R  - - - -
#
#  s1311631         Angus Pearson
#  s1346981    Jevgenij Zubovskij
#

#REACTIVE THRESHOLDS
```

```
CONST_SPEED = 8
CONST_WALL_DIST = 200

#BOREDOM CONSTANTS

CONST_WALL_BORED_MAX = 100
CONST_BORED_TURN_MAX = 20

#TURN SCALING

TURN_LESS = 0.2
TURN_MORE = 1.0

#REACTIVE STATES

STATE_DRIVE_FORWARD = 0
STATE_DRIVE_BACKWARD = 1
STATE_STUCK_LEFT = 2
STATE_STUCK_RIGHT = 3
STATE_LEFT_FOLLOW = 4
STATE_RIGHT_FOLLOW   = 5
STATE_BOREDOM_ROTATE = 6
STATE_BOREDOM_DRIVE  = 7

#called 180 because we were going away from the goal and now are turning towards it
STATE_BUG_180 = 8


#ENCODER_CONSTANTS

TICKS_PER_MM = 12.0    # encoder ticks per milimeter
TICKS_PER_M = TICKS_PER_MM * 1000.0 # encoder ticks per meter

# PHYSICAL CONSTANTS

WHEEL_BASE_MM = 56.0 # mm
WHEEL_BASE_M = WHEEL_BASE_MM / 1000.0 # m



MEASUREMENT_PERIOD_S = 0.05 # s

#Not exactly as expected as 30 seconds is not just delay, but also computations, hence value
     below is 15, not 30
EXPLORATION_CYCLES = 10.0 / MEASUREMENT_PERIOD_S


# SENSOR CONSTANTS

DIST_CUTOFF = 0.0

#Unscaled distance gradient at which we should recompute the M-line
M_DISTANCE = 10 # mm

#Angle threshold until we begin correcting out orientation along the M-line
M_N_ANGLE  = 10 # degrees

#Distance at which we still consider ourselves to be following the M-line
ON_MLINE   = 30 #mm
```

../../requirements.txt

```
cycler==0.10.0
```

```
matplotlib==1.5.3
numpy==1.11.2
pyparsing==2.1.10
pyserial==3.1.1
python-dateutil==2.5.3
pytz==2016.7
PyYAML==3.12
redis==2.10.5
six==1.10.0
whiptail==0.2
prettyprint==0.1.5
```