Overview

The algorithm implemented aims to accomplish two tasks: Exploration and Fastest Path Planning. During exploration, the robot scans an unknown map filled with obstacles, travelling to the Goal Zone and then back to the Start Zone. Performance during exploration is measured by the correctness of scanning and mapping, as well as the time taken to complete the exploration. During fastest path planning, the robot travels from the Start Zone to the Goal Zone, following a computed path that passes through a pre-set way point.

In the actual implementation, right-wall hugging is used for traversing the arena during exploration. And A* Algorithm is used for fastest path planning. Additional features used to optimize both processes will be introduced in the following sections.

The algorithm was initially tested on a simulator written with Pygame. For actual deployment, the algorithm written in Python3 runs directly on the Raspberry Pi. This is to save additional communication time between PC and Raspberry Pi.

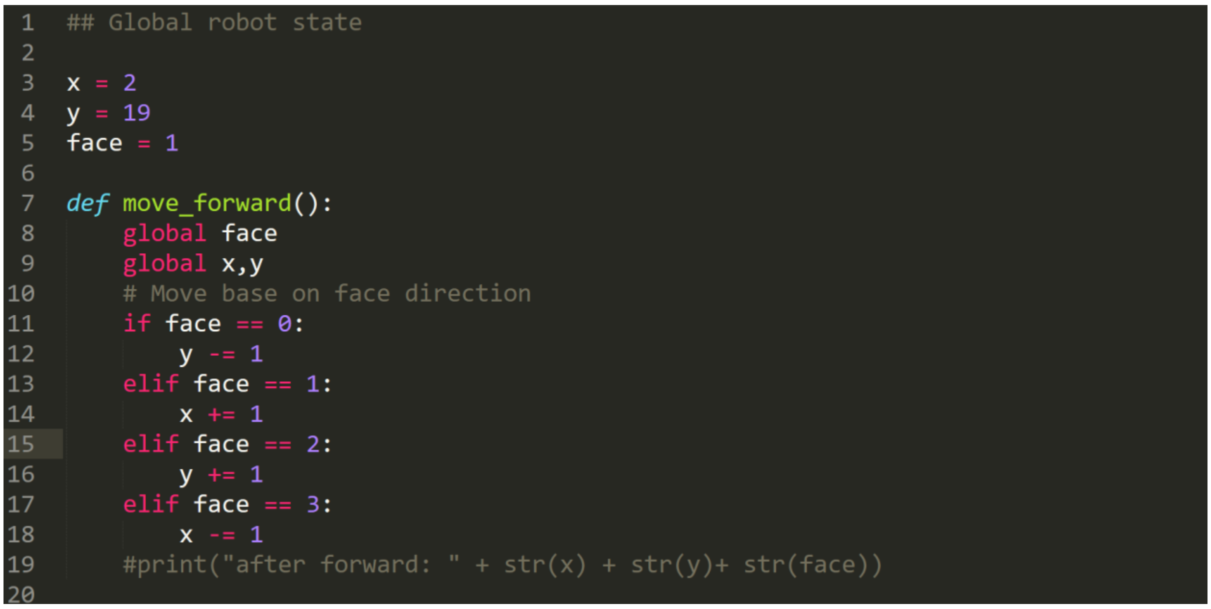
Key Modules

For easy modification and maintenance, the algorithm is split into the following modules:

- *arduino_interface.py* – Arduino communication (see Section *Raspberry Pi*)
- *rfcomm_server.py* – Bluetooth communication (see Section *Raspberry Pi*)
- *AhBot.py* – Robot status and properties
- *Explored.py* – Scanned map and properties
- *Exploration.py* – Computation of exploration movements
- *fastest_path.py* – Computation of fastest path planning and movements
- *process_sensor.py* – Processing of incoming sensor readings
- *detection.py* – (see Section *Raspberry Pi*)
- *main.py* – Coordination of the above modules

AhBot.py

This module stores the key properties of the robot: x-coordinate, y-coordinate and the direction it faces. It also contains methods for local updates on these properties. i.e. maintaining its property correspondence with the actual robot.



