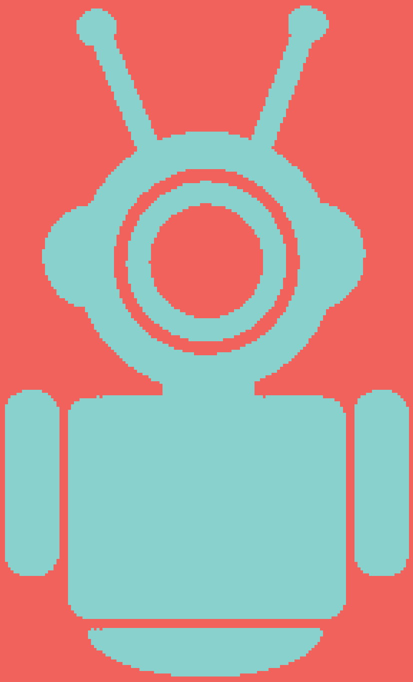
CODE DOCUMENTATION



Automated Kiwifruit Picking

Test Drive

CONTENTS

FUNCTIONALILTY

MOVEMENT

MOVEMENT

ENTITY MOVEMENT

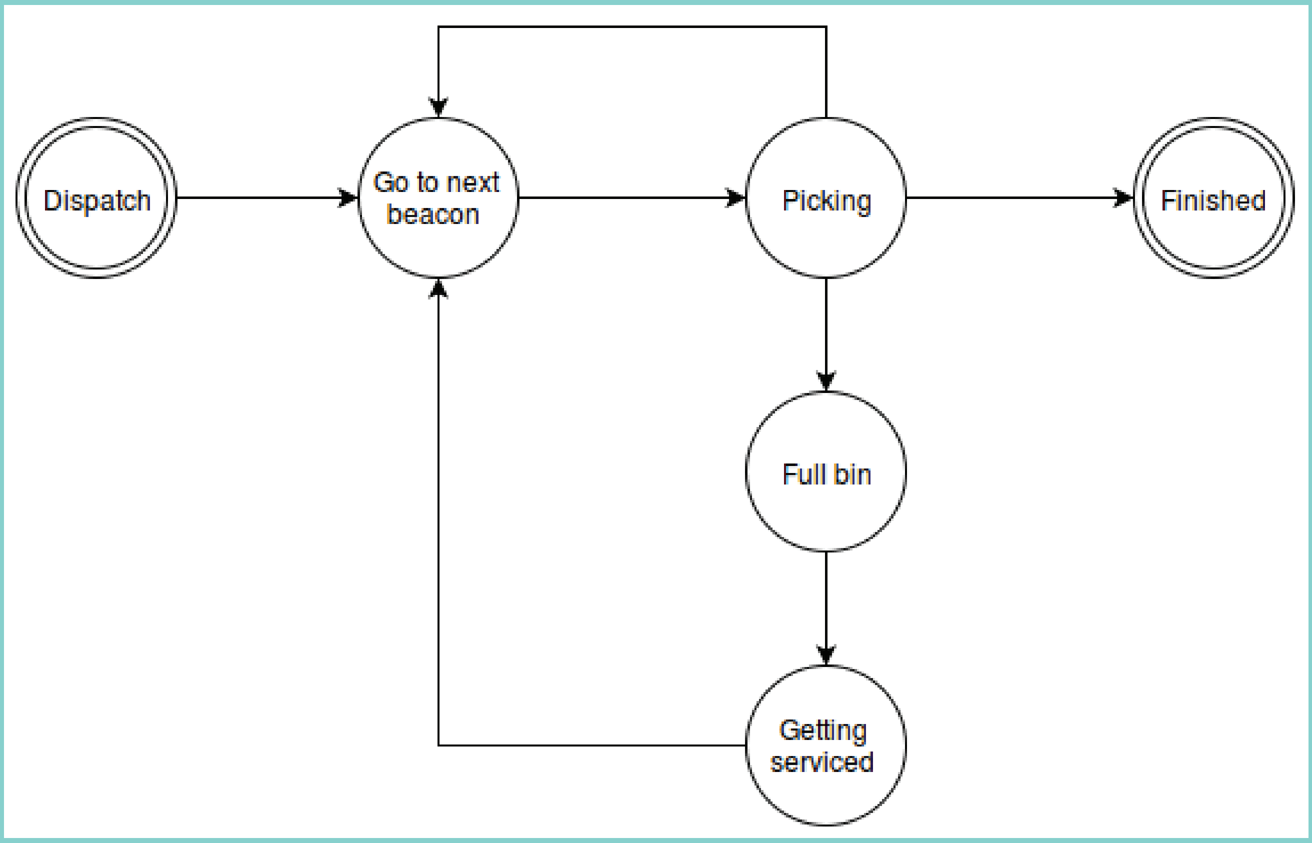
MOVEMENT

PICKER SPECIFIC MOVEMENT

PICKER WORK SPLIT

MOVEMENT

PICKER STATE DESIGN



- The Picker robot's AI was designed to be state-driven. This design was chosen instead of other candidates because by using a finite state machine, this was the most easily modelled and understood. Considering two people were working hand in hand on this mission critical component of the system, it was of utmost importance that they were both on the same page in regards to the design of the Picker's AI.

- Upon being created, the Picker robot begins in the "Dispatch" state. At this point the Picker robot already knows which kiwifruit picking path it has been assigned. It will subscribe to the robotic beacon at the start of its path and will wait until it begins to received pose messages from that beacon. Once it learns the location of the beacon it will move to it. Once it reaches that beacon it will switch to the "Go to next beacon" state.

MOVEMENT

PICKER STATE DESIGN

- In the “Go to next beacon” state the Picker robot checks if it is on the West or East side of an orchard row. Based on this it is able to figure out which beacon is next on its picking path. It then subscribes to this beacon and once it begins receiving the position of the beacon it will move to it. The Picker is in this state when it finishes picking a row and then needs to figure out which beacon to go to so it can enter the next row in its path.
- In the “Picking” state the Picker slows down its speed drastically so it is able to pick kiwifruit without missing any. While in this state, the Picker will be travelling through a row (between two beacons) and hence will be within picking range of kiwifruit pergola structures.
    - During this state the Picker will continuously poll to check if its kiwifruit bin is full (it does so each time it picks a kiwifruit tree). If it is full the Picker will switch its state to “Full bin”.
    - If the Picker reaches the end of a row (when it is at the position of the beacon at the end of that row) it will check to see if it has reached the end of its picking path. If it has not, then it will switch to the “Go to next beacon” state so it can find its next row. If it has reached the end it will switch to the “Finished” state.
- As soon as the Picker is in the “Full bin” state it will contact a Carrier robot to come and switch its full bin for an empty bin so it can continue picking fruit. It will remain stationary until this happens. Once the Picker has been notified by a Carrier that it is coming to help, the Picker will change its state to “Getting serviced”.
- During the “Getting serviced” state the Picker is still stationary. This state is required so other idle Carriers do not also come to the aid of this Picker (they can check to see if the Picker is getting serviced). Once the bin exchange has been made the Picker will switch back to the “Picking” state.

