DEPARTMENT OF
Systems and Computer Engineering

Project Final Report - SYSC 4805

Team Blue Jeans
GitHub
Angela Byun *101004764*
Tarun Kalikivaya *101056745*
Martin Klamrowski *101009909*
Matthew Wesley-James *100944007*

2057 Words

Keywords: Mobile robot, maze, path-finding, search and rescue

9th April, 2021

Abstract
Team Blue Jeans proposes a mobile robot with maze navigating capabilities. The robot enters a maze with the purpose of locating an objective. The robot locates the objective, secures it, and exits the maze within the allowable time. This robot would have applications in search and rescue, and exploration. The physical elements of the robot can be simulated in CoppeliaSim. The robot's underlying intelligence is implemented in Python and integrated with CoppeliaSim using the provided CoppeliaSim Python API.

Contents

1 Charter

1.1 Objective

The objective of this project was to design a robot that enters a maze to find an objective then retrieve it by carrying it out of the maze. The concept is visualized in Fig. 1 below.

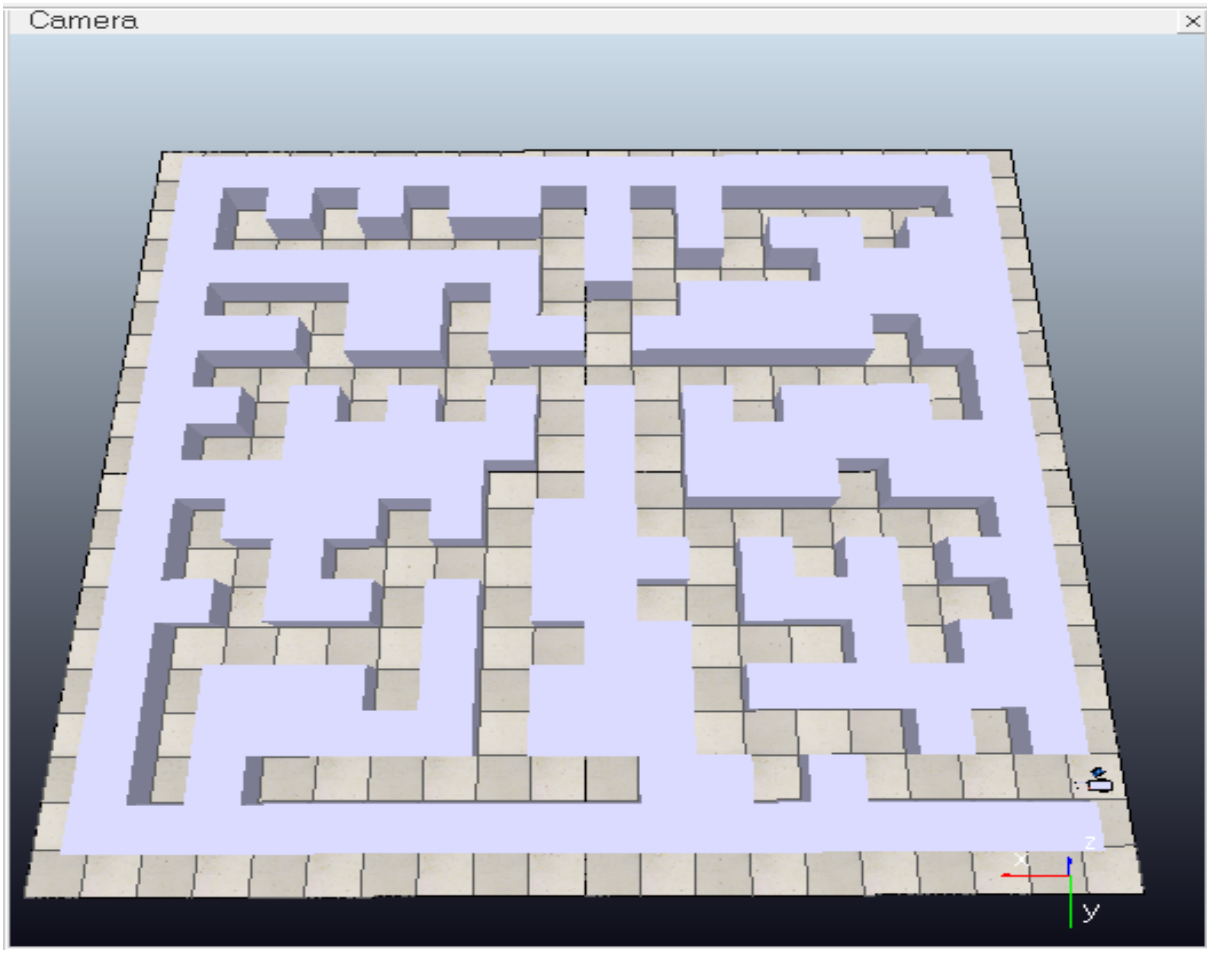

Fig. 1. Maze generated using maze generator

1.2 Deliverables

The purpose of this project was to attain two deliverables; first being the project management based on expectations for this course and the second being the final robot system, which is the maze robot. Project management was applied through the proposal, progress report, presentation, demonstration, and the final report by monitoring the progress. The proposal consists of how we planned on achieving the project goal, and the basic layout of the maze and the robot, including the milestone. The maze robot system consists of the maze, the robot, and the objective. The robot was programmed to find its way by using different sensors, and a path finding algorithm. We observed that our challenge was programming the robot because it required to move through and detect possible paths in the maze, find and carry the objective then find its way back out of the maze again.

