## Abstract (200 – 300 words)

* Follow guide on VLE (<https://docs.google.com/document/d/1jvCs3of8zjf-G_jIX9kA-ZF32Rhi3oZX0a_fdN6bHeI/edit?pli=1>)

## Acknowledgements

I would like to give a big thank you to my project supervisor, Dr Mark Post, for providing guidance and feedback throughout the project; Along with giving me the freedom to explore and develop what interested me most. My exceptional peers at university that always pushed me to a higher standard throughout my time at university; And my family for making my time at university to further develop myself possible and providing endless support.

## Ethics

After consideration of the University’s code of practice and principles for good ethical governance no ethical issues were identified in this project.

## Table of Contents

# Introduction

This section builds, and expands, on material previously included in the project Initial Report (see Appendix %)

## Background and Context

In recent years, there has been rapid development in space systems driven by a global push for increased commercial accessibility. Current commercial systems are designed with a focus on minimizing mass and launch costs, resulting in highly customized configurations that often lack robust maintenance and repair capabilities. Consequently, the population of aging satellites is expanding, and upon reaching the end of their operational life, they are either deliberately deorbited using atmospheric deconstruction methods or left in orbit, contributing to the accumulation of space debris.

At present, there is little available technology to overcome these conditions. The HORIZON 2020 EU-funded MOdular Spacecraft Assembly and Reconfiguration (MOSAR) project [%1] was therefore initiated to develop innovative technologies aimed at standardising satellites and components. Modularising and standardising space systems will benefit the European space industry by enabling mass production of standardised components, reducing assembly costs, shortening the time between customer orders and deployment in space, and facilitating direct in-orbit repair and component upgrades, thereby extending the lifetime of space systems.

MOSAR’s primary objective is to create modular and reconfigurable satellites that can be assembled and adjusted in orbit. The project has developed a demonstrator for reconfiguring cubic modules using a mobile robotic manipulator to simulate module movement. Currently, the manipulator receives fixed instructions for module mobility from software simulations on Earth [%]. This research aims to enhance the system by developing an algorithm to automate module reconfiguration, enabling self-repair and self-assembly. Once implemented, this technology could automate space system assembly and platform construction in space, overcoming current limitations in the space industry.

## Project Objectives and Specification

This project intends to enable autonomous assembly and reconfiguration of modular space systems by implementing a reconfiguration planning program made up of simple algorithms. This program, given the initial state and final state of a modular system, will generate a list of commands to be sent to a mobile manipulator to autonomously rearrange modules on a spacecraft or space platform. The planning program must account for physical constraints imposed by the mobile manipulator present on the modular system; therefore, this project will strive to explore methods of incorporating physical constraints into the planning process.

To achieve the research goal, the primary objective is to implement a functional planning program capable of autonomous module reconfiguration, which will be demonstrated through software simulation. If time allows, an additional goal is to physically demonstrate the planning program by integrating it with the available manipulator arm in the lab to reconfigure real modules.

To achieve the research objectives, the following sub-objectives have been identified:

1. Develop a reconfiguration planning program that generates module movement instructions for a mobile manipulator based on initial and final state configurations.
2. Enhance the reconfiguration planning program to integrate physical constraints imposed by the mobile manipulator.
3. Implement a display function to create reconfiguration slideshows or videos, allowing users to visualise the modular systems reconfiguration process.
4. Conduct systematic testing of the system with various inputs to analyse system performance during solution generation.
5. Demonstrate the system by integrating it with the laboratories robot arm to physically reconfigure real modules.

By pursuing these steps, the project aims to showcase the feasibility and effectiveness of the planning program for autonomous assembly and reconfiguration of modular space systems, potentially paving the way for practical applications in the space industry.

## Report Structure

This document serves as a comprehensive report of the research and development carried out during the Autonomous Re-Configuration of Modular Spacecraft with Manipulator Arm project. The report encompasses the following key components:

1. **Literature Review and Research:** A thorough examination of the current state-of-the-art in modular reconfiguration, including a review of relevant literature and existing technologies in the field.
2. **Detailed Design Development:** Creation of a detailed design plan outlining the implementation strategy for the reconfiguration planning program, specifying key components and methodologies
3. **Implementation Description and Specification:** Description and Specifications of the final implemented design, detailing the development and optimisation.
4. **Design Analysis and Results:** Analysis of the implemented design, records of performance metrics, solution generation times, and failure rates obtained through testing and simulation.
5. **Discussion of Results:** Interpretation and discussion of the analysis results, evaluating their significance and implications within the broader context of the area of study.
6. **Project Management Approach:** Examination of the project management methodology employed throughout the project lifecycle, documenting the evolution of the project plan and strategic adjustments made to achieve project objectives.
7. **Recommendations for Further Work:** Identification of potential areas for future research and development to build upon the findings and achievements detailed in this report, suggesting methods for expanding and refining the implemented system.

# Literature Review

This section builds, and expands, on material previously included in the project Initial Report (see Appendix %)

## Overview of Modular Spacecraft

Modular spacecraft represent a design concept where the overall space system consists of interchangeable modules, each fulfilling specific functions such as propulsion, communication, power generation, or sensing. These standardised modules enable easy assembly to form a unified system, allowing for module movement or replacement to optimize craft efficiency and extend system lifespan. Adopting a modular design approach offers several advantages over traditional methods, including enhanced flexibility, adaptability, and simplified maintenance.

Modules feature standardised interfaces that govern physical and electronic interactions, facilitating seamless integration of modules with different purposes or manufacturers into the overall system architecture. While module sizes and shapes may vary across designs, standardisation principles ensure compatibility for integration. The scalability of modular space system architectures depends on the types and quantities of modules used, providing versatility and cost-effectiveness as the system can be tailored to suit specific mission requirements without necessitating a complete redesign.

## State-of-the-art in Spacecraft Modularity and Automated Reconfiguration

This section explores existing cases of spacecraft modularity and reconfiguration technologies currently or previously in operation. Due to the challenges related to developing automated reconfiguration systems for space operations, there are limited existing cases of automated reconfiguration aside from the International Space Station (ISS). However, modular design principles have been integral in spacecraft development since the 1980s, notably with the introduction of the Multi-mission Modular Spacecraft (MMS).

### Multi-mission Modular Spacecraft (MMS)

The Multi-mission Modular Spacecraft (MMS) was designed and deployed by NASA in the 1980s and 1990s [%] with the intention of decreasing space mission costs. Intended to be recoverable/serviceable by the Space Shuttle Orbiter [%], It is one of the first cases of modular designs seen in the space industry and has paved the way for future innovations.

The MMS consisted of a small number of immobile modules, with the most basic deployed MMS containing only modules for altitude control, communications and data handling, and the power subsystems module [%].

