Simulation and Optimisation of a Two Degree of Freedom, Planar, Parallel Manipulator

A Thesis
Submitted in Fulfilment
of the Requirements for the Degree
of
Master of Engineering
in
Electronic and Computer Systems Engineering
at
Victoria University of Wellington

by

Ben Haughey

2011

For my wife and our one on the way

Abstract

Development in pick-and-place robotic manipulators continues to grow as factory processes are streamlined. One configuration of these manipulators is the two degree of freedom, planar, parallel manipulator (2DOFPPM). A machine building company, RML Engineering Ltd., wishes to develop custom robotic manipulators that are optimised for individual pick-and-place applications. This thesis develops several tools to assist in the design process.

The 2DOFPPM's structure lends itself to fast and accurate translations in a single plane. However, the performance of the 2DOFPPM is highly dependent on its dimensions. The kinematics of the 2DOFPPM are explored and used to examine the reachable workspace of the manipulator. This method of analysis also gives insight into the relative speed and accuracy of the manipulator's end-effector in the workspace.

A simulation model of the 2DOFPPM has been developed in Matlab's® SimMechanics®. This allows the detailed analysis of the manipulator's dynamics. In order to provide meaningful input into the simulation model, a cubic spline trajectory planner is created. The algorithm uses an iterative approach of minimising the time between knots along the path, while ensuring the kinematic and dynamic limits of the motors and end-effector are abided by. The resulting trajectory can be considered near-minimum in terms of its cycle-time.

The dimensions of the 2DOFPPM have a large effect on the performance of the manipulator. Four major dimensions are analysed to see the effect each has on the cycle-time over a standardised path. The dimensions are the proximal and distal arms, spacing of the motors and the height of the manipulator above the workspace. The solution space of all feasible combinations of these dimensions is produced revealing cycle-times with a large degree of variation over the same path.

Several optimisation algorithms are applied to finding the manipulator configuration with the fastest cycle-time. A random restart hill-climber, stochastic hill-climber, simulated annealing and a genetic algorithm are developed. After each algorithm's parameters are tuned, the genetic algorithm is shown to outperform the other techniques.

Acknowledgments

I would like to thank Prof. Dale Carnegie for supervising my research and having someone to bounce ideas off. His support in helping me return to study has been fundamental in undertaking this thesis. To Dr. Will Browne for providing advice on optimisation algorithms, and Dr. Peter Donelan for helping me understand singularity analysis, much gratitude is given. I am most thankful for the help I have received from the technical and administrative staff in the ECS and Physics departments of Victoria University. To my fellow classmates, thanks for providing me with the much needed breaks from study.

This project was made possible through the partnership with RML Engineering Ltd. I would like to thank all the staff there who have assisted me during this project. I would especially like to thank Daryl Joyce, my manager, for providing this opportunity and for committing resources to R & D projects. Appreciation is given to Arron Porter for providing a mechanical design of the manipulator, and Varun Dennis for assisting with technical drawings.

This project was made financially viable with the assistance of TechNZ Capability Funding, for which I am most grateful.

I owe a great deal of gratitude to my parents for teaching me the value of education. They have supported me through my younger years and continue to encourage me to achieve my potential.

To my wife Sarah, I am truly thankful for her supporting me in my decision to return to study. The many hours she spent proof reading my work, has helped make this thesis what it is. Her constant love and encouragement has helped me throughout this project.

I would like to thank God for providing me with the many blessings I have received in my life. Though Him it is all possible.

Contents

List of Tables

List of Figures

Terminology

Herein, various terms are used to describe aspects of manipulators, trajectory planning and optimisation. A summary of their definitions are presented here.

Manipulators

Chain A linkage of independent bodies connected together by joints.

Closed Loop Refers to a mechanism's architecture where a set of bodies are connected in parallel so as to work alongside each other, rather than one after the other as is the case with serial connections. The bodies form a ring structure.

DOF Degree of Freedom. Describes the number of independent axis of motions a manipulator has. Individual degrees of freedom can be translational or rotational movements.

End-effector The final mechanism attached to the end of the manipulator to perform a task. It is also referred to as a *gripper* in pick-and-place applications.

Open Loop Refers to a mechanism's architecture where a set of bodies are connected together serially (i.e. one after the other) with no body re-connecting to a previous body as is the case with closed loop architectures.

Pick-and-Place Describes a task performed by a manipulator. This consists of picking up an object, performing a translation to a different position in space, and placing the object down again.

Revolute Joint A form of joint connecting two bodies together. The motion offered by this joint is revolving around a single axis. This is commonly achieved through electric motors.

Singularity Occurs when the Jacobian matrix describing the motion of the manipulator becomes singular. In physical terms of the manipulators' bodies, this commonly occurs when two or more bodies in a chain become aligned leading to a loss of control of one of the arms. This is shown in Figure 0.1 where *Linkage*_1 and *Linkage*_2 become aligned resulting in the loss of independent control of *Linkage*_2.

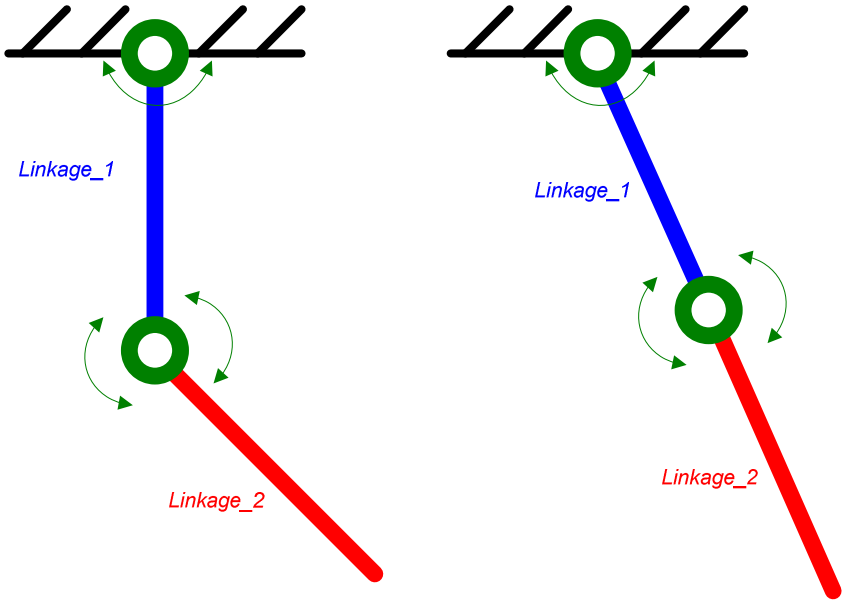

Figure 0.1 Occurance of a singularity between two body linkages of a chain.

Workspace Describes the area reachable by the manipulator's end-effector.

Trajectory Planning

Continuity A measure of how smooth a trajectory is. A continuity of C^1 is continuous in the domain of the first order derivative (velocity). A continuity of C^2 has a continuous second order derivative profile (acceleration).

Database

Primary Key A field, or combination of fields, in a database table that uniquely identifies each record.

Foreign Key A field in a database table that matches the primary key column of another table. The foreign key can be used to cross-reference tables.

Optimisation

Cost Function A measure of how poorly a particular solution performs. It evaluates the negative performance in comparison to the *fitness function* which evaluates the performance from a positive perspective.

Fitness A measure of how good a particular solution performs.

Fitness Function An equation used to evaluate the fitness of a solution. It evaluates the positive performance in comparison to the *cost function* which evaluates the performance from a negative perspective.

Search Space Also referred to as *solution space*. This describes the set of all possible solutions.

1 Introduction

1.1 Motivation

The drive to improve factory efficiencies by increasing throughput has led to increased development in robotic product manipulators. The manipulators are often referred to as pick-and-place robots due to the task that they perform. Traditionally, robotic manipulators have been designed in a generic way which allows the same manipulator to be programmed for multiple tasks on different production lines. However, these robotic manipulators may not be the optimal design for any given task. Therefore, using robotic manipulators to maximise production output, it is imperative that a customised manipulator is developed. The design of the pick-and-place manipulator will allow the fastest possible product handling cycle for the production line.

A New Zealand based company, RML Engineering Ltd., would like to explore the opportunities to develop a parallel robot with two degrees of freedom (DOF). More specifically, they would like to have the tools and knowledge to create custom manipulators that are optimised for a given task. While 2-DOF manipulators are commercially available [1], these are highly priced and are not optimised for any particular application. Consequently they do not offer the high-end performance RML Engineering Ltd. seek.

Many pick-and-place tasks do not require the complexity of traditional 6-axis robots and can instead be achieved using a simplified manipulator operating in a two dimensional plane. As well as the associated cost saving of a simpler design, a manipulator that operates in a single plane can be designed to transfer greater loads at increased speeds, as is discussed in Section 1.3.2.2.

RML Engineering Ltd. is to design a standardised two degree of freedom, planar, parallel manipulator (2DOFPPM). An initial concept drawing is shown in Figure 1.1. This design, while standard, can be scaled and individual dimensions modified to suit a specific application. The ability to modify the dimensions for optimal performance is a feature that will be presented in this thesis.

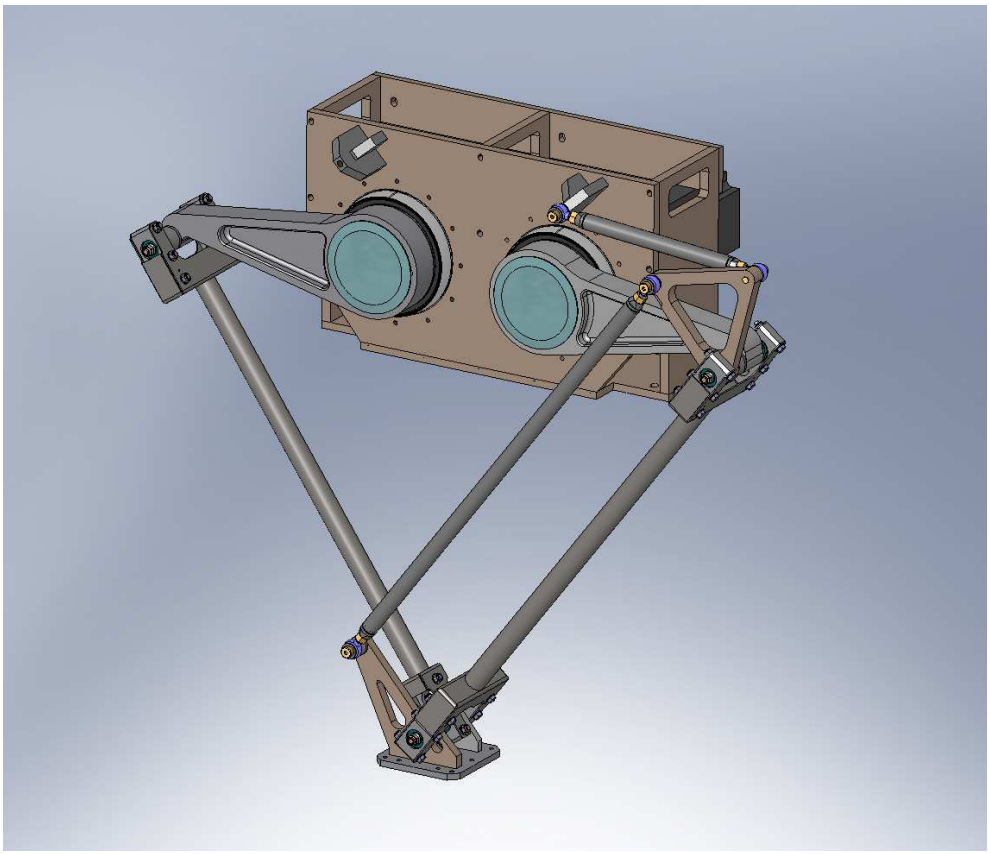

Figure 1.1 A SolidWorks™ rendering of RML Engineering Ltd.'s design of the 2DOFPPM

1.2 Objectives

There exist two main objectives for this project. These objectives, once achieved, will allow RML Engineering Ltd. to analyse and optimise their 2DOFPPM design for specific applications. The objectives are:

- Produce a simulation model of the mechanical system. This will allow analysis of the manipulator's dynamics.
- Implement a method to optimise the manipulator's mechanical dimensions for achieving a near minimum cycle-time for a particular task.

The simulation model will be used to evaluate the dynamic behaviour of the manipulator under applied motor torques and the external force due to gravity. More specifically, the torques and forces acting on the joints and bodies of the manipulator will be examined. The velocity and acceleration of the manipulator's gripper/end-effector are also of interest as these can affect the design decisions of both the manipulator and the gripper which holds the object being manipulated.

