Known Error Prevention and Recovery in Autonomous Ground Vehicles

MSc Robotics Dissertation

Oliver Grubb

School of Computer Science

College of Engineering and Physical Sciences

University of Birmingham

2023-24

Abstract

Due to the unstructured nature of real-world environments, robots struggle to successfully complete long-time horizon tasks in non-controlled settings. The likelihood of encountering a failure inducing error increases with the length of a given task, therefore a reliable method of dealing with such anomalies is essential for the further improvement of autonomous systems. To rectify this problem, an error recovery pipeline was created and applied to a robot vacuum. A simulated world was created and floor coverage instructions were generated by a global task planner, enabling the robot to navigate over the entire surface area of a given map. The robot vacuum was controlled by a library of custom behaviours combined together into high-level behaviour trees, capable of executing the generated instructions. Three industry-identified errors were selected as anomalies: raised doorway boundaries, cables, and states where the robot's wheels were off of the ground. Custom recovery algorithms were created to safely navigate the three problems when they were detected. To make the system autonomous, the recovery pipeline consisted of a detector node responsible for passively monitoring the simulated robot's sensors for known errors to occur, a failure-mode and effect analysis table which matched detected errors to the correct recovery solutions, and a responder node which generated new instructions via the custom recovery algorithms to counter the error. It was shown that without the recovery pipeline enabled the robot would consistently fail its task when any of the three anomalies were added to an environment. When the recovery pipeline was enabled, it was shown that the system could reliably overcome the problems, successfully completing its task. Furthermore, the overhead time of the recovery solutions was analysed and the addition of a re-planning step for the global task planner after a recovery had taken place was shown to be reduced by up to 61%.

Acknowledgements

I would like to express my deepest appreciation to Dr Masoumeh Mansouri for her support and guidance as project supervisor. Her mentorship not only supported my academic growth, but also made the research process enjoyable. I would also like to extend my thanks to Dr Yassin Warsame and Dr Charlie Street for their contribution of expertise which were greatly appreciated.

I am also grateful to my family who have always supported me in my ambitions, and in particular my fiancée Fizz, who has stood by my side throughout this journey. I could not have done it without you.

Abbreviations

2D	Two-Dimensional
3D	Three-Dimensional
A*	A-Star
AI	Artificial Intelligence
AMCL	Adaptive Monte Carlo Localisation
AWS	Amazon Web Services
BFS	Breadth First Search
BN	Bayseian Network
BT	Behaviour Tree
CNN	Convolutional Neural Network
CRM	Conditional Random Field
FMEA	Failure Mode and Effect Analysis
HMM	Hidden Markov Model
IMU	Inertial Measurement Unit
LIDAR	Light Detection and Ranging
LSTM	Long Short Term Memory
ML	Machine Learning
RGB-D	Red Green Blue - Depth
ROS	Robotic Operating System

Contents

List of Figures

List of Tables

CHAPTER 1

Introduction

In a world where software can write poems, compose music, and generate photo-realistic images, it is reasonable to have high expectations of the capability of hardware. Indeed, the potential of robotics and embodied artificial intelligence is immense. The deep integration of robotics into all aspects of society has the potential to free individuals from benign tasks, increase free time, and improve quality of life. Despite such merits, even with the great advances in technology seen in recent decades, the envisaged future of the field of robotics has not yet come to fruition. Although some personal autonomous devices such as robot vacuums exist, they are still not widely adopted. The reasons behind the lack of widespread adoption has been attributed to a number of factors including cost, battery capacity, and maintenance. However, a significant problem reported by users is the unreliability of their performance [1]. The robots often get stuck, are unable to operate around complex environments, and simply do not clean as well as a human could. Such complaints are not specific to robot vacuums. All robots expected to function in a world designed and used by humans naturally struggle to achieve results comparable to those possible by an equivalent human operator. Consider a human tasked with vacuuming a room. As they proceed to navigate around the floor space they may encounter a loose object which they would correctly identify to pose a hazard to their task. Depending on the object, their response may differ. If the object were a sock, they may decide to navigate around, or they may pick it up and move it to the side as it would likely clog the vacuum mechanism. If they were to encounter a cable, they may use the vacuum to push it out of the way, either to an already cleaned area or up against a wall. Either way, rather than naively following the original plan, the human detected, identified, and made an appropriate action to mitigate the effect of the problem. For robots to ever achieve, or exceed, human level performance they must be able to make similar informed choices.

To improve performance in autonomous systems, such as robot vacuums, it is important to consider the complexity and possible failure points of their task.

If a system is expected to act autonomously for a prolonged period of time in an unstructured environment, it is logical to assume that the probability of encountering a problem will increase with the length of the task. To mitigate this risk in long time horizon tasks, robots should be equipped with robust methods of dealing with known problems they are likely to encounter. Such methods should seamlessly allow a robot to deviate from its original plan to prevent, or react, to an identified anomaly. For example, if a robot vacuum is known to get stuck on doorway boundaries between rooms, the ability to autonomously detect, identify, and react to such problematic states should be available to the robot. With this ability, the likelihood of the vacuum completing a long time horizon task, such as navigating the floor surface-area of a house, would increase. Moreover, the more potential failure states identified, and respective recovery strategies implemented within the robot, the higher the chance of successfully completing a given task.

This work aims to improve the execution of long-time horizon tasks in robots. To achieve this, a novel architecture for task execution was developed, allowing the interruption of a main-objective task planner, to run preventative and recovery actions when known anomalies are present in the environment. The application of this work is focused on the context of robot vacuums, with the implementation of novel recovery algorithms for three industry known failure states: the presence of cables in the work environment, the traversal over doorway boundaries, and states where the robot's wheels are not touching the ground. Although the implementation of this work is focused on robot vacuums, the architecture of the system is purposefully modular and context agnostic, allowing the high-level design to be applied to any robot type. This work has made a significant impact, leading to its inclusion in the second annual review meeting for the European Union Horizon CONVINCE project [2].

CHAPTER 2

Literature Review

From the Egyptian water clocks of 3000 BCE to the Jacquemarts created throughout the middle ages, it is clear the concept of creating mechanical devices to automate tasks is not a modern desire. Fuelled by the invention of programmable computers, since the 1950s modern robotics research has seen autonomous systems become more sophisticated and capable. Pioneers such as George Devol and his 'Programmed Article Transfer' patent [3] and subsequent first ever industrial robot 'Unimate' [4] showed the world that machines could perform complex tasks previously only completable by humans. Unimate's ability to record and replay a sequence of user movements, although crude by today's standards, was revolutionary, allowing the true automation of repetitive tasks. The potential of such robots to drastically improve industrial efficiency was apparent, leading to the widespread adoption of industrial manipulators to execute tedious and dangerous tasks. Nowadays, industrial manipulators can perform a vast variety of tasks to incredibly precise tolerances, at high speeds and efficiency, vastly outmatching the ability of humans. With such robots not only existing but also widely used, the lack of robots in everyday life for an average person can understandably be considered surprising.

The impressive abilities of industrial manipulators generally rely on a number of key conditions. Firstly, industrial robots require structured environments where every item needed is present in the workspace, and are ensured to be in specific locations, or are fully observable by accurate sensor arrays [5]. Additionally, thanks to a static base and joint encoders, a manipulator robot's location and pose is always known to a bounded degree of accuracy. Unfortunately, for the majority of non-industrial tasks, these conditions are too limiting for the methods used to be applied in unstructured environments. The complex and often dynamic nature of real world environments mean for a robot to successfully complete a task, it must be able to respond to the unexpected. To do this, a robot must be able to accurately detect, identify and respond to a problem either pre-emptively or once the problem has occurred.