MOVEMENT

CARRIER SPECIFIC MOVEMENT

MOVEMENT

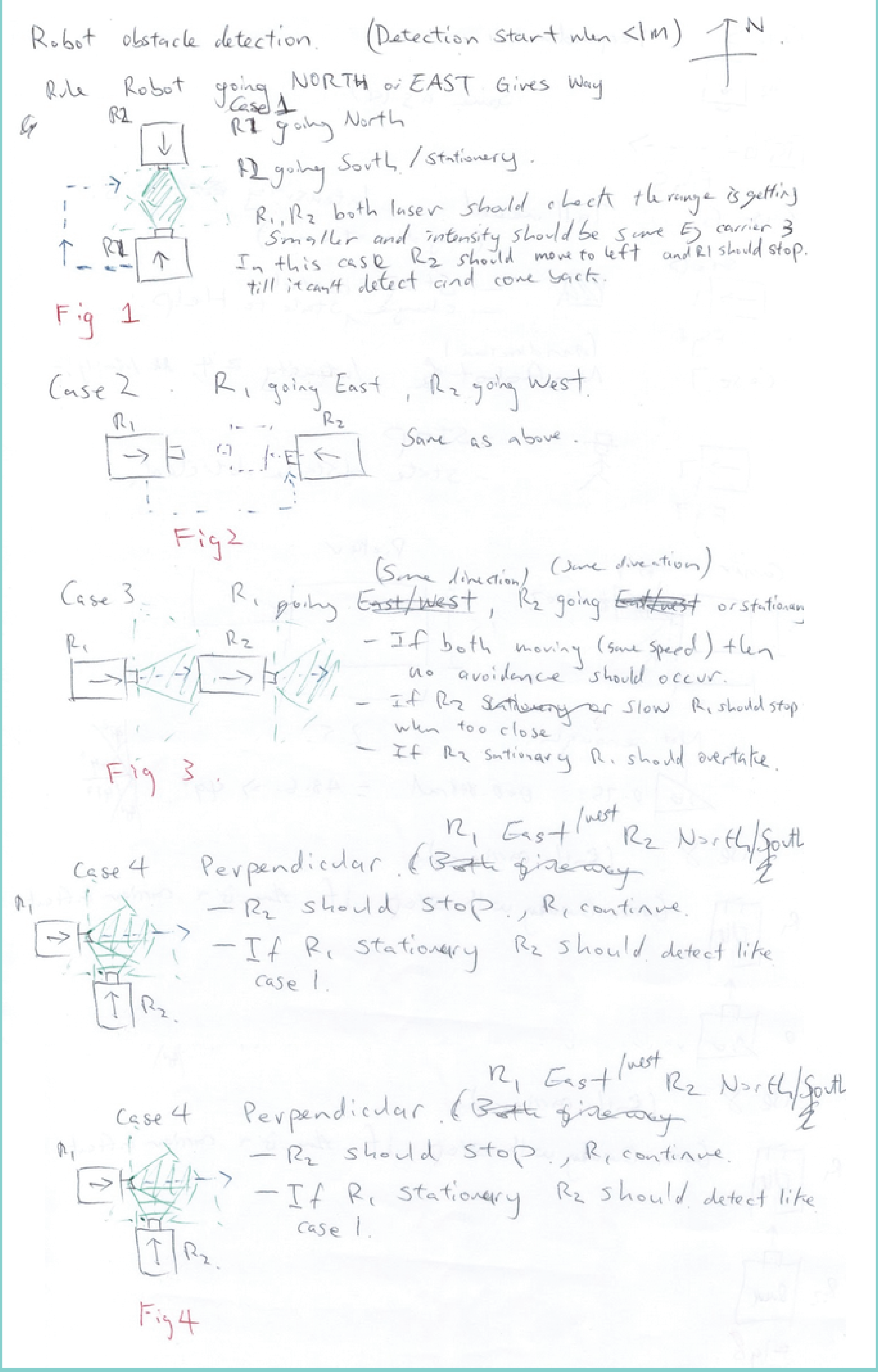
CARRIER STATE DESIGN

MOVEMENT

OBSTACLE DETECTION & AVOIDANCE

LASER INTENSITY

OBSTACLE DETECTION & AVOIDANCE



Robot obstacle detection. (Detection start when <1m) N.

Rule Robot going NORTH or EAST Gives Way

Case 1

R1 going North

R2 going South / stationery.

R_1, R_2 both laser should check the range is getting smaller and intensity should be same E5 carrier 3. In this case R_2 should move to left and R1 should stop till it can't detect and come back.