When optimising a manipulator there are many components that could be considered as parameters for tuning. This study will only consider the major components that have the greatest effect on the cycle-time for performing a task. To validate the optimisation method, a number of techniques will be considered. The performances of all the implemented methods will be compared to find the algorithm best suited to optimising the manipulators dimensions.

In order to achieve these main objectives, a third objective must also be met:

- Implement a trajectory planning methodology that seeks to minimise the time taken to execute a given task.

The trajectory planning method will be used to generate input commands for the motors within the simulation. It will also be used to compare different manipulator configurations during the optimisation process. The trajectories generated by the planner must represent the fastest possible path achievable by a given 2DOFPPM configuration, so that comparisons can be made between individual configurations based on the relative cycle-times of the trajectories.

This project's task is to achieve the objectives stated above. The outcomes of achieving these objectives will provide RML Engineering Ltd. with software tools to assist in the development of bespoke manipulators.

1.3 Robotic Overview

1.3.1 Comparison of Pick-and-Place Architectures

In the quest for faster and more efficient production lines, there have been a number of robotic technologies developed for product handling and manipulation. This area of robotics is often referred to as *pick-and-place* as the robot's sole purpose is to move objects from one location to another, with the possibility of re-orientation at the same time. Pick-and-place robots can be organised into two categories depending on how their manipulating arms are configured. These categories are *serial* or *parallel*.

Of the two types mentioned, *serial* is the most commonly implemented. A *serial* robot is configured such that each arm/axis is linked to another arm/axis in the form of a chain (for example, axis 4 is mounted to axis 3 which in turn is mounted to axis 2, which is connected to axis 1, with axis 1 being located to the base of the robot). This linked structure allows a great degree of dexterity as it is based around the biological structure of a human arm. An example of a serial robot is shown in Figure 1.2.

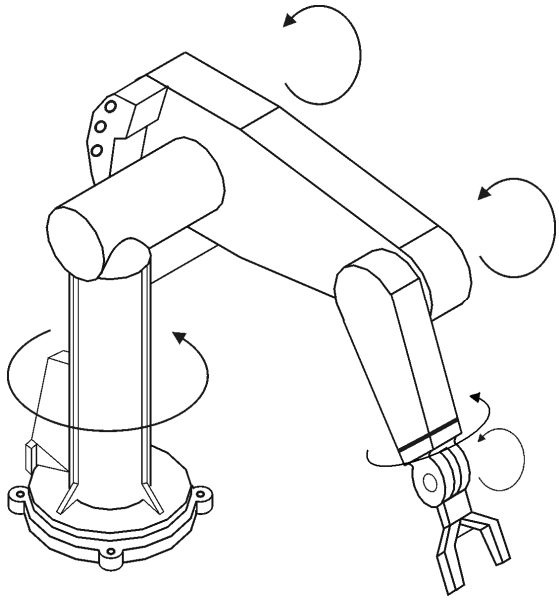

Figure 1.2 PUMA robot, an example of a serial manipulator[1]

The dexterity of *serial* robots makes them a popular choice in production applications where components have to be assembled in ways that are very difficult for other robots to reach. They are also very versatile and can be easily reprogrammed to perform another task (for example, once a production run is completed for widget A, the same robot can be used for a completely different product, widget B). However, *serial* robots do have several draw backs. Inaccuracies are accumulated, so that a small error in axis 1 is multiplied such that the error at the end-effector/TCP (tool centre point) is much greater. Due to the end axes' actuators being carried by the earlier axes, the links have to be reinforced to cope with the additional load. This adds additional inertia to the system, thus reducing the speed of the robot. It is now expected that when speed is the most important issue, an alternative to a *serial* system will be used.

Cartesian robots, also known as *gantry* robots, are a variation of the *serial* configuration. *Cartesian* robots are a form of *linear* robot, meaning that the two or three principal axes are controlled linearly (that is, they move in a straight line). The axes are also at right angles to one another. An example of a *Cartesian* robot is shown in Figure 1.3. Due to the linear nature of the axes, *Cartesian* robots are well suited to transporting heavy loads. Recent advancements in linear actuators have also meant that *Cartesian* systems are now both fast and highly accurate. The main disadvantage of *Cartesian* robots is

[1] Image sourced from Herman Bruyninckx contribution to The Robotics WEBook, www.roble.info. Copyright is held by the authors, who release the text and the figures under the open content WEBook license.

that the working envelope of the system is smaller than the robot itself (that is, the gantry frame limits the work area).

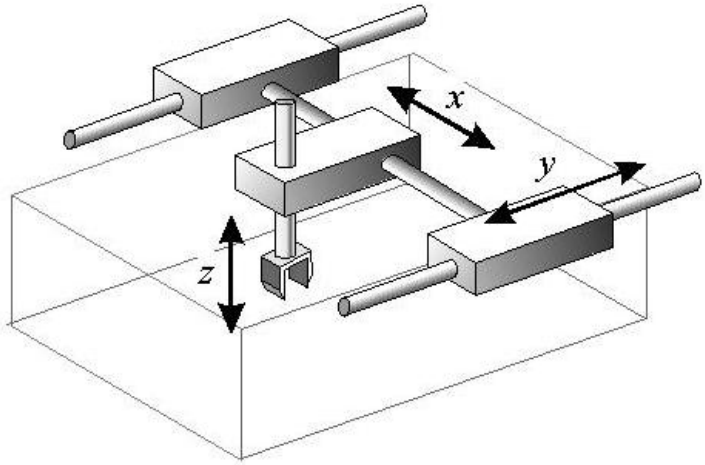

Figure 1.3 Direction of actuation for a Cartesian robot[2]

The alternative to the *serial* configurations presented above is the *parallel* manipulator. *Parallel* robots are closed loop mechanisms that have an end-effector supported by at least two chains, controlled by separate actuators [2]. The most popular of these is the Delta robot, which comes in both three and four axis variations. The Delta robot, shown in Figure 1.4, was first commercialised in the 1990s as the Flexpicker™ by ABB®. Originally under strong patent protection, the system has been replicated by many other vendors [3]. The Delta robot is now the default choice for high speed pick-and-place applications involving objects less than 1 kg in mass.

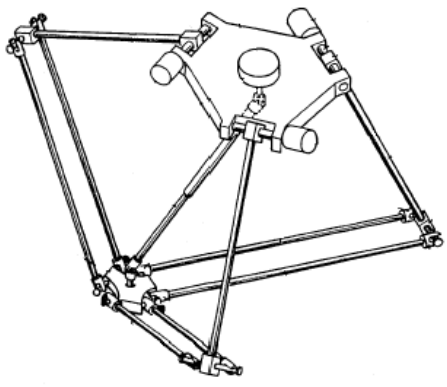

Figure 1.4 Delta Robot - a popular form of parallel manipulator for high speed pick-and-place applications

[2] Image obtained from http://www.pe.tut.fi/akp/images/cartesian on 12/02/10.

Parallel robots offer several advantages over *serial* robots, including greater rigidity due to the closed loop architecture [4], higher payload/weight ratio and reduced inertia as the actuators can be mounted at a fixed base instead of on the arms [5]. They also feature higher precision, due to positioning errors being averaged rather than compounded as they are in serial manipulators [6]. However, the closed loop structure requires more complex mathematical analysis, as well as the disadvantages of more singularities, lack of dexterity and a smaller workspace [7].

The advantages of parallel manipulators make them ideal for high speed pick-and-place movements where the path followed by the end-effector is free of any obstacles. As this is the case for many industrial applications, the parallel architecture shall be considered in this thesis. This will focus on a specific configuration of parallel manipulators, the 2DOFPPM.

1.3.2 The 2DOFPPM

1.3.2.1 Construction

At an elementary level the 2DOFPPM is the simplest useful form a parallel manipulator can be. The manipulator has a single closed loop chain of four arms and a base platform. The manipulator's mechanical components are presented in Figure 1.5. Two motors, *Motor_A* and *Motor_B*, are mounted to the base platform and actuate the proximal arms, *Prox_A* and *Prox_B*. Each proximal arm is then connected to a passive (not actuated) distal arm. These distal arms are labelled *Dist_A* and *Dist_B* in Figure 1.5. The other ends of the distal arms are passively connected together to form a closed loop. A *gripper*, or *end-effector*, is mounted to the point where the two distal points are joined.

In addition to the major components, there is a set of mechanical components to ensure the gripper remains horizontal, that is, parallel to the base platform. This group of components is termed the *stabiliser arm* and is shown in blue in Figure 1.5. The *stabiliser arm* is not actuated and simply shadows the *Prox_B* and *Dist_B* control arms. *Prox_Stab_B* is the same length as *Prox_B* and *Dist_Stab_B* is the same length as *Dist_B*. Several components, labelled *Prox_Crank* and *Dist_Crank* are used to hold the two stabiliser linkages, *Prox_Stab_B* and *Dist_Stab_B,* at an offset, and parallel to *Prox_B* and *Dist_B* respectively. As the *stabiliser arm* is only to prevent the gripper from rotating about its mounting point and does not bear the load, it may be made of lighter or reduced material.

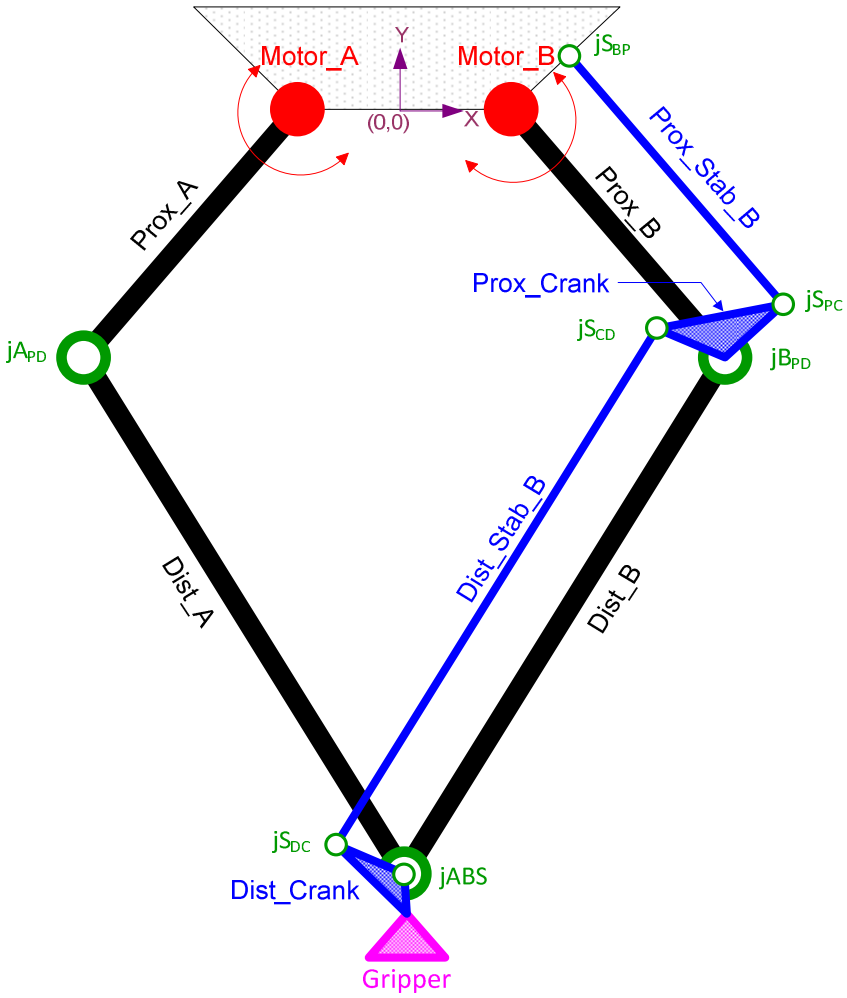

Figure 1.5 2DOFPPM Construction

A coordinate convention is defined with the +Y axis being vertical upwards and the +X axis being horizontal towards the stabilising arm. The Z axis is redundant as the manipulator operates in a single plane. Therefore all the components have a constant position in the Z frame of reference. The origin (0,0,0) is defined as the middle of the base platform. This convention is used throughout this thesis.

A revolute joint exists at each interface between the base and the proximal arms, the proximal arms and the distal arms, and the two distal arms. Each joint rotates about the Z axis. For this project, the joints jA_{PD}, jB_{PD}, *jABS* are assumed to be passive joints, consisting of an ideal bearing with zero resistance. Similarly the joints of the stabilising components (jS_{BP}, jS_{PC}, jS_{CD}, jS_{DC}) are assumed to affect zero rolling resistance load on the system.

