A Virtual Search & Rescue Robot

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

***Abstract*—** This report presents a virtual solution designed to simulate capabilities required of Search and Rescue robots. These include obstacle avoidance, sensor data fusion, environment mapping, precision navigation and victim location. A Python control layer commands a Pioneer P3DX robot modelled in VREP. During victim search ultrasonic sensor data builds an occupancy grid (OG) of free and occupied space probability. The A* path finding algorithm uses the OG to propose an optimal path from victim to home point. All task solutions are successful. Recommendations include performance optimization of grid mapping and resolving identified environment feature missing/*overexposure* issues affecting route planning.

Index – Mobile Robots, Search and Rescue, Simulation

I. INTRODUCTION

Many environments containing natural, radiological, chemical, hydro, structural and other hazards are often extremely challenging and too dangerous for humans to operate in. As the authors of [1] state, "*Shortage of skilled rescue workforces, as well as the risks involved in search and rescue operations, are becoming foremost problems during an emergency situation*". In such demanding conditions with critical resource and time constraints, robots have replaced humans in completing many tasks including search and rescue (SAR), local environment analysis and sampling, material handling and manipulation of valves and levers, acting as "*remote sensing devices reporting information from dangerous places that human operators cannot easily and/or safely reach*" [2].

On 26th April 1986 the Chernobyl unit 4 nuclear reactor exploded, killing 32 from the blast and acute radiation syndrome (ARS), while an estimated many thousands suffered from long-latency cancers [3]. 10 years later the Pioneer robot (Fig. 1), funded by NASA and the U.S. DoE (Department of Energy), and designed by a team of robotics, hardware and software experts from JPL (Jet Propulsion Laboratory), Carnegie Mellon University, Livermore laboratory, Silicon Graphics and other partners was deployed to Chernobyl to help assess the site and it's significantly deteriorating first concrete sarcophagus [4]. Up to this point Ukrainian site workers had carried out assessments in contaminated areas themselves despite severe risk to their health.

As a small, tethered and tracked bulldozer measuring 4ftx3ft able to negotiate rough terrain including stairways, with vital, sensitive computing equipment protected behind lead-lined walls at one end of the tether, and imaging technology borrowed from NASA's Pathfinder project, the Pioneer was tasked with answering such questions as *"How much water leaks in or condenses inside"*, and "*how hot is the water when it drains out*" [4]. Data from the Pioneer robot played a critical role in the ongoing assessment of structural, material and water conditions at the disaster site given that a second, New Safe Confinement (NSC) structure was built and completed operational testing in April 2019 [5].

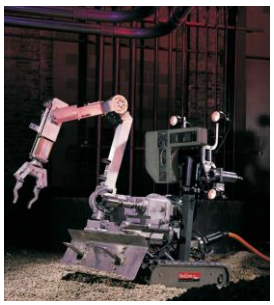

Fig. 1 Pioneer robot

Fig. 2 WTC site box beam explored by robot

The first known use of robots for urban search and rescue (USAR) was 6 hours after the World Trade Centre (WTC) disaster, used in the search of victims, identification of efficient excavation paths through rubble, structural inspection and detection of hazardous materials and explosive gases [6]. With such a huge and dangerous amount of concrete and steel beam debris, the small robots were able to outperform alternatives, able to navigate as far as 20m into the interior of a rubble pile vs 2m with pole-mounted cameras, and manoeuvre into spaces too small for rescue dogs or humans, or where extreme heat and jet-fuel fire was present. An example of the extreme nature of the search site is given in Fig. 2 which highlights one end of a box beam, into which the robot began searching for survivable voids. In this particular case 3 sets of human remains were found [6]. Assessing structural conditions was also a key task. The most frequently used robot was the Inuktun micro-TRAC [7], a small, single track platform more typically used for pipe and duct inspection, selected mainly due to its ability to be carried to site in a backpack and fitting into the small voids to be searched. Although the robots found no survivors, what they did succeed in was acceptance by the rescue community [6].

What the Chernobyl Pioneer and WTC micro-TRAC had in common was human control by tele operation, with communication and power via tether, also considered a necessary safety rope designed to withstand a level of pulling to retrieve a stuck, flipped or failed platform. In the case of WTC site operations, a second, exposed operator had to continuously interact with the tether to avoid tangling around steel bars. Of course, a common constraint is maximum operational range limited by tether length, a significant disadvantage for the Pioneer operators working so closely to an exposed nuclear reactor. There were WTC robot operational errors, due to physical and cognitive fatigue, and the limitations of human perception especially in unfamiliar environments, included oversteering, missed remains only identified later on video reviews and unnecessary adjusting of headlights despite auto gain on video cameras [6].

After the Great Hanshin Earthquake of 1995 where an estimated 6434 people lost their lives [8], the Japanese government looked to resolve challenges with SAR operations and hence promoted "*the development of intelligent, highly mobile, dexterous robots that can improve the safety and effectiveness of emergency responders performing hazardous tasks*" [9] through sponsoring of prizes in RoboCupRescue competitions. The competition's primary task is the location of simulated victims that show sign of life, within a maze of harsh terrain. A map is generated by the robot which should represent landmarks, obstacles and victims.

Code sharing of algorithms is encouraged to foster greater scientific community learning and continued research including the domain of multiagent communication and coordination. Scientific output as a result of league participation has been used in mapping components of a robotic bomb disposal platform, and the robot Quince, developed by Japanese universities, used to inspect the Fukushima Daiichi nuclear power plant following hydrogen explosions, nuclear meltdowns and radioactive material release triggered by a massive earthquake and tsunami on Friday 11th March 2001 [9] [10].

The overall goal of this study is the design and implementation of a virtual robot capable of departing a home point (HP) to explore an indoor environment in search of a "*victim*" target then navigating safely back to starting location, building a map of its surroundings and avoiding all obstacles. It will act autonomously, simulating complete onboard control without human intervention. It will also need to exhibit some key characteristics taking it beyond the capabilities of the Chernobyl and WTC robots, which by the very nature of their human

operation exposed their controllers to serious safety risks and limited their mission performance. Drawing inspiration from contemporary RoboCupRescue participants which demonstrate the ability to sense and map the environment, avoid obstacles, navigate safe passage and locate targets beyond immediate sensory range, this study virtual robot will focus on satisfying the following tasks:

T0 Map the environment
T1 Navigate to the centre of the HP
T2 Exit the HP via the doorway cleanly without touching walls
T3 Avoid all obstacles
T4 Explore the environment to locate the "*target*" beacon
T5 Return to HP centre

The remainder of this paper is organized as follows. Section II introduces the "*base*" environment in which the simulated robot platform, detailed in Section III, will operate. Section IV presents the architectural elements of the solution, while Section V explains the behavioural design. Section VI details the methodology behind the experimental design, followed by Section VII which presents simulation results. Section VIII concludes with final thoughts.

II. Base Simulated Environment

To further illustrate the tasks to be completed, a base scene modelled in VREP is presented in Fig. 3. The start (T1) and completion (T5) location of each simulation run is the HP, with passage to the outer room via doorway T2. T4 represents the mission primary objective *target* which might be a disaster victim, point or material of interest. This base scene will be adapted with configurations of hazardous, impassable obstacles (T3) around which the robot will need to navigate to reach T4. In the course of navigating to achieve mission objectives the robot will assemble an environment map of infrastructure and obstacles that will then be used to plot an optimised path back to HP.

Fig. 3 VREP base scene with tasks


Fig. 4 Pioneer P3DX robot

III. Pioneer P3DX Robot Model

The well-known Pioneer P3DX robotics research platform [11], not to be confused with the Chernobyl Pioneer, is modelled in VREP and used in this study in basic configuration without arms or grippers (Fig. 4). Moving the platform are 2 high-speed, reversible DC motors that each independently drive a large wheel located on the side of the robot body. A positive input value rotates the motor joint in a forward direction, a negative value for reverse and 0 to stop. A 3rd passive, smaller wheel aids stability.

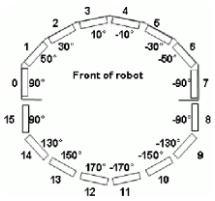

Fig. 5 Sensor array

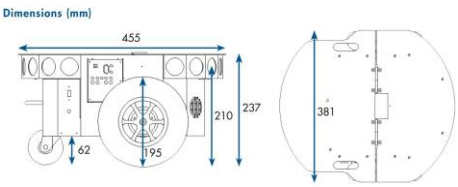

Fig. 6 Pioneer P3DX dimensions

The Pioneer has an effective ring of sensors, combining both a front and rear ultrasonic array of 8 sensors (Fig. 5). The sensors exhibit linear characteristics [12] and have a default object detection range of 1m. Advancing earlier wall-following solutions where only a subset of available sensors were used [12] [13], it is intended to use all sensors for a full 360° view, enabling extended coverage environment mapping and obstacle detection. Robot body dimensions (Fig. 6) will influence minimum navigable distance between it, obstacles and walls, therefore needs to be considered. It is important to note the names of the 16 ultrasonic sensors in VREP were amended to start with a 0 index using a 2-digit suffix. This aligns sensor naming with Fig. 5 and guarantees sensor ordering and correct additional characteristics such as angle position are retrieved when accessed by index.

IV. Architecture Design

A. Control Cycle

The "*sense-think-act*" control model is used to manage both the higher level execution of tasks *T0-5*, and lower level reactive behaviour to rules and conditions within each task. Processing an ordered sequence of planned tasks in turn, the "*sense*" phase continuously refreshes its perception of the environment and location with it from proximity sensors, derived bearing and odometry, with all data stored in a centrally available state repository.