2.1 Detection and Identification

General anomaly detection has a long history of research in signal processing literature [6]. However, since many systems operate with high-dimensional data such as images from a camera feed, supervised learning techniques are a common method attempted to detect anomalies [7]. With recent advancements in Artificial Intelligence (AI) and Machine Learning (ML), many studies have focused on applying such techniques for the detection and identification of errors in robots. Fino-Net [8] uses a Long-Short Term Memory (LSTM) neural method approach which accepts the fused data of multiple sensors on a Baxter manipulator robot and attempts to detect if an error has occurred. The network was trained on a data set made up of 229 manipulation examples gathered from the same robot, and achieved a detection accuracy rate of 98.60%. In [8] the classification of what error type had occurred was not included, however Fino-Net was extended in [9] where classification was added along with 99 additional training data points, resulting in a failure detection rate of 87% and a failure classification F1 score of 0.80. Similar to Fino-Net, a different multimodal LSTM network was introduced in [10]. Also using a Baxter manipulator robot, [10] attempts to detect and identify anomalies in simple manipulator tasks and achieves a 94% successful classification rate. It was also showed that LSTM architecture achieved better results than Hidden Markov Model (HMM) and Conditional Random Field (CRM) approaches on the same data. Furthermore, [11] showed that including a self-attention stage in the visual processing of RGB-D data improved anomaly classification, achieving an F1 score of 0.94. A common theme throughout [8], [9], and [10] is that although they can detect both when and what error has occurred, they are unable to pre-empt the error. This limitation means when a robot is equipped with a similar detection and identification system, unless the robot has the physical ability to recovery from the error, it is unable to overcome the problem and would require external intervention. For example, in [11] one class of anomaly is when a particular object is not present in the camera's field of view. When this error is encountered, the manipulator can execute a search where the arm moves around the workspace until it detects the object. However, in a robot vacuum, if an example error state is that the robot becomes stuck on a raised doorway boundary, when this error is detected it is too late, the vacuum can not recover itself. For that reason pre-emptive detection is vital for robotic contexts with unrecoverable error states.

Pre-emptive error detection, also labelled failure prediction in literature, looks to identify problems before they lead to a failure state. Various methods have been employed to achieve this, with modern methods utilising AI and ML. This approach can replace complicated, context-specific, model-based solutions. Although effective, due to the black-box nature of deep learning models, the reasons as to how a fault is predicted is unexplainable. This becomes problematic when a black-box predictor is wrong. Recent work has been completed to increase the interpretability of such models on robotic failure prediction, such as [12] who show that explanation generation methods like Shapley Additive Explanations (SHAP) can improve explainability without compromising the accuracy of black-box ML models, and [13] who show a method of synthesising a failure predictor with guaranteed bounds on false-positive and false-negative

errors. Both [12] and [13] focus on manipulator robots, with [13] also applying their work to drones, and [12] stating that the lack of available data to learn from was a limitation in their research.

In mobile robotics, the errors likely to be encountered regularly involve the robot colliding with, or becoming stuck on, objects in the environment [14]. Similar to manipulator robots, the use of AI and ML in avoiding failure states in mobile robots is popular. Kahn et al use a reinforcement learning method to learn a neural network model which predicts actions which will lead to a problematic state [15]; problematic states were identified as those where a robot was required to disengage from its current action by a human safety monitor. In a different study, Kan et al created a navigation system which allow a mobile robot to ignore certain geometric obstacles, such as long grass, in an end-to-end self-supervised learning method [16]. Furthermore, Ji et al created a Proactive Anomaly Detection system (PAAD) for mobile robots [17]. PAAD uses multi-sensor data, inputted into a Convolutional Neural Network (CNN) and encoder based neural network, and fuses the outputs to predict the probability of future failure based on the current state and planned next actions. Although achieving a high detection rate, PAAD is unable to classify and autonomously recover from a predicted error.

2.2 Recovery

After an error has been detected or pre-empted, it is essential for the robot to execute a series of actions to recover or prevent failure from occurring. Many studies focusing on robotic anomaly detection rely on manual methods of recovery like [18], however for a robot to be robust, when physically possible, the robot should recover itself. Wu et al developed a complete framework which detects, classifies, and recovers from errors in a manipulator robot [19]. The framework utilises Bayesian Networks (BN) to predict the probability of a particular error occurring given current sensor readings, then selects a response through a Failure Mode and Effect Analysis (FMEA) table, running a previously defined behaviour tree to control the robot's recovery. This method improves upon basic backward-recovery which attempts to return a robot to its previous state and reattempt a skill as seen in [20], and instead implements forward-recovery which takes corrective steps to solve the problem like [21]; [21] also demonstrated an integrated system of autonomous error detection and recovery, but in a cleaning robot. Both backward- and forward-recovery solutions were implemented in [21] with the specific solution depending on the nature of the error with a key limitation being that the recovery solutions were hard-coded.

2.3 Contribution

Considering the existing research outlined above, there is a noticeable gap in general purpose, complete recovery pipelines, applicable to mobile robots. Detection and identification are well researched areas, and modern deep-learning methods have shown they can be done to a high degree of accuracy. Therefore, this project is focused on creating a recovery framework which can allow

existing detection and identification methods to be incorporated within, and which allows the immediate or pre-emptive recovery of known failures in real time. Furthermore, this work demonstrates the frameworks application to robot vacuums and, to the best of the author's knowledge, novel handcrafted recovery solutions for overcoming doorway boundaries and cables in an environment.

CHAPTER 3

System Requirements

3.1 Simulation

3.1.1 Environment

To mitigate technical problems often encountered when working with physical robots, it was decided that a simulated robot and environment was most appropriate. To allow for accurate simulation, the software chosen needed to have physics enabled, allow interactions between the robot sensors and the map, and allow manual manipulation of the robot. Furthermore, the simulation had to allow custom 3D models to be added, to represent both dynamic obstacles and static structures. Since this work is designed to be highly adaptable for future work, the simulation software was required to be commonly used within the robotics industry, thus accessible to other researchers.

3.1.2 Robot

For the robot, its size needed to be somewhat comparable to that of existing robot vacuums to accurately determine if the recovery implementations were appropriate. Moreover, it was also required to have common onboard sensors, allowing it to perceive its environment, such as a camera and LIDAR. Similar to the simulation software, the robot also needed to be common amongst robot researchers to make this work more versatile.

3.2 Algorithms

3.2.1 Vacuum Task Planner

A number of important algorithms to allow the system to function correctly were required. Firstly, the main vacuum task planner was needed. This algorithm had to generate bespoke instructions for the robot to follow, navigating it across a

given map, covering all open floor space whilst avoiding obstacles. The execution time of the algorithm needed to be minimal, as routes were required to be re-planned in real time. In addition, the planner needed to be as optimal as possible, limiting the traversal of the same section of floor to the minimal amount of times possible whilst still ensuring maximal coverage.

3.2.2 Preventative and Recovery Solutions

For each known problem the robot vacuum may encounter, a robust handcrafted solution was needed to provide the robot instructions on how to recover from the issue. When the wheels were detected to be off the ground, due to the physical limitations of the robot, manual recovery intervention was the only viable response. However, for raised doorway boundaries and cables, a solution was required. Since doorways and cables can exist at any location within the map and, particularly for cables, vary in size and pose, the algorithms created needed to be extremely robust to function in any given scenario. Furthermore, after safely dealing with the respective problem, the robot was required to return back to the position from which the problem was detected.

3.3 System Architecture

3.3.1 Instruction Executor

A method of executing instructions generated from the vacuum task planner and the recovery solutions was required to be created. This executor needed to take the given instructions, and convert them to low-level commands for the robot, moving it in the correct way in the simulated environment.

3.3.2 Error Detection and Identification

For errors to be detected, a program was required to continuously monitor the environment through the robot sensors. The raw data output was needed to be analysed for potential errors and, when detected, alert the rest of the system. Once an error was detected, a method of categorising the error was needed. This was needed to match the error to its respective recovery strategy.

3.3.3 Recovery

After an error was detected and classified, its recovery strategy needed to be executed. Therefore, a method of running the correct algorithm to generate the recovery instructions, send those instructions to the instruction executor, and monitor progress was needed.

3.3.4 Communication Network

For all of the aforementioned elements of the system to function cohesively, a comprehensive and fast method of communication was required. The method in which this was done was required to be extremely fast, allowing the robot to be stopped and recover from errors before they become failure points in real time. It also needed to be versatile, facilitating the easy addition or removal of

individual known error detection and recovery strategies. This modularity was important for the future potential of the pipeline.