2 Scope

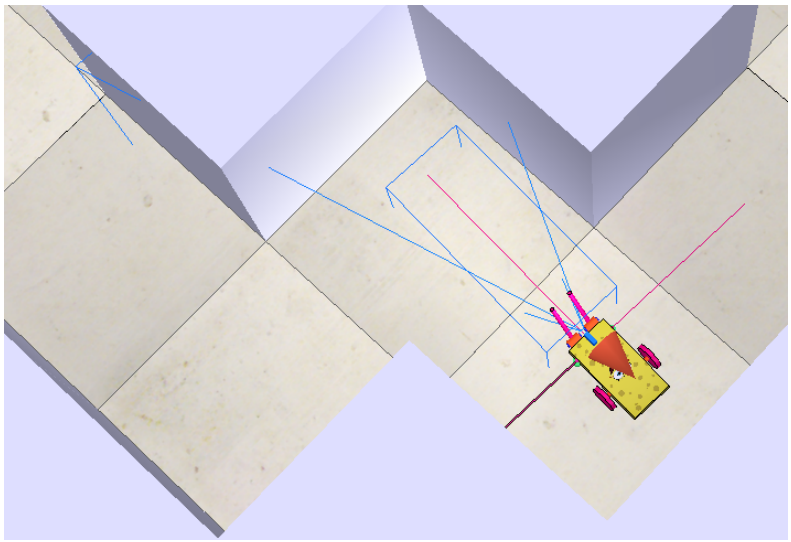

Fig. 2. Robot navigating a maze corridor

2.1 Requirements

1. The robot must be able to maintain constant and smooth forward and backwards motion at a speed that ensures reqs. 2, 3, 4 are also met.
2. The robot must be able to perform turns up to 180° in sharpness.
3. The robot's forward sensors must detect walls in the robot's path at least 0.5m away (the length of a single wall segment).
4. The robot's lateral sensors must detect walls immediately adjacent and parallel to the robot's path at least 0.25m away (half the length of a single wall segment).
   4.1. The robot must maintain 0.25m of distance from any detected wall segments, unless doing so conflicts with reqs. 5.1, 5.2.
5. The robot's sensors must be able to detect the objective in a 180° forward-facing arc of radius 0.5m.
   5.1. When the robot detects the objective, it must turn and move to the objective.
   5.2. The robot must stop when less than 0.05m away and facing the objective.
6. The robot must move forward until a blocking wall segment is detected or req. 5.1 and 5.2 are met.
   6.1. When the robot's sensors detect a wall segment in the robot's path, and only the robot's right lateral sensor detects a wall segment, the robot must perform a 90° turn to the left.
   6.2. When the robot's sensors detect a wall segment in the robot's path, and only the robot's left lateral sensor detects a wall segment, The Robot must perform a 90° turn to the right.

6.3. When the robot's sensors detect a wall segment in the robot's path, and both the left and right lateral sensors of the robot detect a wall segment, the robot must perform a 180° turn in place.

7. The robot must keep track of the path it takes through the maze.
   7.1. The robot must know what direction (in °) it is heading.
   7.2. The robot must have an odometer to track the distance it travels in each direction.
   7.3. The robot must prioritize turns into unexplored areas higher than turns into explored areas.
   7.4. When presented with multiple turns of differing priorities, the robot must take the higher priority turn.
   7.5. When presented with multiple turns of the same priority, the robot must randomly choose a turn to take.
   7.6. The robot must take the shortest path it knows when exiting the maze (seen in Fig. 3).