The MMS flew only six missions through its lifetime which was vastly different from the thirty-one expected in the 1970s [%], it suffered limitation in the form of electronic technologies rather than mechanical restraints. NASA’s first Standard Spacecraft Computer (NSSC-1) [%] was developed to prevent requiring an entire redesign of onboard computers for each mission, requiring only a software redesign though this was still a heavy burden affecting the MMS’s mission flexibility. While no longer in operation as of 2006 [%], the system did show cost-savings in the range of 55% to 65% [%]. “The idea of a modular system serving many purposes was the pioneer of the leading systems within the space technology ecosystem today as it has left a lasting legacy” [%]. In the wake of the MMS’s legacy, new design techniques were developed such as the Modular, Adaptive, Reconfigurable Systems (MARS) system-level architecture [%] that has built the foundation for modern space systems.

### Modular Common Spacecraft Bus (MCSB)

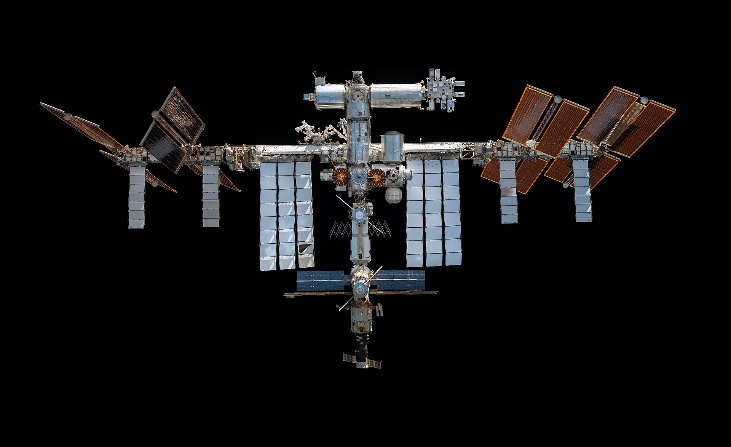
A diagram of a space module

Description automatically generatedThe MCSB is a fast-development, low-cost, general purpose spacecraft platform consisting of a series of 4-5 modules stacked on top of each other, each serving separate functionality [%]. According to NASA, “the spacecraft is roughly one tenth the price of a conventional unmanned mission and could be used to land on the Moon, orbit Earth, or rendezvous with near-Earth objects.” [%]

The MCSB system received the Popular Mechanics 2014 breakthrough Award for innovation in science and technology [%] and is proving to be at the forefront of existing modular space technologies, first deployed on the Lunar Atmosphere and Dust Environment Explorer (LADEE) mission in 2013 [%].

The MCSB system is an example of modularity being used to streamline and reduce costs of the initial development process of the craft, being able to carry up to 50kg of scientific equipment inside its payload module [%], though the end product is still a whole system that has limited in-operation service capabilities and is not capable of being reconfigured to adapt to mission requirements in-orbit.

### International Space Station (ISS)

The International Space Station (ISS) is the largest space platform ever built, created with the purpose of performing microgravity and space environment experiments. First launched in 1998 and expanded through the integration of additional modules and serviced by human occupants up until its planned de-orbit in 2031, it is a monument to advancements in the space industry.

The ISS is capable of reconfiguration using a robotic arm and automated docking with human oversight [%] unlike previous cases, though unsupervised automated reconfiguration is yet to be attempted due to the consequences of failure.

Although the examples provided are not exhaustive, they encompass significant cases of modularity in the history of space exploration. Currently, automated spacecraft reconfiguration remains unimplemented in the industry. This project aims to contribute towards the future widespread adoption of automated modular reconfiguration by developing a system that can be compared with other emerging systems, aiding to identify techniques that offer the most substantial benefits. These techniques can then be utilised to create increasingly advanced reconfiguration systems for space applications.

## Challenges and Limitations of Automated Reconfiguration in Space

The limited deployment of complex automated systems, like automated reconfiguration systems, in space is not due to a lack of interest, but rather stems from the formidable technical challenges and high-risk nature of space missions, which cannot afford failures due to their high cost and critical objectives.

Space systems must exhibit high reliability and operate effectively across a wide range of conditions. As system complexity increases, so does the number of potential failure points, making the validation, verification, and deployment of complex systems in the space industry a lengthy and costly process. Challenges that autonomous space systems face include:

* **Communication latency** – Delays in communications render real-time human intervention impossible, necessitating autonomous systems capable of operating independently without human oversight. Unlike terrestrial applications like self-driving cars that operate under human supervision, autonomous space systems must meet stringent autonomous reliability requirements.
* **Safety Requirements** – Systems will often be hosting valuable scientific equipment while operating in harsh, unpredictable environments with various hazards such as extreme temperature fluctuations, radiation, space debris, ice, and microgravity.
* **Limited Power Sources** – Autonomous systems rely on power sources that may not be constant or reliable. For instance, solar-powered crafts may experience power loss during eclipses or due to unexpected collisions with space debris. Autonomous systems must be capable of recovering from temporary power losses or have reliable backup power sources to prevent mission failure.
* **Isolation** – Unlike on Earth, space missions lack immediate external assistance or observation. Autonomous systems must possess robust sensing capabilities to self-diagnose issues, detect anomalies, and suspend standard operations when necessary to prevent further damage.

Overcoming these challenges demands cutting-edge technology, which has only recently become available, motivating research projects like this one. As computational power and materials sciences advance, we can expect a significant increase in autonomous systems within the space industry in the coming decades.

## Emerging Advancements in Reconfiguration Technologies

### MOSAR Project Outcomes

The MOSAR project has achieved several significant outcomes to date:

* Development of a standardized module framework utilizing the HOTDOCK adapter.
* Design and fabrication of a walking manipulator arm.
* Establishment of a related system architecture for remote control of the manipulator arm.
* Successful ground demonstration showcasing the manipulator arm's capabilities to move and connect modules.

At this stage, the MOSAR demonstrator could theoretically perform reconfiguration in orbit but currently requires manual transmission of reconfiguration instructions to the craft. Further work is needed to enable automated functionality, including:

* Automatic determination of a desired module configuration to meet mission requirements.
* Automated computation of manipulator instructions necessary to reconfigure the craft from one configuration to another.

The following review of automated reconfiguration literature will focus on identifying the best methods for the computation of manipulator instructions.

### Automated Reconfiguration

Automatic planners, algorithms that find a solution for which sequence of operations must be accomplished to achieve a specified goal, have been an area of development attracting wide-spread interest since the earliest days of robotics. Currently there are many different types of automatic planning techniques available. They encompass a large set of algorithmic requirements which trend towards purely discrete or purely continuous search space characteristics. The development of “Hybrid” automated planning approaches with search space characteristics that are not purely discrete or continuous, especially Task and Motion Planning (TAMP) algorithms, represent an area of study of which solutions are considered the most computationally difficult in theory [%]. Consequently, the application of automated planning algorithms to robotic assembly of modular satellites is a very recent development in which little work has been published that implements automatic reconfiguration algorithms while fully considering the range of real-world physical restraints and limitations presented by usage of a mobile manipulator arm in a low-gravity environment.