Fig 1

Case 2. R_1 going East, R_2 going West.

Same as above.

Fig2

Case 3 R_1 going ~~East/West~~ (Same direction), R_2 going ~~East/West~~ (Same direction) or stationary

- If both moving (same speed) then no avoidance should occur.
- If R_2 ~~Stationary~~ or slow R_1 should stop when too close
- If R_2 stationary R_1 should overtake.

Fig 3.

Case 4 Perpendicular. (~~Both give way~~ R_1 East/West R_2 North/South

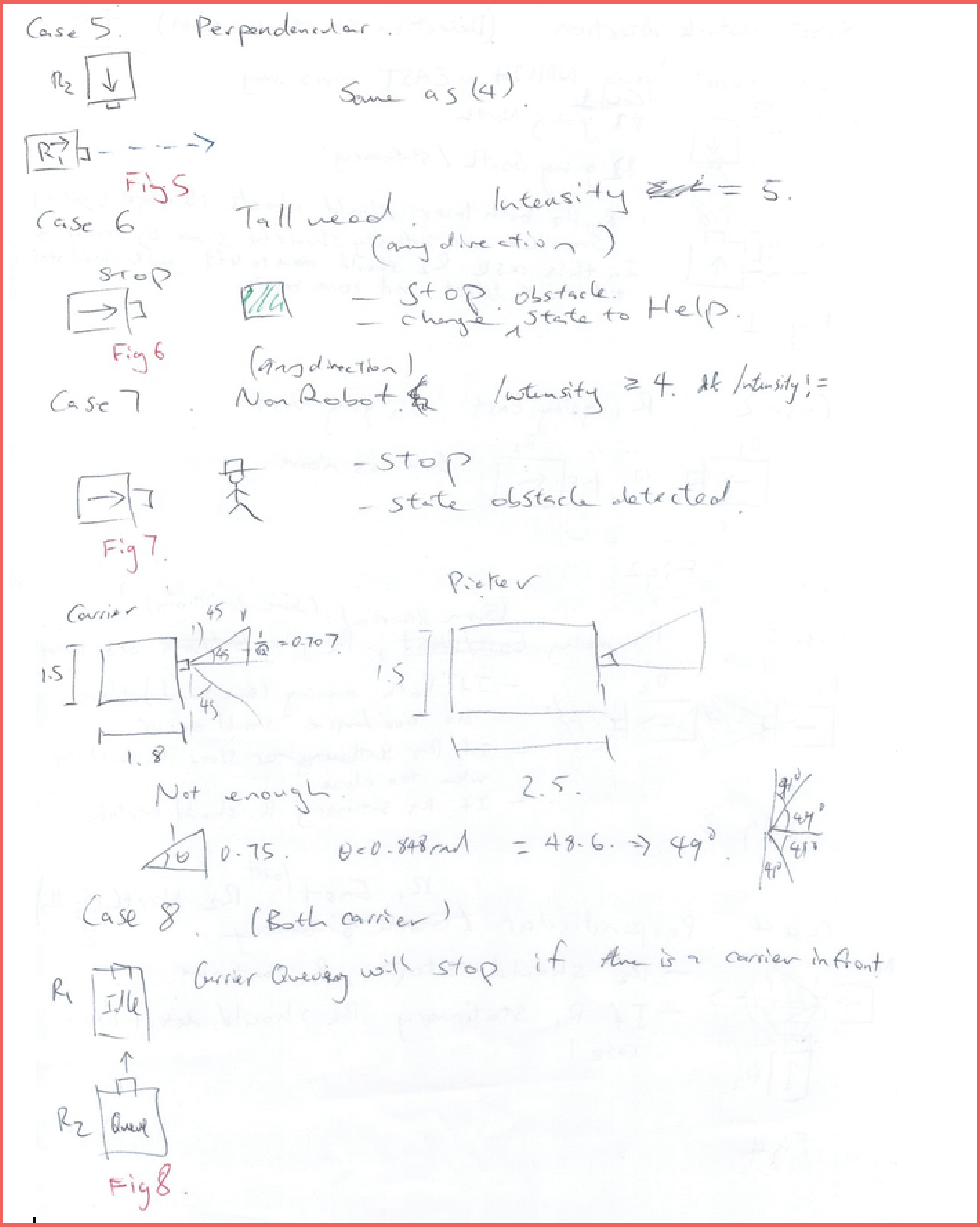
- R_2 should stop, R_1 continue.
- If R_1 stationary R_2 should detect like case 1.