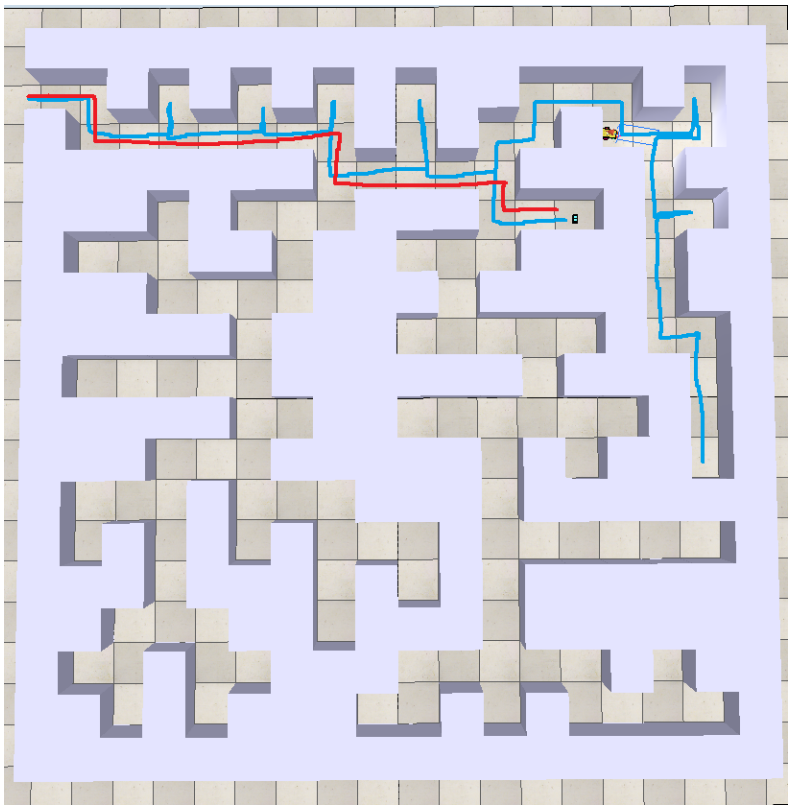

Fig. 3. Path taken by the robot in (Blue) and out (Red) of the maze

8. The robot must have a pair of prongs at its front that can be rotated around a central axis.
   8.1. If the prongs are at 0° when parallel with the ground, the prongs must be able to rotate to an angle of 75° above the ground.
   8.2. The robot lined up the prongs with the holes of the objective when req. 5.2 is active (seen in Fig. 4).

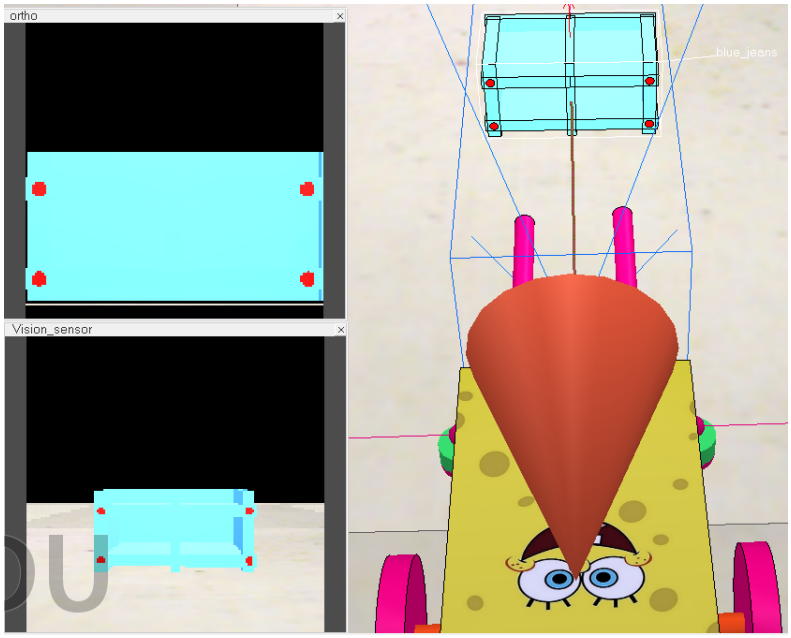

Fig. 4. The robot lining up prongs with objective holes

   8.3. Once req. 8.2 is satisfied, the robot secured the objective by lifting it up off the ground (seen in Fig. 5).

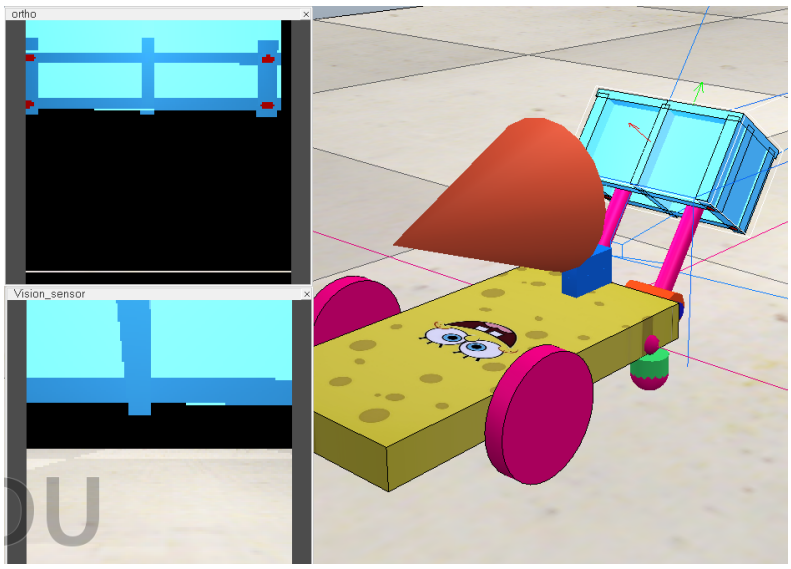

Fig. 5. The robot lifting the objective off the ground

   8.4. Once req. 8.3 is satisfied, the robot must work to complete req. 7.6 while carrying the objective.

2.2 Event vs. Time Triggering

The various subroutines and components that make up the robot are event-triggered.