A diagram of a hybrid planning

Description automatically generated

Taxonomy of automated planning approaches based on their search spaces’ characteristics. Image from [%1].

#### Motion and Manipulation Planning

Motion Planning is finding solutions to move a robot “from one configuration to another configuration without colliding with the objects in the world” [%1]. It involves searching for paths within the robots reach which is a continuous configuration space limited by dimensions represented by the joints of the robot. These collision-free paths are important for robot motion but do not by themselves allow the robot to interact with the world. Further planning must be implemented to allow manipulation of objects through manipulation planning (known as Multi-Modal Motion Planning).

Due to the increased complexity of the problem presented by manipulation planning, the problem is best broken down into a hybrid discrete-continuous search problem of “selecting a finite sequence of discrete action types (e.g. which objects to pick and place), continuous action parameters (such as object poses to place and grasps), and continuous motion paths” [%1].

#### Task Planning

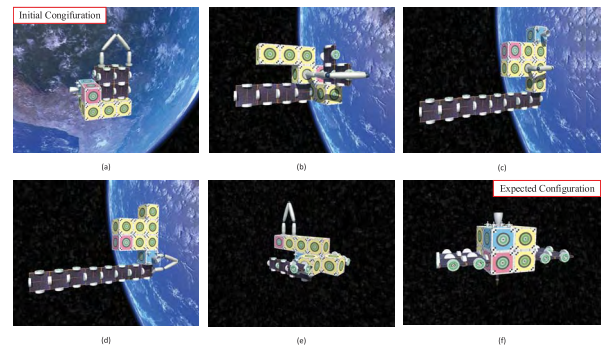
While Motion and Manipulation planning are seen as problems mainly within the robotics field, planning within large discrete domains such as in problems presented by task planning has been more deeply researched within the artificial intelligence (AI) community [%]. Task planning (also known as Action planning) referring to deducing a composition of symbolic actions to achieve a high-level goal (e.g. computing a sequence of actions required to stack boxes in a specified order). The discrete nature of the problem makes it particularly suitable for many machine learning techniques which have particularly advanced in recent years.

#### Task and Motion Planning

Current research in task and motion planning (TAMP) primarily aims to combine the robotics solutions for manipulation planning under physical constraints with the usually unrestricted machine learning approach to task planning. With the goal of deriving automated planning systems capable of reasoning symbolically with discrete “high-level” robotic action sets while geometrically taking into account continuous “low-level” robotic motion planning and restrictions. To date, several papers have developed algorithms for similar TAMP problems to the scenario of modular satellite reconfiguration that unfortunately are not compatible due to the method of module mobility, but act as a proof of concept that a solution is possible [%][%][%].

#### Related Work

The 2010 Intelligent Building Blocks for On-Orbit Satellite Servicing and Assembly (IBOSS) project [%] by DLR provided many advances in the area of satellite modularization with the development of standardised building blocks and interfaces [%]. Simple task planning techniques were implemented using Hierarchical task network (HTN) planning to produce high-level mobile arm instruction sets to then be verified through inverse kinematic checks and motion planning. This implementation solved the discrete and continuous planning problems separately, which simplified the problem however does not allow the separate systems to properly integrate. The system was not capable of efficiently solving more difficult tasks of identifying were solutions where not feasible.



Alternatively, another approach was taken here [%] through the implementation of the melt-grow algorithm [%]. The physical restraints of the robot were not including in the reconfiguration planning stage of the system, effectively reducing the problem to task planning. This reduces complexity though can only be achieved due to the behaviour of the melt-grow algorithm, which deconstructs (melts) the initial module configuration into chains of modules defined as the intermediate configuration, seen in configuration d in figure [%], before then reconstructing (growing) the modules into the expected configuration. The system then does not need to consider whether a move is possible for the mobile arm through manipulation planning as due to the algorithms inclusion of an intermediate state between the melting and growing operations, the algorithm essentially reconstructs the satellite instead of modifying the current state, all required moves are possible for the mobile manipulator and simply require motion planning. While proven to work, this method is shown to be highly inefficient for the mobile manipulator, especially as the number of modules increases in the system. Though, the paper [%] suggests this could be offset by the inclusion of additional manipulators which would consequently increase construction and operational costs.

More recent research has taken inspiration from these previous works to propose a comprehensive Task and Motion Planning (TAMP) problem solver [%] to intrinsically include the robot constraints into the system. The system, seen in figure [%], includes a logic layer, a physical layer, and a feedback system. Where the logic layer acts as a task planner finding a semantic solution by considering the solution as a sequence of states, with module movements defining the transition between states. A graph is developed to represent the possible states where nodes are system states and edges represent module movements which are verified by the physical layer which provides manipulation planning results through the feedback system. Using this graph, the shortest and hence most efficient set of operations to reconfigure the system into the desired state can be identified. The removal of the intermediate configuration present in the melt-grow algorithm improves the efficiency of the solution set of operations, especially as the number of modules in the system increases, requiring less movement from the mobile manipulator.

Diagram of a machine

Description automatically generated

“Architecture of the autonomous robot planning system. The system receives as inputs the start and goal satellite configurations, and iterates between the logic and physical layer until a solution is found.” [%]

The paper notes “the goal of this work was not to set a baseline for planning problems in terms of absolute times, but to demonstrate the usefulness of integrating feedback from the physical layer on the logic layer.” [%], suggesting that there is an opportunity for further research into the components of the planning system and the related feedback strategies to further advance the system towards space applications.

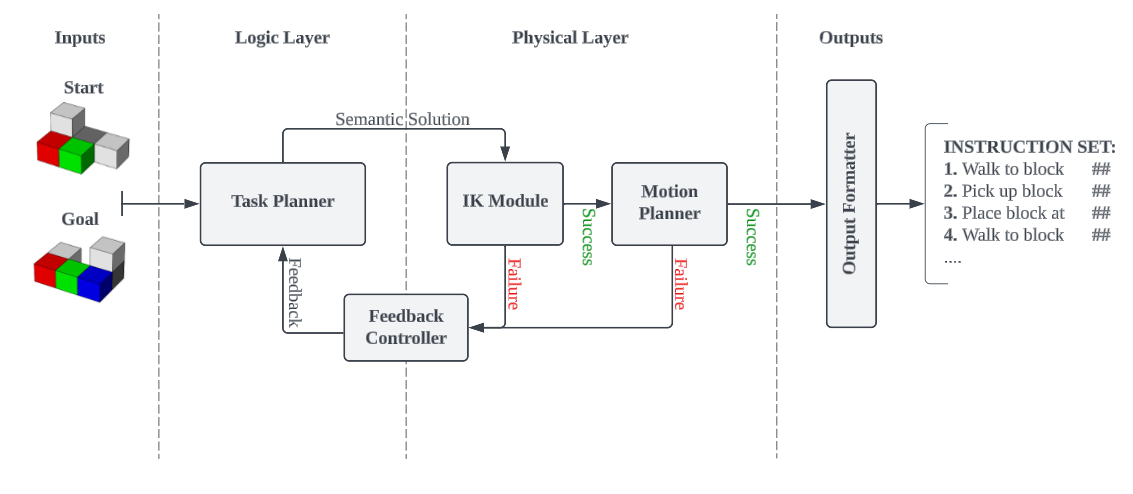
## Gaps and Opportunities

Modular reconfiguration defines a subclass of the generic planning problems usually addressed by TAMPs. Although research has previously demonstrated effective systems that can handle both symbolic and geometric reasoning, their application to robotic assembly and in particular robotic re-assembly is currently limited. There is additionally a distinct lack of discriminating modular blocks by type in existing algorithms which could potentially be implemented without a substantial hit to system performance.