Case 4 Perpendicular. (~~Both give way~~ R_1 East/West R_2 North/South

- R_2 should stop, R_1 continue.
- If R_1 stationary R_2 should detect like case 1.

Fig4

OBSTACLE DETECTION & AVOIDANCE



Case 5. Perpendendular.

R_2

Same as (4).

R_1

Fig 5

Case 6 Tall need. Intensity = 5.
(any direction)

stop

- stop obstacle.
- change state to Help.

Fig 6

Case 7. Non Robot. (any direction) Intensity ≥ 4. && Intensity !=

- stop
- state obstacle detected.

Fig 7.

Carrier

45

1

$\frac{1}{\sqrt{2}} = 0.707$.

45

1.5

45

1.8

Picker

1.5

Not enough. 2.5.

1

θ

0.75. $\theta < 0.848$ rad = 48.6. → 49°.

41°

49°

41°

41°

Case 8. (Both carrier)

Carrier Queuing will stop if there is a carrier in front

R_1 Idle

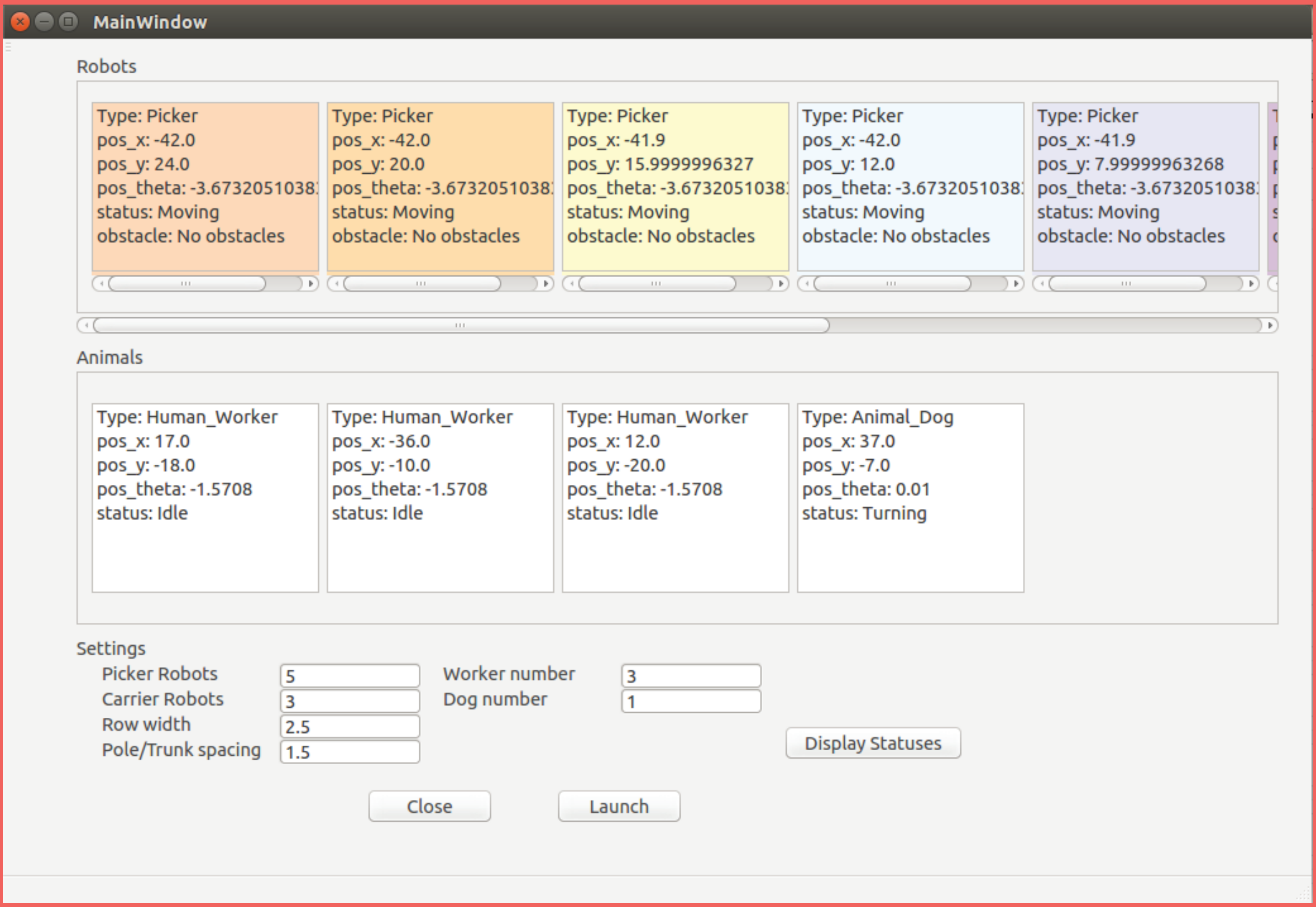
R_2 Queue

Fig 8.

ANIMAL & PERSON BEHAVIOUR

- The class structure contains an Entity superclass which is subclassed by the Animal and Person class. Specific animals and people such as dogs, cats, workers, gardeners, and neighbours are subclassed from Animal or Person. This allows generic behaviour specific to all entities, people or animals and does not need to be repeated. The subclasses only contain behaviours that are specific to them.
- The workers and gardeners model the orchard workers, so they will move around the orchard regularly and trim the trees or remove weeds. The worker trims all the trees in a single column instead of trimming any tree in the orchard to reduce the complexity of the implementation. This allows us to spawn multiple workers and assign each worker with a column without having to worry about multiple workers trimming the same tree.
- The neighbour and blind person are not trained to be around robots so they may move around more randomly. The blind person may follow the dog, which models the use of a guide dog .