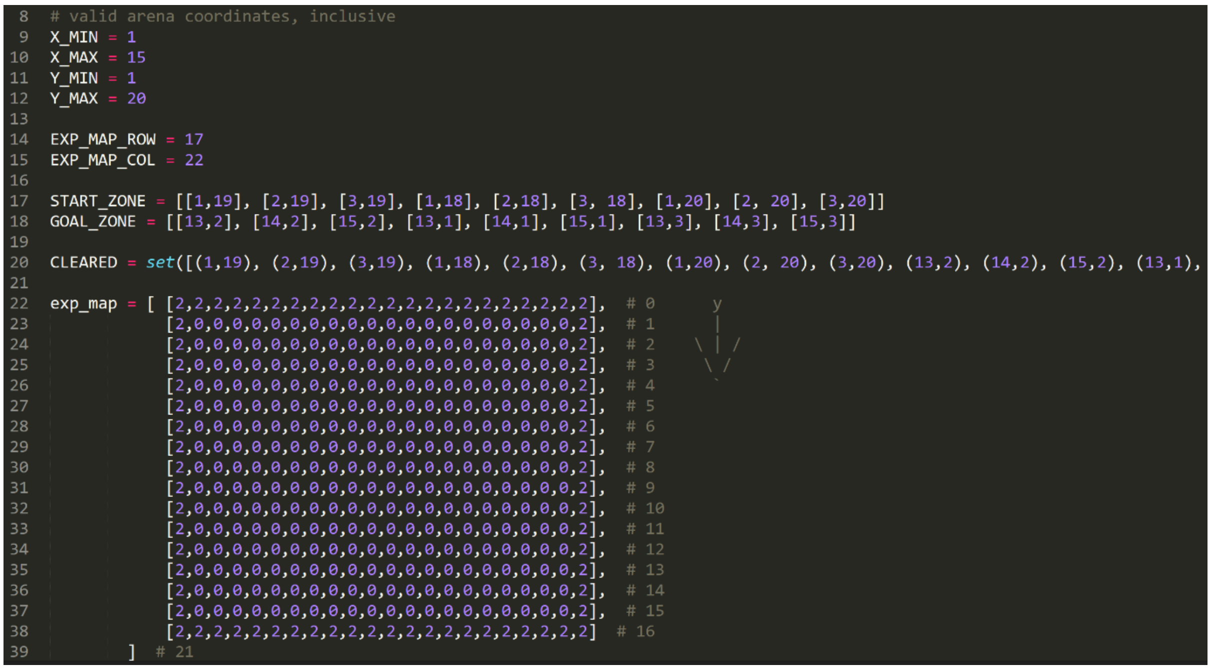
```python
## Global robot state

x = 2
y = 19
face = 1

def move_forward():
    global face
    global x,y
    # Move base on face direction
    if face == 0:
        y -= 1
    elif face == 1:
        x += 1
    elif face == 2:
        y += 1
    elif face == 3:
        x -= 1
    #print("after forward: " + str(x) + str(y)+ str(face))

```

Movement updates includes coordinate updates and face direction updates. Coordinate updates are dependent on the face direction. These methods are called when a movement command is sent to the Arduino.

Explored.py

Attributes

This modules stores the explored map with its properties such as number of rows and columns. The map is initialized to a 20×15 blank grip map, which corresponds to the 200×150 cm actual arena. This map is updated throughout the maze exploration.



```
# valid arena coordinates, inclusive
X_MIN = 1
X_MAX = 15
Y_MIN = 1
Y_MAX = 20

EXP_MAP_ROW = 17
EXP_MAP_COL = 22

START_ZONE = [[1,19], [2,19], [3,19], [1,18], [2,18], [3, 18], [1,20], [2, 20], [3,20]]
GOAL_ZONE = [[13,2], [14,2], [15,2], [13,1], [14,1], [15,1], [13,3], [14,3], [15,3]]

CLEARED = set([(1,19), (2,19), (3,19), (1,18), (2,18), (3, 18), (1,20), (2, 20), (3,20), (13,2), (14,2), (15,2), (13,1),

exp_map = [ [2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2],  # 0      y
            [2,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,2],  # 1      |
            [2,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,2],  # 2    \ | /
            [2,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,2],  # 3     \ /
            [2,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,2],  # 4      `
            [2,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,2],  # 5
            [2,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,2],  # 6
            [2,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,2],  # 7
            [2,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,2],  # 8
            [2,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,2],  # 9
            [2,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,2],  # 10
            [2,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,2],  # 11
            [2,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,2],  # 12
            [2,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,2],  # 13
            [2,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,2],  # 14
            [2,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,2],  # 15
            [2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2]  # 16
          ]  # 21
```

‘0’ represents unexplored grid
‘1’ represents explored free grid
‘2’ represents explored obstacle grid

Phantom Block Prevention

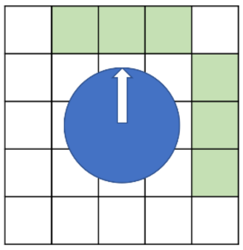
The set ‘CLEARED’ contains grids which are never blocks. Such grids include: grids in the Start Zone, grids in the Goal Zone and grids that have been occupied by the robot. This set facilitates the prevention of ‘phantom blocks’, i.e. non-existent blocks detected by the sensors.