The system proposed in figure [%] is promising due to the robustness of solutions and flexibility of the logic layer, however, there lacks the extensive performance testing required to recognise weaknesses and future improvements and identify why this system could not be used in real-world application currently.

# System Design and Development

## Overview



The reconfiguration task and motion planner (TAMP) program is designed around a python implementation as there is already an existing python implementation for controlling the robot arm present in the lab [%][%]. The program takes an initial and final state as inputs, and outputs a list of instructions required to reconfigure the initial state into the final state using a mobile manipulator.

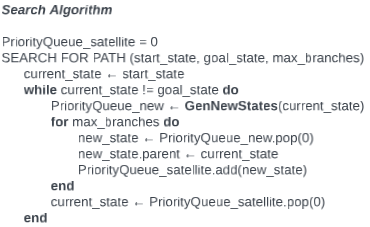
The overall system consists of a logic layer, physical layer, and feedback controller. Each of the two layers handle a separate role in the overall system allowing the discrete and continuous components of the solution search to be handled individually, while the feedback controller handles the integration and communication between the layers. The Logic layer is responsible for discrete task planning, while the physical layer is responsible for the continuous inverse kinematic checks and motion planning. Together, the integration of the two system layers through the feedback controller form the overall TAMP program.

The logic and physical layer communicate between each other using feedback strategies. These strategies are implemented by the feedback controller to create control loop type behaviour, which works to iteratively find a feasible solution. The solution is then converted to the required instruction set format by the output formatter.

## Logic Layer

### Overview

### The logic layer is primarily a Task Planner handling the discrete portion of the TAMP search to find semantic solutions. These solutions being a sequence of state configurations with a difference of one module movement between them, that transforms our initial state configuration into our goal state configuration. The best examples of existing task planners used today use machine learning techniques to find solutions. However, these techniques are inappropriate for use in the space industry due to introducing black box behaviour to the system. So instead, simple graph search techniques are implemented. The pseudo-code for our algorithm can be seen in figure [%].



### Searching the Graph

To search the graph to find the path to the goal state configuration, there are 2 major search algorithms available. Depth-first and Breadth-first search. In depth-first search, the algorithm travels all the way through a possible path before attempting to search another path. We can implement this algorithm through the use of a state priority queue that sorts state according to how close to the goal state they are to find a solution very quickly. Though the found semantic solution would be one of many possible paths to the goal state and is rarely the most efficient semantic solution for the mobile manipulator as each state transition requires extra movement for the mobile manipulator, hence the less transitions, the better for the application of this reconfiguration program to the space industry. Taking this into consideration, it makes sense to instead implement a breadth-first search which searches states one layer at a time. The algorithm searches all states one level deep away from the starting state, before searching all states two levels deep, then three levels and so on. The breadth-first search algorithm is far slower than the depth-first search algorithm which hinders the algorithms’ ability to scale to a larger number of modules; But semantic solutions found will always be paths with the smallest number of transitions to the goal state.

### Generating States

To expand the graph, the task planner needs to take a state and generate a set of new states using the GenNewStates() function referenced in our search algorithm pseudo-code. States are generated using a set of basic rules that prioritise which blocks to move to generate a set of states. The priority of a module in the generation of a move set is defined as follows:

1. Modules not yet in their final position
2. Modules adjacent to modules not yet in their final position
3. Remaining modules

Using the above rules, we guarantee that more sets can always be generated, though modules that require moving first will almost always be prioritised. The construction of a priority queue that would prioritise blocks by their distance to modules not yet in their final position was considered to allow blocks that are for example, deep within a structure, to be dug out efficiently; Though this was seen as adding unnecessary complexity and would increase computation time just to cover a relatively rare scenario for our current program usage. If this planner was to be applied to larger structures though, this technique should be considered to improve computation time.

### Trimming States

When handling problems with large numbers of modules, the generation of new states often produces a huge number of states which quickly takes up memory and computation time. To speed up the search, states are sorted into a priority queue upon generation, then a chosen number of highest priority states are added to the search graph, while the remaining states are discarded. States are sorted according to how close they are to the desired goal configuration according to the following defined heuristics:

1. The number of modules in the current state that are already in their final positions.
2. The number of modules in the current state that are not in their final positions, however, are in positions that do not contain a module in the goal state.
3. The sum of the Euclidean distances of the module positions in the current state to their final positions in the goal state.

This heuristic is used to measure how far a state is from the goal state and compare the state against another state to find which state is closest to the goal state. The first rule is used to sort states into the priority queue, but in the case of a tie, the second rule is used to compare the states instead. In the case of another tie, the distance each module is from their final position is used instead which is a simple however computationally intensive comparison to find. The initial two rules help to significantly speed up the comparison of states by reducing how often the planner must run expensive calculations.

### Physical Layer Feedback

When a semantic solution is found, it is sent to the physical layer to verify the solution. In the case of a failure, the transition that caused the failure is trimmed from the tree and all generated transitions on the corresponding branch of the graph are removed. The search then resumes without the failing transition. An alternative method would be to perform physical layer checks on each individual move when generating states, though in comparison to the logic layer, the calculations conducted by the physical layer are far more intensive. So, it is preferred to verify only the transitions involved in the semantic solution and spend longer searching for semantic solutions.

## Physical Layer

### Overview

The physical layer is primarily composed of an inverse kinematics module and motion planner. It is responsible for the physical verification of transitions in the semantic solution proposed by the task planner in the logic layer. It takes as input a semantic solution and returns the solution with either success or a failure.

When a module transition is input into the inverse kinematics module, it will verify that the pose for picking up and placing the module is possible from the current base position. If either pose is not possible, the module will move the base to attempt to find a base position which can both pick up and place the module. Only when both poses are possible from the same position will the inverse kinematics allow the transition to continue to the motion planner, otherwise the module returns an inverse kinematics failure and feeds back the transition to the logic layer.