- Animals such as dogs and cats have specific behaviours such as moving in circles and lying close to trees, though they may get out of the path of other robots if it sees the robot.

GRAPHICAL USER INTERFACE



Robots

Type: Picker
pos_x: -42.0
pos_y: 24.0
pos_theta: -3.6732051038
status: Moving
obstacle: No obstacles

Type: Picker
pos_x: -42.0
pos_y: 20.0
pos_theta: -3.6732051038
status: Moving
obstacle: No obstacles

Type: Picker
pos_x: -41.9
pos_y: 15.9999996327
pos_theta: -3.6732051038
status: Moving
obstacle: No obstacles

Type: Picker
pos_x: -42.0
pos_y: 12.0
pos_theta: -3.6732051038
status: Moving
obstacle: No obstacles

Type: Picker
pos_x: -41.9
pos_y: 7.99999963268
pos_theta: -3.6732051038
status: Moving
obstacle: No obstacles

Animals

Type: Human_Worker
pos_x: 17.0
pos_y: -18.0
pos_theta: -1.5708
status: Idle

Type: Human_Worker
pos_x: -36.0
pos_y: -10.0
pos_theta: -1.5708
status: Idle

Type: Human_Worker
pos_x: 12.0
pos_y: -20.0
pos_theta: -1.5708
status: Idle

Type: Animal_Dog
pos_x: 37.0
pos_y: -7.0
pos_theta: 0.01
status: Turning

Settings

Picker Robots 5
Carrier Robots 3
Row width 2.5
Pole/Trunk spacing 1.5
Worker number 3
Dog number 1

- The design includes a graphical user interface which is designed to show the status of the system in a clear way to the users. It uses the C++ QT4 library which is cross-platform and built into the lab image of Ubuntu.
- Each dynamic object shows its current position, velocity, and current state. This will update dynamically using background threads so that the interface can continue to be responsive.
- A colour key is used which links the background colour of the panels with the colour of the objects, so it is easy for the user to tell which panel refers to which object.