This supports fast "*thinking*" what is next and may include aborting the current task due to impending collision, calculation of safe navigation paths, logging of important location waypoints and calculation of distance and bearing to objectives. State information continues to be updated to ensure the latest, most accurate representation of robot and environment for successful task execution.

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 or correcting navigational errors to bring the robot closer to the current task objective such as moving to the centre of HP or rotating to target bearing. When a task is complete its status is updated, informing the control cycle to proceed with the next.

B. Control Architecture

The robot will execute tasks T1-5 in sequence while continuously mapping the surrounding environment and avoiding obstacles. The *sense-think-act* control cycle will be supported by a reactive control strategy using a rule-based system consuming up-to-date state information that allows the immediate direction of robot behaviour in reaction to environment events. Fig. 7 presents the system control architecture in which for every loop of the control cycle the environment perception is refreshed before progressing an ordered collection of tasks as configured in a mission scenario.

An integrated suite of specialized modules together produces the robot's overall behaviour. The mission scenario of tasks is specified (yellow) before the first phase of the control cycle (blue) "*senses*" the environment (purple), before "*thinking*" on how to "*act*" (red), such as calculating navigation errors and setting motor velocity appropriately. Mapping is continuous (grey). Path planning includes a stochastic random wander searching the environment for the target beacon and a deterministic A* method planning shortest path back to HP.

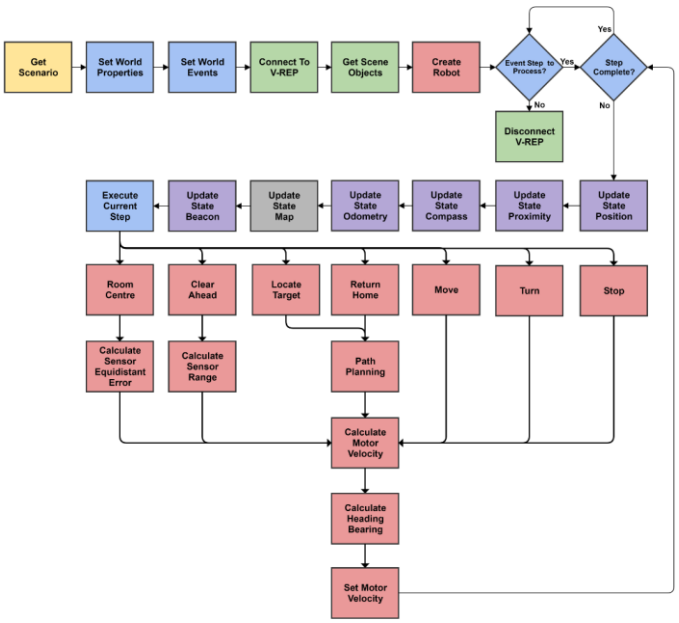

Fig. 7 High-level system diagram

C. Control Implementation

The client software architecture, written in Python to simulate onboard hardware management of sensors, motors and tasks, and first developed for a wall-following mission [12], is presented in Fig. 8. The existing framework was used as its design makes it easy to incorporate new scenarios by way of plugin methods handling a specific task. It should be noted that in the rest of the paper any example code removes reference to "*self*" for the sake of brevity.

Each coloured block on the Python client side encapsulates specific concerns, for example the "*sense-think-act*" cycle is handled by *controller.py*. Component *robots.py* is now extended with additional actions to solve tasks T1, T2, T4 and T5. A new mapper component to address task T0 generates a world map for both human visualisation and autonomous navigation path planning.

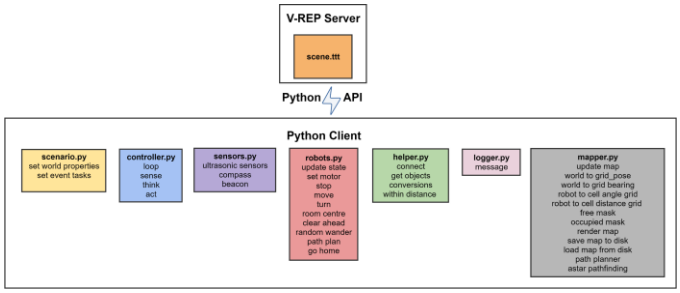

Fig. 8 Implemented software controller architecture

An API (application programming interface) facilitates communication between client and VREP server. See [12] for information on *logger.py* and *helper.py* which is not repeated here. Access to the "*victim*" beacon is via an enhanced *sensors.py* which persists the distance to a new beacon external state variable. The sensors component now also includes the degree angle position of each ultrasonic sensor on the robot (Fig. 5), converted to radians. This is required when using each sensor reading to map the environment in relation to it's specific orientation.

```
sensor_angle = {0: 1.57079, 1: 0.872665,
                2: 0.523599, 3: 0.174533,
                4: -0.174533, 5: -0.523599,
                6:-0.872665, 7: -1.57079,
                8: -1.5708, 9: -2.26893,
                10: -2.61799, 11: -2.96706,
                12: 2.96706, 13: 2.61799,
                14: 2.26893, 15: 1.5708}
```

Note the angle index and VREP sensor naming use the same 0 index convention. However, retrieving a list of sensors from VREP may result in ordering by technical handle, therefore the sensor list is reordered by name:

```
state['int']['prox_s_arr'] = {k: v for k, v in
sorted(state['int']['prox_s_arr'].items(),
key=lambda item: item[1])}
```

1) Scenario

A configured scenario sequences the tasks for controller execution:

```
world = {'props': {'min_dist_enabled': True, 'min_dist': 0.23,
                   'max_dist': 0.28},
         'events': [{'task': 'room_centre', 'mapping_enabled': True},
                    {'task': 'clear_ahead', 'mapping_enabled': True},
                    {'task': 'move', 'robot_dir_travel': 1, 'distm':
                       2.0, 'velocity': 0.2, 'mapping_enabled':
                       False},
                    {'task': 'set_waypoint', 'wp': 'HP Doorway',
                       'mapping_enabled': False},
                    {'task': 'random_wander', 'mapping_enabled':
                       True},
                    {'task': 'stop',  'mapping_enabled': False},
                    {'task': 'set_waypoint', 'wp': 'Beacon',
                       'mapping_enabled': False},
                    {'task': 'path_plan', 'mapping_enabled': False},
                    {'task': 'go_home', 'mapping_enabled': False}]}
```

2) Controller

The robot's ability to successfully and accurately execute its mission tasks is heavily dependent on timely sensory input. To guarantee latest environment perception information is available to task and event rules, the *sense* controller phase updates state at the very start of each control cycle, now including beacon and map inputs:

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

With updated sensor information available, the *think* phase of the controller is supported in making informed decisions during each cycle, for example issuing a stop directive when within range of the target beacon and saving mapping data to permanent storage:

```
if robot.is_prox_to_beacon()and step['task'] in
   beacon_tasks:
   robot.stop(step_status, world_props, task_args)
   robot.state['ext']['mapper'].save_map_to_disk()
```

Methods such as *room_centre* and *go_home* in the *PioneerP3dx* class within *robots.py* implement tasks T1-5, *acting* on sensor and route plan information persisted in state to adjust motor velocities and correct navigational errors. Only when the task is determined complete will the cycle proceed to the next planned task. The controller executes robot task methods dynamically, so there is a one to one relation with a scenario task.

```
# scenario.py
world = {'events': [{'task': 'room_centre'},

# controller.py
if step_status['complete'] is None:
   task_args = {arg: step[arg]
     for arg in step if arg not in 'task'}
   getattr(robot, step['task'])(step_status,
     world_props, task_args)

# robots.py
def room_centre(step_status, world_props, args):
```

V. Behaviour Design

This section takes a closer look at each scenario task and the design proposed to satisfy its requirements. They are processed in sequence, with the controller monitoring execution state and logging performance metrics including time taken and distance travelled.

A. T0 Mapping

For robot localisation, obstacle avoidance, task execution and path planning to both be possible and efficient the world in which the robot operates needs to be understood. In this study there is a problem in that a map of the static environment is not available before commencing the mission scenario, therefore one needs to be built on the fly that the robot can use to plan an optimal route back to HP centre once the search objective is complete.

With the help of VREP API function *simxGetObjectOrientation*, the robot pose is available to provide both a two dimensional absolute location and a heading bearing. Raw data from the ultrasonic sensor array indicates range to obstacles. As the Pioneer has only 2 degrees of freedom, combining the pose and sensor elements enables a 2D floorplan representation of the environment to be assembled which should prove sufficient.

This study adds a mapping feature using the occupancy grid (OG) method, first introduced by Moravec and Elfes in 1985 [14]. It is a "*tessellated 2D grid in which each cell stores, fine grained, quantitative information regarding which areas of a robots operating environment are occupied and which are empty. Specifically, each individual cell in the grid records a certainty factor relating to the confidence that the particular cell is occupied*" [15]. An existing open source algorithm developed in Python [16] implements OG mapping as presented by *Thrun et al* in "*Probabilistic Robotics*" [17], and is integrated into the existing controller framework. It was selected due to its elegant use of numpy arrays for sensor range, sensor angle and occupancy probability computation in a compact solution which clearly demonstrates each of the mapping problems to be solved.