CHAPTER 4

Design and Implementation

4.1 Environment

To meet the requirements outlined in Chapter 3, Gazebo [22] was chosen as the simulation environment, and a TurtleBot3 robot [23] was selected to simulate the robot vacuum. Gazebo is a popular, physics-based simulation software, and TurtleBot3 is a common robot with publicly available software packages, thus an appropriate choice for this project. Throughout the project, three 3D environments were used to test the system: a custom made house with minimal obstacles, a small furnished house provided by Amazon [24], and a bookstore also provided by Amazon [25]. Each of the three maps can be seen in Figure 4.1.

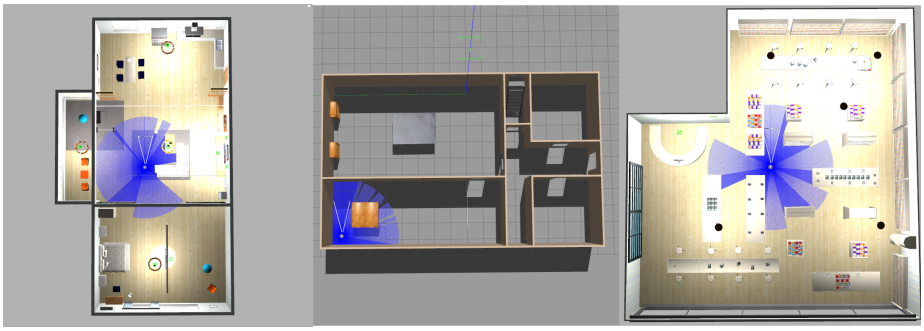


Figure 4.1: Birdseye view of all maps. From left to right: AWS small house, custom map, and AWS bookstore.

To allow the system to interact with the map, an internal representation of the 3D world was required. It was decided that a discrete, grid-style map would be the best internal representation medium, as navigation would be simpler and

faster to compute as there already exists vast amounts of research on agents traversing grid worlds. Converting a 3D world to a 2D grip map involved a number of steps. First, for each map, a 2D map was created by navigating a robot around the environment, mapping obstacles sensed through the robot's LIDAR. This was achieved by using the robot's inbuilt cartography node. These maps were then manually cleaned in a image-editing software, ensuring artifacts and errors were smoothed over, leaving only obstacles denoted by black pixels and free space denoted by white pixels. An algorithm was then created which parsed the cleaned image and over-layed a square grid with each square being of size n x n pixels (tile resolution). For each tile in the grid, if any black pixels were present, the tile was marked as an obstacle. Similarly, if only white pixels existed within a tile, the tile was marked as free space. The algorithm then outputted a 2D array consisting of 1's and 0's where 1's denoted obstacles and 0's denoted free space. The dimension of the resulting grid map depends on the original size of the cleaned image, and the chosen tile resolution. This process is displayed in Figure 4.2.

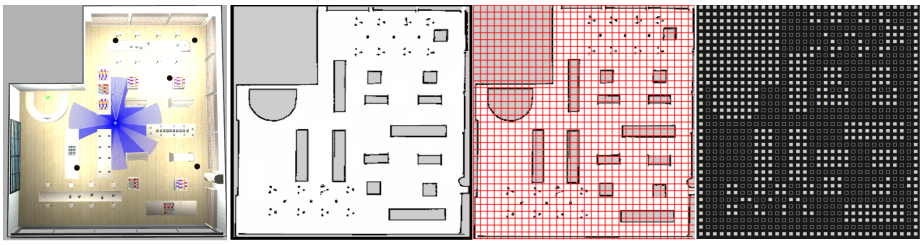


Figure 4.2: Process of converting a 3D world to a 2D grid map used for internal representation. From left to right: 3D map in simulation, map generated from LIDAR cartography, 9x9 pixel tile resolution overlay, 2D array internal representation (For ease of viewing, 1's have been replaced with filled squares and 0's with hollow squares).

4.2 Vacuum Task Planner

Since the chosen context for this project is robotic vacuums, a task planner was required to be developed to implement their behaviour. The main objective of the task planner was to generate instructions to navigate the robot across every free cell within the grid map. To this end, instructions were limited to four primitive choices: up, down, left, and right, where each of the four instructions correspond to movements within the grid map. To select the best combination of instructions to complete the floor coverage task, a Breadth-First Search (BFS) approach was implemented. Given a starting location, the task planner iterates through the four primitive movement options and selects the first unvisited cell. Once found, the movement instruction is added to the instruction path, visited cells are added to a set, and the current position is updated as though the robot took the movement to the unvisited cell. If no free cell exists at any of the four movement options around the current position, the closest unvisited

cell is selected and the A-Star (A*) algorithm is implemented to generate an optimal path to it. This process repeats until all cells have been visited. If the map provided contains cells which are inaccessible, these are added to the visited set before searching, ensuring a plan can be created for any given grid world. Once all cells have been visited, the robot navigates using A* back to its starting location, emulating a static recharge station. This approach ensures full floor coverage and optimal navigation. An example of a map and the generated instructions are shown in Figure 4.3.

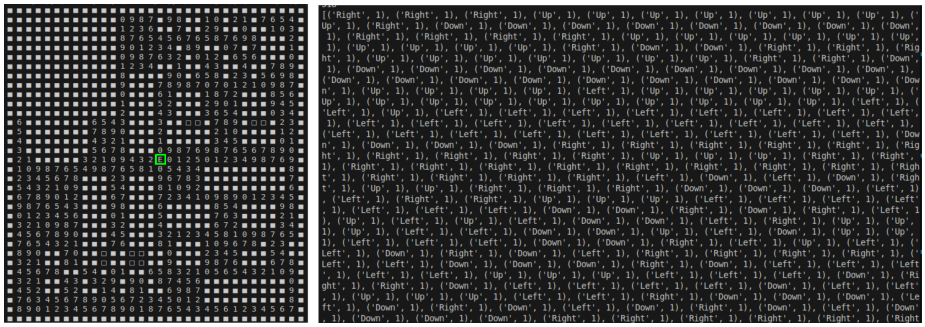


Figure 4.3: The generated instructions and resulting path generated by the task planner for the AWS Bookstore map. The robot starting location is denoted by the highlighted 'E' on the map. The instructions are truncated for easier viewing.

4.3 Robot Controller

Once instructions had been generated, they needed to be converted into low-level commands and given to the robot to execute in the simulation. Since the robot is required to act fully autonomously, the robot needed to execute various behaviours depending on environmental factors and recovery solutions. To facilitate a conditional and dynamic control structure, behaviour trees were selected due to their flexibility to be modified and extended. Specifically, the Pytrees library [26] was used. A library of high-level behaviours was created which interfaced with the robot, converting desired actions into low-level instructions. The created behaviours available to the robot are shown in Table 4.1. After the behaviour library was created, select behaviours were combined together into a number of behaviour trees, controlling the desired execution of high-level tasks. For example, one tree created was the 'vacuum_planner' tree responsible for planning, executing, and monitoring the vacuum task planner instructions generated by the process outlined in Section 4.2. This behaviour tree is illustrated in Figure 4.4.

Behaviour Name	Description
cancel_goal	Send a cancel request to Nav2 to stop the current instruction being executed
check_if_finished	Return true is all instructions have been executed, false otherwise
face_closest_wall	Locate closest wall to the robot via LIDAR data and rotate to face it
follow_wall	Follow the wall located at a given direction at a set distance until wall ends
go_to	Given a coordinate target location, invoke a Nav2 action request to navigate towards the target
move_forward	Move forwards at a set speed for a set amount of time
move_towards_obs	Given a direction of an obstacle, rotate and move towards it until located a set distance from it
plan_route	Run the task planner to generate floor coverage instructions
process_route	Process an instruction and convert it to a coordinate target location

Table 4.1: Behaviours in the behaviour library and description of their function

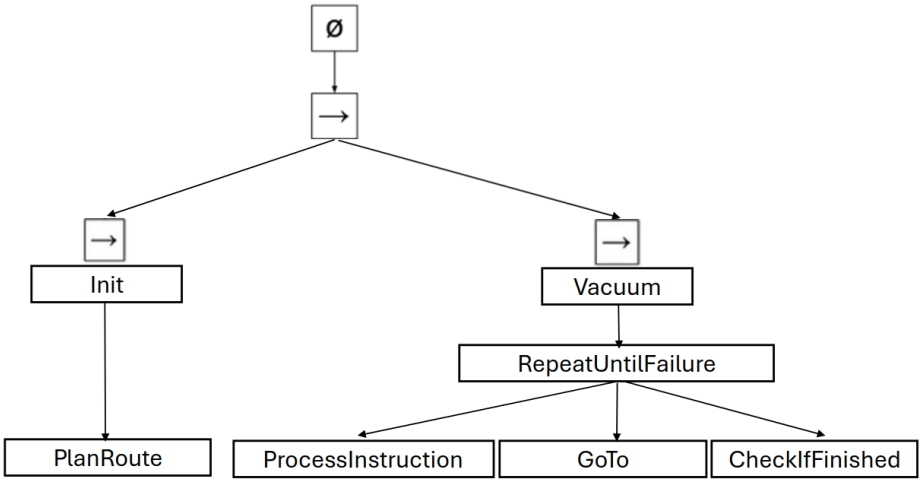


Figure 4.4: An example behaviour tree consisting of behaviours from the behaviour library. This tree implements the robot vacuums default floor coverage task by planning the route instructions, then executing them.