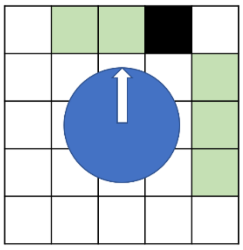
Exploration.py

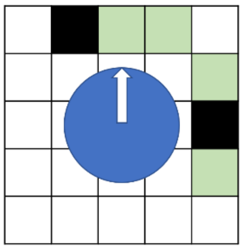
The Right Wall Hugging algorithm is the core of this module. During right wall hugging, the movement of the robot is computed based on the following rules:

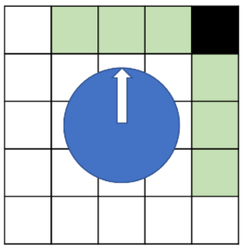
- If right-hand side is clear, turn right and move one step forward.
- If right-hand side is not clear and front is clear, move one step forward.
- If right-hand side is not clear and front is not clear, turn left.

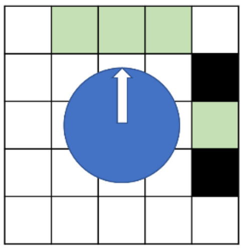
Here ‘clear’ means all three grids adjacent to the robot’s certain side are all spaces, given that the robot occupies a 3×3 grid space.

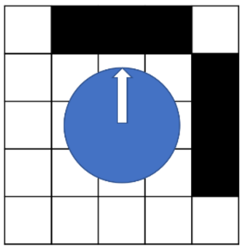

Front clear & RHS clear


Front not clear & RHS clear


Front not clear & RHS not clear


Front clear & RHS clear

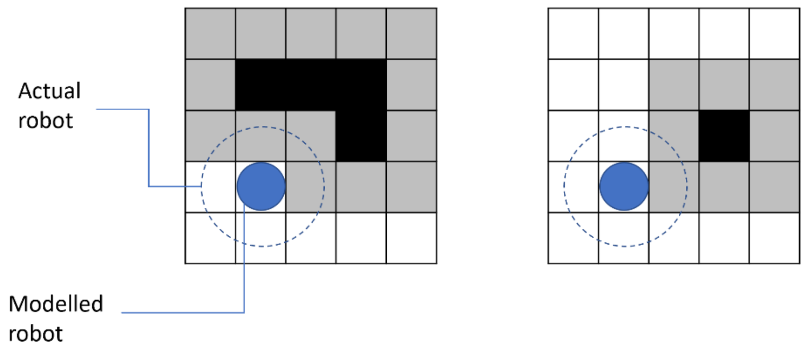

Front clear & RHS not clear


Front not clear & RHS not clear

For each iteration during the exploration phase, we first update the map in Explored.py with the sensor readings. Subsequently, Exploration.py is called to compute the next step base on the newly updated explored map.

fastest_path.py

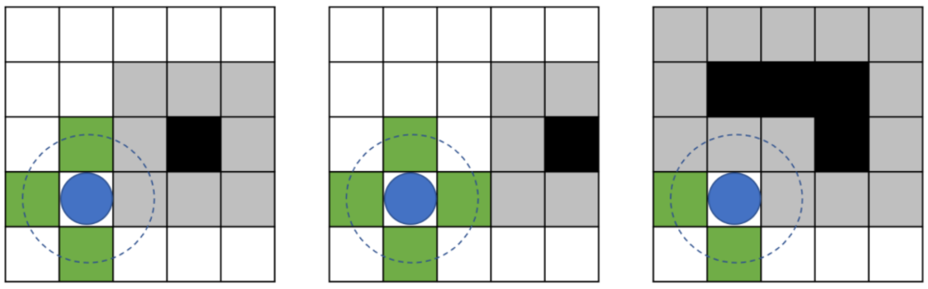
The fastest path is computed with the A* Algorithm. In this module, the robot is modeled to occupy only one grid. To prevent the robot from hitting the walls, neighboring grids of the explored blocks are also set as obstacles.