1. Sensors:
   (a) The various sensors on the robot trigger on specific events. For example: the robot computed a change of course if a wall is detected (event) in its path.
   (b) Proximity sensor was used to navigate its way through and out of the maze by detecting where the walls are. The sensors detect 0.5m away from the robot body.
   (c) Vision sensor was used to detect where the object (jeans) is by detecting its color. The sensor detects 1 block ahead from the robot body. The perspective sensor allow us to see a wide angle view of what is in front of it. "Seeing" blue, triggers the objective "hooking" mode.
   (d) The orthographic sensor allows us to see an orthographic view of the objective. Orthographic means the objective is projected onto a 2D plane. This helps keep a consistent view of the view markers. The robot then attempts to align with the view markers. Using different colours, we can segment the orthographic view into red - and everything else. I.e., we mask out everything that is not red. The resulting image is segmented into 5 areas (or centroids). We ignore the first, which is "everything else", and compute distances between the red view markers, which are now defined positionally with their own centroids.

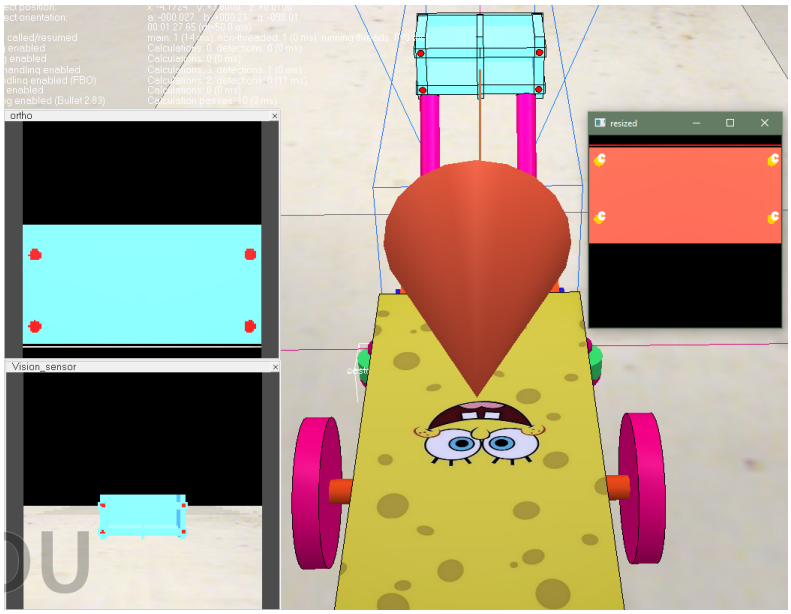

Fig. 6. Vision sensor cameras for perspective and orthographic in the left windows, and the orthographic view segmented based on colour on the right with view markers labelled as "C".

2. Navigation:
   (a) The underlying assumption in the navigation component is that the robot is able to know where it is within *its internal view* of the maze. It is important to reiterate that the robot does not know what the maze looks like upon simulation start.
   (b) Directions to take are determined by considering sensor readings (events) and the state of the robot's internal map (updated through events).

3. Prongs:
   (a) The objective was considered secure when the prongs reach an angle of 75° relative to the ground. This was ensured by setting the prongs' target joint position when req. 8.2 is satisfied. We designed the structure of the objective and the prongs in such a way that the objective would not fall off.

2.3 Maze Generation

The maze generation algorithm was built in Python using Prim's Random Maze Generation Algorithm.The underlying maze pattern would be generated by the Python Client and communicate the information to CoppeliaSim's remote API by calling a Lua function to generate and visualize the maze on the Resizable Floor object. Prim's Random Maze Generation Algorithm is a greedy algorithm that uses tree traversal to create random paths in a rectangular area. Before the actual maze generation occurs, the user is prompted to enter the bounds of the maze (i.e., length and width of the maze). This would determine how many blocks wide and how many blocks long the maze would be. Once the algorithm acquires these inputs, a two-dimensional matrix is created internally filled with wall blocks (denoted as a 'w' character in the code). A random wall block is then selected by the algorithm at a random edge of the bounds of the maze and turned into a path block (denoted as a 'c' character in the code). Every iteration of the algorithm, a random wall black adjacent to a path block will be turned into a path block, thus generating a random maze. The information of whether a specific block is a wall block or path block is all stored in the aforementioned two-dimensional matrix. This can be seen pretty-printed as shown in Fig. 7 below.

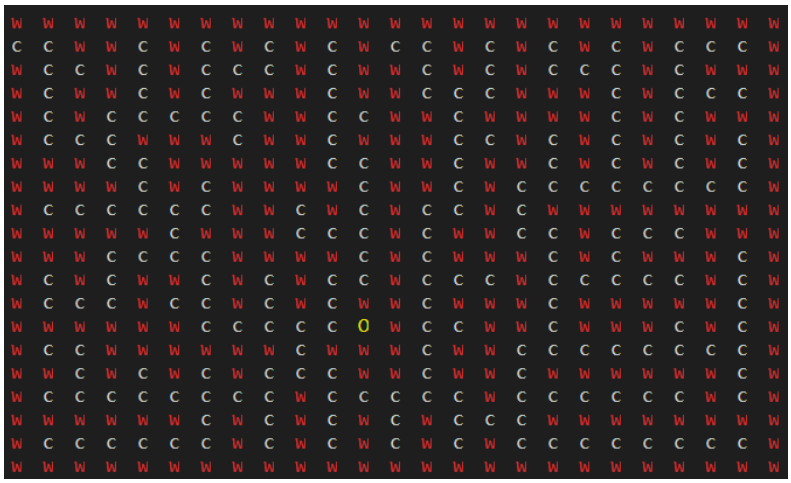

Fig. 7. A 20 x 25 block maze with an opening on the West wall

Another important feature in the maze is creating the opening, such that the robot can enter and exit the maze (with or without the object). This was done by changing a random wall object on the perimeter of the maze to a path block such that it would flow with the rest of the maze. Lastly, the object's position in the maze is also determined by the Python Client - it is always positioned at a dead-end in a maze (and denoted by 'O' in the code). Using this two-dimensional array, a function in the non-threaded Lua simulation script (associated with the Resizable Floor object) would be called in the boundary class of the overall system. This