1.3.2.2 Features

The 2DOFPPM benefits from the same features as more complex parallel manipulators. The inertia of the manipulator is relatively small when compared to serial equivalents. This is due to the two motors being mounted at the base, leaving only the arms, gripper and the load being manipulated, moving in space. This permits fast rotation of the motors, which, when combined with the lever system formed by the arms, generates very high acceleration and speed at the end-effector. Experiments have seen the 2DOFPPM's end-effector velocity reaching 8.5 ms^{-1} with accelerations exceeding 230 ms^{-2} with a 1 kg load [8].

With the end-effector only two linkages away from the fixed base, the positional errors are much less than those of the common six DOF serial manipulators. Errors are formed by averaging the inaccuracy of each of the two chains, where the inaccuracy of each chain is due to positional errors in the actuators, slop in the passive joint formed by the proximal and distal arms, and flex in the arms. This compares favourably to serial manipulators with greater DOF. An increase in the number of DOF introduces extra errors which are accumulated in the serial architecture, rather than averaged as they are in parallel systems.

The stiffness of a parallel structure is further improved with the 2DOFPPM acting in a single plane. The forces being transmitted run along the same plane of actuation. This means that there is no shearing force along the Z axis, which has traditionally been a limiting factor in the load bearing capability of parallel manipulators. Working in a single plane, revolute joints can be used, which can bear higher loads than the spherical joints found in other parallel manipulators like the Delta. This means the 2DOFPPM can fundamentally carry heavier loads.

1.4 Thesis Structure

This chapter describes the motivation for the project. Several manipulator configurations are presented and their strengths and weaknesses discussed. A detailed overview of the 2DOFPPM is included. The research objectives are presented and an outline of the thesis is provided below.

Chapter 2 offers background information on the subjects of industrial manipulators, trajectory planning and manipulator optimisation. The existing research related to this thesis is presented and discussed. Based on the current state of the art, justifications are given for the approaches used in this thesis.

Chapter 3 covers the analysis of the mechanical system via simulation methods. The workspace of the 2DOFPPM is analysed after considering the forward and inverse kinematic model. The manipulator is

simulated using a simulation engine (SimMechanics®) and the model's output analysed. This simulation model provides insight into the manipulator's dynamics and is used again in Chapter 5 to review any optimised manipulator configurations.

Chapter 4 presents a trajectory planning method that seeks to find a time-minimum trajectory for the manipulator to traverse a given path. This trajectory planning method is used to provide input into the SimMechanics® simulation of Chapter 3 as well as being used to generate optimisation results in Chapters 5 and 6.

Chapter 5 considers altering certain dimensions of the manipulator to achieve a faster path cycle-time. A method of generating and storing large amounts of simulation data is presented. The search space of possible manipulator configurations is explored. An introduction into optimisation techniques is given along with a justification of the techniques implemented in this thesis.

Chapter 6 presents the full implementation of the optimisation algorithms. The performance of the different methodologies is statistically evaluated and one method is chosen as being the most suitable for the task. An optimised manipulator configuration is then viewed in the SimMechanics® simulation of Chapter 3 to validate the configuration's performance.

Chapter 7 concludes the research presented in this thesis and makes recommendations for future work.

2 Literature Review

This chapter provides an overview of existing research relevant to this project. It is separated into three sections. The first section presents the approaches of other researchers in analysing the workspace of parallel manipulators and specifically, the 2DOFPPM. The second section discusses existing trajectory planning methods, outlining the strengths and weaknesses of each of them. The third section examines current methodologies in optimising manipulators for improved performance.

2.1 Parallel Manipulators and Their Workspace Analysis

The first parallel mechanism was developed in 1956 by Gough [9] as a universal tyre testing machine. This was followed a few years later by the more famous Stewart Platform [10], created as the base for a flight simulator. These mechanisms, shown in Figure 2.1, had six degrees of freedom (DOF). The first pick-and-place manipulator developed with a parallel architecture was the three DOF Delta robot [11]. Clavel, the inventor of the Delta mechanism, questioned why large and heavy serial manipulators were being used to perform lightweight pick-and-place operations. His research resulted in a manipulator with base mounted actuators and low-mass arms that could easily outperform the serial counterparts when moving light objects. Clavel's design was strongly patented and commercialised by ABB® as the FlexPicker™, shown in Figure 2.2. Since the expiration of the FlexPicker™ patent, many three and four DOF variations of the Delta manipulator have been commercialised.

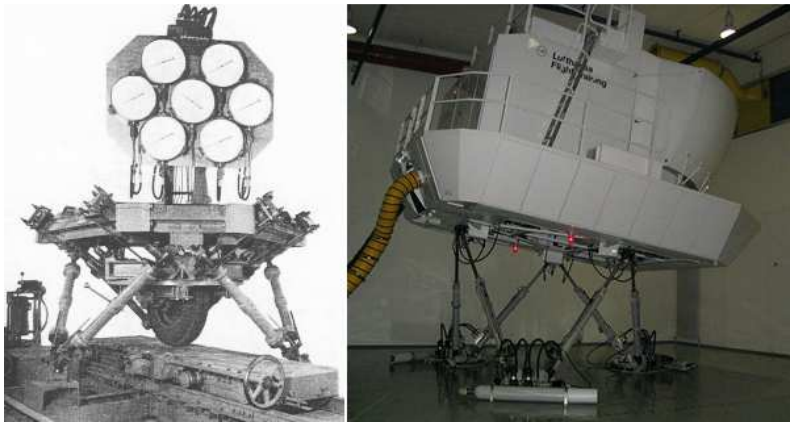

Figure 2.1 Early examples of parallel manipulators: Gough's Universal Tyre Tester (left) and a flight simulator using a Stewart Platform (right)[3].

[3] Images obtained from http://commons.wikimedia.org on 22/12/10 under the Creative Commons Licence.

Figure 2.2 ABB® Flexpicker™ - the first commercialised parallel pick-and-place manipulator[4].

While much research has been undertaken into the Delta and its four DOF variation, there have been a relatively small number of researchers who have investigated the simpler two DOF variant. Although the three and four DOF variants have greater versatility by being able to operate in all three dimensions, there are certain applications where a two DOF planar manipulator is suitable and in some cases advantageous. Wherever a product needs to be moved in a single plane (for example, between two conveyors), a two DOF manipulator will suffice. As explained in Section 1.3.2.2, by moving in a single plane, heavier loads can be carried thereby giving the 2DOFPPM a greater advantage over three dimensional manipulators. In recent years the 2DOFPPM has grown in popularity and has been studied by Huang [12-15], Gao [16][17], Baradat [8], Li [18][19], Cervantes-Sánchez [20], Stan [21] and others.

Piras et al. [22] shows through finite element analysis (FEA) of a 2DOFPPM, the effects of vibration on accuracy are minimal, but that they are also highly dependent on the precise configuration. Li [19] went further to say that the analysis of flexible linkages in the 2-DOF parallel robot is of significant importance when high speed and high precision are required. Li [18] also found that the tubular structure of the arms, specifically the outer diameter, was of great importance to the system's rigidity. This indicates that for the most part, the system simulation can be limited to rigid body analysis, with dynamic vibration analysis being undertaken near the end of the design cycle to further tune the manipulator for highest performance.

[4] Image obtained from www.abb.com on 22/12/10.

Several modifications have been made to the basic structure of the 2DOFPPM to improve one or another performance aspect. Baradat et al. [8] presents a configuration with improved stiffness in the plane perpendicular to the plane of motion. This is achieved by introducing two redundant stabilising arms, mechanically coupled together in the perpendicular plane, to counteract any vibration or movement in the direction normal to the plane of motion. The additional arms however, greatly increase the manipulator's footprint while also adding extra inertia to the system. Hu et al. [23] present a 2DOFPPM with increased stiffness via the introduction of several passive chains connecting the base to the end-effector via a translational sliding mechanism. Unlike Baradat's design, this does not add to the size of the mechanism's footprint, however it does increase the inertia of the system.

Huang et al. [13] extends the studied manipulator to a third degree of freedom, by incorporating the existing mechanism onto a translational actuator in the plane perpendicular to the typical plane of operation. This side shifting mechanism allows the manipulator to be used in all three dimensions of space, while maintaining the primary benefits of the parallel architecture.

When defining the workspace of the 2DOFPPM, it is common to refer to it as being the area reachable by the end-effector without passing through any singularities of the manipulator [16][20]. However Huang et al. [13][14] defines the workspace of the 2DOFPPM as being of rectangular shape within the actual area reachable by the end-effector. This reduction in workspace, while theoretical, provides more simplistic parameters to evaluate the manipulator's performance using an index method. A number of researchers use the idea of a conditioning index, originally developed by Gosselin and Angeles [24], to give an indication of global performance aspects of the manipulator within the workspace [13][14] [17][25]. The conditioning index uses the Jacobian matrix of the system to determine the behaviour of the manipulator in the workspace. More specifically, the performance index provides a single value indicating the accuracy and speed of the end-effector within the workspace. This allows comparisons of the global performance within workspaces formed by different manipulators but does not allow direct comparison of manipulators over a given path. The index also gives an indication into how susceptible the manipulator is to reaching a singularity pose. This can be useful in determining the reachable workspace.

The stabilising arms are used to keep the end-effector parallel with the base. Huang et al. [15] identified a potential risk in the stabilising arms over-constraining the manipulator if manufacturing tolerances are too tight. If there is no clearance in the joints on the stabiliser arms, the unavoidable imperfections in the manufacturing process will result in stabiliser arms locking the entire manipulator into a fixed

position. Huang et al. has provided a method to allocate tolerances to the manufactured manipulator components.

2.2 Trajectory Planning

The control process of a manipulator is typically separated into levels of abstraction. The lowest level consists of a closed-loop control system tracking a given trajectory. The next level involves planning the trajectory of the actuators. Above that is the generation of the path for which the end-effector is to travel through. Higher levels can exist that are often related to the tasks being performed by the manipulator. This project is not concerned with the closed-loop control tracking, nor the high task level control, but rather involves taking a desired path for the end-effector and transforming it into time dependent trajectories for each of the manipulator's actuators. This process is commonly referred to as trajectory planning.

There exists a wide range of trajectory planning methodologies used to control manipulators. They vary in computation complexity, path accuracy, trajectory smoothness and path cycle-time. This section discusses the key methods used in industry and for research, highlighting their performance and applications.

A manipulator's productivity can be increased by executing a task's path in minimum time or by minimising down time. The trajectory planning technique often affects both of these. The cycle-time of trajectories can vary based on the method used to generate them, but also some methods of generation can take excessive time such that the manipulator must wait for computation to complete before executing the path. Such methods are referred to as being *off-line* algorithms, which are less favourable, in terms of computational intensity, than *on-line* algorithms [26].

Another consideration when choosing trajectory planning techniques is how smooth the resulting trajectory is. Smoothness is normally considered in terms of the joint actuators, rather than the smoothness of the path travelled by the end-effector [27]. This is because a trajectory which is not smooth for the actuators will create vibration in the manipulator and can result in poor path tracking. A popular method for rating the smoothness of a trajectory is to refer to its level of continuity. A trajectory which has C^1 continuity means that the velocity, or 1^{st} order derivate of the path with respect to time, is continuous over the entire path. Similarly, a path with C^2 continuity has continuous acceleration over the whole path.

There are two options in choosing a frame of reference for planning a trajectory. The first option involves converting the Cartesian path into joint paths by inverse kinematics and then controlling the manipulator at the joint level. Alternatively, the joint limits (velocity, acceleration, torque, etc.) are converted into Cartesian bounds and then the trajectory is planned at the end-effector level. Luh and Lin [28] sought to minimise a trajectory using the latter method but found the conversion of the joint limits too difficult due to the non-linear and highly coupled manipulator dynamics. Therefore, the first method of converting the Cartesian path into joint space is favoured.