- Robot is modelled to occupy one grid
- The nine grids around an actual block are set as obstacles and hence untraversable

The A* Algorithm

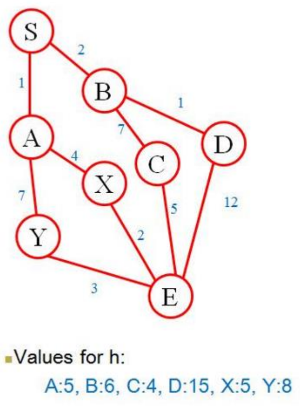
For each grid, its estimated distance to the destination is computed as ***(abs(destination.x – grid.x) + abs(destination.x – gird.y))***. Diagonal distances are not involved as the robot does not move diagonally. For the same reason, only grids on the robot's front, back, left and right are considered neighboring grids.

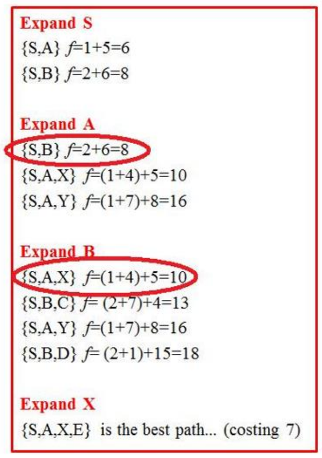


- The green grids are the child nodes of the robot's node
- The four adjacent blocks are considered children of the robot's node
- If a child node has been set as an obstacle, it won't be considered a candidate child.

Each grid on the map is represented as a 'node' data structure in this module. Each node stores information such as its coordinates, the parent node, and actual distance traveled from the start node to the current node (if applicable). We refer the neighboring nodes of the current node as its children. During each iteration, all children of the current node will be inserted into the candidate node queue. The estimated total distance through each child is computed as ***estimated_total_distance = current_node.distance_traveled + child.estimated_remaining_distance*****.** Children in the candidate node queue are then sorted based on their estimated total distance in increasing order. The first child, which has the shortest estimated total distance, will be added to the fastest path.

After selecting the optimal child, the current node will be set as an obstacle to prevent re-computation as the current node could be considered a child to its neighboring nodes.




Expand S
{S,A} f=1+5=6
{S,B} f=2+6=8

Expand A
{S,B} f=2+6=8
{S,A,X} f=(1+4)+5=10
{S,A,Y} f=(1+7)+8=16

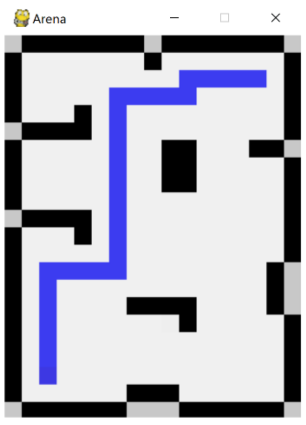
Expand B
{S,A,X} f=(1+4)+5=10
{S,B,C} f= (2+7)+4=13
{S,A,Y} f=(1+7)+8=16
{S,B,D} f= (2+1)+15=18

Expand X
{S,A,X,E} is the best path... (costing 7)

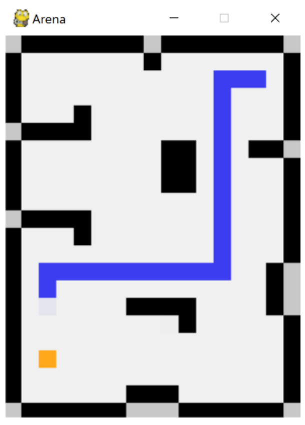
An illustration of the A* Search Algorithm
Source: https://stackoverflow.com/questions/5849667/a-search-algorithm

Promoting Straight Paths

Since each turn in the fastest path would take up extra time during the actual run, the algorithm is modified to promote straight paths, i.e. reduce the number of turns. If a draw occurs when selecting the candidate child, we prioritize the one that aligns with the current node and its parent node.