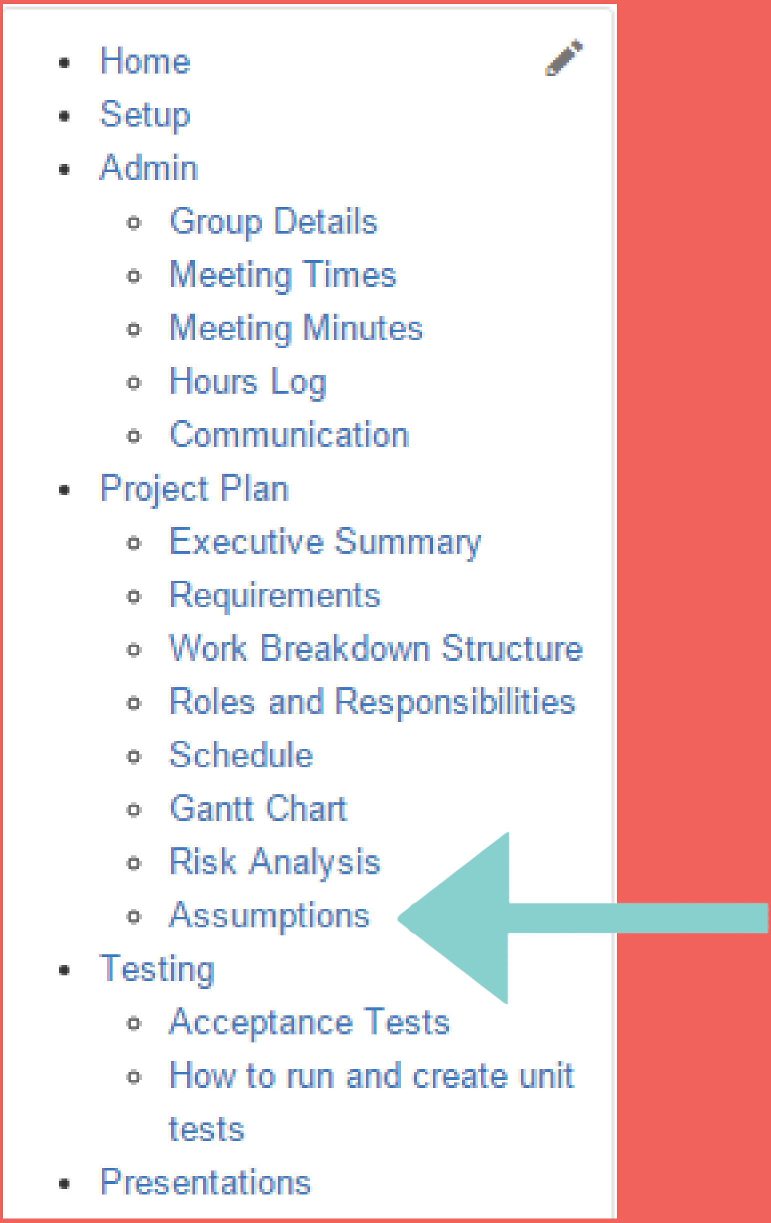
CONFIGURABLE ORCHARD

- Due to the specifications in the brief mentioning the possibility of configurable dimensions in the orchard, it was decided that a world and launch file generator would be used which would generate the orchard based on the values that the user enters into the user interface.
- The user interface can also be used to set the number of robots, animals, and people in the orchard, to allow the user to easily customise the simulation.
- Restrictions are placed on the number of robots that the user can create to represent a real world scenario, maintain performance and prevent overloading Stage.