A popular method for generating smooth trajectories is to use polynomial functions. In this process, Cartesian defined knots are converted to *N* sets of single dimension joint positions, where *N* is the number of knots. A polynomial is then fitted to pass through each of the knots in joint space, thus forming the trajectory for each actuator. Zhihong [29] and Spong et al. [30] show that the polynomial can either be a single high-order polynomial of order *N*, or be formed as piecewise segments. The single high-order polynomial provides a high level of smoothness but can become computationally intractable as the number of knots, and consequently the order of the polynomial, increases [29]. A more common approach, and one that is implemented in industrial controllers [30], is to define individual polynomials of a lower order between each knot. To ensure continuity between individual polynomial segments, constraints are applied forcing the velocity, acceleration and/or jerk profiles to be continuous over the entire path. At the simplest level, 3rd order cubic polynomials are used. These can provide a trajectory with continuous velocity and acceleration, but will likely have a discontinuous jerk profile. An example of how a cubic polynomial is used to describe a joint trajectory is given in Equation (2.1), where the angular position, θ, of joint i is described by a third order polynomial of time t, with coefficients a_0 to a_3.

$$\theta_i(t) = a_0^{(i)} + a_1^{(i)}(t) + a_2^{(i)}(t)^2 + a_3^{(i)}(t)^3 \tag{2.1}$$

In the 1970s Paul [31] and Finkel [32] investigated using cubic polynomials to interpolate knots for manipulator trajectory planning. These required solving 3(*N*-1) or 4(*N*-1) systems of linear equations where N is the number of knots. These methods proved to be smooth and have small overshoot of joint displacement. In 1983, Lin et al. [33] popularised cubic polynomial use for manipulator trajectory planning when they developed a method to minimise the time between knots. They used Nelder and Mead's [34] flexible polyhedron search to iteratively alter the path-time at each knot until a near minimum cycle-time is found which satisfies the constraints of the manipulator. A trajectory planner of this type was implemented specifically for the 2DOFPPM by Hu et al. [23].

Chand and Doty [35] developed an *on-line* cubic spline trajectory planning methodology. By only considering a limited number of knots immediately ahead of the current position, and not the entire path, the trajectory could be computed quickly and alterations to the path made on the fly.

Boryga and Grabós [36] presented a study of trajectory planning for a serial manipulator using piecewise 5^{th}, 7^{th} and 9^{th} order polynomials. They found the 7^{th} order polynomial fitment to be optimal for avoiding the limits of the manipulator over a given path and cycle-time. This however, is highly dependent on the configuration of the manipulator.

Thompson and Patel [37] used an alternative method of fitting B-splines to control points, or knots, along the path. Unlike cubic polynomials, B-splines do not pass through the knots but instead are 'pulled' towards them. Formulating the splines is computationally easy and can be computed fast enough to allow the trajectory planner to be executed *on-line*. B-splines provide a smooth trajectory with continuous position, velocity and acceleration that is easily followed by real-world joint actuators. Thompson and Patel's method allowed velocity and acceleration constraints to be set at each knot along the path. Wang and Horng [38] sought to minimise the cycle-time of a B-spline trajectory controller by using a recursive flexible polyhedron search method to alter the path time between knots. Despite the research into B-splines, they have failed to be implemented into industrial controllers. This is largely because the trajectory generated does not pass through the control points, hence lacks the accuracy levels needed in industry.

In general, the trajectory planning methods previously discussed only consider the kinematic limitations of the manipulator, that is, the bounds on velocity, acceleration and jerk. However, in reality the manipulator's actuators are also limited by dynamic constraints, such as torque and torque rate. To truly maximise the manipulator's capabilities and find a time-minimum trajectory, the trajectory planner must take into account a dynamic model of the manipulator. Kahn and Roth [39] first attempted to produce a trajectory planner that took into account the dynamic model. This consisted of optimising an unconstrained path subject to torque limitations using Pontryagin's principle [40]. However, the result was computationally intractable, and the dynamic model had to be linearised for an optimal trajectory to be found. This linearization of the dynamics results in significant errors, rendering the method unsatisfactory [41].

Geering et al. [42] showed that for various manipulator configurations the time-minimum trajectory subject to torque constraints must be either a bang-bang or bang-singular-bang trajectory. A bang-bang trajectory is where the actuator is exerting maximum acceleration up until a switching point where it

applies maximum deceleration. A bang-singular-bang trajectory is similar except that a maximum velocity is reached before the switching point and the actuator cannot continue to accelerate [29]. Figure 2.3 shows the velocity profiles of both a bang-bang and bang-singular-bang trajectory. Chen and Desrochers [43] proved that for the trajectory to be traversed in minimum time, at least one of the actuators must be in torque saturation along the entire trajectory.

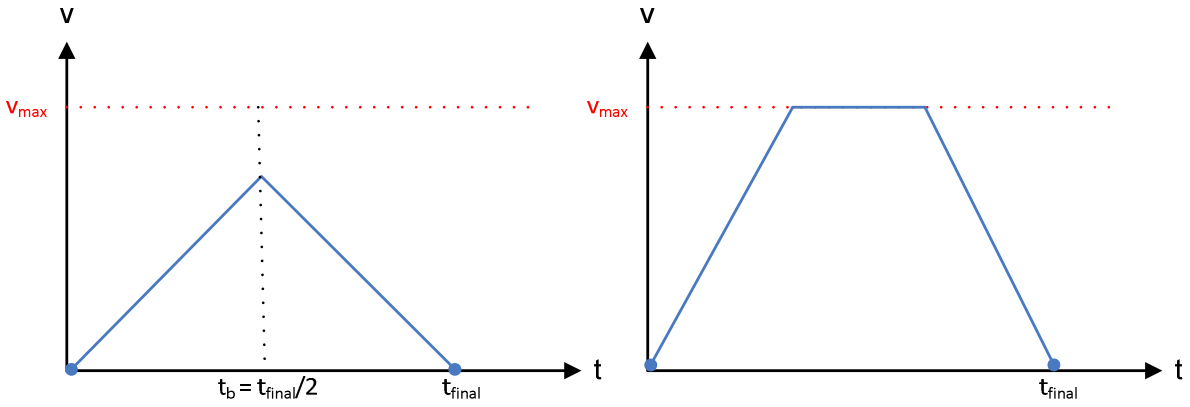

Figure 2.3 Velocity-Time profiles of bang-bang (left) and bang-singular-bang trajectory (right).

Bang-bang trajectory planning algorithms, when considered in isolation from the rest of the control system, appear to be optimal. However, when tracking a purely time-optimal trajectory with a simple controller, actuator saturation occurs which causes poor tracking, vibrations in the machine and increased machine wear [44-46]. In order not to exceed the actual capabilities of the manipulator, the actuator bounds must be chosen conservatively, possibly forcing the manipulator to be underutilised [47].

Bobrow et al. [48] and Shin and McKay [49] independently developed a trajectory planning method that would allow both the kinematic and dynamic constraints to be taken into account. The method first determined a function that describes the maximum velocity along the path, dependant on the position along the path. Knowing this function, switching points are calculated which switch the actuator from maximum acceleration to maximum deceleration at points along the path so as to minimise the overall cycle-time. Other variations have been made to this method seeking to improve the performance of the algorithm [46][50-52], or make it suitable for *on-line* calculation [26][53]. Huang et al. [54] has implemented a variation of Bobrow et al. and Shin and McKay's trajectory planner, specific to the 2DOFPPM.

Due to the importance in minimising the cycle-time of manipulators for industrial applications, a large amount of research has been undertaken into trajectory planning techniques. This area of research continues to be active as even the smallest increase in performance can equate to large financial benefit in high production industries. Within this project, the exact trajectory planning technique is not important. The trajectory planner is used to compare the performance of different 2DOFPPM configurations. Therefore, the only consideration is that the methodology chosen must produce near-minimum cycle-times that give a good indication into the performance of the 2DOFPPM configuration relative to alternative 2DOFPPM configurations. An in-depth discussion of the trajectory planner is presented later in Chapter 4.

2.3 Manipulator Optimisation

Improving the performance of manipulators in industry can be achieved by improved trajectory planners or through developing superior manipulators. While there has been an abundant amount of research into trajectory planners, the concept of optimising the manipulator itself has seen somewhat less attention. This may be due to many manipulators needing to remain generic in order to serve multiple applications. However, for many pick-and-place applications, the task that is performed remains the same for the life of the manipulator. It is in these situations, where a custom manipulator could be developed that would outperform a generic equivalent. This section presents existing research related to optimising a manipulator's dimensions.

The importance of customising manipulators is further backed up by Merlet [55], who argues it is absolutely necessary in order to ensure the highest performance is obtainable from the mechanism. Merlet states that this is especially true of parallel manipulators due to a high degree of coupling that is intrinsic in parallel architectures.

Zhuang et al. [56] argued that cost functions related to manipulators are highly non-linear, and as such often have many local minima. This means that gradient based optimisation methods like hill-climbers are inadequate. Zhuang et al. instead used an optimisation technique known as simulated annealing to find the manipulator configuration that gave the least positional error. With correctly chosen optimisation parameters, simulated annealing can overcome local minima to find the global minimum [57]. The style of manipulator used for this study was a serial robot.

Similarly, Stan et al. [58][59] uses a simulated annealing process to maximise the workspace of the 2DOFPPM. This process, while finding the relative dimensions that give the maximum workspace, does

not look at other performance aspects such as path cycle-time or end-effector accuracy. This is useful in evaluating the versatility of a manipulator, but does little for the pick-and-place performance indicators of speed and accuracy. Stan et al. [21] carries on from his earlier work and seeks to maximise transmission quality, manipulability and stiffness indexes of the 2DOFPPM workspace. This is achieved through a genetic algorithm.

Cochron and Bidaud [60] developed a genetic algorithm to optimise a serial manipulator for a specific task. The task consisted of an end-effector path with obstacles to be avoided. Six criteria were optimised including maximising the reachability, proximity to obstacles and dexterity, and minimising the complexity/inertia of the linkages, as well as the linear and angular distances travelled.

Feddema [61] found that the placement of a manipulator within the work area can alter the path cycle-time by up to 25 %. Using both six and two DOF serial manipulators as examples, a gradient descent method was used to find the optimal position to place the manipulator. This proved to be several orders of magnitude faster than locating the position by exhaustive search. In Feddema's method, the gradient descent was preceded by a coarse exhaustive search of the solution space, thus providing a good seeding value for the gradient descent to start from. This also largely avoided the effects of becoming stuck in local minima.

Pashkevich and Pashkevich [62] took a different approach to this same problem by seeking to find a Pareto-optimal set of solutions based on a multi-objective criterion. A genetic algorithm was used to find a set of solutions that were optimal in at least one of the objectives.

Mitsi et al. [63] also recognised the importance of where the manipulator's base is positioned in the work area. By using a specially developed genetic algorithm combined with a hill-climbing routine, a system was developed that minimised the travel distances and maximised the dexterity of the joints of an industrial six DOF serial manipulator. This custom method was shown to perform better than a genetic algorithm alone.

While there has been some research around optimising manipulators, there has not been a comparative study of optimising algorithms as applied to finding the best manipulator dimensions to achieve the fastest possible path cycle-time. This comparison is presented in this thesis for optimising the 2DOFPPM.

2.4 Summary

In summary, the 2DOFPPM is a simplified version of the very popular Delta robot. By only operating in a single plane, there is an increase in stiffness and load bearing ability but at the cost of some versatility. However, pick-and-place movements are often fixed for the lifetime of the manipulator, thus in such a scenario, a 2DOFPPM will be sufficiently effective. A number of subtle improvements to the general structure of the manipulator have been presented here. While not attracting as much research as the delta robot, several researchers have sought to optimise the workspace of the 2DOFPPM through use of a condition index that gives a value to the performance of the manipulator within the workspace.

A selection of trajectory planning methodologies has been discussed. The trend of researchers in this area is to plan the trajectory in joint space as this allows integration of the manipulator's dynamic constraints. Trajectory planners can be grouped as *on-line* or *off-line* depending on their computational complexity. *On-line* methods are preferred as they allow the trajectory to be computed on the fly while the manipulator is moving, thus increasing up-time of the machine. Using piecewise polynomials to define the trajectory between knots is a common technique that allows the kinematic limitations of the manipulator to be taken into account. More complex methods also exist that minimise the cycle-time within the dynamic constraints of the manipulator. For this project, the exact trajectory planning method is not critical provided it achieves near-minimum cycle-times. It may be an *off-line* or *on-line* system.

This project seeks to minimise the cycle-time of a 2DOFPPM by finding the optimal dimensions of the manipulator. Several researchers have looked at similar problems to this and tested various optimising methodologies. This project compares the performance of different optimising methodologies as they apply to optimising the 2DOFPPM for a specific task. An optimised manipulator, while specialised in only a single task, can provide increased production in industry where the task is repetitive and consistent.

3 Mechanical Simulation Analysis

This chapter covers the simulation of the studied mechanism. Kinematic equations are first developed and then used to analyse the reachable workspace of the manipulator. Once this is achieved, Matlab's® SimMechanics® simulation package is used to model the manipulator.