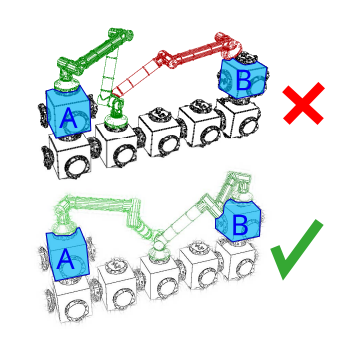
The motion planner upon receiving a transition, will find a path from the start and end pose positions which avoids the arm body and grabbed module colliding with other modules. If no solution is possible, the motion planner returns the transition to the logic layer with a motion planning failure.

### Inverse Kinematics Module

Inverse kinematics is used to find the joint angles required to place a mobile manipulators end-effector at a desired position and orientation. There are multiple methods for finding inverse kinematics, primarily using an analytical approach or the Pseudoinverse Jacobian method. The analytical approach is far more difficult as it requires analysing and finding a mathematical solution for each unique manipulator arm specification, but due to producing a basic formula for finding each joint angle, it is much faster to calculate. The Pseudoinverse Jacobian method however simply takes a guess of the required joint angles, and then iteratively increments/decrements joint angles to move towards the target position and orientation to find a solution. This method is more generic and easily works for any arm configuration but is slow to calculate.

As we are expecting to calculate inverse kinematics often, we will use an analytical approach which can be calculated by representing our mobile manipulator as a set of Denavit-Hartenberg Frames.

### Manipulator Base Location Planning



The mobile manipulator present in the lab has a stationary base, making this implemented portion of the overall TAMP planner currently an unnecessary inclusion for demonstration. However, the mobile manipulator present in the MOSAR [%] project is capable of walking around the modular satellite and connection to module at either end of the manipulator arm. Therefore, if the inverse kinematics module fails to connect to a module from a position, it is possible that moving the base further towards the module that failed the inverse kinematic check could result in a successful solution. This can be seen in figure [%]. When the inverse kinematics module fails a check, it attempts to move the base to another available surface in between the 2 movement points. The inverse kinematics module will only return a failure once all appropriate surfaces have been verified as unsuccessful.

### Motion Planning

Originally, the design and plan were to implement a motion planner utilising the RRT-algorithm [%] to find and return collision-free motion paths for the robot arm. The advantage of a complex motion planner is that it would not be designed for any specific scenario, allowing it to produce plans for a wide range of environments and different mobile manipulator types.

Though due to time constraints, instead a simple set of rules are to be used to develop motion path plans for the specific scenario available in the lab where modules are reconfigured on a platform by a stationary robot arm. We simplify planning by assuming that:

* A module can be picked up by the robot arm if no modules are above, it and the module is in reach according to the inverse kinematic solution.
* Modules can be moved between positions by simplify moving the module up to the z-limit of the arm, across to above the new position, and straight back down.

In this scenario the modules are not being reconfigured as they would be in space, so additional physical constraints must be implemented through feedback:

* Modules can not be moved to negative-z values due to the platform.
* Modules must be placed on top of another module due to gravity.

Including these physical restraints will induce many physical layer failures, so it is expected that the logic layer time will largely increase compared to regular operation of the system.

### Failure Feedback

The physical layer will return several types of failures when conflicts are detected according to where the failure occurred. Examples of returned failures include:

* Out of reach – the module beginning, and final positions are out of reach of each other for the current mobile manipulator.
* No base location – the modular beginning and final positions are in reach of each other however, there is no appropriate base position on the module configuration that can reach both points.
* Collision – There is no path available to move the module without colliding with another module.

## Feedback Strategies

Without feedback strategies, the system can:

1. Find a step-by-step solution to reconfigure modules into the desired goal configuration.
2. Verify whether the mobile manipulator can perform the resulting semantic solution.

Feedback strategies are how the system implements a control loop that gives the logic and physical layers the ability to communicate; Allowing them to work in tandem to find a solution that satisfies the goal of each layer. There are several different considered feedback strategies that work to either optimise the time efficiency of the overall system or improve the validity and efficiency of the final output for the mobile manipulator. Due to project scope, the implementation aims to incorporate feedback strategies related to final output validation and not computation time optimisation, though strategies for both were considered in case of early project completion or for further work beyond project completion.

### Semantic Solution Verification

The feedback strategy to be primarily implemented in this project is simple verification of semantic solutions. Solutions that fail are returned to the logic layer, and the transition causing the failure is then trimmed from the tree along with all configurations that originated from that transition. The logic layer will then continue the search without those transitions. There is concern that, for example, the physical layer will fail most transitions from a node early in the search process, which results in the search space being reduced to such a small scope that no reasonable solution is found. Further testing and research will be required to deduce whether this is a problem that can be solved with basic modification and advancement of the feedback strategy, or whether other methods of communicating physical layer failures should be pursued. The testing and resultant analysis of the verification feedback strategy will be used as a primary performance metric upon completion.

### Failure Memory

Implementing a failure memory much like the one developed in this paper [%] was considered; But after further research, was deemed outside the scope of this project due to the primary benefit being time performance based. The failure memory in the mentioned paper was a machine learning algorithm that would use a database of past failures as a training set, to predict whether a transition in the logic layer would later fail in the physical layer. Transitions with a high probability of failure were then trimmed from the set of generated states during graph expansion, reducing the overall run time of the system by reducing the number of calls to the physical layer along with reducing the set of states the algorithm was required to search through. For space application, it was desired to develop a system that avoids the use of black box machine learning algorithms, however still implemented memorising failures to give it the ability to gain efficiency over time. Specifically implementing a failure memory would give the system scalability over time, while originally the system would be too slow for large systems, gaining experience through saving failures would allow the system to over time gain enough experience to generate solutions for a larger number of modules in reasonable time for real-world applications.

# System Implementation and Specifications

## Hardware Specifications

### Processing Hardware

The current software implementation requires only a simple processor though is limited by the speed of the processor. The only hard limit on the software is the available RAM, it is recommended to have at least 2 – 4 GB of RAM available for use by the planner, depending on the size of the state configuration input into the system.

During the project, the hardware used to develop results related to system timing was a general-purpose desktop computer with the following specifications:

Operating System: Windows 10 Pro 64-bit (10.0, Build 19045)

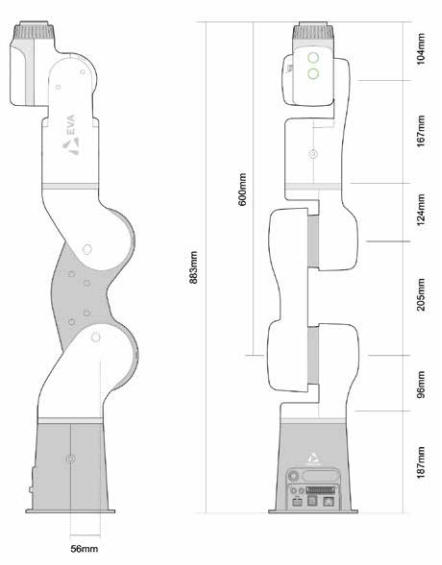
Processor: AMD Ryzen 7 3700X 8-Core (16 CPUs), running at ~3.6 GHz

GPU: NVIDIA GeForce RTX 2070 Super 8GB

RAM: 3 x 8 GB 3200 MHz DDR4 Memory

Memory: 1 TB M.2 SSD running at 7,300 MB/s read, 540 MB/s write speeds

### Mobile Manipulator

The mobile manipulator available in the lab is the Automata EVA [%] which has the configuration and joint limits seen in figure [%] and [%], further details can be seen in Appendix [%].