The first challenge is the design of a discretized grid system onto which the simulated world can be mapped. A 2D array of 1500 x 1500

cells is constructed in *mapper.py* to represent the simulation scene (15m²) at 1cm resolution. This should provide accurate localisation of obstacles and adequate definition of navigable paths between them. Each cell is defaulted with 0 to indicate unknown obstacle presence:

```
grid_res = 1
x_size = int(1500 * grid_res)
y_size = int(1500 * grid_res)
map_grid = np.zeros((x_size, y_size))
```

The system of units localising a position differs between the VREP simulated scene and OG. For example, the VREP top left corner *X*, *Y* is represented as (-7.50, +7.50) whereas in OG it is (0,0) (Fig. 9). Therefore, unit conversion transforms VREP absolute coordinates in metres (using international system of units [18]) to grid reference:

```
pose[0] =
int((x_size / 2) + int(pose[0] * -100) * grid_res)

pose[1] =
int((y_size / 2) + int(pose[1] * -100) * grid_res)
```

Similarly, a further enhancement to the OG original source converts VREP robot Euler bearing to grid bearing:

```
bearing =
robot.state['int']['compass'].last_read_euler
if bearing > 0:
   bearing -= math.pi
else:
   bearing += math.pi
```

Consider Fig. 10 showing the base scene modified to shakedown and validate the occupancy grid integration, configuration and results. It is a rather complex map with 17 individual features varying in shape and orientation. At 15m² the environment is of course larger than the maximum range of the ultrasonic sensors, which has been increased from 1m to 4m in sensor volume properties to help improve obstacle detection generally, and deal with positioning within the central HP room measuring ~ 3.5m². A test scenario instructs the robot to follow a fixed, pre-planned route, highlighted red, that starts from scene centre to an inner square path before traversing across to the outer wall for a final circumnavigation.

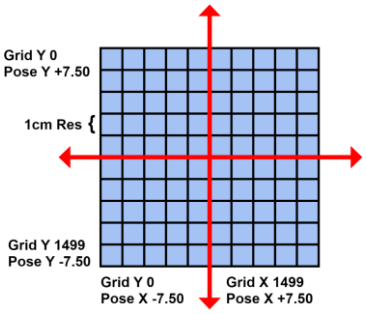

Fig. 9 Discretized grid representation

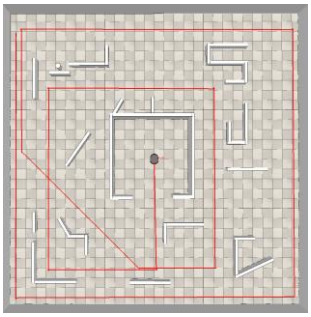

Fig. 10 Mapping test scene

The map is updated during every control cycle. First, with robot position at grid dead centre (750, 750) as an example, a 1500x1500 matrix of distances to all other cells can be calculated in cm using Euclidean norm (Fig. 11). An additional, same resolution matrix of angles in radians from robot to cell is also computed (Fig. 12):

```
def robot_to_cell_distance_grid():
    return scipy.linalg.norm(grid_to_pose, axis=0)

def robot_to_cell_angle_grid():
    return np.arctan2(grid_to_pose[1, :, :],
      grid_to_pose[0, :, :]) - bearing
```

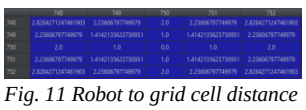

Fig. 11 Robot to grid cell distance

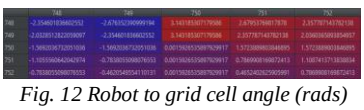

Fig. 12 Robot to grid cell angle (rads)

Each sensor is then processed, ignoring any with a last reading flagged as *false* to throw away abnormal readings, and rescaling *good* range readings from *m* to *cm* to conform with grid system units. The position of the sensor on the robot (Fig. 5) is also retrieved from the internal state store:

```
for i, sen in
  enumerate(robot.state['int']['prox_s'].last_read):
  if sen[1] is False:
     continue

  sensor_distance = sen[0]
  sensor_distance *= (100 * grid_res)
  sensor_angle =
    robot.state['int']['prox_s'].sensor_angle[i]

  fm = free_mask()
  om = occupied_mask()

  map_grid[om] += occupied
  map_grid[fm] += free
```

With the robot localised on the grid, cell distance and angles computed, sensor angle known and returned sensor detection range scaled, a further two matrices *FM* (free mask) and *OM* (occupied mask) with binary *True*/*False* cell values are built. FM represents cells determined free with clear path from robot up to ranged sensor distance, accounting for the ultrasonic sensor angle and aperture, with the value *True*, otherwise *False* (Fig. 13). The free cells typically form a beam-like pattern spreading out from the robot according to sensor radius configuration. Similarly, OM will flag cells as *True* at sensor ranged distance and volume spread, forming a curved line:

```
def free_mask():
    return (np.abs(r_to_c_angle_grid - sensor_angle)
              <= beta  2.0) &
           (r_to_c_distance_grid < (sensor_distance
              - alpha / 2.0))

def occupied_mask():
    return (np.abs(r_to_c_angle_grid - sensor_angle)
              <= beta / 2.0) &
           (np.abs(r_to_c_distance_grid -
              sensor_distance) <= alpha / 2.0)
```

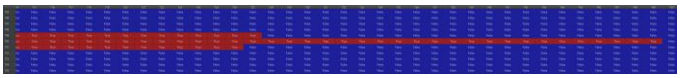

Fig. 13 Free mask with cells confirmed free of obstacles marked True (red)

Finally, the probabilistic OG map is updated. As the OM and FM share the same resolution as the map, the mask vectors flagged as *True* are projected onto the same map vectors which adds value 0.693 to vectors determined by OM to be likely occupied, and -0.693 to vectors from FM probably free. These values were determined to best represent the ground truth after extensive testing. As the robot navigates the environment and *pings* its surroundings with a multi-sensor array, any given obstacle may reflect beams several times, resulting in cells where the probability of free or occupied becomes greater. Fig. 14 shows the occupancy representation of space around an L-shaped structure below and to the right of HP at grid coordinates (737, 1044), where greater negative values along the top and left of the grid indicate cells likely free. It illustrates point to point path planning for safe route navigation should be possible with such a grid system.

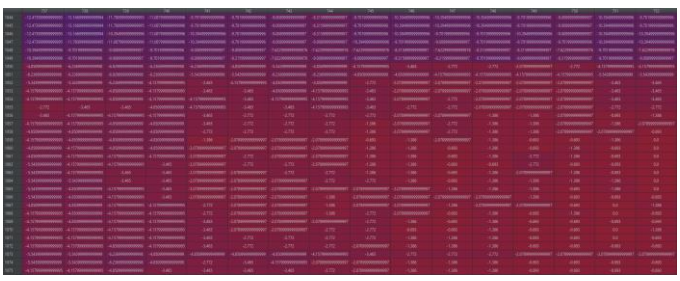

Fig. 14 OG snapshot of free and occupied space around L-shaped obstacle

B. T1 Room Centre

This scenario task requires the robot to find HP centre starting from a random point and bearing within it and can be considered preparation for the launch of a SAR operation or simulating the scanning of an environment from a central position. 4 distinct challenges are recognised. Firstly, measured at ~3.5m², the dimensions of the HP are greater than the default ultrasonic sensor range. Second, there is an open doorway that will return large sensor readings and is not to be navigated until T1 is complete. Thirdly, commencing the mission with a random starting pose implies HP centre can be above, below, to the left, right and in front or behind the robot, which therefore introduces a little complexity to the motor control required. Fourth, in simulating location within a previously unexplored, real-world environment, there are no room centre coordinates available for triangulation.

The first issue is straight-forward to resolve by changing the default volume detection range property for all sensors from 1m to 4m, giving the robot the ability to measure distance to furthest wall irrespective of starting point within HP. New method *room_centre* in component *robots.py* is created to find room centre, handling challenges 2, 3 and 4. One possible approach would be to use odometry and wall following to measure room dimensions from which room centre could be calculated. However, a proposed more elegant solution is taken using opposing sensor distances. The underlying principle is to use distance readings from sensors 0 (left), 3 (front offset left), 7 (right) and 11 (rear offset right) and calculate the standard deviation (SD) from their value distribution as an error function. A value below a set threshold indicates similar distance readings which infers the robot is located approximately in the centre of the square HP room.

A proportional controller sets motor velocity using the error and a gain constant set to a value refined through testing. Sensor distances from the full array are read. The room is 3.5m² with an open doorway beyond which open space and obstacles are reachable by sensors with a 4m range. Therefore, any readings greater than 3m are clipped to remove noise in the execution of this task:

```
kp = 0.155
clip_distance = 3

sensors = [s[0] if s[0] < clip_distance
  else clip_distance for s in
         state['int']['prox_s'].last_read]
```

A subset of opposing sensor distance readings of interest is used to calculate an error as a function of SD from their value distribution:

```
sensor_index = [0, 3, 7, 11]
sensor_list =
  list(itemgetter(*sensor_index)(sensors))
error = statistics.stdev(sensor_list)
```

An error below a set threshold e.g. 20cm indicates the robot is within acceptable distance to room centre and sets task T1 as complete.

```
if abs(error) < 0.20:
   step_status['complete'] = True
   return
```

Image a line drawn through each pair of opposing sensors. This creates two axis that extend out to the measured sensor distances. The aim is to move the robot so that each end of the line on the same new axis becomes equidistant, which can then be inferred as the robot in the centre of the room. New error functions are calculated as the sensor distance difference between the left and right, and front and rear sensors respectively:

```
left_right_error = sensor_list[0] - sensor_list[2]
front_rear_error = sensor_list[1] - sensor_list[3]
```

The omnidirectional, highly manoeuvrable turning characteristics of the Pioneer can now be used to guide the robot to room centre, applying each axis error to the left or right motor, in effect moving it horizontally or vertically in the direction of centre:

```
 if left_right_error < 0:
   state['int']['motor_l_v'] = left_right_error * kp
   state['int']['motor_r_v'] = front_rear_error * kp
else:
   state['int']['motor_l_v'] = front_rear_error * kp
   state['int']['motor_r_v'] = left_right_error * kp

set_motor_v()
```