3.1 Workspace Analysis

The reachable workspace of a robotic manipulator is useful to know. This defines where the end-effector of the manipulator can reach and as such determines whether or not the manipulator can achieve a given task. For a given path, Figure 3.1 and Figure 3.2 demonstrate a reachable and unreachable workspace respectively. In Figure 3.1, the path of the end-effector is completely contained within the workspace, whereas the same path shown in Figure 3.2 has some parts outside the workspace. A manipulator with the workspace of Figure 3.2 would not be able to complete the given task.

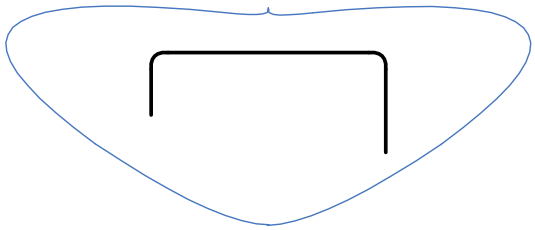

Figure 3.1 Reachable workspace, i.e. end-effector path is within reach of manipulators limits

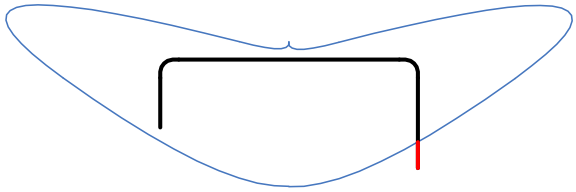

Figure 3.2 Unreachable workspace, i.e. part of the end-effectors path lies outside the manipulators workspace

As the manipulator being studied only has two degrees of freedom, the workspace can be obtained by moving each actuated arm (degree of freedom) through its full range of motion while holding the other arm stationary, iteratively stepping the stationary arm on through its full range of motion after each sweep of the non-stationary arm. By plotting the position of the end-effector at each of these points a diagram of the workspace can be formed. While this may be easily obtained when using a physical model, in software the kinematics of the system must be known.

The kinematics of the system describes the motion of the bodies without consideration of the forces that cause the motion [64]. This can be broken up into forward and inverse kinematics. While only forward kinematics are needed to find the workspace, the inverse kinematics will also be presented here for later reference as the calculations are often related.

3.1.1 Forward Kinematics

The forward kinematics specifies the valid positions of all bodies/arms for given angles of the two actuated joints. Figure 3.3 specifies the angles relative to +*Y*-axis in an anti-clockwise direction for θ_A, and clockwise for θ_B. This convention was chosen initially as it would ensure that all realistic angles would be positive and less than 360°. While many researchers would rather use the clockwise angle of the +*X*-axis as a plane of reference, it was decided that it would be preferential to align to the convention that exists in the most popular of parallel pick-and-place manipulators, the Flexpicker™ by ABB® [65].

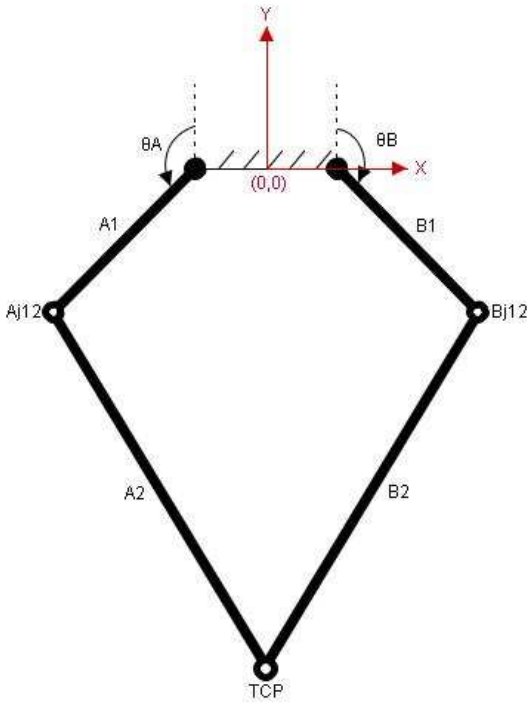

Figure 3.3 Configuration with driven angles referenced relative to the +Y-axis

It will be assumed that the mechanism is symmetrical about the origin, that is, the lengths of the proximal and distal arms shall be considered equal in length on both sides:

$$\|A_1\| = \|B_1\| \quad , \quad \|A_2\| = \|B_2\| \tag{3.1}$$

The following is the derivation of the TCP (tool centre point, i.e. end-effector) coordinates given θ_A and θ_B. Figure 3.4 is used as a reference for the following calculations.

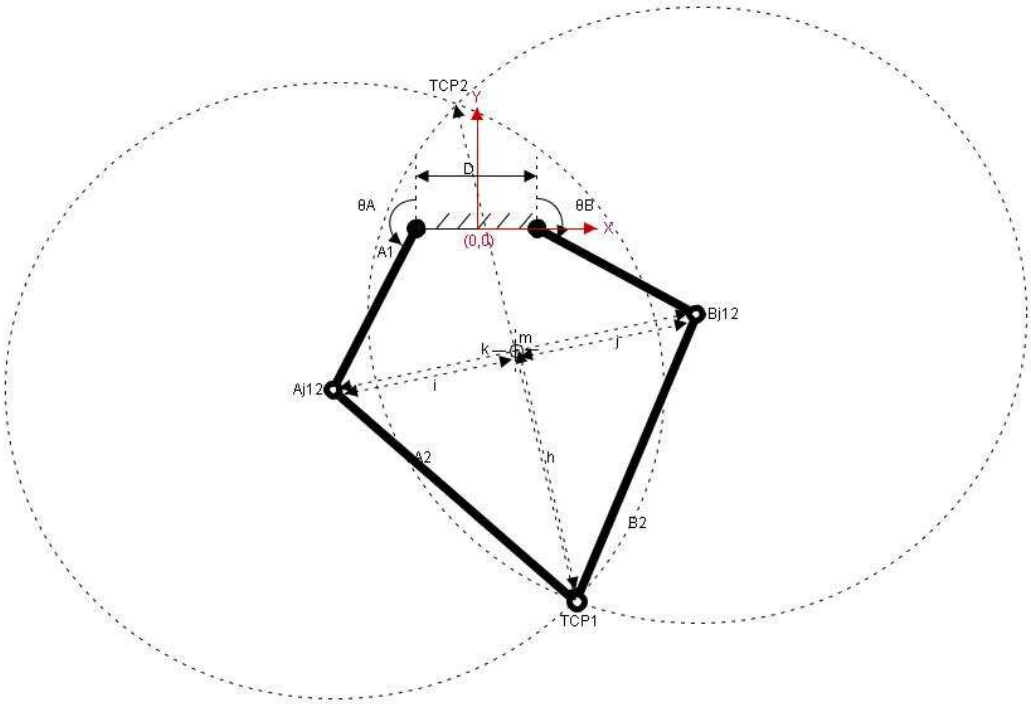

Figure 3.4 Derivation of manipulator's forward kinematics

The positions of the joints A_{j12} and B_{j12} are easily resolved using Pythagoras:

$$A_{j12}\begin{pmatrix}x\\y\end{pmatrix} = \begin{pmatrix}-D/2 - A_1\sin(\pi - \theta_A)\\ -A_1\cos(\pi - \theta_A)\end{pmatrix} \tag{3.2}$$

$$B_{j12}\begin{pmatrix}x\\y\end{pmatrix} = \begin{pmatrix}-D/2 + B_1\sin(\pi - \theta_B)\\ -B_1\cos(\pi - \theta_B)\end{pmatrix} \tag{3.3}$$

Given the fixed position of the actuated proximal arms, A_1 and B_1, the TCP coordinate can be obtained by finding the intersection of the two circles traced by rotating the distal arms, A_2 and B_2 about the joints A_{j12} and B_{j12}. This is displayed graphically in Figure 3.4 and can be resolved mathematically as shown below:

$$A_2{}^2 + i^2 = B_2{}^2 + j^2 \tag{3.4}$$

$$\therefore i = \frac{{A_2}^2 - {B_2}^2 + k^2}{2k} \tag{3.5}$$

$$h = \sqrt{{A_2}^2 - i^2} \tag{3.6}$$

$$m\begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} A_{j12}(x) + \dfrac{i(B_{j12}(x) - A_{j12}(x))}{k} \\ A_{j12}(y) + \dfrac{i(B_{j12}(y) - A_{j12}(y))}{k} \end{pmatrix} \tag{3.7}$$

$$\therefore TCP\begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} m(x) \pm \dfrac{h(B_{j12}(y) - A_{j12}(y))}{k} \\ m(y) \mp \dfrac{h(B_{j12}(x) - A_{j12}(x))}{k} \end{pmatrix} \tag{3.8}$$

As shown in Equation (3.8), for given values of θ_A and θ_B, there can exist up to two distinct arrangements of arms while maintaining a closed loop structure. When considered in practical terms, in pick-and-place applications, only one of these configurations, at most, is permissible. This is due to the planar mechanism having a work area underneath itself, and as such any configuration with the end-effector above the X-axis must be considered invalid. Based on this rule, Figure 3.4 shows the difference between an allowable configuration, *TCP1*, and an invalid one, *TCP2*.

Further to this, a configuration may be disallowed due to the potential for singularities either at the position or while moving to it. A singularity occurs when two or more connected arms become aligned, causing a loss in control of a degree of freedom. Singularities can be avoided by disallowing any configuration with internal angles (interior angles formed between arms-arms or arms-base) greater than or equal to 180°. Figure 3.5 shows an example of such a configuration which would be disallowed.

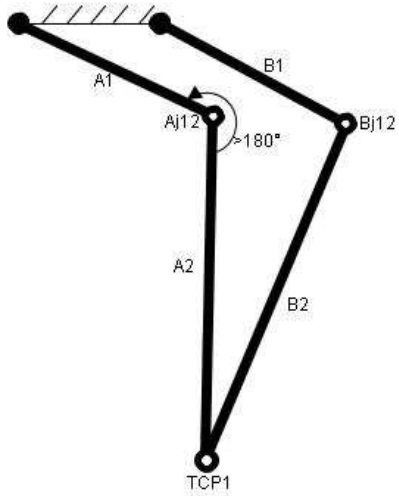

Figure 3.5 An example of an invalid configuration which has had to pass through a singularity to result in this position.

3.1.2 Inverse Kinematics

Realising the forward kinematic relationship is only half the kinematics. The inverse kinematics must also be obtained. The inverse kinematics answers the issue: "given a desired TCP coordinate (X, Y), determine the required angles for θ_A and θ_B". Deng et al. [66] shows for a similar manipulator that there are four possible configurations for a given TCP, however, only one of these is permissible if singularities are to be avoided. Huang et al. [13] have shown that the solutions can be limited by using formulae (3.9) and (3.10). These hold for a five-bar mechanical linkage with 2-DOF:

$$\theta_A = -\frac{\pi}{2} + 2\tan^{-1}\left(\frac{-e_A - \sqrt{{e_A}^2 - {g_A}^2 + {f_A}^2}}{g_A - f_A}\right) \tag{3.9}$$

$$\theta_B = \frac{\pi}{2} - 2\tan^{-1}\left(\frac{-e_B + \sqrt{{e_B}^2 - {g_B}^2 + {f_B}^2}}{g_B - f_B}\right) \tag{3.10}$$

where:

$$e_A = -2A_1TCP_y, \qquad e_B = -2B_1TCP_y,$$

$$f_A = -2\left(TCP_x + \frac{D}{2}\right)A_1, \qquad f_B = -2\left(TCP_x - \frac{D}{2}\right)B_1,$$

$$g_A = (TCP_x + \tfrac{D}{2})^2 + {TCP_y}^2 + {A_1}^2 - {A_2}^2, \qquad g_B = (TCP_x - \tfrac{D}{2})^2 + {TCP_y}^2 + {B_1}^2 - {B_2}^2$$

3.1.3 Reachable Workspace

With the knowledge of the mechanism's forward kinematics, the workspace can be determined. This was done in Matlab® by deriving the end-effector's X-Y coordinates for $0° \leq \theta_A \leq 360°$ and $0° \leq \theta_B \leq 360°$. Samples were taken at 5° increments. Additional joint angle constraints were included to represent mechanical limits imposed by the real-life design. These constraints were obtained from RML Engineering Ltd. The coordinates were then plotted to reveal the reachable workspace of a given manipulator configuration. The pseudo code of how this is done is shown in Figure 3.6.

```
procedure produce reachable workspace
begin
θA = θA_min
θB = θB_max
while θA < θA_max do
        evaluate TCP using forward kinematics from θA and θB
        if θB < θB_min then
                θA = θA + stepsize
                θB < θB_min
        else
                θB = θB – stepsize
        end
end
end
```

Figure 3.6 Pseudo code for producing the reachable workspace of the manipulator

The position of the TCP is evaluated using forward kinematics for different motor positions. The positions of θ_A and θ_B are iteratively altered between the minimum and maximum values with a granularity based on the parameter *stepsize*. This provides a close estimate to the reachable workspace of the manipulator.