• Picker	1 - 7
• Carrier	1 - 7
• Cat	0 - 5
• Dog	0 - 5
• Worker	0 - 4
• Garden Worker	0 - 4
• Blind Person	0 - 3
• Neighbour	0 -3
• Row Width	3 - 6m
• Pole Spacing	1 - 10m

- The world components specified by the user are randomly generated inside the orchard. All components, other than the tall weeds, are generated inside the kiwi-fruit farming area, not the entire orchard. To ensure that these components do not spawn inside or too close to the kiwifruit trees they're spawn points are ensured to be equal distance apart from the trees surrounding them.
- The tall weeds spawn randomly around the orchard. However, they are not allowed to spawn on or near the driveway. While weeds can be removed on the simulation; this decision was made as weeds spawning on these areas hindered the simulation's speed unnecessarily.
- The trees that surround the perimeter of the orchard have their colours randomly picked from an array of colours.

DESIGN ASSUMPTIONS



- Home
- Setup
- Admin
  - Group Details
  - Meeting Times
  - Meeting Minutes
  - Hours Log
  - Communication
- Project Plan
  - Executive Summary
  - Requirements
  - Work Breakdown Structure
  - Roles and Responsibilities
  - Schedule
  - Gantt Chart
  - Risk Analysis
  - Assumptions
- Testing
  - Acceptance Tests
  - How to run and create unit tests
- Presentations

- The project’s design assumptions can be found under the “Assumptions” category in the GitHub Wiki.

CONSTRAINTS

ENVIRONMENT

TECHNICAL

DESIGN PRINCIPLES

DESIGN PRINCIPLES

FLEXIBILITY

- The design is flexible as it allows for further people, robots, or animals to be added without needing to modify existing entity classes, and duplicating movement code wouldn't be necessary since these are included in parent classes.
- Behaviours are kept separate from low level movement details, so changing the behaviour of one entity will be completely separate from other entities
- The use of callbacks and publisher/subscribers means that specific dependencies between entities are not needed, for example if any other robot also wants to subscribe to the picker robot, they can simply subscribe to the topic.

MAINTAINABILITY/ EXTENSIBILITY

- Movement queue allows for easy addition of new behaviours.
- Generator class means that new entities can be generated automatically into the world file and launch file.
- Use of beacons to guide robots means that further parameters such as row length and number of rows could be added if time allowed.
- GUI showing the status is good for maintenance as it makes the system state obvious.

DESIGN PRINCIPLES

ROBUSTNESS

- The use of spinners in the GUI ensure that no illegal arguments are passed, as these allow for maximum and minimum values to be set. Additionally, non-numeric values cannot be passed.

USABILITY

- The GUI ensures that configuring the simulation is done quickly & easily. There is a label for each component of the simulation that can be configured and a spinner to adjust the value for each component. Additionally, there are buttons to launch & close the simulation as well as starting the innovative feature, controlling the tractor. Furthermore, the GUI easily allows tracking of each world component as their colour on the GUI corresponds to the component's colour in the world. These panels also allow the user to easily track the component's position, orientation & status.

DESIGN DECISIONS

PATTERNS USED

- The design uses a publisher/subscriber pattern as this is how ROS was designed to work, and it mimics the behaviour of swarm bots. There is no “master” node and instead the robots communicate with others through publishing and subscribing to topics.

C++ vs Python

- C++ was chosen instead of python because it is more widely used in the robotics industry so it was thought that using C++ would be a more valuable experience. It also has the QT4 gui library built in to the lab computers, which includes the ability to use background threads to process information and sends signals to update the user interface, which is what was required for the status panels.

SOFTWARE DESIGN

MODULARITY

SOFTWARE DESIGN

PERFORMANCE

SOFTWARE DESIGN

CONFIGURATION

2