function blindly iterates over the two-dimensional matrix, and whenever it encounters a wall block a cuboid is placed, an "object block" an object is placed, and when it encounters a path block nothing is placed. This all occurs at the time the simulation starts and the Python Client runs.

2.4 Navigation

The navigation algorithm was designed with the intent of dictating the main run loop for the robot (outlined in the sequence diagram, figure 16). It has 2 phases; first is the exploring phase & second is the phase where the robot moves through known territory to get to the exit. The exploration phase runs on a loop of pulling proxy sensor data, updating the map, deciding where to move, then moving. The moves are broken down to pivots (L,R), moving forwards (F), & finally capture image (C) to check for the Objective. To decide where to move the wallflower algorithm was used, to follow the wall to the left of the robot, making sure it will eventually find the object in every case. To get through known territory & exit the maze, the robot used a depth first search algorithm, only searching through known empty spaces. Pointers were implemented my copying numpy maze matrix references, they are used here to speed up both phases, by making efficient comparisons between neighboring blocks (or cells). For further information please see navigation.py.

2.5 Work Breakdown Structure (WBS)

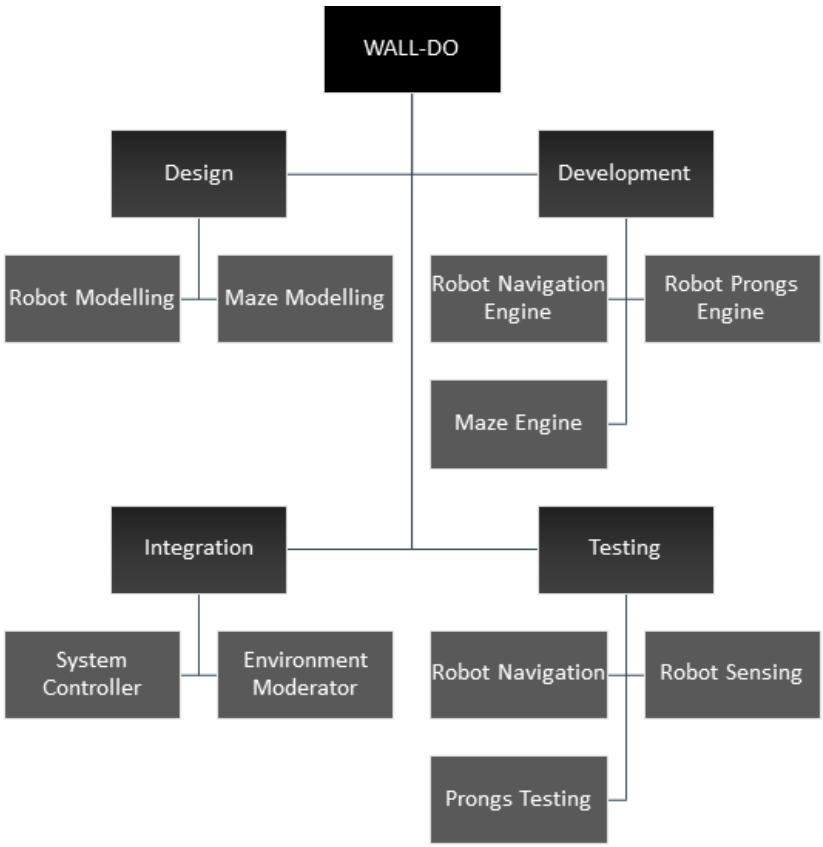

Fig. 8. WBS Diagram

2.6 Testing

- **Unit Testing**
  - Robot Navigation
    - Robot was tested to move straight at a constant speed, turn 90°, and 180°, and stop. This was performed to confirm requirements 1 and 2. These were verified by calling getObjectPosition() in a loop.
  - Robot Sensors
    - Each sensor were tested by having the sensor move toward a flat wall, the position of the sensor was tracked to verify the condition is met. This confirmed requirements 3 and 4 (using get/setObjectPosition).
    - The sensor was placed with a wall on one side and the objective on the other side (0.5m away), it rotated in place and detect what is in front of it. This confirmed requirement 5 (using get/setObjectPosition).
  - Hook Testing (prongs)
    - To complete requirements 8 (and the subsequent requirements), the hook was activated (after requirement 5.2 is met) then the hook was activated. Once the hook has signaled it is finished. The robot moved forward. If the robot's velocity equals the objective's velocity the requirements were satisfied.
- **Integration Testing**
  - The robot was placed facing towards a wall, and run forward, if it stops 0.5m away from the wall requirement 6 is met.
  - For requirements 6.1, 6.2, and 6.3 the robot was placed in the center with walls in four different cases (as shown in Fig. 9 below.) Once the correct direction is chosen the requirement was met.
  - For case A the robot was pre-programmed with each of the three directions having different unexplored area values, this tested requirements 7.4, 7.5, and 7.6.