3.2 SimMechanics® Simulation

Matlab® was used as the software development environment for this project due to its efficient computation ability, object-oriented programming support and its large selection of add-on features. One of these features is the Simulink® simulation environment which allows the modelling, simulating and analysing of multi-domain dynamic systems. Of particular usefulness to this project were the SimMechanics™ and Simscape™ tool-boxes. SimMechanics™ allows the development of a three dimensional model of the mechanical system. It should be noted however, that as the 2DOFPPM operates in a plane, only two dimensions are needed which in turn saves computation time.

The following sections outline the components used in constructing the model, the formation of constraints between mechanical bodies as well as the settings for the simulation.

3.2.1 Model Components

SimMechanics™ uses the concept of joints and bodies to create a mechanical model. Constraints can then be placed on components. Both joints and bodies can be actuated in the time domain and the model's behaviour is simulated under these conditions. While this allows a large degree of flexibility for various applications, it does require an accurate understanding of the desired system to permit correct implementation.

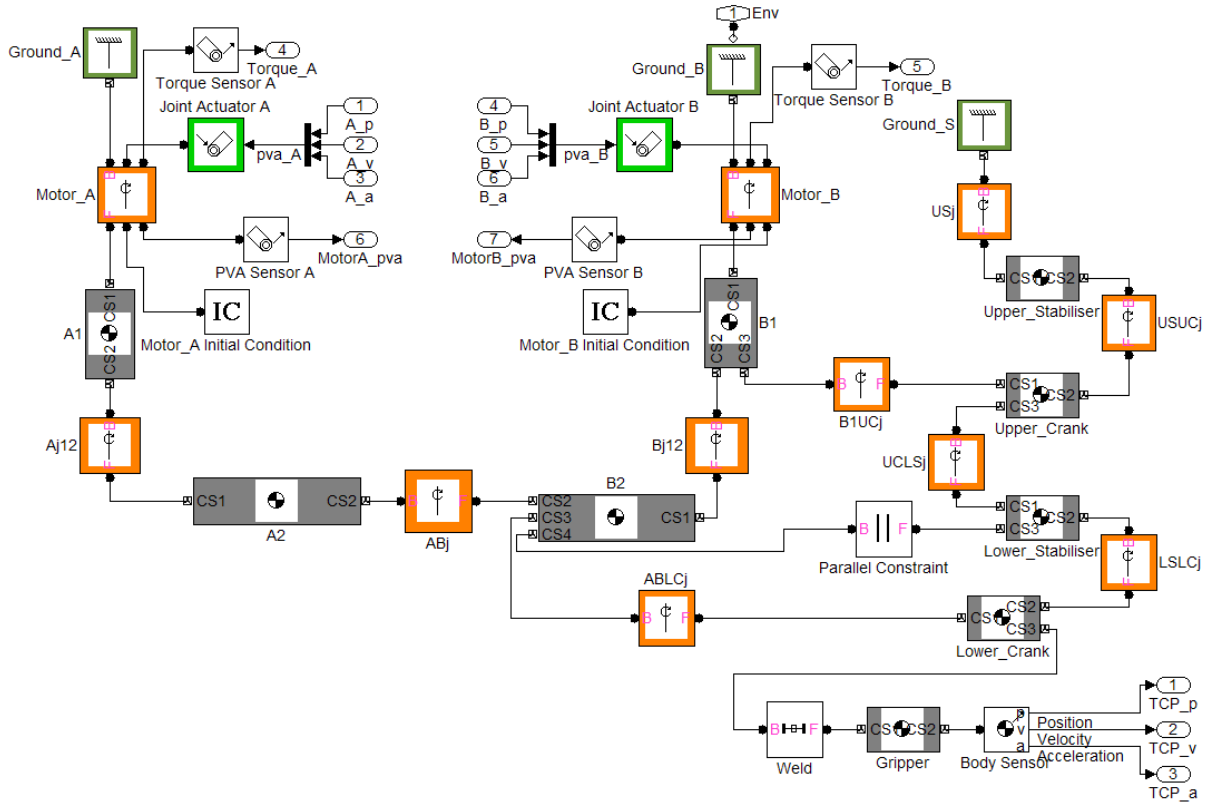

Figure 3.7 SimMechanics™ model of the 2-DOF Parallel Planar Mechanism

The mechanical components of the SimMechanics™ simulation are shown in Figure 3.7. This features a system of bodies connected by joints. The proximal arms, *A1* and *B1*, are connected via joints, *Aj12* and *Bj12*, to distal arms, *A2* and *B2*. The joints *Motor_A* and *Motor_B* represent the motors of the system and are acted on by joint actuators, *Joint Actuator A* and *Joint Actuator B*. This represents the core component of the 2DOFPPM. Additional bodies and joints are shown on the right-hand side of Figure 3.7

which represent the stabiliser arm components. The following sections explain the various components of the model in greater detail.

The model was developed using variables to define all dimensions, positions, constraints and settings. This allows the same model to be used for all simulations of the 2DOFPPM with only the value of the variables needing to be changed. When a simulation is run under SimMechanics™, a coded script is run (refer Appendix F, Figure F.68) to initialise these variables before conducting the simulation.

3.2.1.1 Bodies

Bodies are the fundamental mechanical linkages in the system. Bodies are characterised by their mass and inertia, position and orientation in space, as well as any attached coordinate systems.

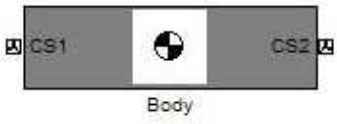

Figure 3.8 SimMechanics™ Block - Body

It is a requirement that the positions of each body are validly defined before the model can be simulated, that is, the positions of each body can be resolved. This means that the coordinates of the connection points on each body need to be specified, and that the coordinates of connection points for adjacent bodies be the same. This was done by using the inverse kinematics routine, developed earlier in the project (Section 3.1.2), to resolve the coordinates during the start-up script.

Bodies have coordinate systems assigned to them. These can define points on the body which other joints connect to, location of the centre of gravity, or any arbitrary point of potential significance. Coordinate systems are defined relative to another coordinate system. For example the centre of gravity could be defined as being at a 10 mm offset along the X-axis of the base coordinate system of the body, where the base coordinate system is at an (X,Y,Z) location of (20,30,40) in the world (global) coordinate system. Each coordinate system also has a defined orientation which allows a coordinate system to be rotated about one or more axes. A two dimensional example of coordinate systems is shown in Figure 3.9.

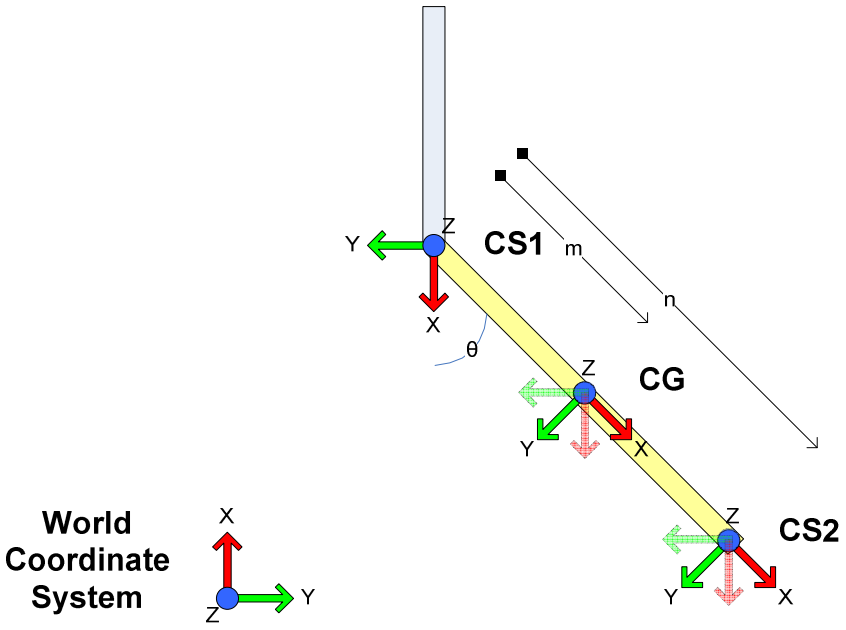

Figure 3.9 An illustration of how the body coordinate systems relate to each other.

In Figure 3.9 the yellow body has three coordinate systems attached to it. CS1 is attached at zero offset and zero rotation from the previous grey body. CG has been offset from CS1 by a distance *m* and rotation of θ. CS has been offset from CS1 by a distance *n* and rotation θ.

Two options exist as to how best to define the system. Either, each body could be defined in the world coordinate system, or the bodies could be defined relative to one another. Both these options were explored to see which would be the most convenient. While using the world as a reference point made it easy for a human to read during debugging, it proved more beneficial to use the relative frame of reference of the adjacent body. This was due to the order in which the kinematics were calculated, with the position and orientation of the bodies being calculated sequentially along the chain of arms. This was done starting at the ground points, through the proximal arms to the distal arms and TCP.

In the system being analysed, each body has three coordinate systems:

CS1 – the coordinate system connected to the previous joint in the chain, referenced relative to the adjoining coordinate system.

CS2 – the coordinate system connected to the next joint in the chain, referenced relative to CS1.

CG – the coordinate system defining the centre of gravity for the body, referenced relative to CS1.

Each body has a mass assigned to it along with an inertia tensor matrix to define the distribution of the mass. The inertia matrix was calculated in a custom Matlab® procedure (refer Appendix F, Figure F.82), and based on the assumption that the bodies would be regular hollowed cylinders as is commonly the case with parallel manipulators. The method of calculation can be seen in Equation (3.11).

$$I = \begin{bmatrix} \frac{m}{12}(3{r_1}^2 + {r_2}^2 + h^2) & 0 & 0 \\ 0 & \frac{m}{12}(3{r_1}^2 + {r_2}^2 + h^2) & 0 \\ 0 & 0 & \frac{m}{12}(3{r_1}^2 + {r_2}^2) \end{bmatrix} \quad (3.11)$$

where m = mass, h = length, r_1 = internal diameter, r_2 = outer diameter.

3.2.1.2 Joints

In SimMechanics™, joints are block components that represent one or more mechanical degrees of freedoms. Joint blocks are used to connect two body blocks to one another. There exist several different types of joints in SimMechanics™ however, the studied system only uses revolute joints. These are joints that rotate about a single line of reference (often a primary axis). The system has been set up such that the manipulator moves in the X-Y plane with the revolute joints rotating about the +Z axis. A revolute joint block is shown in Figure 3.10.

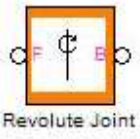

Figure 3.10 SimMechanics™ Block - Joint

SimMechanics™ cannot resolve closed loop topologies directly. Instead, it splits each closed loop into two serial chains and analyses them individually while preserving the fact that they are really a single closed loop. The joint at which SimMechanics™ cuts the chain can be specified. By specifying the most logical joint to cut, the TCP joint, the model behaves in a more appropriate manner than when left to determine the cut joint itself. When no joint is specified, the simulation selects one of the actuated joints to be cut. This causes the mechanism to be operated in an inverse manner where instead of selecting the TCP below the base, it selects the invalid TCP point above the base (see Figure 3.4). Although this is easily resolved in this situation, the ability for SimMechanics™ to determine a starting configuration that is invalid in reality presented a problem.

To ensure the starting configuration is valid for a 2DOFPPM, the initial conditions on the actuated joints, *Motor_A* and *Motor_B* in Figure 3.7 were set. These initial conditions were calculated and specified in the start-up script of the model. This proved successful in limiting the mechanical configuration to realistic positions.

SimMechanics™ allows several features to be added to joints to improve its realism. Both damping and stiction values can be added as additional blocks in the program. This project has not utilised these options as the values are arbitrary unless the exact bearing system is known, which will not be the case at this stage in the design process. If desired, this can be easily added to the system during the final design stages to further validate the design decisions.

3.2.1.3 Joint Actuation

SimMechanics™ allows both joints and bodies to be actuated by an external force or motion. For the system being evaluated it is necessary to only actuate the two joints A_{j1B} and B_{j1B}. These joints would normally be actuated by servo motors in reality.