4.4 Preventative and Recovery Solutions

As detailed in Chapter 2, current robot vacuums can encounter many potential problems, with this work focusing on errors involving the wheels coming off the ground, raised doorway boundaries, and cables. Therefore, handcrafted solutions were required to deal with each of the three identified problems.

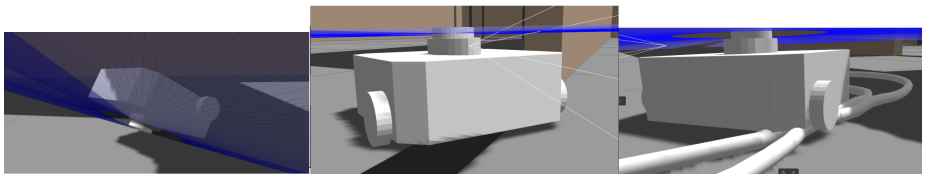


Figure 4.5: The three error states. From left to right: wheels off ground, raised doorway boundary, cables.

4.4.1 Wheels off Ground

By the time a robot vacuum has sensed it is upside down, there is very little it can do to recover itself. Unless the robot is equipped with a physical mechanism capable of flipping itself back over, a third party is needed to help. This holds true for many problems a robot may encounter, such as a mechanical malfunction, or a bad actor purposefully inhibiting progress. Therefore, it is reasonable to suggest that a valid recovery response is to ask for assistance. In this fashion, the recovery solution decided for when the wheels are detected to be off the ground is to broadcast a request-for-assistance message, asking for human intervention. However, rather than the vacuum task failing when such an error occurs, once returned to an upright position, the robot will continue on its mission. This ability greatly increases the robustness of the robot to complete tasks. This error state is shown in Figure 4.5.

4.4.2 Doorway Boundaries

Through experimentation, it was discovered that a major problem with navigating through a doorway was approaching a raised boundary at a non-perpendicular angle. When approaching in such a way, one wheel of the robot collides with the boundary and the momentum of the robot pushes the wheel over. However, there is not enough momentum for the second wheel to make it over, causing the base of the robot to make contact with the boundary and trap the vacuum, stranding it on top. This state is shown in Figure 4.5. To overcome this problem, a preventative measure is taken; when a doorway is detected the recovery pipeline activates, with the goal of redirecting the robot to make a perpendicular approach to the doorway.

Ensuring the robot is perpendicular with a doorway boundary requires a number of steps. Firstly, since the internal map available to the robot only contains obstacles and free space, the location of the doorway must be determined. To do this the map is parsed and gaps between obstacles are highlighted as potential doorways. Once potential doorways are detected, the closest one to the robot is selected as the most likely choice and is thus set as the target to approach. Instructions are then generated to navigate to the cell which lies on the same side as the robot, and two cells perpendicular to the doorway. By navigating to this cell, it ensures the robot can approach perpendicularly. The reason as to why the cell two spaces away from the doorway is chosen is to ensure the robot has space to turn if, after navigating to the cell, it was facing away from the doorway. Once the correct orientation is ensured, the robot re-plans the

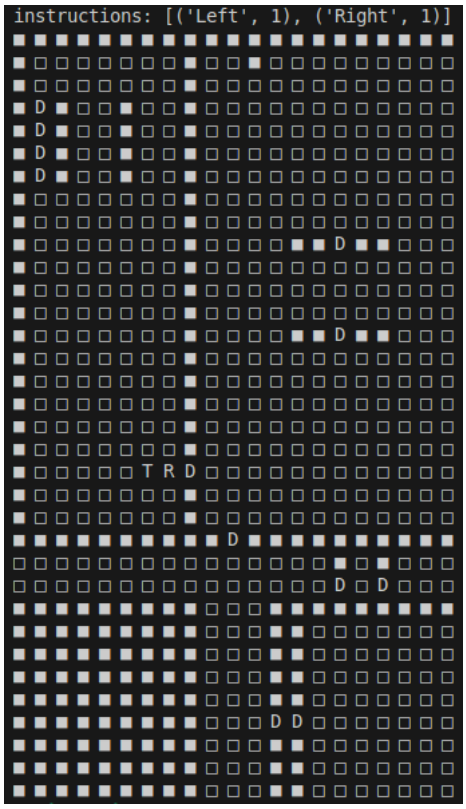


Figure 4.6: The custom map internal representation with potential doorways marked as 'D', the robot location marked as 'R', and the perpendicular target cell located by the recovery solution marked as 'T'. The generated instructions to re-orientate the robot perpendicularly to the doorway are seen at the top of the map.

main vacuum task route and continues with its mission. This recovery process is illustrated in Figure 4.6.

4.4.3 Cables

Loose cables on the floor pose a big risk for robot vacuums. Depending on the cable type, they can get stuck in the mechanism, stuck on the wheels, or get wrapped around the robot causing it to become stuck (shown in Figure 4.5). Therefore, moving cables when they are encountered is beneficial for a robot vacuums progress. Since cables can occur in a variety of lengths and poses, a general method of moving a cable is a non-trivial task.

To reliably move cables when they are encountered a six-step method was created: finding accessible target cells at the ends of the longest side of the cable, identifying the direction the cable should be pushed in, navigating to the

target cell from the cell the cable was detected from, sweep-pushing the cable to an appropriate location in the map, navigating back to the cell from which the cable was detected, and finally re-planning the vacuum route.

Since cables are non-static and deform when moved, it is essential that the deformation induced from the robot does not cause interference with the robot. To minimise this risk, it was decided that cables should only be manipulated from their longest exposed side. This approach reduces the chance of localised stress causing unwanted bending by distributing the force applied by the robot over a greater length of the cable. Through manual control of the robot, it was seen that this approach mitigated the risk of the cable kinking or bunching, limiting the impact of the cable on the robot. The longest accessible side of a cable was determined by scanning the internal map and looking at the cells marked as containing the cable by the error detector. The four directions around each cable cell were searched, keeping track of accessible cells adjacent to the cable. This was repeated until all cable cells were inspected, and the longest side was determined by the longest continuous path of accessible cells. This process is illustrated in image A in Figure 4.7.

Once the longest side had been established, the areas either side of it were inspected. To ensure the cable does not interfere with the robot vacuum again after it is moved, the cable needs to either be pushed to an area which has already been cleaned, or against a wall. The map is analysed and the areas either side of the longest side of the cable are searched until an already cleaned area or a wall is detected. Since the best solution would require the cable to be moved the least amount, the area with the smallest number of free cells between it and the cable is selected. This decision is illustrated in image B in Figure 4.7.

After a target area has been decided, the robot is required to move to its starting location at the end of the longest side of the cable, on the side opposite to the target area. This is done by implementing an A* path finding algorithm from the robot's current cell to the target cell.

Once at the end of the cable, the pushing process starts. To ensure a smooth movement and minimal risk of the cable becoming bunched or kinking, the robot pushes the cable a small amount before retreating, moving along the length of the cable, and then pushing the next section a small amount in a sweeping motion. Once the robot has reached the end of the cable, it moves back along the cable in the opposite direction, again pushing along sections of the cable by small amounts. This process, dubbed a 'sweep-push', is repeated until the cable has been moved to the designated target area. The sweep-push is illustrated in image C in Figure 4.7.