As task T5 requires return to HP centre, a new method *set_waypoint* in *robots.py* tags the robot's current absolute world location with a passed waypoint name:

```
def set_waypoint(wp):
  res, state['ext']['waypoints'][wp] =
    vrep.simxGetObjectPosition(h.client_id,
      handle, -1, vrep.simx_opmode_buffer)
```

Which is then called upon completion of the room centre task:

```
set_waypoint('HP Centre')
```

C. T2 Room Exit

The robot is now in the centre of HP but not necessarily facing the open doorway. This requirement simulates leaving the enclosed environment after aligning with the exit to avoid bumping into any walls on the way out. It is satisfied by 2 scenario tasks, the first a new *clear_ahead* method in *robots.py* which simply rotates the robot slowly in an anti-clockwise direction until sensor 4 (front offset right, Fig. 5) detected range is greater than 3m. This means open space beyond the HP inner walls is detected, and furthermore the limit considers the robot being able to exit the room even with obstacles a minimum of 1m from the doorway. A reading VREP flags as *false* with no return reflection defaults to the maximum 4m range so the robot of course is able to exit HP into local empty space.

```
def clear_ahead(step_status, world_props, args):
    if state['int']['prox_s'].last_read[4][0] > 3:
       step_status['complete'] = True
       return

    state['int']['motor_l_v'] = -0.01
    state['int']['motor_r_v'] = 0.01
    set_motor_v()
```

The second scenario task executes a modified *move* method with argument *distm* as distance in metres designed to take the robot out of HP by moving it 2m at a velocity of 0.2m/s, then stopping to mark a new waypoint "*HP Doorway*".

D. T3 Avoid Obstacles Locating T4 Target

With the robot "*parked*" outside the HP room it is now in a position to proceed with locating the victim or scene of interest marked by a beacon in VREP, reactively avoiding obstacles throughout its search with proximity checks triggering a stop directive when collidable objects are within a set scenario minimum distance threshold. As stated earlier, the controller sense phase is enhanced to persist the distance to beacon to external state:

```
def update_state_beacon():
    state['ext']['beacon'].read(vrep.simx_opmode_buffer,
      handle=handle)
```

Distance between robot and beacon less than 0.5m is flagged true:

```
def is_prox_to_beacon():
    if state['ext']['beacon'].last_read < 0.5:
       return True
    else:
      return False
```

Close proximity to the beacon terminates this "*locate beacon*" scenario task only. At this point the OG map, continuously compiled during the search, is saved for later analysis and use:

```
if robot.is_prox_to_beacon()and step['task'] in
     beacon_tasks:
     robot.state['ext']['mapper'].save_map_to_disk()
     robot.state['int']['lb_status']['complete'] = True
     robot.stop(step_status, world_props, task_args)
```

Although position and distance to beacon are known, the information will not be used to triangulate and directly navigate to it, as the robot method *locate_beacon_random* in *robots.py* is intentionally designed to randomly wander around the scene to measure the effectiveness of this approach.

The underlying design principle implemented for search continually executes a pair of directive elements until the beacon is located - a stochastic *turn* followed by a fixed *move*. As each element must be executed in strict sequence, leveraging enhanced versions of previously developed turn and move methods originally designed for execution standalone as individual scenario tasks, robot internal state has been updated to maintain status for the overall "*locate beacon*" task as well as these individual sub-elements. Note metrics are captured for time taken, distance travelled and number of turns.

```
'lb_status': {'complete': None, 'turns_count': 0, 'start_t':
             0, 'start_m': 0},
'lb_turn_status': {'complete': None, 'degrees': 0, 'args':
                   {}},
'lb_move_status': {'complete': None, 'startm': 0, 'args':
                   {}},
```

Determining which way to turn incorporates a little intelligence by comparing the distance measured by the left and right sensors. Greater unobstructed space to the left implies it is safer to turn in that direction, with a random degrees of rotation between 2 and 180. Otherwise a right turn is executed. Upon completion a *move* directive moves the robot forward until a distance of 20m is covered or an obstacle is encountered within a minimum distance of 23cm as configured in scenario properties. With this design, the random wandering behaviour works to prevent being trapped by maximising the exploration of open space while avoiding obstacles at a safe distance.

The first challenge to overcome is preventing the robot from re-entering HP given that random turn rotation may cause the robot to face the open HP doorway from outside. The following stops the robot if a *move* directive is active, the last *turn* directive parameterised rotation with a random number of degrees from the fully available turning arc, and the robot's position is within 2.1m of the HP centre waypoint set in task T1. The status for *move* and *turn* are reset to trigger a new combination:

```
if state['int']['lb_move_status']['complete'] is False and
   (h.within_dist(state['ext']['waypoints']['HP Centre'],
    state['ext']['abs_pos_n'], dist_threshold=2.1)) and
    'degrees' in state['int']['lb_turn_status']['args']:
    stop(state['int']['lb_move_status'], world_props, {})
    state['int']['lb_turn_status']['complete'] = None
    state['int']['lb_move_status']['complete'] = None
```

Sensor distances are now retrieved from internal state. Unlike T1 that clipped sensor range to 3m, the full ranging distance is utilised to ping the scene:

```
clip_distance = state['int']['prox_s'].max_detection_dist
sensors = [s[0] if s[0] < clip_distance else clip_distance
  for s in state['int']['prox_s'].last_read]
```

A new turn element in the random wander process is prepared. If the robot is within 2.1m of HP Centre as measured by Euclidean distance from current location to centre waypoint, then the random bearing is limited to a restricted range of 120-240 degrees and parameterised as "*fixed*". This instructs the *turn* method to turn the robot to the explicit compass bearing specified, rather than adding a number of degrees to the current bearing. This forces the robot to move in a generally southerly direction away from the doorway. The 2.1m radius limits the triggering of this corrective action to robot locations south of HP centre and within very close proximity to the doorway, to prevent this rule from being triggered when the robot is to the north.

Beyond the restricted doorway zone, the left and right sensor distances are compared. With more space to the left a counter-clockwise turn rotating a random number of degrees from current bearing is configured, otherwise a clockwise turn is set up. As the turn randomly determines rotation degree from an available wide arc there is little need for high precision, therefore the radius threshold is relaxed to speed up the turn directive managed by a proportional controller:

```
if state['int']['lb_turn_status']['complete'] is None:
   sensor_index = [0, 7]
   sensor_list = list(itemgetter(*sensor_index)(sensors))

   if (h.within_dist(state['ext']['waypoints']['HP Centre'],
       state['ext']['abs_pos_n'], dist_threshold=2.1)):
       random_degree = random.randrange(120, 240, 2)
       state['int']['lb_turn_status']['degrees'] =
         random_degree
       state['int']['lb_turn_status']['args'] = {
         'fixed': state['int']['lb_turn_status']['degrees'],
         'radius_threshold': 0.3}
   else:
       random_degree = random.randrange(2, 180, 2)
       if sensor_list[0] > sensor_list[1]:
          random_degree *= -1
          state['int']['lb_turn_status']['degrees'] =
            random_degree
          state['int']['lb_turn_status']['args'] =
            {'degrees':
              state['int']['lb_turn_status']['degrees'],
               'radius_threshold': 0.3}
```

The turn is executed until signalled complete. Only then will the second element be executed which directs the robot to move forward:

```
if state['int']['lb_turn_status']['complete'] is not True:
   turn(state['int']['lb_turn_status'], world_props,
     state['int']['lb_turn_status']['args'])
   return
```

With a turn complete, the *move* method directs the robot forward for 20m, continuously sensing for approaching obstacles, which if detected below the minimum proximity of 23cm forces a stop action and flags this element as done. A faster velocity is set following detection of the robot within proximity of the open HP doorway, to "*nudge*" it away before the next cycle of rule checks:

```
if state['int']['lb_move_status']['complete'] is None:
   state['int']['lb_move_status']['start_m'] =
     get_distance()
   dir = 1

   velocity = 0.26
   if 'fixed' in state['int']['lb_turn_status']['args']:
       velocity = 0.32
      state['int']['lb_move_status']['args'] =
        {'velocity': 0.26, 'distm': 20,
         'robot_dir_travel': dir}

move(state['int']['lb_move_status'], world_props,
  state['int']['lb_move_status']['args'])
```

On successful location of the beacon the random wander task is flagged complete. A waypoint labelled "*Beacon*" is saved to robot external state as localising information useful to human responders and other automated services.

E. T5 Return Home To HP Centre

With task T4 using random search to locate the beacon now complete, the final objective requires the robot to return to HP Centre. Following a direct route back by triangulating heading and calculating Euclidean distance using logged waypoints is not sufficient as there are obstacles in the way. T4 compiled an OG map as a 2 dimensional array representing the probability of obstacles and free space in the search environment. A deterministic approach takes this information to plan an optimal route back to HP centre avoiding all obstacles.

There are a variety of methods to generate a route plan between 2 points in a static location using a map of the environment space, including Probabilistic Roadmaps (PRMs), Rapidly Exploring Random Trees (RRT), Generalized-Sampling Based Methods, Visibility Graphs, Voronoi Diagrams and cell decomposition methods [19]. However, in this study the A* (A-star) algorithm is used, first published by Hart et al in 1968 [20] as project work enabling Shakey the robot to plan its own paths. The technique takes a grid and applies graph search techniques to "*find the minimum cost path through a graph*" [20] between 2 points. A particular characteristic of A* is the

implementation of a heuristic specific to the problem domain which determines a path cost for each grid location. In this study the heuristic is defined as Euclidean distance from each OG grid cell to HP centre, supporting the algorithm in focused search of optimal paths.