While the manipulator was not physically used in this project due to time constraints. The specifications of the arm were used for simulation in the implemented software layer; Meaning the current state of the reconfiguration planner software could be integrated with the arm in a future project.

## Software Specifications

### Overview

A diagram of a process

Description automatically generated

An overview of the modular structure of the final software implementation can be seen in figure [%]. The main file utilises the Logic Layer and Physical Layer separately to simplify communication between the modules and to apply feedback strategies between modules. The Output Formatter is used after a solution is found to correctly display the instructions to users and additionally is used to create state reconfiguration transition animations for visual analysis of the process.

### Logic Layer

#### Task Planner

The Task Planner begins by verifying the start and goal states have the same number and composition of modules; And generates states utilising the State Priority Queue and State Classes. It returns an array of states representing the transitions required to reconfigure the start state into the goal state configuration. The Planner consists of 2 major methods, “find\_path “and “generate\_states”. While “find\_path” implements the search algorithm pseudo-code seen in figure [%], “generate\_states” expands the tree according to the heuristics specified in the design section [%] as can be seen in figure [%]. After states are generated, the find\_path function saves their state of origin as the states parent, allowing the program to iterate through the state’s ancestors back to the starting state configuration, to view the transition path required to transition from the starting state, to the generated state.

**GENERATE\_STATES**(state, goal\_state)

state\_queue <- new StateQueue(goal\_state)

from <- state.get\_non\_final\_modules()

to <- state.get\_available\_positions()

state\_queue.push(state.generate\_moves(from, to))

**if** State\_queue.empty() **do**:

from <- state.get\_adjacent\_modules(b)

state\_queue.push(state.generate\_moves(from, to))

**end**

**if** State\_queue.empty() **do**:

from <- state.get\_modules()

state\_queue.push(state.generate\_moves(from, to))

**end**

**return** state\_queue

#### State Priority Queue Class

The State Priority Queue class uses a simple array data structure to construct a sorted list of states, ranging from closest to furthest state from the specified goal state. State priority is decided through the heuristics detailed in design section x[%]. When a state is inserted into the queue, a binary search algorithm [%] is used to search through the queue, comparing the inserted state with states in the queue to identify where the inserted state belongs in the sorted queue to maintain priority order. Originally a linear search algorithm was used to simplify the implementation process, but later during development while undergoing optimisation, the binary search algorithm was implemented which showed an exponential decrease in overall time spent inserting states into the queue.

#### State Class

The state class is responsible for representing a state configuration and storing Module positions. The class contains methods for state comparisons, measurements, validation, and visualisation. A dictionary data structure is used for the efficient storage and access of modules, where modules are stored in key, value pairs where the key is the module position, and the value is a module object. A dictionary data structure was chosen as it does not require storing zero values such as in a position matrix, making it memory efficient. It also is capable of quickly returning a list of modules or module positions by simply looking at the values or keys. A position matrix data structure was initially utilised early during development to store modules and simplify testing and modification of internal logic, along with increasing development speed, but was later updated to use a dictionary for optimisation purposes.

The class verification function is used to verify all modules within the state are connected. The original position matrix data structure cleverly utilised an out-of-the-box clustering algorithm [%] to verify the state. If more than one cluster was detected, not all blocks in the state were connected. After switching to a dictionary data structure, the verification algorithm required updating; So, a new simple search algorithm was implemented to verify states seen in fig [%].

VERIFY\_STATE(state)

found\_list <- []

search\_list <- [state.first\_module\_in\_dictionary]

**while** search\_list not empty **do:**

module <- search\_list.pop()

**for** neighbour in module.get\_neighbours() **do**:

**if** neighbour not in found\_list **do**:

**if** neighbour not in search\_list **do**:

search\_list.push(neighbour)

**end**

**end**

**end**

**end**

**return** found\_list.length() == state.num\_modules()

Several functions are implemented to measure the number of modules in final positions, the number of modules in free positions and the Euclidean distance between all modules in non-final positions and their final positions. These measurements are often requested many times for the same state, so the calculated values are saved within the state after the first time the get measurement function is run, and only changed if the goal state changes.

For visualising reconfigurations and aiding in-depth testing and analysis, the state class contains a display function which translates the dictionary into a position matrix to be display on as a 3d matrix using matplotlib [%]. Configurations such as the one seen in fig [%] can then be display. Additionally, the function is used to develop reconfiguration videos to visualise system output.

A colorful cubes with different colors

Description automatically generated

As pointed out in figure [%], the state contains a function for generating a list of states for mass movement of modules. The list takes 2 lists, a list of positions of modules, and a list of positions the modules can move to. For each module, the function will validate the state without the moving module (to ensure the state doesn’t break apart mid-movement) then move the module to each of the movement positions and if the state is valid after movement, the new state is added to a return list. The custom mass movement function is used to optimise the generation of states during search tree expansion.

#### Module Class

The Module class is a simple class holding information about the module. It is primarily used to compare modules using an equal’s function. The function can be modified to adjust what module properties are used to decide whether two modules are equal. In the classes current state, colour is used to decide if 2 modules are equal. There is implemented functionality to instead decide using module type or module identification number, however comparing by colour simplified analysis during testing, as modules could be visually differentiated when displayed.

### Physical Layer

#### Inverse Kinematics Verifier

As stated in the design section [%], the implemented Inverse Kinematics Verifier uses an analytical inverse kinematics solution to calculate manipulator joint angles required to place the manipulator end-effector at a specified position. Initially, an analytical formula was developed for the Automata EVA [%] arm available in the lab. This made the program unique to only the specific mobile manipulator, reducing the scope of hardware compatible with the completed reconfiguration planner. There are python libraries available that can create analytical solutions for mobile manipulators from a Unified Robotics Description Format (URDF) file.

To increase the compatibility of the reconfiguration planner, a URDF file was developed for the Automata EVA seen in appendix [%]. The IKPy package [%] was then used to generate an analytical solution for use by the Inverse Kinematics Calculator and Motion Planner. Users can then update which mobile manipulator the reconfiguration planner is solving physical solutions for by simply replacing or modifying the URDF file.