Following the successful sweep-push, the robot navigates back to the cell from which it originally detected the cable. This is again achieved through a path generated by the A* algorithm.

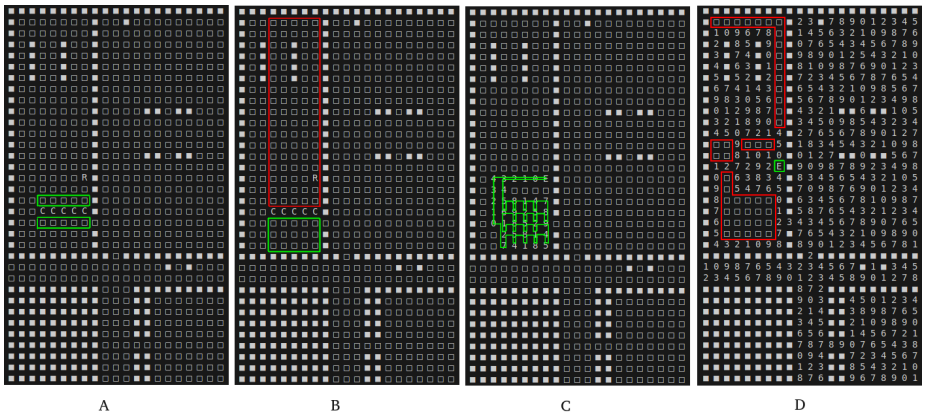


Figure 4.7: Breakdown of cable recovery process. A: Identifying the longest accessible side of the detected cable. B: Locating the smallest area (green highlighted) of cells needed to be traversed to push the cable to an appropriate area adjacent to the longest accessible sides. C: The path taken by the robot during the sweep-push movement. D: Already traversed cells highlighted to show the need for re-planning.

During the process of moving the cable, the robot heavily deviates from its original plan to traverse the map. As such, the area the robot travelled over during the pushing (shown in image D in Figure 4.7) would be cleaned, and therefore would not need to be covered again. To ensure the same cells are not visited again, after the robot has returned to its original cell from which the cable was detected, the vacuum task planner is reran, generating a new path which avoids the already traversed cells. By doing this, the efficiency of the robot vacuum is increased, and the overall overhead of the cable recovery solution is reduced.

4.5 System Architecture

To encompass the individual parts of the the system outlined in the project, the Robotic Operating System (ROS) [27] was used; specifically the Humble distribution from ROS2. ROS is a widely recognised set of software libraries facilitating the control and design of robotic applications, and was thus appropriate for use in this project. Consistent with ROS good practise, the different elements of this system were split into three distinct nodes: a behaviour tree executor node, a detection node, and a recovery node. Additionally, an FMEA table and a fake-detection publisher were created to allow the system to function. The system architecture is illustrated in Figure 4.8.

4.5.1 Behaviour Tree Executor Node

To run the behaviour trees created and outlined in 4.3, a node is required to interface with the robot vacuum in the simulation. This node is responsible for

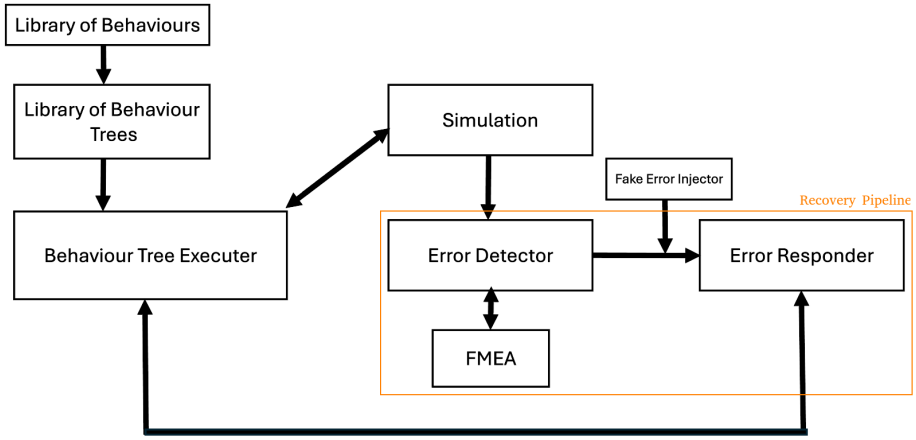


Figure 4.8: High-level overview of the system architecture. Arrows denote the flow of information.

all active interactions by this system with the robot and simulation. Therefore, this node had access to the library of behaviour trees, enabling it to select, execute, and monitor whichever tree was required to be ran. Furthermore, the behaviour tree executor node can be set into an 'error state' mode where it pauses all current instructions to the robot and awaits commands on how to proceed. This mode is utilised by the responder node when a different behaviour tree needs to be ran, and where the robot must deviate from its path to overcome an error.

4.5.2 Detection Node

The detection node's purpose was to passively monitor the robot's sensors by subscribing to each sensor's corresponding ROS topic and determine whether an error has, or is likely to, occur. The node was modular by design, with standalone detectors looking for particular errors, and a central controller iterating through each detector. This way error detectors could be added or edited with minimal changes to the node, increasing the versatility of the system. Furthermore, such a design simplifies the classification of errors as the error detector which senses a problem was only searching for a singular type, thus when an error is detected the type of error is automatically known. After an error is detected, the node is responsible for searching the FMEA table to determine the correct response. If a response exists in the table, the node triggers the responder node and passes it the name of the response required. The FMEA table can be seen in Table 4.2.

Throughout this work, the 'wheels off ground' detection method was implemented by monitoring the robot's Inertial Measurement Unit (IMU), and triggering when the rotation of the robot in the x or y plane exceeded 0.4 radians. Developing detection methods for encountering a doorway boundary or cable was deemed out of scope for this project, however as covered in Chapter 2, the detection and identification of such errors has been shown to be possible and

Abnormal Skill	**Affected Subsystem**	**Cause**	**Solution**
navigate	environment issue	doorway boundary	BT: perpendicular alignment to doorway
navigate	environment issue	cable	BT: move cable
navigate	drive issue	wheels off ground	MANUAL: place robot upright

Table 4.2: Abnormal Skills and Their Solutions. BT indicates the name of the behaviour tree needed to be ran to respond. MANUAL indicates the message needed to be sent to the user to intervene.

done to a high degree of accuracy. Therefore, to still be able to react to doorway boundary and cable errors, a fake-detection publisher was created. This publisher allowed detections to be injected into the system, bypassing the detector node, however still triggering the responder node as though it had been detected organically. Throughout this work, all doorway boundary and cable errors were invoked this way.

4.5.3 Responder Node

Once triggered and given the required response's name by the detection node, the responder node's job was to generate new instructions for the robot to overcome the detected problem, and ensure they were executed. Similar to the detection node, the responder node is modular by design. Each individual response is standalone and managed by a central controller. Once given the response name by the detector node, the central controller sends a signal to the behaviour tree executor node to enter an error state and to pause its current instructions. Simultaneously, the correct standalone responder generates new instructions for the robot to overcome the problem and passes them back to the central responder node controller. The central responder node controller then passes the new instructions to the behaviour tree executor node along with the name of the behaviour tree that it should run. The responder node then waits for the behaviour tree executor node to finish running the new behaviour tree and instructions. Once successfully completed, the responder node sends a signal that the behaviour tree executor node should end its error state and continue with its vacuum path. For the cable and doorway boundary recovery strategies, the floor coverage task instructions were re-planned, and the new instructions were passed to the behaviour tree executor node. If a recovery solution fails, the node alerts the user that manual intervention is required.

CHAPTER 5

Evaluation Method

To evaluate the effectiveness of the created recovery framework, a series of experiments were designed. Since the design and implementation of the recovery algorithms in this work are unique to the robot vacuum context, environment, and robot selected, there is no direct comparison to be made to existing literature. However, to ensure the comprehensive evaluation of this system, the methods of experimentation used are standard across similar robotics research. For example, [20] break down the success of their recovery implementation by anomaly class as do [28] who also analyse the additional execution time of recovery strategies.