Fig. 9. A: Front-facing wall. B: Front-facing wall and adjoining left wall. C: Front-facing wall and adjoining right wall. D: Dead end.

- **System Controller Testing**
  - For the requirements of 7 (and the subsequent requirements) a custom made testing maze was made, with trackers for each requirement. These will be waiting until the robot goes to a certain positions that was calculated based on the custom testing maze, then these requirements were complete.

- **Full System Testing**
  - Robot enters the maze (assuming robot moves at 0.5m per second) the robot have to exit the maze within the time limit to be successful. The time limit in seconds t_{max} is determined by Eq. (1), where l_M, h_M represent the maze length and height respectively in meters.

$$t_{max} = 2l_M h_M + 5 \tag{1}$$

3 Schedule

3.1 Schedule Network Diagram

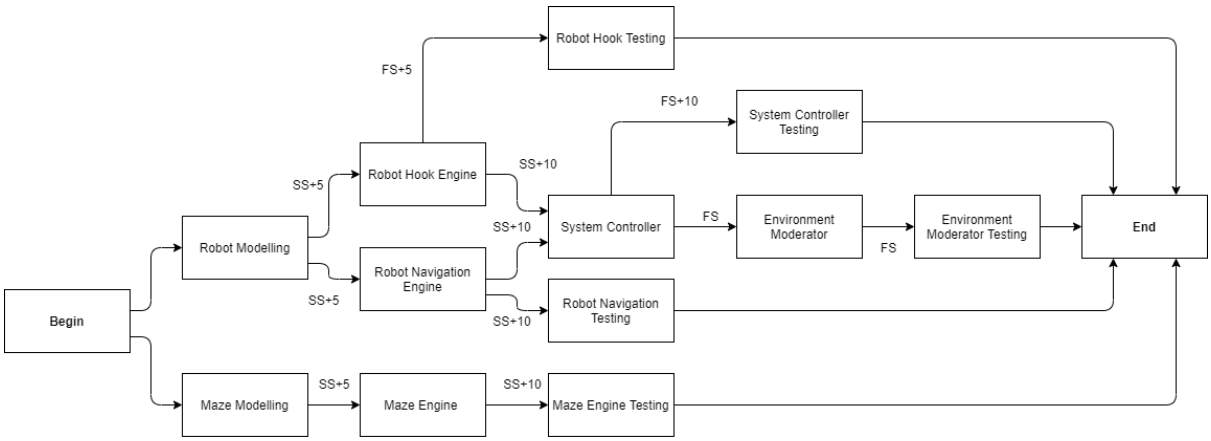

Fig. 10. Schedule Network Diagram

3.2 Gantt Chart

Fig. 11. Gantt Chart for Activities

3.3 Control Charts

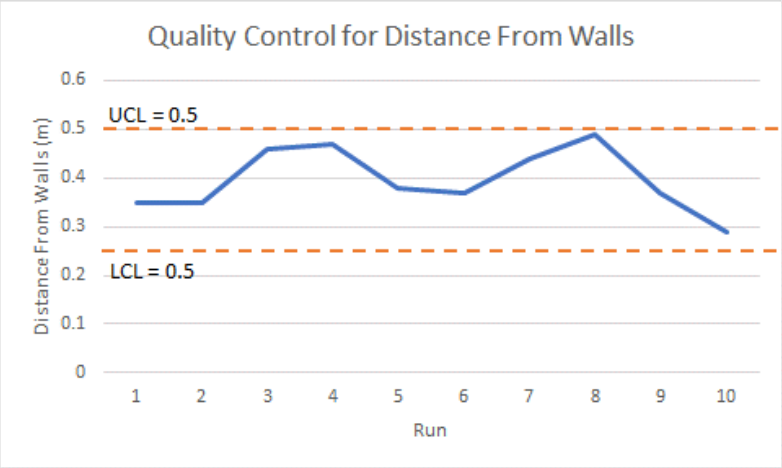

Fig. 12. Control Chart with robot's distance from walls

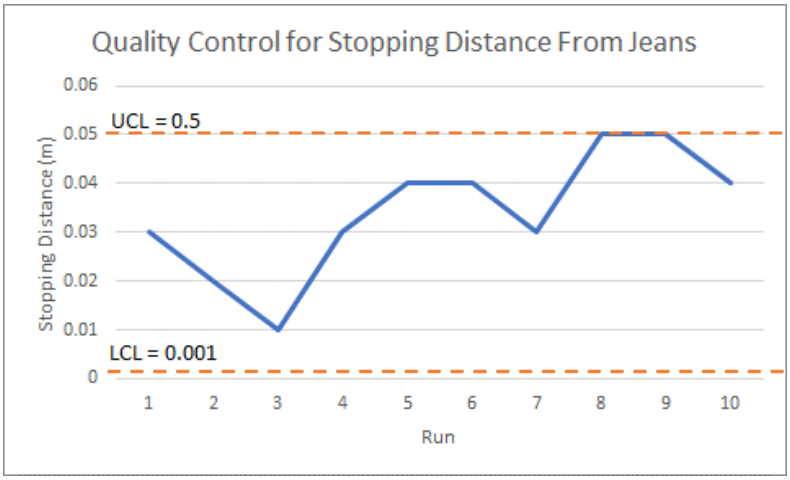

Fig. 13. Control Chart with robot's stopping distance from object

4 Diagrams

4.1 Architecture

The system architecture is visualized in Fig. 14 below. Only the "physical" robot is in CoppeliaSim. The underlying motor function, navigation logic, object detection is done in Python, with a connection being established with CoppeliaSim through the provided RemoteAPI module. The details for communicating and needed function calls are abstracted away in the Boundary class.