The inverse kinematics verifier is used to verify that the start and final position of each module movement in the reconfiguration semantic solution is reachable by the mobile manipulator. In the case of a module being out of reach, the verifier returns the state transition that caused the failure. Otherwise, the semantic solution is sent to the motion planner for further processing.

#### Robot Description File

A URDF file is used to define the mechanical structure, dimensions, joint configurations, and physical constraints of the mobile manipulator the physical layer is using to verify the logic layers semantic solution. URDF files are an XML-based file format that is widely used in robotics [%] to describe robots to software systems. The file describes a robot as a collection of links and joints that can articulate around each other according to specified constraints. URDF files are also modular meaning they can include other URDF files, aiding in the design of particularly complex robots. This for example means that a user can develop a URDF file for an arm end-effector and simply include it in the already existing arm file to attach it to the arm.

URDF files also allow for the visualization of the defined arm joints, as seen in figure [%] which can be overlaid on top of our module state display to visualise mobile manipulator pose on the modular space system. Additionally available online packages such as urdf-loader [%] can display the visual meshes described in the URDF file to view the mobile manipulator in more detail such as seen in figure [%].

A graph of a line graph

Description automatically generated

A white robot with a black foot

Description automatically generated with medium confidence

#### Motion Planner

Due to time constraints during the project, instead of implementing an advanced motion planner, a simple motion planner was implemented that makes assumptions based on the lab environment surrounding the mobile manipulator available in the lab. Physical constraints of the robot arm and the environment are applied to the system during module movements using 2 basic rules:

1. Modules can only be picked up by the mobile arm if no blocks are above them.
2. Modules can only be placed at a supported position (above another module and on the ground) and cannot be placed at negative z values (below the ground)

The combination of these two rules applies the physical constraints of gravity and the presence of the ground. If either rule is broken, the planner returns the state transition causing the failure. Otherwise, it generates and returns an instruction set in the form of an array containing each manipulator action sequentially required to perform the state reconfiguration. The available instructions the motion planner can generate can be seen in fig [%]

|  |  |  |
| --- | --- | --- |
| Instruction | Supplied Information | Action |
| Connect | None | Signal the arms end-effector to grab/connect |
| Disconnect | None | Signal to the arms end-effector to drop/disconnect |
| Move to | Position, Joint angles | Move the arm to position the end-effector at the supplied position by transitioning the arms joint angles to the supplied joint angles |

### Feedback Strategies

In the systems current state, only a simple feedback strategy is implemented. When a failure is detected within the physical layer, the state transition that caused the failure is returned. The branch of the tree resulting from the failing state transition is then removed from the existing search tree. Leaving behind only states that the failing state is not an ancestor of. The trimmed search tree is then input back to the logic layer to continue the search for another semantic solution.

## Implementation Challenges

The original complete implementation suffered from several initially unforeseen problems that required in-depth analysis and modifications to the system. These issues resulted in the system either being unable to find a solution, requiring more powerful hardware, or taking so long to find a solution that testing was not feasible.

### Memory Usage

The data structure implemented simply used a dictionary with positions as keys, storing only necessary information and making modifications to internal logic and computations simple. At a point during development when testing configurations with a larger number of modules, the hardware used to run the system ran out of memory (24 GB) and the program was forced to stop the search. The nature of the anomaly was suspicious as the memory usage was orders of magnitude more exponential than expected when compared to lower module numbers, especially considering the miniscule size of the dictionaries involved. Further analysis found the in-built copy module [%] used in python would not only copy objects, but also objects referenced within the object, resulting in the entire search graph duplicating during each transition. The oversight of this behaviour can only be put down as lack of experience with the python programming language and how variables are stored (whether in pointer or absolute form). Development of a custom duplication method for the class resulted in significant reduction of memory usage along with an additional 90% reduction in search algorithm speed, greatly increasing the capabilities of the logic layer.

### Repeated Computations

While investigating which processes in the system had the highest contribution to the calculation time of the final solution, the Generate\_States [%] function was identified as one of the slowest processes within the system. The function inserts state into a priority queue that holds a custom comparison function that requires the calculation of several different measurements for each state being compared to decide priority. The speed the function inserted states into the priority queue was identified as a major time consumption source, and the initial suspicion was that the linear search being used to find the required insertion index was the cause. Though once a more efficient search algorithm was implemented (binary search), the time consumption of insertion did not increase as much as initially expected. Further analysis showed that when inserting a state, when there are many states in the queue, resulted in a single state during insertion being compared to many states and repeatedly calculating its own measurement values against the goal state to decide priority. Meanwhile states already in the queue would often be compared to each state inserted into the queue, also repeating the same calculations to find measurement values. As these measurement values are based on their module layout and composition, and the goal state, which does not change during runtime, there was no need to be repeating calculations.

The solution was to save measurement values upon calculation for each state, and only calculating the values when needed, unless the goal state was to change upon which all saved values would be deleted. Saving values along with using a binary search drastically reduced overall calculations required during insertion.

### Optimising Python

The greatest overall roadblock to further success and experimentation simply came down to the speed of the system and identifying slow processes. Due to the nature of the python programming language, often a considerable time-wasting process was a simple function that unknowingly had large overheads involved. The most successful method of improving the speed of the system during development was writing custom functions that utilise libraries built to utilise the C programming language. While there was no issue writing the custom utility functions, identifying where custom functions needed to be implemented took up most of the dedicated development time.

# Testing and Results

## Testing Method

To test the system, a generated set of module configurations were input into the system, and the total time spent in the logic layer and hardware layer were recorded separately for both the system with and without physical constraints applied. An average time is then calculated for configurations consisting of 4 – 9 modules. The branching factor was also varied throughout the test to analyse the effect on total time, failure rate and number of moves returned in the final solution.

## Performance Metric

To measure performance of the overall system, the testing methods measure:

* The number of failures encountered during the solution search with feedback strategies.
* The number of modules effect on failure count
* The number of search branches effect on failure count
* The number of search branches effect on number of moves in the solution.
* The number of search branches effect on search time.
* Total calculation time.

To Quantify the performance of the implemented feedback strategy, the number of semantic solutions that fail in the physical is used. The physical layer is the most computationally demanding section of the overall system, so it is desired to be used as little as possible. Time spent calculating results is recorded for comparison with other systems and to prove the systems capabilities for real-world use but is not considered a measure of project success.

## Analysis of results

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
|  |  | **branches** | | | | | |
|  |  | **1** | **2** | **3** | **4** | **5** | **6** |
| **modules** | **4** | 0.425s | 0.441s | 0.497s | 0.597s | 0.77s | 1.018s |
| **5** | DNF | DNF | 0.462s | 0.677s | 1.049s | 1.657s |
| **6** | DNF | DNF | 4.408s | 13.053s | 38.173s | 113.846s |
| **7** | DNF | DNF | 1.381s | 1.95s | 2.735s | 3.745s |
| **8** | DNF | DNF | DNF | DNF | 65.921s | 256.216s |
| **9** | DNF | DNF | DNF | DNF | DNF | 284.359s |

* Critical analysis how well my product would work in certain applications given the obtained results.
* Reference hardware specifications used to develop timing results.