In brief, A* calculates a cost for each unoccupied grid cell as the number of cells away from beacon starting point plus heuristic distance to HP centre destination point. The shortest path connects start and destination with the minimum sum total of cell cost. This study integrates an existing Python implementation of the A* algorithm [21], additionally resolving 2 challenges to allow it to work with the generated OG grid map for successful pre-planned robot navigation in this problem domain. The first is one of mapping data interpretation, with grid cells contain probabilities of occupied space yet A* requires Boolean 1's and 0's. Second, as standard, A* does not account for robot dimensions, which risks proposed plans attempting to route through impassable space.

Returning home to HP introduces 2 new scenario tasks to be executed by the controller; *path_plan* which handles the planning aspect and *go_home* to navigate the proposed path. The new *path_planner* method in *mapper.py* first loads the saved OG grid from disk then uses a probability threshold between -0.5 and 0 to transform it to an array of binary values where 0 represents a passable cell and 1 an impassable cell to be discounted in path planning. Unknown grid locations not covered by ultrasonic pings during search are considered occupied. The A* now has a map representation it can work with.

```
load_map_from_disk()
map_grid_binary = np.zeros((x_size, y_size))
for iy, ix in np.ndindex(map_grid.shape):
    if map_grid[iy, ix] == 0:
       map_grid_binary[iy, ix] = 1
    if -0.5 < map_grid[iy, ix] < 0:
       map_grid_binary[iy, ix] = 1
```

As stated, A* does not consider the configuration space of the Pioneer. However, it is essential to take the physical dimensions into account to ensure only a route guaranteeing safe passage is proposed. The solution used here applies a technique known as binary dilation, originally used in image processing. With the grid having 1cm resolution, a synthesized occupied structure tuned to 35cm² is centred over existing occupied cells:

```
map_grid_binary = binary_dilation(
  map_grid_binary, structure=np.ones((35,
35))).astype(int)
```

In essence, existing path clearance defined as clear space is narrowed through expansion of each occupied cell. Figs. 15 and 16 show the binary representation of the OG before and after dilation, clearly showing obstacles larger than ground truth which has the effect of building a safety margin of error into the grid map used by A*.

Fig. 15 Binary map before dilation

Fig. 16 Binary map after dilation

Following binary grid map pre-processing, the A* algorithm is executed to find the optimal route, which is persisted to disk as a list of coordinate tuples:

```
planned_route = astar_pathfinding()
save_planned_route_to_disk()
```

A route of *X*, *Y* grid references through empty space from beacon to HP is now available for the robot to follow to guide it home, used by the new *go_home* task. It is similar in overall design to the sequencing of motion control via paired *move* and *turn* directive elements in T4 random wander. However, in place of a random bearing rotation and fixed move distance, the robot navigates between route points by first calculating the angle between current absolute position pose bearing and next point. A turn to face the new destination direction is executed, making use of the compass implementation to determine how far to rotate to new target bearing:

```
delta_y = state['int']['gh_status']['route_y'] -
          state['int']['gh_status']['pose'][0]
delta_x = state['int']['gh_status']['route_x'] -
          state['int']['gh_status']['pose'][1]

angle_robot_to_route_point =
  math.degrees(math.atan2(delta_y, delta_x))

angle_robot_to_route_point += 90

if angle_robot_to_route_point > 360:
   angle_robot_to_route_point -= 360

turn(state['int']['gh_turn_status'], world_props,
     state['int']['gh_turn_status']['args'])
```

The distance between current location and next point is calculated in metres, with a move action putting the robot in forward motion towards the new destination. A slow velocity prevents overshoot.

```
dist = (math.hypot(state['int']['gh_status']['pose'][1] -
        state['int']['gh_status']['route_x'],
        state['int']['gh_status']['pose'][0] -
        state['int']['gh_status']['route_y']) / 100)

state['int']['gh_move_status']['args'] = {'velocity':
  0.15, 'distm': dist, 'robot_dir_travel': 1}

move(state['int']['gh_move_status'],
  world_props, state['int']['gh_move_status']['args'])
```

The A* proposed route shares the same 1cm resolution as the binary representation of the OG map used in search of lowest cost path. However, this is too granular for smooth and timely navigation, with testing determining 20cm resolution satisfies the objective of returning to HP. Hence, upon completion of single point-to-point navigation the next 20 route steps are deleted:

```
n = 20
del state['ext']['mapper'].planned_route[:n]
```

VI. Experimental Design

A single scene modifying the base is presented in Fig. 17. The beacon location remains constant at the top left, but the new test scene introduces a number of trapping and deflecting elements primarily designed to test the performance of T4 locate beacon and T5 return home. Obstacle O1 presents a "*natural*" obstacle inherent in the design of the HP inner room doorway, first to be safely navigated on leaving HP, and only passed through again on completion of T4. O2 deliberately tests the robot's ability to progress beyond the immediate vicinity of the HP and its doorway. Obstacles O3, O4, O5, O10 (lower) and O11 form a second layered attempt to deflect and constrain the robot to the area around the HP doorway.

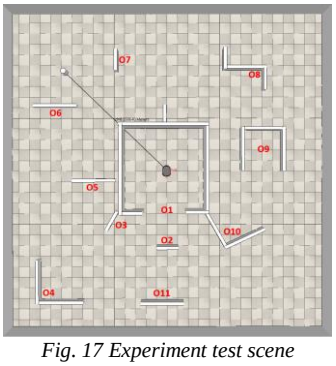

Fig. 17 Experiment test scene

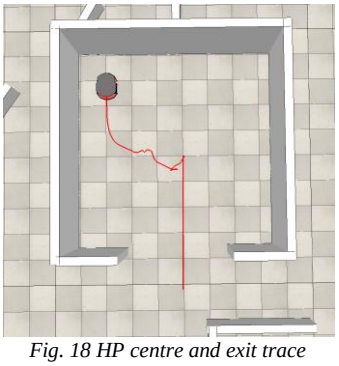

Fig. 18 HP centre and exit trace

Obstacle O5 is placed in such a way that a small gap exists between it and the HP left wall. This is designed to test path planning which

may propose the shorter route to HP centre through the gap impossible for the robot to navigate, rather than the longer, safe route around the wall. O6 and O7 to an extent box in the beacon and make it more difficult to find, as they are located beyond the fixed beacon detection range of 0.5m. O8 is shaped as multiple deflecting surfaces while O9 is designed as a concave, dead-end trap to catch navigation in an area of local minima.

VII. RESULTS

As each mission task is a distinct, rather unique directive issued to the robot, results of each are covered in detail individually.

A. T0 Mapping

Upon completion of the mission scenario the OG is visually rendered as a greyscale image where darker colours indicate higher probability of occupied space, and lighter colours likely free space. Fig 19 shows the visualized OG after completing the test scene in 544s with robot forward motion fixed at 0.2m/s and the full sensor array used. Compared with a slower 0.05m/s completing the scenario in 643s (Fig. 20), the first, faster scenario exhibits greater areas of uncertainty and blind spots, for example to the left and above the inner HP room.

Despite executing the scenario excruciatingly slowly from an observer perspective, the slower speed facilitates greater coverage of the environment and higher mapping precision. This is simply due to a higher frequency of distance readings for a given region, which results in a crisper, cleaner representation where cells converge to free or occupied. As Moravec and Elfes state, "*the map definition improves as more readings are added*" [14]. There are areas on both maps where straight black lines in a known free region indicates a Pioneer design flaw by not having a direct, forward facing sensor at 0°.

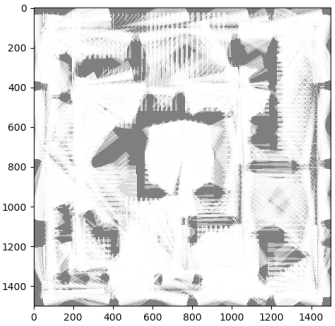

Fig. 19 All sensors, speed 0.2m/s

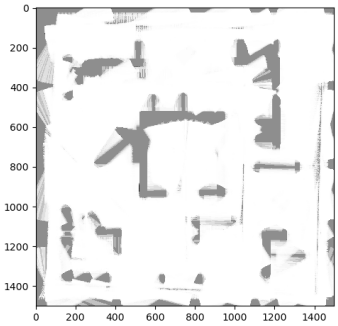

Fig. 20 All sensors, speed 0.05m/s

Further observation of rendered mapping identifies some anomalies which are present on both Figs. 19 and 20. The world space where the right side wall of the HP is located appears less occupied than the left and top walls of the same room despite being *pinged* a similar number of times. Material, mass and other model properties are consistent for all walls, as are sensor volume types and range across the full ultrasonic array. Hence, the root cause is unknown and requires further analysis. Also note a spherical free space *signature* within the HP. This is proposed due to cell occupation determined using sensor distance originating from the geometric centre of the robot, however each sensor is physically offset a number of cm on the body outer shell. This implies the OG represents occupied cells a number of cm closer than they actually are, as the distance between sensor and robot centre is not accounted for. Overall, the map implementation generates an acceptable, consistent and usable world image.

Unfortunately, this mapping comes at huge cost to controller performance. Without mapping or any sleep timer between controller loops, the cycle runs at ~2575Hz. Mapping using all 16 ultrasonic sensors is a significant bottleneck, with control cycle speed reduced to 1Hz, or 1 cycle per second. The problem is one of dimensionality, with cost proportional to map resolution, with a 1500x1500 grid implementation requiring distance and angle computations for 2,250,000 cells. Furthermore, the computational expense and achievable controller Hz is linear to the number of sensors used. Fig. 21 shows Avg runtimes after 20 map updates for a given number of sensors used. Similarly, Hz is measured, with a fairly linear detrimental impact as number of sensors to process is increased (Fig. 22).