Figure 3.11 SimMechanics™ Block - Joint Actuator

The joint actuation blocks, shown in Figure 3.11, have two modes of operation. Actuation can be in the form of either a force or a motion. A force applies a given torque to the joint. A motion requires three arguments, angular position, angular velocity and angular acceleration. The manipulator's TCP is to follow a given trajectory and therefore the joint actuator must follow a separate but related trajectory. This means that if the joint is controlled by the force technique, a mathematical relationship must be developed to relate the joints path to the torque applied. As seen later in Chapter 4, this is not easily obtained and therefore the joint must be actuated with the motion parameters.

The developed system uses an off-line trajectory planner (see Chapter 4) to determine the motor joints angular position, velocity and acceleration with respect to time. These values are stored in a file which is then accessed by SimMechanics™ during the simulation.

3.2.1.4 Sensors

The simulation would not be useful without any outputs providing data on how the simulation performed. As such, SimMechanics™ includes a range of sensor blocks that can be connected to both bodies and joints. For measuring joint outputs this project utilises *Torque Sensors* on the actuated 'motor' joints' as well as angular position, velocity and acceleration sensors. The *Gripper* body uses a *Body Sensor* to measure the end-effector's position, velocity and acceleration in Cartesian space. Examples of these sensor blocks are shown in Figure 3.12.

Figure 3.12 SimMechanics™ Blocks - Body Sensor (left), Joint Sensor (right)

3.2.1.5 Physical Constraints

Several additional constraints are used to help ensure the simulated model is configured correctly at the start of the simulation. Firstly, *Initial Condition* blocks allow a predefined position to be assigned to the joints. This assists in ensuring the arms are configured as desired and not inverted. Secondly, a *Parallel Constraint* block is added between the lower distal arm and the lower arm of the stabilising section. These components, shown in Figure 3.13 ensure the manipulator is configured as would be expected and prevents SimMechanics™ from potentially placing some arms in an inverted position.

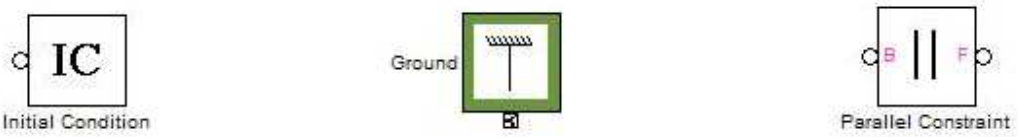

Figure 3.13 SimMechanics™ Blocks - Initial Condition Constraint (left), Ground Constraint (centre), Parallel Constraint (right)

3.2.2 Simulation Settings

There are a number of settings that define the environment and how the simulation is processed. Several points will be covered here however, for complete details the SimMechanics® documentation should be referenced.

Gravity is defined as a vector. This allows gravity to be either added or removed from the simulation which can be useful during debugging. Without gravity, the direct and inverse kinematic procedures

produced earlier in the project were used to crosscheck results returned from the SimMechanics® simulation.

There exist several modes of analysis for closed loop systems. Namely these are *forward dynamics* and *kinematics*. *Forward dynamics* computes the positions and velocities of the system's bodies, given forces, torques and initial conditions. *Kinematics* computes the forces and torques required to produce the specified motions. This project has used the default mode of *forward dynamics* analysis, although the *Kinematics* mode was also explored. No noticeable differences in computation performance or simulation results were found.

The simulation can be resolved at either fixed step intervals or by allowing SimMechanics® to detect the time intervals to produce an accurate simulation. After some trialling of different settings with various cycle-paths, the variable step option was found to be best suited as it produced an accurate result while not taking too long to process.

The simulation can be resolved using one of several numerical analysis techniques. The different methods produce the same general result but with differing degrees of accuracy and execution speed. The default option in SimMechanics™ for a variable step solver is the *Runge-Kutta, Dormand-Prince (4,5) pair* method. This method proved the most suitable for solving the 2DOFPPM, as configured in this section, due to it providing suitable precision in the fastest possible time.

3.2.3 Running the Simulation

Figure 3.7 shows the model of the mechanical system in terms of bodies, joints etc. This is turned into a subsystem and included as part of the larger system which handles the inputs, outputs and simulation settings. The higher level abstraction can be seen in Figure 3.14. The mechanical system is contained within the large block labelled '*Mechanical Robot'*. Data streams are read in from files on the left hand side and different data streams are read out of the '*Mechanical Robot*' block on the right hand side. These output data streams are stored into separate files.

The SimMechanics™ model was developed to take input from a pair of files. These files contain time dependent data about each of the motors' angular position, velocity and acceleration. At this stage in the thesis, the formation of these data files will not be considered as it will be covered later in Chapter 4 on Trajectory Planning. These files will instead be used as a given set of commands for which the simulation must carry out.

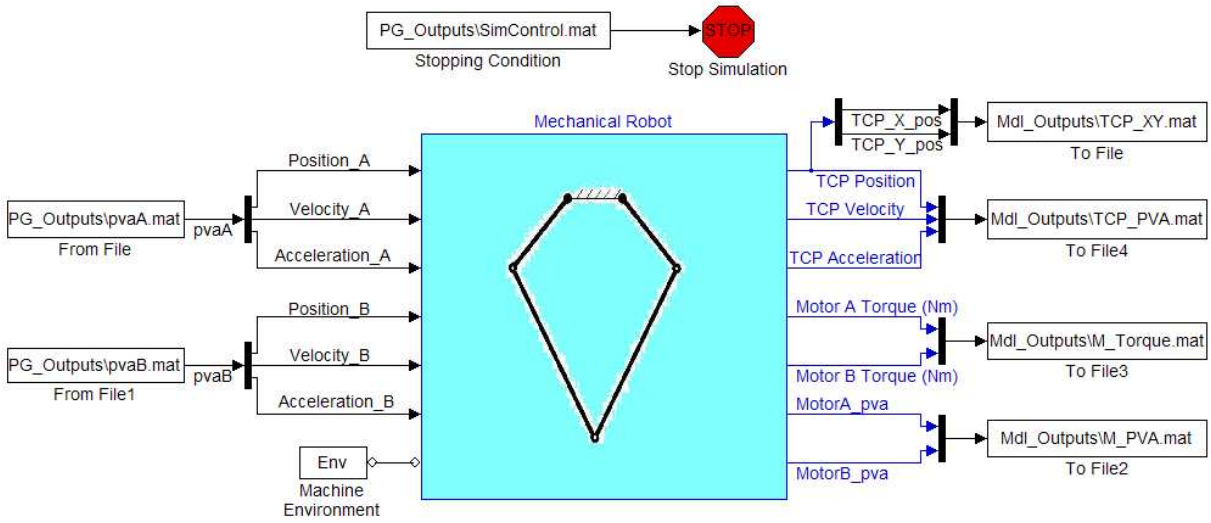

Figure 3.14 SimMechanics™ high-level view of the simulation construct

At the top of Figure 3.14 a file stream is being used as a stopping condition for the simulation. This file is a time dependent file consisting of '0' data values up until the desired end of the simulation where a '1' value triggers the simulation to stop. This is also produced as part of the trajectory planning process.

The '*Machine Environment*' block at the bottom left of Figure 3.14 contains parameters, constraints and settings for the simulation. This is where the gravity vector is defined along with mechanical assembly tolerances settings.

When the simulation is run the SimMechanics™ simulation engine evaluates the time dependent inputs and, at internally determined points in simulation time, calculates the resulting actions of the mechanical components. The sensors within the system then record their measurements and the values are stored in files. These files can be plotted and the system's performance analysed. An automated process for plotting these results has been developed.

3.3 Mechanical Simulation Results

3.3.1 Workspace Analysis

As discussed previously, the workspace of a manipulator is defined as the area which the TCP of the manipulator can reach given the constraints of the system. In the case of the manipulator being studied, the constraints are limited by the minimum and maximum angles of each joint, the upper and lower arm lengths and the spacing of the servo motor actuators. The results in this section are based around default dimensions and constraints obtained from RML Engineering Ltd. shown in Table 3.1. These

default values were obtained based on good engineering principles and a 'rule-of-thumb' approach. Later in this thesis some of these values will be optimised for a particular task. The results presented here are to highlight the effects of each dimension on the shape and size of the workspace, as such the default values and their results are not important in themselves but rather the relative changes in the results are of interest. Figure 3.16 through to Figure 3.19 highlight the changes in workspace shape when the manipulator's dimensions are varied.

Table 3.1 Default parameters for workspace analysis. Values obtained from RML Engineering Ltd.

Parameter	**Default Setting**
Base length (separation of servo motor actuators)	0.3 m
Proximal (upper) arm length	0.36 m
Distal (lower) arm length	0.88 m
Minimum angle between proximal arm and +Y-axis	43°
Maximum angle between proximal arm and +Y-axis	164°
Minimum internal angle between proximal arm and distal arm	43°
Maximum internal angle between proximal arm and distal arm	134°
Minimum internal angle between distal arms	48°
Maximum internal angle between distal arms	71°

The dimensions and constraints of RML Engineering's first design are displayed in Figure 3.15, along with the workspace reachable under these constraints.

The effects of applying constraints on minimum and maximum angles at each joint can be seen most clearly in Figure 3.16, where the default limits are compared to the maximum workspace limited only by joint singularities.

Upon consideration of Figure 3.16, an interesting observation can be made regarding the relative speed and accuracy of the TCP within the workspace. If the larger workspace (green) is analysed in the knowledge that each point is plotted with a constant angular displacement from each other (8°), it can be seen that if the servo motor actuators are rotated at a constant rate, the displacement of the TCP becomes smaller in higher density areas (by higher density, it is meant the density of the plotted points in the workspace graphs). Conversely, if the plotted points are further dispersed (e.g. near the outer limits of the workspace), this indicates an area of the workspace where greatest TCP speed can be

achieved. Relative accuracy, due to errors in the servo motor positions, can also be deduced in the same manner. Densely populated areas of the workspace are less prone to error in TCP position due to servo motor errors, whereas sparsely populated areas will encounter greater TCP errors from any servo motor inaccuracies.

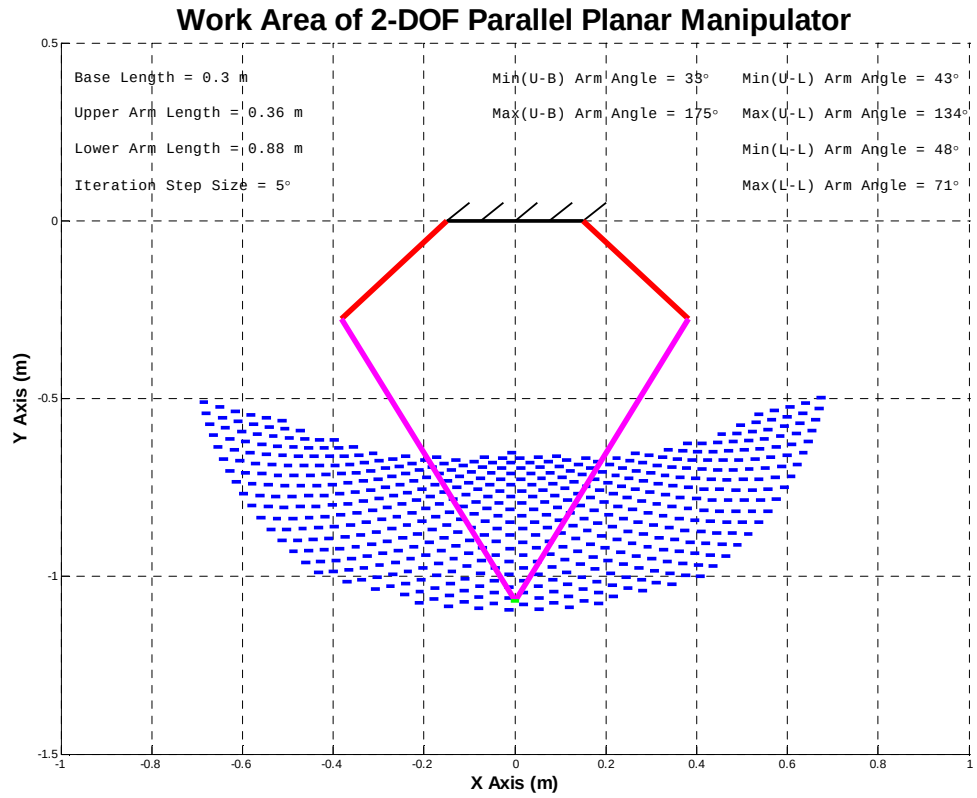

Figure 3.15 Workspace of manipulator using RML Engineering's default dimensions and constraints.

It should be noted however, that the oversimplified perception of relative speed and accuracy in the workspace obviously does not take into account the motor dynamics, torque requirements from the arms nor the highly coupled nature of the parallel mechanism. A much more thorough analysis is required to accurately compare even the relative performance of manipulator configurations, let alone being able to evaluate the actual performance.