# Discussion

## Interpretation of results – (what results say about current system)

## Comparison to existing work

* Melt-and-grow
* Paper used as primary reference

## Implications – (potential impact of work on the field)

* Demonstration day feedback – usage in warehouses
* Proof of concept of use in a wide range of applications, not just space industry as it isn’t designed for the specific use

# Planning and Time Management

## Project Management Procedures

To streamline the design and development of the project, the project followed a traditional engineering product development cycle consisting of 5 phases:

**Initiation** - The definition of the problem and the projects goals, requirements, and risks. This phase was completed by the given description of the project and further questioning of the project supervisor.

**Planning -** The definition of how to solve the problem by outlining the details and goals to meet the defined requirements. This phase was completed by the production of the initial report seen in appendix [%], the project plan seen in figure [%], and a conceptual high-level product design.

**Execution -** The working phase where the plan designed in the previous phase is put into action and the product is developed. This was completed according to the created project plan and was finished in its majority by the project demonstration day on the 29th of April 2024.

**Controlling & Monitoring -** This phase runs alongside the execution phase and involves tracking progress and adjusting the workflow to remove potential roadblocks.

**Closure -** Reflecting on the progress and results to officially end the project. This phase is conducted through analysis of project results, documentation of completed work and reflection of project success which is represented by this document.

Each phase was given a set of weeks to complete within the project plan, and every Friday a review of the plan was conducted to monitor progress and aid in modifying the plan in the case of unexpected roadblocks.

## Project Management Reflection

The project went according to plan through to the development of the physical layer. Due to unfamiliarity with robot kinematics, little in-depth design was created in the planning phase of the project with the assumption that with the knowledge of what each section of the physical layer needed to accomplish, figuring out how to accomplish it would not be a notable obstacle. This led to the physical layers’ development taking far longer than expected, over-running its planned development time by a week despite completing the logic layer a week earlier than expected. Due to overrunning the deadline, the project goal was instead completed by defining a simple physical layer rule to use for feedback such as “is the module at the top of the stack and hence, can be picked up in an environment with gravity by a stationary arm”; allowing the remainder of the project to be completed, making it possible to develop and analyse a range of feedback strategies without sacrificing time to develop a mostly unused and complex simulation.

Despite the delay, the final product does match what was planned at the beginning of the project, and as such the goals of the project have been filled. This can be attributed to appropriate levels of slack in task timing guidelines and creatively making use of out-of-the-box implementations to decrease production time drastically and reduce complexity.

## Risk Assessment

This section builds, and expands, on material previously included in the project Initial Report (see Appendix %)

Risk register:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| ID | Risk Description | Impact | Risk Probability | Mitigation of Risk | Effectiveness during project |
| 1 | Missing or corrupted documents | High | Medium | Documents are backed up to a GitHub repository | Highly effective |
| 2 | Ambitions for project are too great for the project time limit | High | High | Setting appropriate scope expectations from the beginning of the project | Slightly effective – did not properly take into consideration prior knowledge |
| 3 | Illness or work unavailability | High | Medium | Record illness and provide proper explanation for missing work in final report. Decrease scope to provide meaningful results | Highly effective – Illness affected several weeks of the project; However, scope was reduced appropriately |
| 4 | Losing test results | Medium | Medium | Produce lab reports to document progress | Highly effective |

## Evolution of Project Plan

The project plan saw little modification over the project. During progress reviews during the project, if it was seen that a section of the project would overrun its deadline, alternative methods of reaching a functional overall system were found that involved sacrificing small features such as including module orientation and module connectivity directivity direction. These features were still considered in the completed implementation allowing them to be designed and implemented with relative ease when the project is further developed in the future.

# Conclusion

* Refer back to objectives/specification in introduction

# Further Work

* Making the program work in real-time through a control loop, allowing the program to continue running and working towards rearranging a satellite towards a solution even when blocks have been removed mid-program or there are other moving objects within the surroundings. This would also allow the system to recover from temporary power failure pointed out in the literature review challenges section
* Support for multiple manipulator arms
* Support for clustering (moving multiple modules at once)
* Support for modules of different sizes
* Further testing through random generation of input/output states
* Improved measurement for similarity between states

# References

# Appendix A – System input/output

A screenshot of a computer screen

Description automatically generatedInputs:

A screenshot of a computer game

Description automatically generated

Output:

Instruction Set

['START']

['MOVE\_TO', (0.05, 1.05, 0.1), (0.0, -0.048, -1.664, -0.190, -3.169, -1.253, 0.0)]

['CONNECT']

['MOVE\_TO', (1.05, 0.05, 1.1), (0.0, -1.52, -0.770, -0.190, 3.568, -2.070, 0.0)]

['DISCONNECT']

['MOVE\_TO', (2.05, 1.05, 1.1), (0.0, -1.097, -1.171, -0.190, -2.075, -1.560, 0.0)]

['CONNECT']

['MOVE\_TO', (0.05, 0.05, 1.1), (0.0, -0.786, 0.230, -0.580, -3.527, -2.705, 0.0)]

['DISCONNECT']

['MOVE\_TO', (2.05, 1.05, 0.1), (0.0, -1.097, -1.615, -0.190, -0.598, -1.200, 0.0)]

['CONNECT']

['MOVE\_TO', (1.05, 1.05, 0.1), (0.0, -0.785, -1.637, -0.190, -1.336, -1.242, 0.0)]

['DISCONNECT']

['MOVE\_TO', (2.05, 0.05, 0.1), (0.0, -1.546, -1.620, -0.190, -5.366, -1.214, 0.0)]

['CONNECT']

['MOVE\_TO', (0.05, 1.05, 0.1), (0.0, -0.048, -1.664, -0.190, -3.169, -1.253, 0.0)]

['DISCONNECT']

['MOVE\_TO', (1.05, 0.05, 1.1), (0.0, -1.523, -0.769, -0.190, 3.568, -2.070, 0.0)]

['CONNECT']

['MOVE\_TO', (2.05, 1.05, 0.1), (0.0, -1.097, -1.615, -0.190, -0.598, -1.200, 0.0)]

['DISCONNECT']

['MOVE\_TO', (0.05, 0.05, 1.1), (0.0, -0.786, 0.230, -0.580, -3.527, -2.705, 0.0)]

['CONNECT']

['MOVE\_TO', (0.05, 1.05, 1.1), (0.0, -0.0476, -0.769, -0.190, -4.074, -2.070, 0.0)]

['DISCONNECT']

['END']

# Appendix B – Code (need to cite libraries used) (multiple appendix’s)

# Appendix C – URDF file

# Appendix D – Robot arm spec sheet

# Appendix E – Initial Report