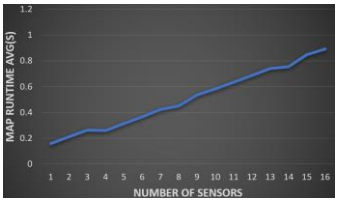

Fig. 21 Map runtime vs # sensors

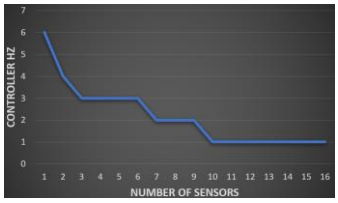

Fig. 22 Controller Hz vs # sensors

Python profiler package *cProfile* [22] was used to get quantifiable runtime statistics on each component part of the mapping process using 16 active sensors. For a total mapping runtime of 0.925s, building the occupied mask takes 0.506s, the free mask 0.292s, the robot to cell angle computations 0.051s and robot to cell distances 0.043s. Comparing code for the free and occupied mask methods it's noted the later has an additional *numpy.absolute* function call, which returns a new array of absolute values. It is proposed the memory allocation for the new array imposes a performance cost. See *Appendix B* for screenshots of map runtime profiling when using a full and front-only sensor array.

Having proved mapping cost scales linearly with number of sensors utilised, 2 further runs mapping the test scene with only the front sensors active were done, at the same forward speeds as per full array testing. At 0.2m/s (Fig. 23) the scene representation looks reasonable although there are some unmapped regions, for e.g. a triangular blind spot to the right of HP (1200, 900), and what appears fuzzier, blurred patches of uncertainty to left of HP (100, 900). When motion is reduced to 0.05m/s (Fig. 24) the world representation sharpens considerably due to better convergence to the free and occupied extremes, resulting in far fewer grey areas. That same triangular blind spot is still present, due to its location beyond front array line-of-sight. It is anticipated this issue will not affect search and rescue simulations where motion is primarily forward facing and hence obstacles in direction of travel should be covered by front array pings.

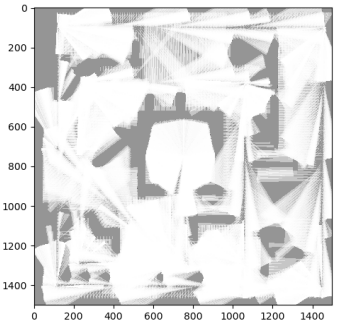

Fig. 23 Front sensors, speed 0.2m/s

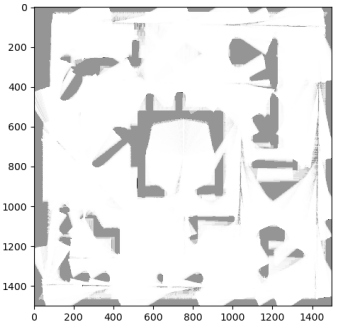

Fig. 24 Front sensors, speed 0.05m/s

Further consideration is given to the overall impact of mapping that despite the OG being update real time and always available for rendering, it is only visualized offline upon mission completion due to the need to protect the controller cycle from any further performance degradation, knowing that the performance of Python package *matplotlib* used for rendering is poor.

It should also be noted that due to the demands mapping places on controller performance, 2 refinements to existing robot tasks were necessary. The existing *robots.py turn* method developed to turn the robot a specified number of degrees relied on a high frequency controller to stop turning motion when actual bearing was within a target bearing threshold. This simple method was sufficient when checking actual vs target bearing occurred several thousand times per second. With the mapping feature in place reducing the frequency of this check to once per second, quite often the robot turned beyond the threshold and thus prevented a stop command being issued to signal completion of task. A new proportional controller continuously reducing error between current and target bearing by setting clockwise or counter-clockwise motion constrained by a rotational speed gain removed the turning task dependency on controller performance and resolved the issue.

The original implementation of the *robots.py move* method, tasked with moving the robot in a direct forward motion, was rather rudimentary. With a fixed velocity of 0.4m/s, a scenario configured

with a move task specified distance in seconds, e.g. 10s indicating 4m forward travel. This sufficed with a fast controller comparing task start time with current time frequently. With a slow controller the robot would travel further than planned. The original design also meant direct forward motion was limited to 1 fixed speed which prevented testing of the mapping process at different speeds. Therefore, a new *move* task argument *distm* now allows specification of distance in metres. Odometry from continuous aggregation of motor revolute joint turns determines when the specified distance to travel is complete, irrespective of speed.

B. T1 Room Centre

A number of scenarios testing task ability to manoeuvre to HP centre regardless of robot location and bearing within it are performed. The room is 3.5m² with an open doorway beyond which open space and obstacles are reachable by sensors with a 4m range, clearly illustrated in Fig. 25.

Fig. 26 shows the path taken, traced red, from a location just SW of centre with the robot facing SSW, completing in 30.42s. Locating centre from the top left with the robot facing NE takes 21.97s (Fig. 27). From above with the robot directly facing HP centre takes 23.97s (Fig. 28). With the robot facing SW away from HP centre, and starting from the bottom left, completes in 16.21s (Fig. 29). The final test places the robot to the right of centre facing it, taking 30.42s (Fig. 30).

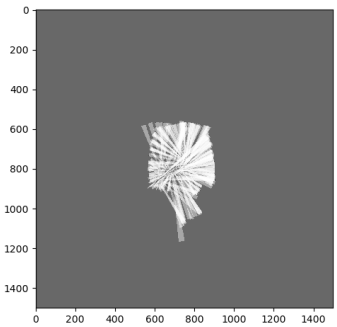

Fig. 25 Ranging beyond doorway

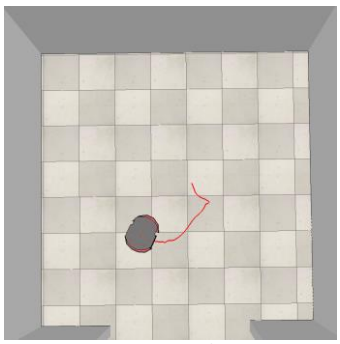
Fig. 26 Near centre facing SSW

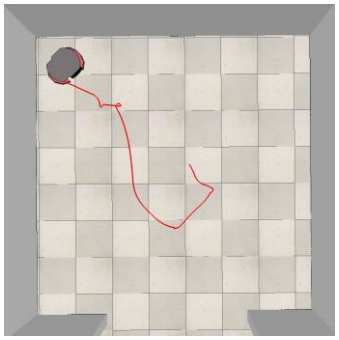
Fig. 27 Top left facing NE

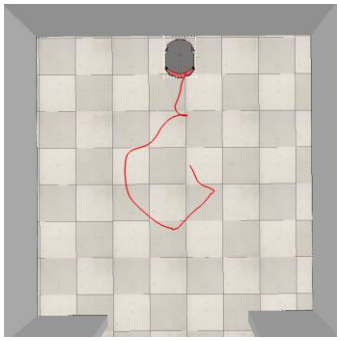
Fig. 28 Top centre facing S

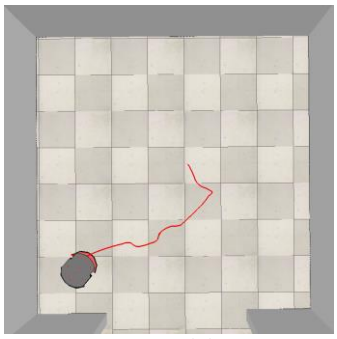
Fig. 29 Bottom left facing SW

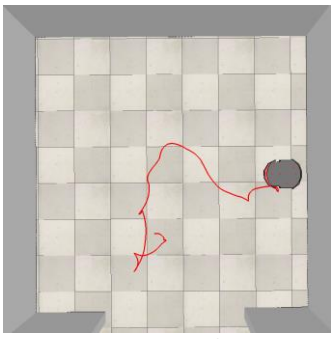
Fig. 30 Centre right facing W

Some tests clearly show a rather circuitous route to centre (Figs. 28, 30), where starting location is directly above or to the side. Improvements may be realised from the optimisation of proportional gain and sensor clipping point, which additionally is believed the cause of signalled HP centre further away from actual ground truth than is ideal, but still within acceptable parameters for completion of the overall scenario. The tests also demonstrate the ability to locate centre even when facing away and starting in a tight corner position. It is considered an elegant, understandable and minimalist solution given the problem complexity is solved in 17 lines of code.

C. T2 Room Exit

Fig. 18, p.7 illustrates a combined HP room centre and exit trace, which clearly shows the solution satisfies T2 requirements. The robot exits HP in 10s without touching open doorway edges.

D. T3 Avoid Obstacles Locating T4 Target

The idea behind the implementation of a random wander putting the robot in continuous stochastic motion until the beacon is located was to measure the effectiveness of this probabilistic exploration strategy. As results will show, unsurprisingly, not very effective, and critically in the problem domain of SAR, not time efficient.

A direct line from HP centre to beacon, ignoring obstacles, is ~5m. The shortest path between both points is ~13m. The first example illustrated in Fig. 31 traces the 34.66m random path from HP to beacon, making 14 turns in 523.23s (8.71 mins). This may be considered a success, taking only 2.5 times the optimal shortest path distance. The corresponding OG map visualised in Fig. 32 looks reasonably good, with scene features recognisable, although there is a concern around the HP left wall shown as free space which suggests an optimal route path back to HP centre that is not physically viable.