To accurately determine the merit of each recovery implementation, a baseline of comparison was required. To this end, the robot vacuum was tasked to run its floor coverage operation on three distinct maps with no doorway boundaries, loose cables, or instances of the robot being flipped over, and without the recovery pipeline being activated. The three maps chosen were a custom made floor plan of a house with very few default obstacles, an AWS small house, and an AWS bookstore map (shown in Figure 4.1). Between the three maps there is a large degree of variation in map size, obstacle geometry, and number of obstacles allowing the created robot vacuum task planner and recovery framework to be tested in multiple realistic environments, showcasing its robustness. By initially removing the three error-inducing obstacles and states, it was possible to measure the time taken by the robot to complete its vacuum task on each map and determine whether the task was successfully completed without additional obstacles. With the recovery pipeline still disabled, once the standard run-times and successes were established with no obstacles, each of the three obstacles were added to the map one at a time, where for each obstacle if the robot successfully completed its task the runtime was recorded, and if not the reason of failure was recorded. The location of the added obstacles was chosen intuitively and logically for the obstacle type. For example, doorway boundaries were added in doorways and tight gaps which could resemble a doorway, and

cables were placed with them extending into open floor-space so the robot was required to navigate over the same area. For instances where the robot was flipped, this was done manually shortly after the start of the robot's task. The locations of the additional obstacles added are illustrated in Figure 5.1. After all obstacles were tested individually, a combination of all three were applied to the maps and the robot was again tasked with executing its floor coverage mission without the recovery pipeline activated.

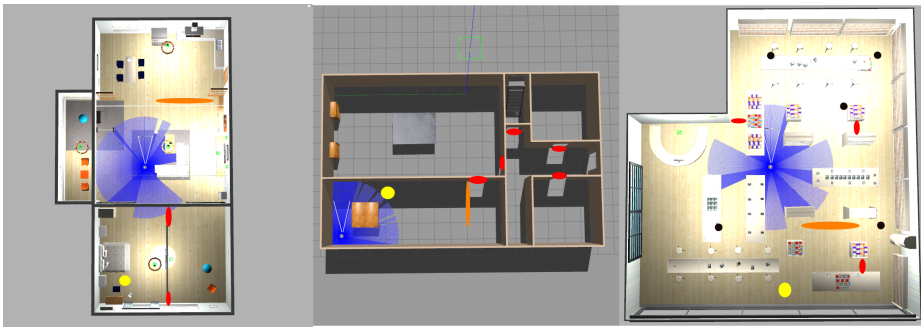


Figure 5.1: Birdseye view of all maps with the locations of the where obstacles were added highlighted. Red markers indicate raised doorway boundaries, orange markers indicate cables, and yellow markers indicate the location where the robot was manually flipped.

With baseline times and qualitative descriptions of failure states without the recovery pipeline being activated established, the experiments were repeated with the recovery pipeline enabled. For each map, each obstacle was individually, manually, inserted within, and the robot was instructed to execute the floor coverage task. Again, the error state of the robot's wheels being off the ground was manually implemented in the simulation, and after each individual obstacle was tested in isolation, a combination of all three obstacles were tested. The location of the obstacles for each map was kept the same as for the experiments when the recovery pipeline was disabled. Differing from the baseline measurements, the recovery pipeline was activated and its instructions were executed. For instance, when a manual intervention was required, the robot was manually assisted and the system was informed of when this had been completed. For the recovery from doorway boundaries and cables, the detection was manually injected into the system, however the recovery solutions were autonomously performed. This process of testing with the recovery pipeline enabled was repeated twice, once with the re-planning of the floor coverage path after a recovery solution, and once without re-planning. This was done to investigate how much of a benefit re-planning was to the overall execution time after an autonomous recovery had taken place.

Overall, the key metrics gathered to evaluate the created system were execution success, overall execution time, and the overhead the recovery solutions had on execution time. To consider the created system a success, two criteria needed to be met during the floor coverage task: a significant increase in exe-

cution success and minimal execution time overhead when the recovery pipeline was enabled compared to when it was disabled.

All experiments were completed in real time on a desktop computer with the specifications outlined in Table 5.1.

Computer Specifications	
CPU	AMD Ryzen 7 5800x
GPU	GeForce RTX 3080
RAM	16 GiB

Table 5.1: The hardware specifications of the computer on which all experiments were ran.

CHAPTER 6

Results and Discussion

6.1 Baseline Results

Table 6.1 shows the baseline results for the robot executing a floor coverage task in the three maps with the recovery pipeline disabled.

The successful completion of the floor coverage task in all three maps without additional obstacles demonstrates the vacuum task planner's ability to create comprehensive instructions, and the system's capacity to execute these instructions successfully in the simulation for a prolonged period. The floor coverage percentage was calculated as the total percentage of cells visited by the robot which were accessible to it. The 100% floor coverage seen in each map shows the completeness of the created task planner as a successful path was generated and executed which visited all cells. The lack of success in each map with the recovery pipeline disabled where obstacles have been added clearly shows the negative impact the obstacles have on the robot's ability to execute its task. Each of the failures were solely caused by the addition of the obstacles, and in the mixture of obstacles, the first encountered obstacle was the failure point. Images of each of these failure states are shown in Figure 6.1.

6.2 Recovery Pipeline Results - No Re-planning

Table 6.2 displays the results for the robot vacuum executing a floor coverage task in the three maps with the recovery pipeline enabled.

Comparing the execution times in Tables 6.1 and 6.2 for each map with no additional obstacles, it is evident that there is some discrepancy in values. Since the floor coverage task planner is deterministic, in an ideal world these values

	Custom Map	**AWS House**	**AWS Bookstore**
No Additional Obstacles			
Execution Success?	YES	YES	YES
Execution time (If success)	33m 23s	44m 48s	49m 44s
Reason for Failure (If not success)	-	-	-
Floor Coverage %	100	100	100
Wheels off Ground			
Execution Success?	NO	NO	NO
Execution time (If success)	-	-	-
Reason for Failure (If not success)		Wheels off ground	
Floor Coverage %	-	-	-
Doorway Boundary			
Execution Success?	NO	NO	NO
Execution time (If success)	-	-	-
Reason for Failure (If not success)		Stuck on doorway	
Floor Coverage %	-	-	-
Cable			
Execution Success?	NO	NO	NO
Execution time (If success)	-	-	-
Reason for Failure (If not success)		Robot stuck on cable	
Floor Coverage %	-	-	-
Mixture			
Execution Success?	NO	NO	NO
Execution time (If success)	-	-	-
Reason for Failure (If not success)		Wheels off ground	
Floor Coverage %	-	-	-

Table 6.1: Recovery Pipeline Disabled - Results for robot vacuum floor coverage task in each map.

would be the same, however small errors and inefficiencies in the execution of instructions accumulate over time resulting in either slightly faster or slower execution times at random. By taking the execution times of each map, both with the recovery pipeline enabled and disabled, and looking at the number of cells traversed, it was calculated that with no additional obstacles present, the robot vacuum takes on average 4.75s to navigate per cell. It is notable that the time taken to execute a path with a high number of rotations takes longer than the average. Thus, the average navigation time per cell for the custom map was 3.94s, and for the bookstore 5.76s, due to the significantly increased number of turns required to navigate the map. Nevertheless, the success of the recovery pipeline is evident with all obstacles being overcome, resulting in a 100% success rate for task completion in each map. This demonstrates the robustness of the recovery algorithms developed for the three obstacle types, and the reliability of the system to deal with a multiple obstacles existing in the map simultaneously.

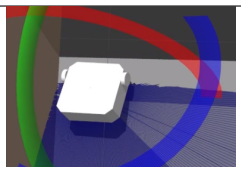
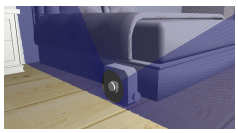
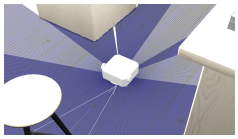
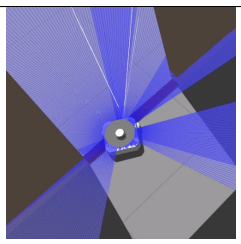
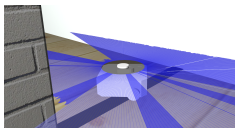
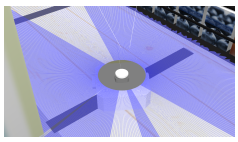
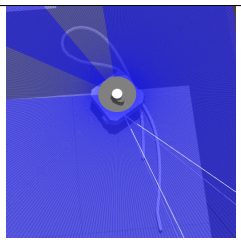
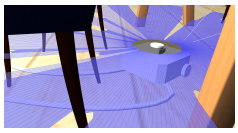
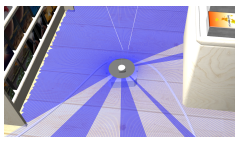
	Custom Map	AWS House	AWS Bookstore
Wheels off Ground			
Doorway Boundary			
Cable			

Figure 6.1: Screenshots of the robot vacuum in its failure state for each map from experiments with recovery pipeline disabled.