Before



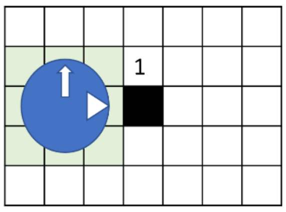
After

process_sensor.py

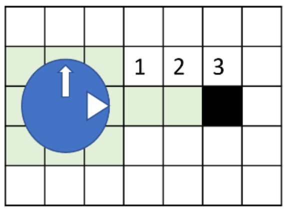
This module converts raw sensor readings to number of grids away and updates the map in Explored.py accordingly. It stores the maximum range of each sensor. Conversion logic is obtained from the Arduino Team.

When allocating spaces and blocks during updates, we first check if the grids returned is smaller than the maximum grid number **max**. Suppose the sensor readings indicates **n** grids (i.e. the block is **n** grids away) and **n < max** , the **1st** to **(n-1)th** adjacent grid will be updated as space and the **nth**adjacent grid as a block. If **n > max**, all the **1st** to **maxth** adjacent grid will be updated as space. Given the robot face direction and sensor position, we can determine which adjacent grids to update.

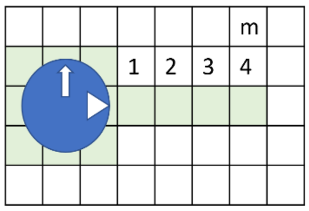
LHS sensor max grids: m = 4



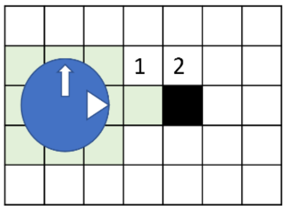
LHS = 1



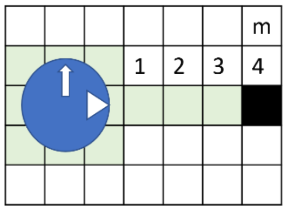
LHS = 3



LHS = 5,6,7.....



LHS = 2



LHS = 4

main.py

The module coordinates other modules in the system. There are four threads running in this module: the main thread, the serial communication thread for Arduino, the Bluetooth communication thread for android, and the detection thread using Raspberry Pi Camera.

The main body runs the exploration loop and the fastest path loop. Issues addressed are mainly synchronization and efficiency.

Main Logic

The exploration loop consists of the following steps:

1. Read sensor values from the serial communication thread.
2. Pass the raw sensor readings to *process_sensor.py* to probe the surroundings and update the map in *Explored.py.*
3. Based on the current map, compute the next movement(s) through *Exploration.py.*
4. If image detection is enabled, call *detection.py* to detect arrows on the right-hand side.
5. Based on the movements computed in Step 3, send the corresponding commands to the serial communication thread.
6. Call *AhBot.py* to update the local records of the robot's position and face direction after the movement.
7. Send the MDF string of the updated map, robot status and detection result to the Android tablet through the Bluetooth communication thread.

The fastest path loop consists of the following steps:

1. Based on the explored map, set the corresponding obstacles through *fastest_path.py*.
2. Compute the fastest path from the start position to the way point through *fastest_path.py*.
3. Compute the fastest path from the way point to the goal position through *fastest_path.py*.
4. Join the two paths above to form the final fastest path.
5. For each sub-path in the final path computed, send the corresponding command to Arduino through the serial communication thread.

Synchronization

Since constant polling of information is required in the serial and Bluetooth communication , multiple threads are used to save the overall processing time. However, it could cause the communication among devices to become out of sync. Therefore, four queues are implemented to facilitate synchronization of communications:

- *btinQ* – buffer for data from the Android tablet over Bluetooth
- *btoutQ* – buffer for data to be sent to Android tablet over Bluetooth
- *ardinQ* – buffer for data from Arduino over serial port
- *ardoutQ* – buffer for data to be sent to Arduino over serial port

During actual fetching and sending of data, length of the corresponding queue is constantly checked to see if data is available. This ensures that data are read and delivered in the correct order at the right time.

<u>Efficiency</u>

Since timing and correctness are the key measures in the challenge, the algorithm is tuned to achieve decent performance on these two attributes.

During actual testing, missing blocks was not much of a concern. In comparison, the presence of phantom blocks was compromising the correctness of exploration. While our Arduino team tried to fine tune the sensors, the algorithm was also improved by hard-coding certain configurations of the map, i.e. set space at grids where blocks can never be present (see explanations for *Explored.py*).

As mentioned in previous sections, multi-threading is used to save the overall processing time. Moreover, running the algorithm locally on Raspberry Pi saves communication time with PC through WIFI. Last but not least, the complexity of the given challenge does not require intense computation. Thus, running the algorithm in Python does not significantly affect the overall timing.