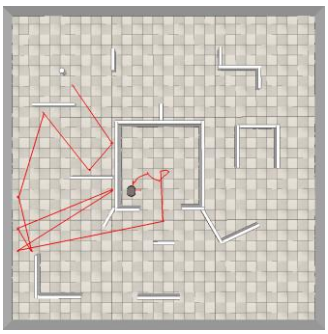
Fig. 31 Random locate beacon e.g. 1

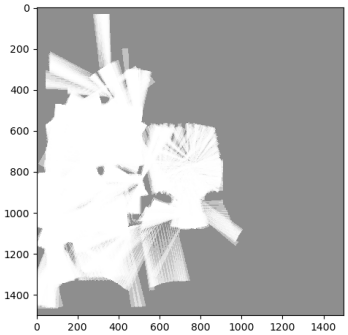

Fig. 32 OG mapping random e.g. 1

The metrics of random wander example 2 (Fig. 33) tell a different story, travelling 429.94m in 16674.87s (277.9 mins, or 4.63 hrs) making 178 turns. It took so long that the red trace does not show line to beacon due to the graph buffer size exhausted despite being extended from 1000 to 300000. This second result is unacceptable, where a matter of minutes can determine the success of a SAR mission, affecting mission planning and victim status. Furthermore, it is unlikely that any autonomous, untethered robot has access to battery life longer than the potential time take by random wander. A secondary side-effect of such long search times is the apparent "*overexposed*" nature of the rendered OG (Fig. 34), which further study might look to resolve by setting a probability capping limit.

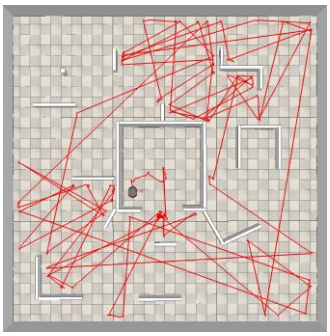
Fig. 33 Random locate beacon e.g. 2

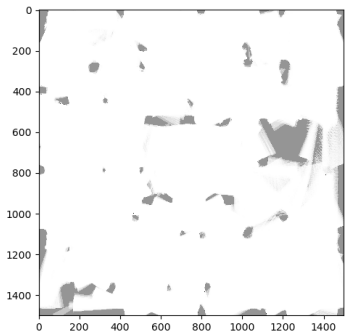

Fig. 34 OG mapping random e.g. 2

Although the random wander strategy is not effective from a SAR perspective, this study's technical implementation of it works, with the robot prevented from re-entering the HP several times (Fig. 33) and eventually locating the beacon without being rendered locked into regions designed as traps. The navigation itself is collision-free, leveraging the built-in proximity checking capabilities that use both front and rear sensor arrays to safeguard the robot at a minimum obstacle distance. The particular design of the experimental

environment and placement of deflecting obstacles appears a really effective test of the solution.

E. T5 Return Home To HP Centre

Behaviour design described the technical implementation of 2 tasks for T5. The first, *path_plan*, determines the lowest cost route home to HP from beacon location as an ordered list of grid coordinates, which *go_home* then navigates. The OG map generated from the first successful random wander to beacon location (Fig. 32) is transformed into a binary representation for path planning, with an optimal route (Fig. 35) proposed in 274s. As clear, the concern regarding the presence of free space in the location of the HP left wall was warranted, with A* correctly identifying these cells as on a shortest path to target destination as far as the mapping data is concerned. Modifying the VREP scene HP left wall to reflect the mapping image allowed the binary dilation mechanism that pads obstacles with safe margins and the navigation routine that follows a pre-planned route to be proved successful (Fig. 36), with the robot trace resembling the planned path. In this example, returning home comprised individual navigation to 25 route points, taking 155.86s to cover 6.38m.

Fig. 35 Planned route to HP e.g. 1

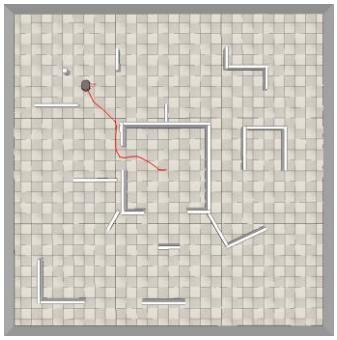

Fig. 36 Navigation to HP e.g. 1

It was desirable to test an alternative, longer, more complex route to HP, therefore the scene was remapped with the rover following a manually configured path that included scanning "*sweeps*" (Fig. 37), the result shown rendered in Fig. 38.

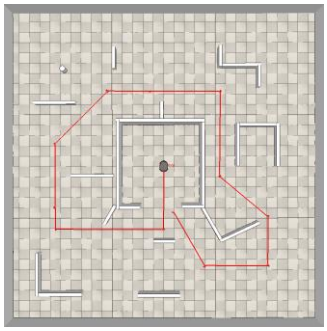

Fig. 37 Remapping experiment scene

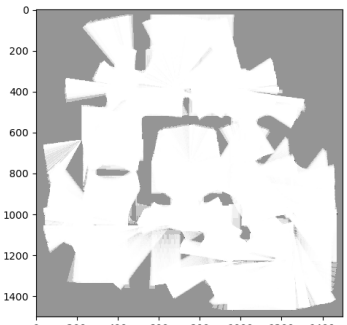

Fig. 38 Experiment scene OG remap

Despite the test obstacles identifiable on the map, there now exists an issue with the HP right wall where it is incomplete. This was temporarily resolved by filling it in with:

```
self.map_grid_binary[550:915, 900:950] = 1
```

With the modified map, path planning took 302s to suggest the route illustrated in Fig. 39. Clearly A* follows the edge of occupied space and does not specifically account for robot dimensions. However, as the map is artificially enhanced with binary dilation, detected obstacles are enlarged to include a safe buffer. This is nicely illustrated in Fig. 40, with the red path trace showing sufficient room for the robot to safely navigate a collision-free, shortest path around multiple obstacles to destination. In this example, returning to HP took 384.28s to travel 14.95m in 65 steps. With return to HP centre, the implemented technical solution, based on available mapping data, is considered successful and the overall scenario complete.

Fig. 39 Planned route to HP e.g. 2

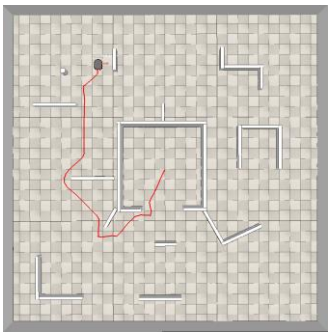

Fig. 40 Navigation to HP e.g. 2

VIII. CONCLUSION

This study implemented and evaluated 6 tasks typically required of a SAR robot, including mapping of the environment, obstacle avoidance, autonomous location of target objectives, precision navigation fusing multi-sensor data and optimised path planning. As demonstrated with results, the proposed solutions accomplished the task objectives successfully.

The study highlighted that achieving performant, accurate and well-controlled results even in a synthetic environment far simpler than those complex, highly unstructured sites at Chernobyl and the WTC is extremely challenging, with several specific areas of the solution identified for future improvement. Foremost are the computational resource demands of the current environment mapping implementation, which reduces controller frequency to just a few Hz with linear degradation in line with number of ultrasonic sensors used. The underlying issue is the resolution at which the environment is mapped. Given the ultrasonic sensors have a 4m range there is no need to calculate free and occupied space masks with angles and distances for the entire 15m² environment at every sensor update. A possible improvement here would limit masking to a striding environment window, centred on the robot and covering sensor range only, that would reduce the number of grid cells to be processed by ~50%.

Another performance improvement would introduce separate processing threads for both the physics controller and mapping processes. This would allow the control cycle managing motor control and navigation to run at a consistent, guaranteed Hz and not "*hang*" pending completion of secondary tasks. However, one positive outcome of the mapping performance challenges experienced was the further refinement of existing *move* and *turn* robot actions. Proportional controllers were introduced to negate the dependency on controller cycle frequency to check task completion in a timely fashion, using error calculation and reduction specific to each problem.

Further work would also explore more efficient alternatives to the random wander search approach. Despite demonstrations of this approach avoiding obstacles and traps to successfully locate the beacon, the time taken to search a 15m² environment in a SAR scenario is unacceptable. The authors of [1] would agree, stating that SAR robots "*must determine the location and status of victims as quickly as possible*". [23] emphasises the importance of rapid response further with "*In a typical collapsed structure scenario, responders have roughly 72h to find trapped survivors, otherwise the likelihood of finding victims still alive and restoring them to full health drops nearly to zero. Disaster mitigation requires rapid and efficient search and rescue as the magnitude of the devastation of urban environments exceeds the available resources USAR specialists, USAR dogs, and sensors needed to rescue victims within the critical first 72h*". Ideally a new approach would also negate the need to stop to avoid obstacles, but rather slow down yet maintain continuous motion. With forward planning a set of alternative navigation directives could be made immediately available to the controller and then acted upon when obstacles come into close proximity.

Finally, further detailed analysis of the environment mapping and robot sensor configuration might identify why certain scene features such as the HP walls were missing or overexposed. Effective path planning is reliant on high quality, reliable map data and such scene mapping issues adversely affect route calculation. All solution code is available at https://github.com/corticalstack/vrep-search-rescue

REFERENCES

[1] M. Moniruzzaman, M. S. Rahman Zishan, S. Rahman, S. Mahmud and A. Shaha, "Design and Implementation of Urban Search and Rescue Robot," *International Journal of Engineering and Manufacturing,* vol. 8, no. 2, pp. 12-20, 2018.

[2] D. Calisi, A. Farinelli, L. Locchi and D. Nardi, "Multi-objective exploration and search for autonomous rescue robots," *Journal of Field Robotics,* vol. 24, no. 8-9, pp. 763-777, 2007.

[3] Wikipedia, "Deaths due to the Chernobyl disaster," [Online]. Available: https://en.wikipedia.org/wiki/Deaths_due_to_the_Chernobyl_disaster. [Accessed 07 01 2020].

[4] J. Abouaf, "Trial by fire: teleoperated robot targets Chernobyl," *IEEE Computer Graphics and Applications,* vol. 18, no. 4, pp. 10-14, 1998.