The Robo class maintains an instance of Boundary. The Robo class defines the robot's underlying behaviour. Robo also maintains an instance of Navigation is used by the Robo to interact and understand its environment. The composing Navigation object is used to maintain a representation of the maze, internal to the Robo. Maze contains algorithms for generating a maze and the objective within.

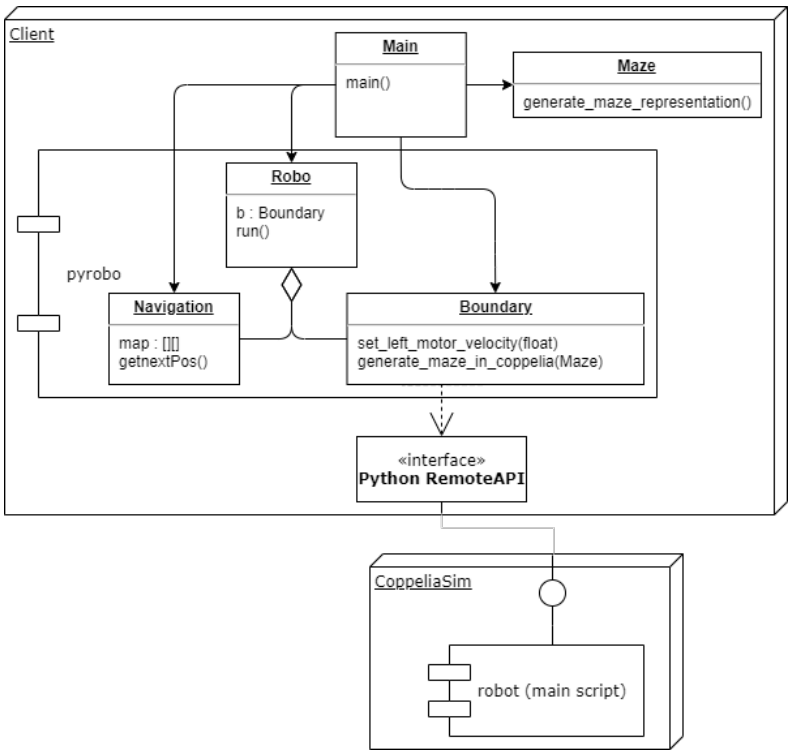

Fig. 14. System Architecture

4.2 State Chart Diagram

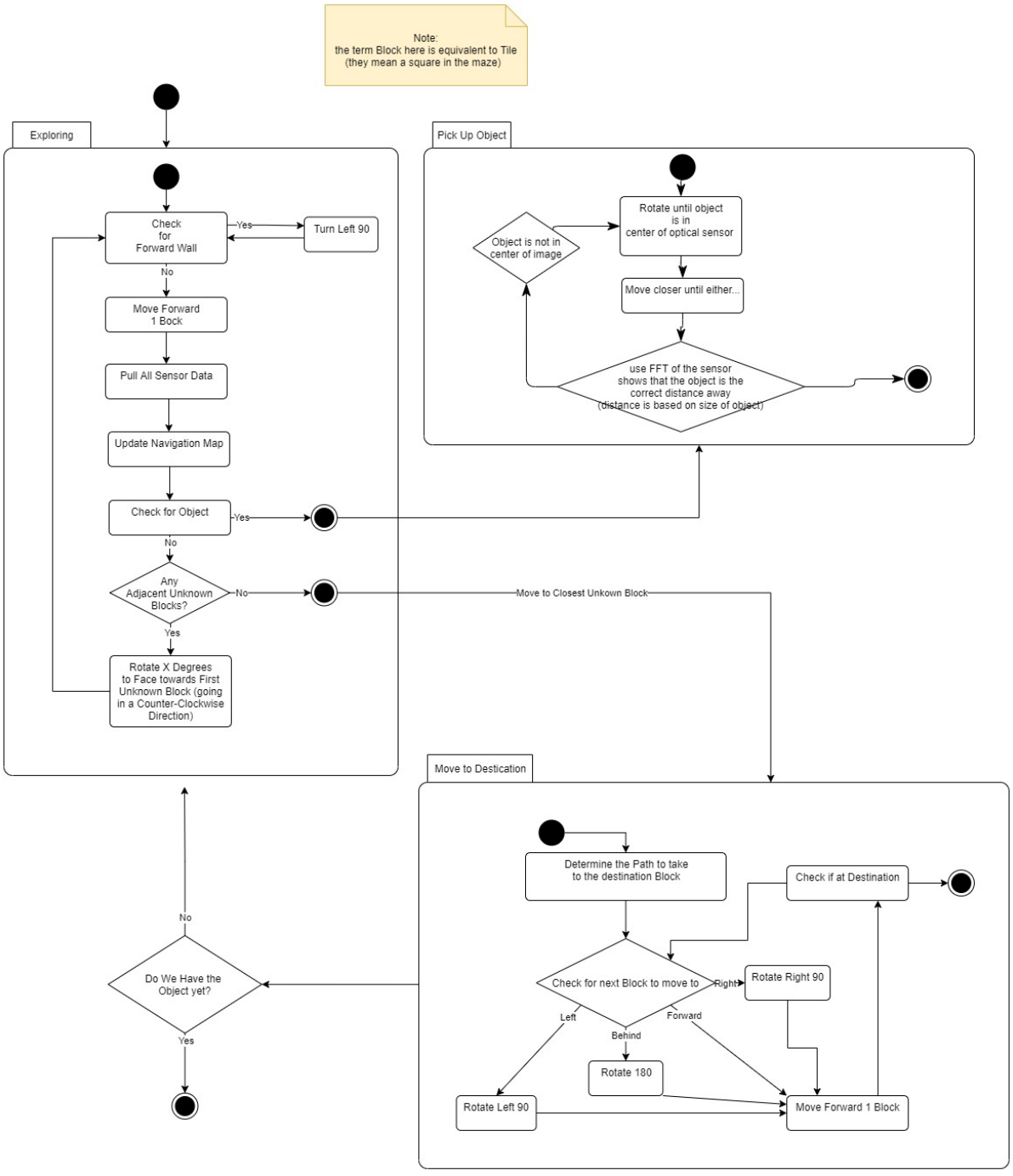

Fig. 15. State Chart Diagram

4.3 Sequence Diagram

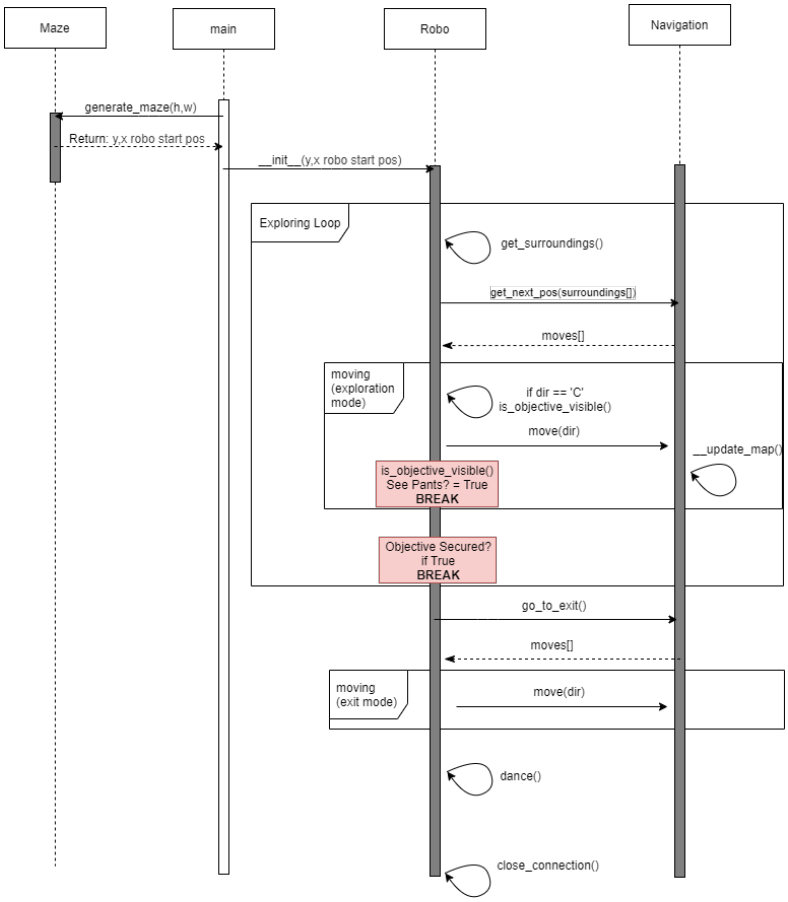

Fig. 16. Sequence Diagram

5 Human Resources (HR)

5.1 Responsibility Assignment Matrix

	Angie	Tarun	Martin	Matthew
Robot Modelling	R	A	R	A
Maze Modelling	R	R	A	A
Robot Navigation Engine	A	A	R	R
Robot Hook Engine	A	R	A	R
Maze Engine	R	A	R	A
System Controller	R	R	A	A
Environment Moderator	A	A	R	R
Robot Navigation Testing	A	R	A	R
Robot Hook Testing	R	A	R	A
Maze Engine Testing	R	R	A	A
System Controller Testing	A	A	R	R
Environment Moderator Testing	A	R	A	R

Legend: R -- Responsible, A -- Approver

Fig. 17. Responsibilities.

- **Martin**
  - Robot Boundary, prongs
- **Matthew**
  - Robot Navigation
- **Tarun**
  - Maze Generation
- **Angie**
  - Sensors