## Abstract (200 – 300 words)

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

## Acknowledgements

## Ethics

* No current identified ethical considerations? Need to apply for ethical approval if any identified

## Table of Contents – (Include section on Figures, Tables and abbreviations)

## Glossary?

# Introduction

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

## Background and Context

Space systems have rapidly developed in recent years, with a global drive to increase commercial availability. Current commercial systems designed under the limitation of mass and launch costs, are traditionally highly customized whole systems which consequently have very limited or no maintenance and repair capabilities. The number of ageing satellites is rapidly increasing and upon reaching end-of-life, are discarded through atmospheric deconstruction methods if possible, or left in orbit contributing to space debris build-up.

Technology to circumvent these conditions is not currently available and as such, the HORIZON 2020 EU-funded MOdular Spacecraft Assembly and Reconfiguration (MOSAR) project was launched to develop novel technologies that would allow standardising satellites and components [%1]. The modularisation and standardisation of space systems will benefit the European space industry by facilitating mass production of standard components and therefore decreasing assembly costs, reducing time between customer orders and commissioning in space, and allowing repair and upgrading of components directly in-orbit.

MOSAR primarily aims to produce on-orbit modular and reconfigurable satellites. At present the project has developed a demonstrator for re-configuring cubic modules to simulate the movement of modules through the use of a mobile robotic manipulator. Currently fixed instructions facilitating module mobility are sent to the manipulator from a software simulation on earth [%]; This research project aims to further develop the capabilities of the overall system by developing an algorithm to automate the module reconfiguration process, facilitating self-repair and self-assembly. Following development, this technology has the potential to facilitate the automated assembly of space systems and platforms directly in space, expanding the limitations currently imposed on the space industry.

## Project Objectives and Specification

This project intends to introduce the capability of autonomous assembly and reconfiguration of a modular space system by implementing a reconfiguration planning program made up of simple algorithms. This program, given the initial state and final state of a modular craft as parameters, will produce a list of commands to send to the mobile manipulator. These commands can then be used to autonomously rearrange modules on a spacecraft or space platform in operation. This program must consider the physical constraints placed on the system by the mobile manipulator present on the modular system, therefore this project will also strive to explore methods of introducing physical constraints into the planning program.

To accomplish the research goal, a functional planning program must be implemented and be demonstratable through software simulation. Though if time permits, there is the additional goal of demonstrating the planning program by physically re-configuring real modules in the lab through integrating the system with the available manipulator arm.

The above goals of producing, testing and demonstrating the system has been broken down into the following sub-objectives:

1. Development of a reconfiguration planning program to produce module movement instructions to send to a mobile manipulator. The program must take as arguments the initial and final state configuration and return an instruction set.
2. Further expansion of the reconfiguration planning program to take into consideration the physical constraints placed onto the modular system by the mobile manipulator present.
3. Development of a display function to produce reconfiguration slideshows/videos, allowing users to view a simulation of the modular system reconfiguration from start to finish.
4. Analysis of the system through testing a variety of inputs and recording the time taken to find a solution along with failure rates.
5. Demonstration of the system through integration with the robot arm available in the laboratory.

## Project Scope

outline boundaries of the project and any restraints. There are little restraints other than the requirement to develop hardware for demonstration which puts development of a physical demonstration out of scope, however it can be completely prepared for from a software perspective

## Report Structure

This document aims to provide a detailed report of the research and development completed during the Autonomous Re-Configuration of Modular Spacecraft with Manipulator Arm project. The report includes:

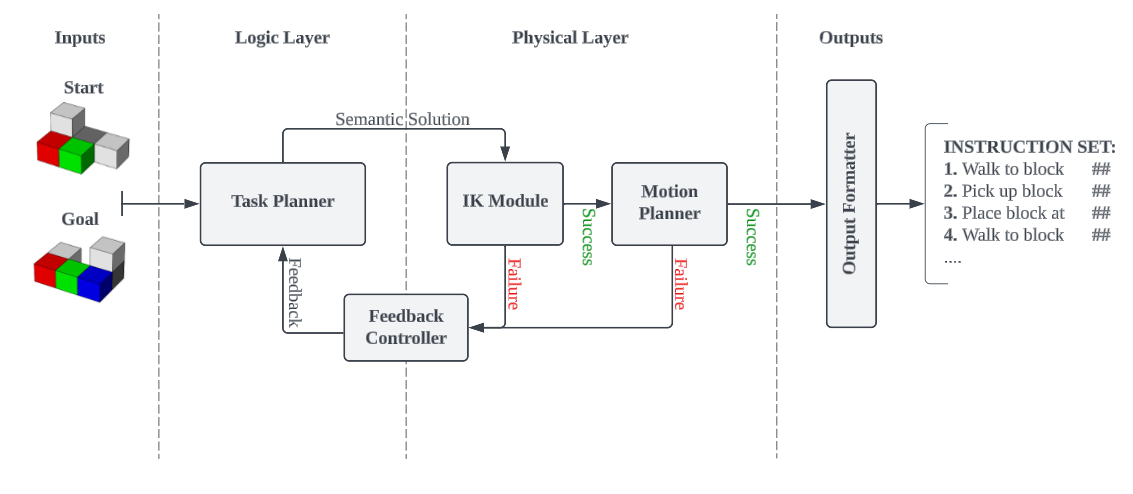
* Research on the state-of-the-art of modular reconfiguration and review of existing literature.
* Development of a detailed design to guide the implementation of the reconfiguration planning program.
* Description of the final implemented design.
* Analysis of the final implemented design and records of analysis results.
* Discussion into the results and their relevance to the field of study.
* Explanation of the project management approach and the evolution of the project plan throughout the duration of the project.
* Suggestions for further work to expand on completed work detailed in this report.