6.2.1 Wheels off Ground Overhead

To understand the additional overhead time of the recovery pipeline, it is important to analyse the increase of execution time caused by each obstacle. Considering the recovery of a robot with its wheels off the ground, the overhead is entirely reliant on the overseeing human. Since the recovery strategy is manual intervention, the time taken for a human to right the robot makes up the vast majority of the overhead time. In fact, from detection to the display of the message to manually intervene, the system only takes approximately 150ms. Similarly, the time taken to continue the floor coverage task after the user has manually intervened takes approximately 150ms.

6.2.2 Doorway Boundary Overhead

For the pre-emptive recovery from doorway boundaries, the average increase in execution time was 14.33s per encountered doorway. Since the pre-emptive algorithm only produces an additional two instructions per doorway, the average time of navigation per cell during recovery is 7.17s, 2.42s slower than the average. This increase in execution time can be explained by the recovery solutions design which incorporates moving to a cell, then navigating to the cell behind, thus incorporating a significant amount of rotation. However, since the solution needs to ensure perpendicularity to the doorway, this is unavoidable, and thus an acceptable increase in execution time.

	Custom Map	**AWS House**	**AWS Bookstore**
No Additional Obstacles			
Execution Success?	YES	YES	YES
Execution time (If success)	33m 19s	44m 55s	49m 42s
Reason for Failure (If not success)	-	-	-
Floor Coverage %	100	100	100
Wheels off Ground			
Execution Success?	YES	YES	YES
Execution time (If success)	34m 4s	45m 36s	50m 20s
Reason for Failure (If not success)	-	-	-
Floor Coverage %	100	100	100
Doorway Boundary			
Execution Success?	YES	YES	YES
Execution time (If success)	33m 33s	45m 10s	49m 56s
Reason for Failure (If not success)	-	-	-
Floor Coverage %	100	100	100
Cable			
Execution Success?	YES	YES	YES
Execution time (If success)	37m 58s	47m 35s	54m 51s
Reason for Failure (If not success)	-	-	-
Floor Coverage %	100	100	100
Mixture			
Execution Success?	YES	YES	YES
Execution time (If success)	39m 4s	38m 40s	55m 57s
Reason for Failure (If not success)	-	-	-
Floor Coverage %	100	100	100

Table 6.2: Recovery Pipeline Enabled without Re-planning - Results for robot vacuum floor coverage task in each map.

6.2.3 Cable Overhead

When cables are encountered the increase in execution time depends upon the size and location of the cable. Since the cable recovery algorithm attempts to push the cable to either an already cleaned area or against a wall, the number of cells needed to be traversed can be large if the initial detected location of the cable is far away from such areas. Moreover, due to the sweep-push algorithm manipulating the cable from its longest accessible side, the larger the cable the more cells are traversed in the recovery. For each of the maps, the length of the cable added to the environment was kept constant, spanning 6 cells. However, the distance the cable was required to move varied for the three maps with the custom map moving the cable an area of 30 cells, the AWS house 18 cells, and the AWS bookstore 34 cells. To compare the additional overhead time during the sweep-push, the average traversal time per cell was calculated to be 9.3s for the custom map, 8.9s for the AWS house, and 9.1s for the AWS bookstore, resulting in a total average of 9.1s across all maps. This value is 4.35s higher than the average traversal time during the floor coverage task. The reason for such a large increase is two-fold: firstly, similar to the doorway boundary recovery, the sweep-push process requires a significant amount of rotations, thus increas-

ing the execution time; secondly, during the sweep-push, the robot sometimes rotates while in contact with the cable, causing the robot to tilt in such a way that the LIDAR detects the ground. When this occurs, the navigation system incorrectly senses an object in its path and it takes a small amount of time for this false detection to be overridden by the correct LIDAR readings once the robot has stabilised. During this time the robot either attempts to navigate around the non-existent object or cannot make progress, thus increasing the execution time. This effect was minimised by tuning the Adaptive Monte Carlo Localisation (AMCL) parameters, specifically by increasing the update frequency rate, however the problem could not be completely avoided. To stop this issue occurring the sweep-push algorithm would need to be adjusted so that the robot avoids rotating while being in contact with the cable.

6.3 Recovery Pipeline Results - With Re-planning

By implementing re-planning after a recovery solution had been executed, it was hoped that the system overhead could be reduced. When re-planning, the cells already traversed both before and during the recovery solution was executed, were excluded from the list of cells needing visiting. This way cells were not visited twice unless they were needed to be traversed to get to an unvisited cell elsewhere. Only recovery solutions which deviated the robot from its original floor coverage task were required to re-plan. The results of experiments with re-planning enabled are shown in Table 6.3.

	Custom Map	**AWS House**	**AWS Bookstore**
No Additional Obstacles			
Execution Success?	YES	YES	YES
Execution time (If success)	33m 25s	45m 01s	49m 37s
Reason for Failure (If not success)	-	-	-
Floor Coverage %	100	100	100
Doorway Boundary			
Execution Success?	YES	YES	YES
Execution time (If success)	33m 29s	45m 06s	49m 45s
Reason for Failure (If not success)	-	-	-
Floor Coverage %	100	100	100
Cable			
Execution Success?	YES	YES	YES
Execution time (If success)	35m 41s	46m 15s	52m 26s
Reason for Failure (If not success)	-	-	-
Floor Coverage %	100	100	100

Table 6.3: Recovery Pipeline Enabled with Re-Planning - Results for robot vacuum floor coverage task in each map for recovery solutions which deviated from original floor coverage instructions.

It is apparent that the total execution time of the floor coverage task was reduced when re-planning was enabled compared to disabled. For doorway boundary recoveries, the average overhead of recovery per cell across all maps

was 2.83s, over a 60% decrease from the 7.18s observed with re-planning disabled. Furthermore, for cable recoveries, the average overhead per cell that the cable needed to be moved, across all maps, was 4.54s, a 51% decrease of the 9.30s when no re-planning was done. The overhead times for each obstacle with and without re-planning can be seen in Table 6.4.

	Overhead Time Per Cell In Seconds			
	Custom Map	**AWS House**	**AWS Bookstore**	**Average**
Without Re-Planning				
Doorway Boundary	7.00	7.50	7.00	7.17
Cable	9.30	8.90	9.10	9.10
With Re-Planning				
Doorway Boundary	2.00	2.50	4.00	2.83
Cable	4.53	4.11	4.97	4.54

Table 6.4: Overhead Time Per Cell for each map and relevant recovery solution with re-planning enabled, and disabled.

CHAPTER 7

Conclusion

7.1 Limitations and Future Work

Although successful, a number of limitations to this work exist. Due to a limited time frame and a larger focus on recovery strategies, the detection of doorway boundaries and cables within the environment were manually injected into the system when testing. For the recovery pipeline to be fully autonomous, a non-manual method of detection for both objects would need to be added. However, as discussed in Chapter 2, such methods of error detection and identification have been studied and shown to be reliable. Thanks to the modular design of the detection section of the developed system architecture, additional detection methods can be 'plugged' in to the system with minimal work required. Likewise, recovery solutions can be added or changed in a modular fashion, making the system easy to extend and apply to different contexts. The overall system architecture is context agnostic, allowing it to be applied to any autonomous system, with context specific implementations of task planners, detection methods and recovery solutions required.