[5] Wikipedia, "Chernobyl New Safe Confinement," [Online]. Available: https://en.wikipedia.org/wiki/Chernobyl_New_Safe_Confinement. [Accessed 07 01 2020].

[6] "Activities of the rescue robots at the World Trade Center from 11-21 september 2001," *IEEE Robotics & Automation Magazine,* vol. 11, no. 3, pp. 50-61, 2004.

[7] E. Expert.com, "Inuktun - Microtracs," [Online]. Available: https://www.environmental-expert.com/products/inuktun-model-microtracs-crawler-unit-376753. [Accessed 07 01 2020].

[8] Wikipedia, "Great Hanshin earthquake," [Online]. Available: https://en.wikipedia.org/wiki/Great_Hanshin_earthquake. [Accessed 07 01 2020].

[9] A. H. Levent, I. Nobuhiro, J. Adam, K. Alexander and P. Johannes, "RoboCup Rescue Robot and Simulation Leagues," *AI Magazine,* vol. 34, no. 1, pp. 78-86, 2013.

[10] Wikipedia, "Fukushima Daiichi Nuclear Disaster," [Online]. Available: https://en.wikipedia.org/wiki/Fukushima_Daiichi_nuclear_disaster. [Accessed 07 01 2020].

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

[12] J. Boyd, "A Virtual Wall Following Robot," 2019.

[13] J. Boyd, "A Virtual Wall Following Robot With PID Controller," 2019.

[14] H. Moravec and A. Elfes, "High resolution maps from wide angle sonar," in *IEEE International Conference on Robotics and Automation*, 1985.

[15] T. Collins, J. Collins and C. Ryan, "Occupancy Grid Mapping: An Empirical Evaluation," in *Mediterranean Conference on Control and Automation*, Athens, Greece, 2007.

[16] G. -. superjax, "An occupancy grid mapping example," [Online]. Available: https://gist.github.com/superjax/33151f018407244cb61402e094099c1d. [Accessed 14 01 2020].

[17] S. Thrun, W. Burgard and D. Fox, "Occupancy Grid Mapping," in *Probabilistic Robotics*, MIT Press, 1999, pp. 221-243.

[18] Wikipedia, "International System of Units," [Online]. Available: https://en.wikipedia.org/wiki/International_System_of_Units. [Accessed 08 01 2020].

[19] V. Sezer and M. Gokasan, "A novel obstacle avoidance algorithm: "Follow the Gap Method"," *Robotics and Autonomous Systems,* vol. 60, no. 9, pp. 1123-1134, 2012.

[20] P. E. Hart, N. J. Nilsson and B. Raphael, "A Formal Basis for the Heuristic Determination of Minimum Cost Paths," *IEEE Transactions on Systems Science and Cybernetics,* vol. 4, no. 2, 1968.

[21] C. Careaga, "Python A* Pathfinding (with binary heap)," [Online]. Available: http://code.activestate.com/recipes/578919-python-a-pathfinding-with-binary-heap/.

[22] python.org, "The Python Profilers - Python Documentation," [Online]. Available: https://docs.python.org/2/library/profile.html. [Accessed 15 01 2020].

[23] R. Voyles and H. Choset, "Editorial: Search and rescue robots," *Journal of Field Robotics,* vol. 25, no. 1-2, pp. 1-2, 2007.

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

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

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 – MAP RUNTIME PROFILING

```
ncalls  tottime  percall  cumtime  percall filename:lineno(function)
     1    0.000    0.000    0.925    0.925 C:\Users\JP\Documents\My Developments\mobilerobots\vrep\vrep-search-rescue\robots.py:183(update_state_map)
     1    0.028    0.028    0.925    0.925 C:\Users\JP\Documents\My Developments\mobilerobots\vrep\vrep-search-rescue\mapper.py:42(update_map)
    16    0.506    0.032    0.506    0.032 C:\Users\JP\Documents\My Developments\mobilerobots\vrep\vrep-search-rescue\mapper.py:94(occupied_mask)
    16    0.292    0.018    0.292    0.018 C:\Users\JP\Documents\My Developments\mobilerobots\vrep\vrep-search-rescue\mapper.py:90(free_mask)
     1    0.051    0.051    0.051    0.051 C:\Users\JP\Documents\My Developments\mobilerobots\vrep\vrep-search-rescue\mapper.py:84(robot_to_cell_angle_grid)
     1    0.000    0.000    0.043    0.043 C:\Users\JP\Documents\My Developments\mobilerobots\vrep\vrep-search-rescue\mapper.py:87(robot_to_cell_distance_grid)
     1    0.005    0.005    0.043    0.043 C:\Users\JP\AppData\Local\Programs\Python\Python37\lib\site-packages\scipy\linalg\misc.py:19(norm)
     1    0.019    0.019    0.039    0.039 C:\Users\JP\AppData\Local\Programs\Python\Python37\lib\site-packages\numpy\linalg\linalg.py:2293(norm)
     1    0.010    0.010    0.010    0.010 {method 'astype' of 'numpy.ndarray' objects}
     1    0.010    0.010    0.010    0.010 {method 'reduce' of 'numpy.ufunc' objects}
     1    0.006    0.006    0.006    0.006 {method 'copy' of 'numpy.ndarray' objects}
     1    0.000    0.000    0.000    0.000 C:\Users\JP\AppData\Local\Programs\Python\Python37\lib\site-packages\numpy\lib\function_base.py:434(asarray_chkfinite)
     1    0.000    0.000    0.000    0.000 C:\Users\JP\Documents\My Developments\mobilerobots\vrep\vrep-search-rescue\mapper.py:76(world_to_grid_pose)
     2    0.000    0.000    0.000    0.000 C:\Users\JP\AppData\Local\Programs\Python\Python37\lib\site-packages\numpy\core\numeric.py:469(asarray)
     2    0.000    0.000    0.000    0.000 {built-in method numpy.array}
     1    0.000    0.000    0.000    0.000 {built-in method builtins.issubclass}
     1    0.000    0.000    0.000    0.000 C:\Users\JP\Documents\My Developments\mobilerobots\vrep\vrep-search-rescue\mapper.py:69(world_to_grid_bearing)
     1    0.000    0.000    0.000    0.000 {method 'conj' of 'numpy.ndarray' objects}
     1    0.000    0.000    0.000    0.000 {built-in method builtins.isinstance}
     2    0.000    0.000    0.000    0.000 {built-in method builtins.len}
     1    0.000    0.000    0.000    0.000 {method 'disable' of '_lsprof.Profiler' objects}
```

Fig. 41 Map process runtime profiling with full sensor array

```
ncalls  tottime  percall  cumtime  percall filename:lineno(function)
     1    0.000    0.000    0.482    0.482 C:\Users\JP\Documents\My Developments\mobilerobots\vrep\vrep-search-rescue\robots.py:183(update_state_map)
     1    0.017    0.017    0.481    0.481 C:\Users\JP\Documents\My Developments\mobilerobots\vrep\vrep-search-rescue\mapper.py:42(update_map)
     7    0.230    0.033    0.230    0.033 C:\Users\JP\Documents\My Developments\mobilerobots\vrep\vrep-search-rescue\mapper.py:94(occupied_mask)
     7    0.131    0.019    0.131    0.019 C:\Users\JP\Documents\My Developments\mobilerobots\vrep\vrep-search-rescue\mapper.py:90(free_mask)
     1    0.052    0.052    0.052    0.052 C:\Users\JP\Documents\My Developments\mobilerobots\vrep\vrep-search-rescue\mapper.py:84(robot_to_cell_angle_grid)
     1    0.000    0.000    0.045    0.045 C:\Users\JP\Documents\My Developments\mobilerobots\vrep\vrep-search-rescue\mapper.py:87(robot_to_cell_distance_grid)
     1    0.006    0.006    0.045    0.045 C:\Users\JP\AppData\Local\Programs\Python\Python37\lib\site-packages\scipy\linalg\misc.py:19(norm)
     1    0.020    0.020    0.040    0.040 C:\Users\JP\AppData\Local\Programs\Python\Python37\lib\site-packages\numpy\linalg\linalg.py:2293(norm)
     1    0.010    0.010    0.010    0.010 {method 'astype' of 'numpy.ndarray' objects}
     1    0.010    0.010    0.010    0.010 {method 'reduce' of 'numpy.ufunc' objects}
     1    0.006    0.006    0.006    0.006 {method 'copy' of 'numpy.ndarray' objects}
     1    0.000    0.000    0.000    0.000 C:\Users\JP\AppData\Local\Programs\Python\Python37\lib\site-packages\numpy\lib\function_base.py:434(asarray_chkfinite)
     1    0.000    0.000    0.000    0.000 C:\Users\JP\Documents\My Developments\mobilerobots\vrep\vrep-search-rescue\mapper.py:76(world_to_grid_pose)
     2    0.000    0.000    0.000    0.000 C:\Users\JP\AppData\Local\Programs\Python\Python37\lib\site-packages\numpy\core\numeric.py:469(asarray)
     2    0.000    0.000    0.000    0.000 {built-in method numpy.array}
     1    0.000    0.000    0.000    0.000 {built-in method builtins.isinstance}
     1    0.000    0.000    0.000    0.000 C:\Users\JP\Documents\My Developments\mobilerobots\vrep\vrep-search-rescue\mapper.py:69(world_to_grid_bearing)
     1    0.000    0.000    0.000    0.000 {method 'conj' of 'numpy.ndarray' objects}
     1    0.000    0.000    0.000    0.000 {built-in method builtins.issubclass}
     2    0.000    0.000    0.000    0.000 {built-in method builtins.len}
     1    0.000    0.000    0.000    0.000 {method 'disable' of '_lsprof.Profiler' objects}
```

Fig. 42 Map process runtime profiling with front-only sensor array