# Literature Review - around 3000 words

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

# 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.

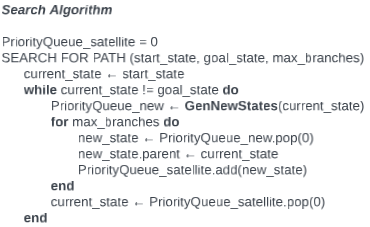
The overall system consists of a logic layer, physical layer, and feedback controller. Each of the two layers handle a separate role in the overall system allowing the discrete and continuous components of the solution search to be handled individually; While the feedback controller handles the integration 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 through the use of 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 a large number of modules, the generation of new states can generate 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 kept to be added to search graph. The remaining states are trimmed from the graph. 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.

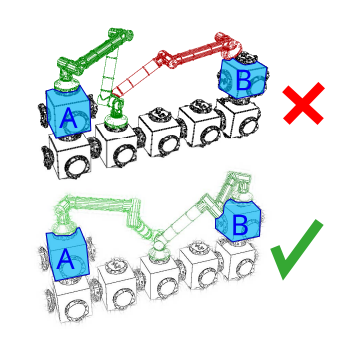
The motion planner upon receiving a transition, will find a path from the start and end pose positions which avoids the arm body and grabbed module colliding with other modules. If no solution is possible, the motion planner returns the transition to the logic layer with a motion planning failure.

### 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 particular manipulator arm 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

* Detail algorithm used for motion planning

### 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, however will be 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)

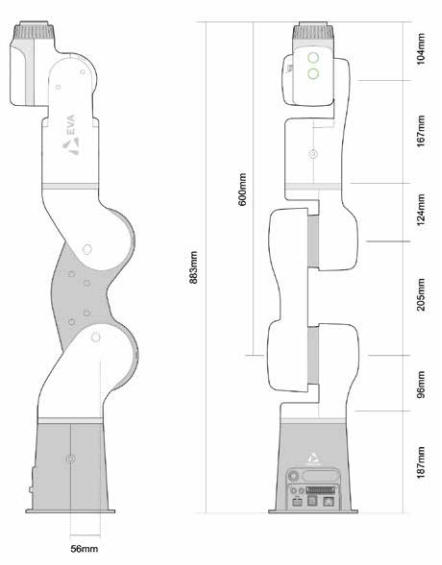
Processor: AMD Ryzen 7 3700X 8-Core (16 CPUs), running at ~3.6 GHz

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 [%].

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 [%].

**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

* Include usage of cv2 to analyse validation structure

#### Module Class

### Physical Layer

#### Inverse Kinematics Calculator

As stated in the design section [%], the implemented Inverse Kinematics Calculator uses an analytical solution. Initially, an analytical solution 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.

#### 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

* Implemented Software Structure
* Data Structures used.

## 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 initial data structure implemented simply used a position matrix to store module locations to allow easy and simple modification to internal logic and computations, with the intention of using modular design methodology to allow the easy replacement of the data structure to a more memory efficient structure later in the project that did not store unnecessary zero values. 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. 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 utilising the speed of the numpy module 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. Before modification, the insertion time complexity was O(6n \* state\_size^3) where n is the length of the queue, the multiplication of 6 was due to the 3 measurement value comparisons taking place during each state comparison, multiplied by the 2 \* state\_size^3 which is from the calculation of the measurement values in each state. This was reduced to O(log(6n \* state\_size^3)) in a worst-case scenario. The average case scenario however was only O(log(3n + 3\*state\_size^3)); the log coming from the binary search implementation and the reduction of 6n \* state\_size^3 to 3n + 3\*state\_size^3 being the fact often most measurement values have already been calculated on insertion of each state so only measurement values for the state that required insertion are calculated.

# Testing and Results

## Testing Method – (varying inputs to find edge cases)

* Varying module number

## Performance Metric – (failure rate, timing)

* Time
* Failure rate of logic layer
* Failure rate of physical layer (number of control loops)
* Failure rate of overall system (unable to find solution)

## Analysis of results

* 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.

## 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. The project goal could have been completed by firstly 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”. Then it would be possible to develop and analyse a range of feedback strategies without developing a mostly unused and complex simulation.

Despite the delay, the final product does match exactly 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

Refer to Appendix B section [%]

Rewrite risk assessment and put on it anything I had to do to face a risk such as getting sick and not being able to see mark (he marked work over email instead)

## Evolution of Project Plan

# 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 - Initial Report

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

# Appendix C – URDF file