The implementations of the recovery algorithms for doorway boundaries and cables rely on a few key assumptions which may not hold true in certain real-world scenarios. For example, doorways are assumed to only exist parallel to the vertical or horizontal axis of the internal grid map; doorways existing at an angle are not currently supported. For cables, it is assumed that cables are either not attached to an object at either side, like being plugged into a wall, or have enough slack in them to be moved. For scenarios where this is not true, there is a risk that the recovery will fail. However, in such a scenario, a robot vacuum is unlikely to have the physical capabilities to unplug or move a heavy electrical appliance. Autonomously detecting when it is appropriate to attempt a recovery solution based on permutations of the same anomaly would be an area which requires further research.

The created system is only designed to respond to known errors. However, for long time horizon tasks executed in an unstructured environment, an autonomous robot is likely to encounter unknown problems. The ability for a robot to recognise and respond to an unknown error state reliably and safely would be an excellent avenue for further research. Furthermore, if more than one error occurs simultaneously, the current system would only attempt to recover from the error detected first. For example, if a loose cable was positioned over a raised doorway boundary and the cable was detected before the boundary, the recovery solution for the cable would be executed without consideration for the doorway boundary. Further research into how to deal with such scenarios would also be an interesting direction for further work.

7.2 Summary

In this work a new architecture of autonomous error recovery was created and applied to robotic vacuums. Novel recovery algorithms were created to either take pre-emptive action to prevent an error, or recover from a sensed error, were designed, implemented, and tested. Each error guarded against was highlighted by an industry leader as being a real problem facing robot vacuums; these errors were when a robot encounters a raised doorway boundary, comes across a cable, or has its wheels off of the ground. Through the creation of a simulated world, floor coverage task planner, custom robot behaviours, custom behaviour trees, and a comprehensive robot control network, a complete simulation of a robot vacuum was created. By executing the instructions of the floor coverage task planner, the effect of doorway boundaries, cables, and flipping the robot were investigated. It was found that when none of these obstacles were present, the robot was able to fully complete its task, achieving a 100% floor coverage rate, proving the completeness of the instructions generated by the task planner, and the reliability of the created control structures to execute these instructions over a long time horizon task. However, when any of the three error types were added to the simulation, the vacuum failed its task. Consisting of an error detector which monitored the robot's sensor data and an error responder which selected and monitored the execution of recovery solutions, the recovery pipeline was enabled and investigated. It was found that the handcrafted recovery solutions were effective at overcoming the added sources of error, allowing the robot vacuum to successfully complete its task in three different maps in the presence of each and all obstacles. Furthermore, the recovery solutions for doorway boundaries and cables were completely autonomous and ran in real-time. The effect of adding a re-planning step after a recovery solution had finished executing was also shown to be effective, reducing the overhead of recovery per cell by 60% and 51% for doorway boundaries and cables respectively. This work has made a significant impact, leading to its inclusion in the second annual review meeting for the European Union Horizon CONVINCE project [2].

Bibliography

[1] K. Carames, K. Mui, A. Azad, and W. C. Giang*, "Studying robot vacuums using online retailer reviews to understand human-automation interaction," *Proceedings of the Human Factors and Ergonomics Society Annual Meeting*, vol. 65, p. 1029–1033, Sep 2021.

[2] CONVINCE, "Context-aware verifiable and adaptive dynamic deliberation." https://convince-project.eu/.

[3] G. Devol, "Programmed article transfer," Jun 1961. https://patents.google.com/patent/US2988237A/en.

[4] A. Marsh, "In 1961, the first robot arm punched in - ieee spectrum," Aug 2022.

[5] M. A. Isa, M. A. Khanesar, R. K. Leach, D. Branson, and S. Piano, "High-accuracy robotic metrology for precise industrial manipulation tasks," *Repository@Nottingham (University of Nottingham)*, Aug 2023.

[6] M. Basseville, "Detecting changes in signals and systems—a survey," *Automatica*, vol. 24, p. 309–326, May 1988.

[7] L. Ruff, J. R. Kauffmann, R. A. Vandermeulen, G. Montavon, W. Samek, M. Kloft, T. G. Dietterich, and K.-R. Müller, "A unifying review of deep and shallow anomaly detection," *arxiv.org*, Sep 2020.

[8] A. Inceoglu, E. E. Aksoy, A. C. Ak, and S. Sariel, "Fino-net: A deep multimodal sensor fusion framework for manipulation failure detection," *2021 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, Sep 2021.

[9] A. Inceoglu, E. E. Aksoy, and S. Sariel, "Multimodal detection and classification of robot manipulation failures," *IEEE Robotics and Automation Letters*, vol. 9, p. 1396–1403, Feb 2024.

[10] D. Altan and S. Sariel, "What went wrong?: Identification of everyday object manipulation anomalies," Jan 2020.

[11] D. Altan and S. Sariel, "Clue-ai: A convolutional three-stream anomaly identification framework for robot manipulation," Mar 2022.

[12] A. Alvanpour, S. K. Das, C. K. Robinson, O. Nasraoui, and D. Popa, "Robot failure mode prediction with explainable machine learning," Aug 2020.

[13] A. Farid, D. Snyder, A. Z. Ren, and A. Majumdar, "Failure prediction with statistical guarantees for vision-based robot control," 2022.

[14] J. Carlson and R. Murphy, "How ugvs physically fail in the field," *IEEE Transactions on Robotics*, vol. 21, p. 423–437, Jun 2005.

[15] G. Kahn, P. Abbeel, and S. Levine, "Land: Learning to navigate from disengagements," 2020.

[16] G. Kahn, P. Abbeel, and S. Levine, "Badgr: An autonomous self-supervised learning-based navigation system," *arXiv (Cornell University)*, Jan 2020.

[17] T. Ji, A. N. Sivakumar, G. Chowdhary, and K. Driggs-Campbell, "Proactive anomaly detection for robot navigation with multi-sensor fusion," *arXiv (Cornell University)*, Jan 2022.

[18] A. B. Beck, A. D. Schwartz, A. R. Fugl, M. Nauman, and B. Kahl, "Skill-based exception handling and error recovery for collaborative industrial robots," *IROS 2015*, 2015.

[19] R. Wu, S. Kortik, and C. H. Santos, "Automated behavior tree error recovery framework for robotic systems," *IEEE International Conference*, May 2021.

[20] H. Wu, S. Luo, L. Chen, S. Duan, S. Chumkamon, D. Liu, Y. Guan, and J. Rojas, "Endowing robots with longer-term autonomy by recovering from external disturbances in manipulation through grounded anomaly classification and recovery policies," *arXiv (Cornell University)*, Jan 2018.

[21] K. Yamazaki, R. Ueda, S. Nozawa, Y. Mori, T. Maki, N. Hatao, K. Okada, and M. Inaba, "Tidying and cleaning rooms using a daily assistive robot - an integrated system for doing chores in the real world -," *Paladyn, Journal of Behavioral Robotics*, vol. 1, Jan 2010.

[22] OpenRobotics, "Gazebo." https://gazebosim.org/home.

[23] TurtleBot3, "Robotis e-manual." https://emanual.robotis.com/docs/en/platform/turtlebot3/overview/.

[24] AWS, "Aws robomaker small house world ros package," Apr 2023. https://github.com/aws-robotics/aws-robomaker-small-house-world.

[25] AWS, "Github - aws-robotics/aws-robomaker-bookstore-world: A bookstore world with shelving and tables for aws robomaker and gazebo simulations," 2019. https://github.com/aws-robotics/aws-robomaker-bookstore-world.

[26] JAX, "Py trees — pytrees 2.1.6 documentation." https://py-trees.readthedocs.io/en/devel/.

[27] OpenRobotics, "Ros 2 documentation — ros 2 documentation: Humble documentation." https://docs.ros.org/en/humble/index.html.

[28] A. D. Luca, L. Muratore, and N. G. Tsagarakis, "Autonomous navigation with online replanning and recovery behaviors for wheeled-legged robots using behavior trees," *IEEE Robotics and Automation Letters*, vol. 8, p. 6803–6810, Oct 2023.

7.3 Appendix

The code found for this project can be found here:

https://git.cs.bham.ac.uk/projects-2023-24/oxg685

The GitLab repository linked above contains all files to run the simulation and the recovery pipeline. To set up and run the code, please follow the instructions found in the repositories README file. Specific software is needed to correctly run the environment, however the distributions and installation instructions are included in the README.