Figure 3.17, Figure 3.18 and Figure 3.19 display the effects of changing the spacing between the servo motor actuators, and lengths of the proximal and distal arms respectively. In each example the dimensions were altered by ±100 mm, and the corresponding workspace plotted. As can be seen in the plots, a small change in any of these dimensions can vastly alter the effective workspace. It is for this

reason that an effective means of optimising, not only the workspace, but the overall system performance is needed.

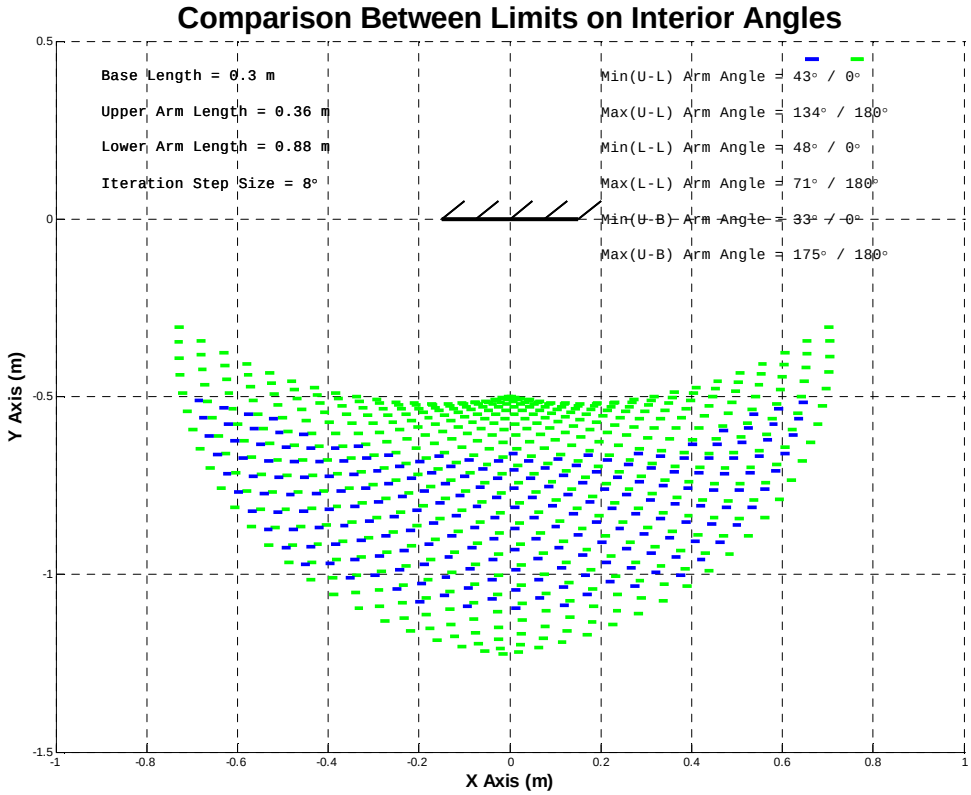

Figure 3.16 Comparison of workspace limited by RML Engineering's concept manipulator's angle constraints (blue) and angle limits before encountering singularities (green).

Figure 3.17 shows that if the angle constraints remain the same, increasing the separation of the servo motors causes the workspace to become a more hollowed, deeper and narrower 'U' shape. By moving the motors closer to each other, the workspace becomes wider and flatter.

When considering the effect that the length of the proximal arm has on the workspace, Figure 3.18 shows that a shorter arm produces a smaller workspace closer to the base. A longer upper arm results in a deeper and more hollowed 'U' shape with similar width to the original.

Figure 3.19 demonstrates that altering the distal arm length has the greatest effect in modifying the available workspace. This is due to the distal arm being the link furthest from the point of actuation and therefore altering its length is 'multiplied' by the leverage of the proximal link. By increasing the distal

arm length, the workspace becomes spread along the X-axis and slightly compressed in the Y-axis. Reducing the distal arm length causes the workspace to become noticeably more 'U' shaped.

Pick-and-place applications often require a rectangular workspace and as such, 'U' shaped workspaces become ineffective and difficult to utilise. It is therefore preferential to select a workspace that is most evenly dispersed in both the X and Y planes. This method of analysis can be useful to achieve a desirable workspace.

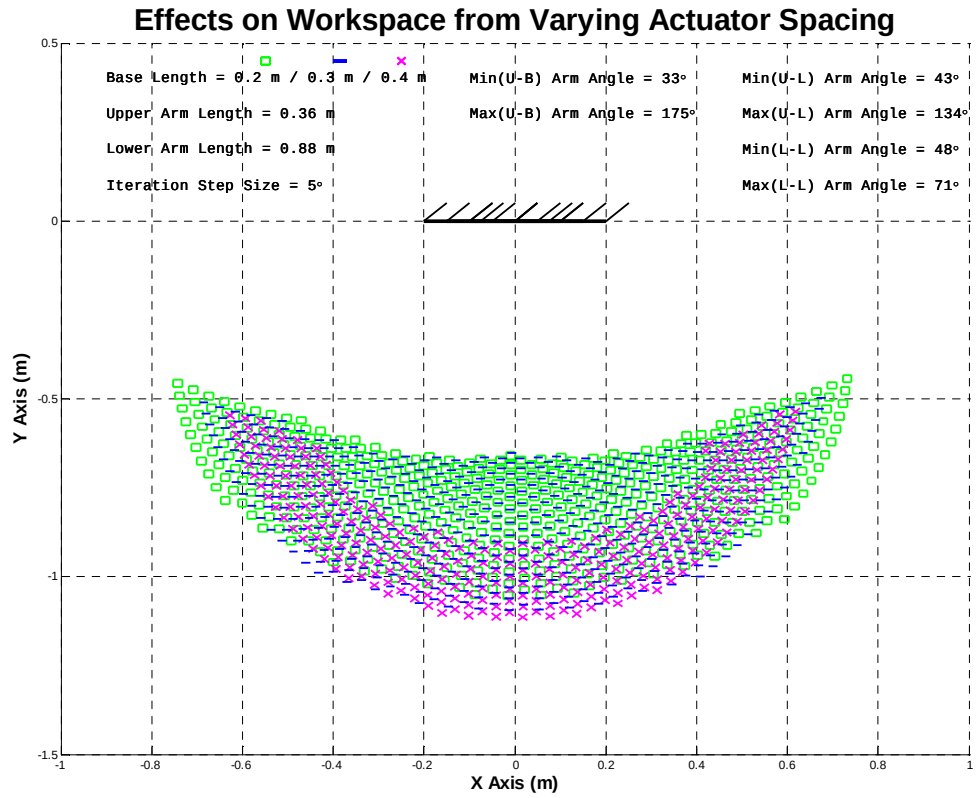

Figure 3.17 Comparison between workspaces when the base length (separation of actuated joints) is altered. The default distance of 0.3 m (blue) is compared to a smaller distance of 0.2 m (green) and a larger distance of 0.4 m (pink).

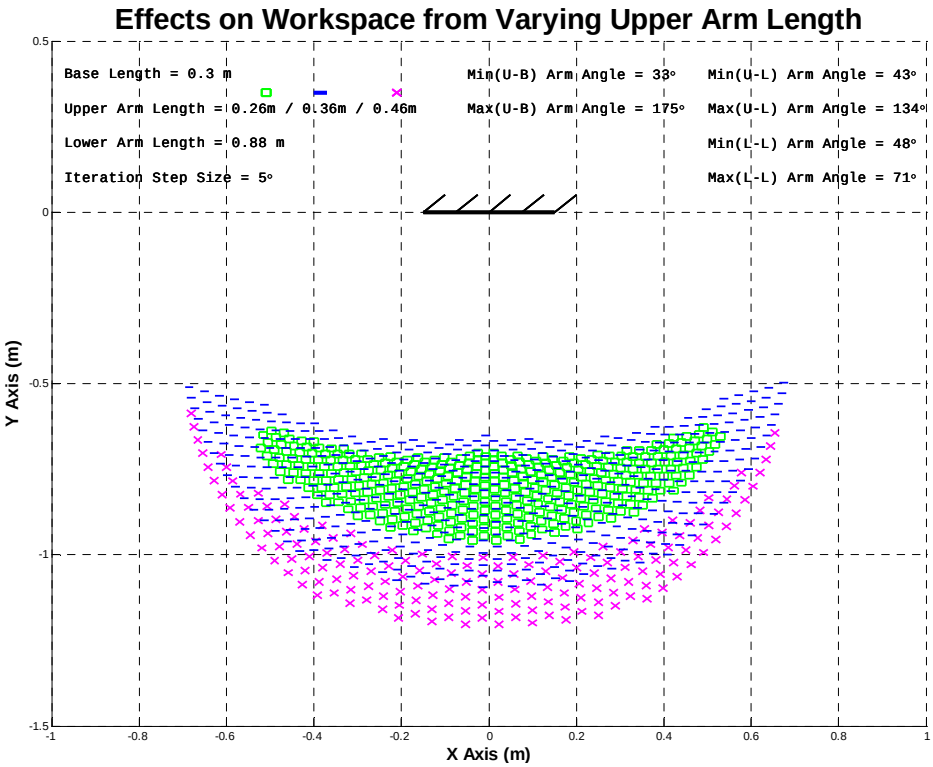

Figure 3.18 Comparison between workspaces when the proximal (upper) arm length is altered. The default length of 0.36 m (blue) is compared to a smaller length of 0.26 m (green) and a longer length of 0.46 m (pink).

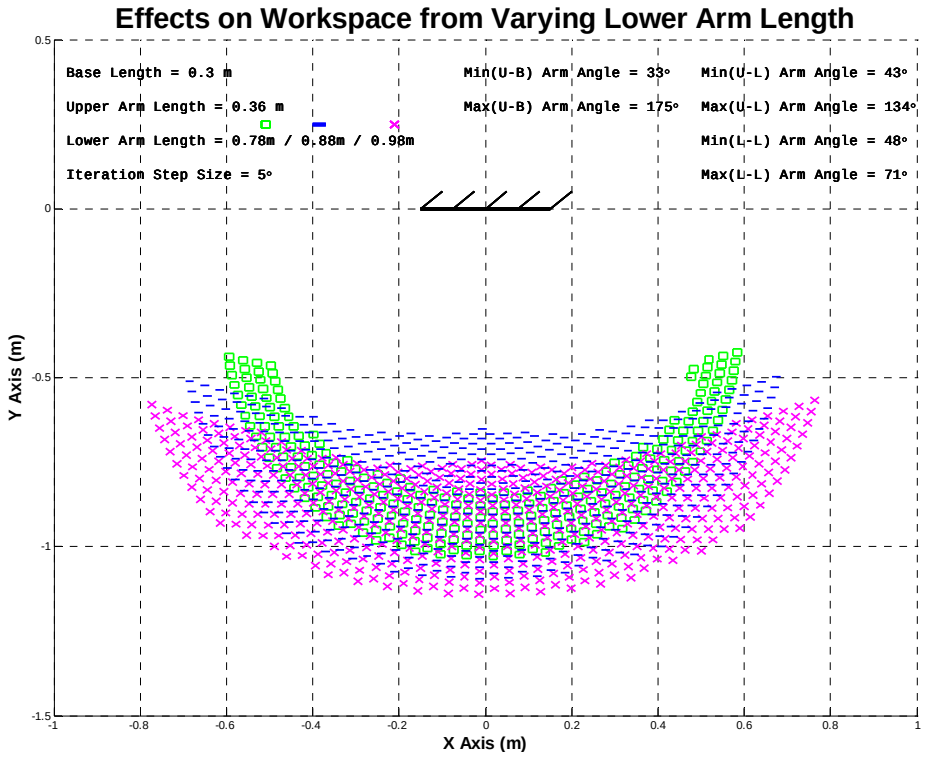

Figure 3.19 Comparison between workspaces when the distal (lower) arm length is altered. The default length of 0.88 m (blue) is compared to a smaller length of 0.78 m (green) and a longer length of 0.98 m (pink).

3.3.2 SimMechanics™ Analyis

In order to present the output and capabilities of the SimMechanics™ simulation, a sample path must be defined. Figure 3.20 shows the end-effector's cycle-path that is being used in this project. This path was chosen as it represents a typical pick-and-place cycle for product manipulation using the machines that RML Engineering Ltd. currently manufactures. The trajectory planning method used to generate the example path in this section is described in the following chapter. The simulation was run using this path, along with the additional parameters and constraints specified in Table 3.2. A complete list of mechanical parameters can be found in Appendix A.

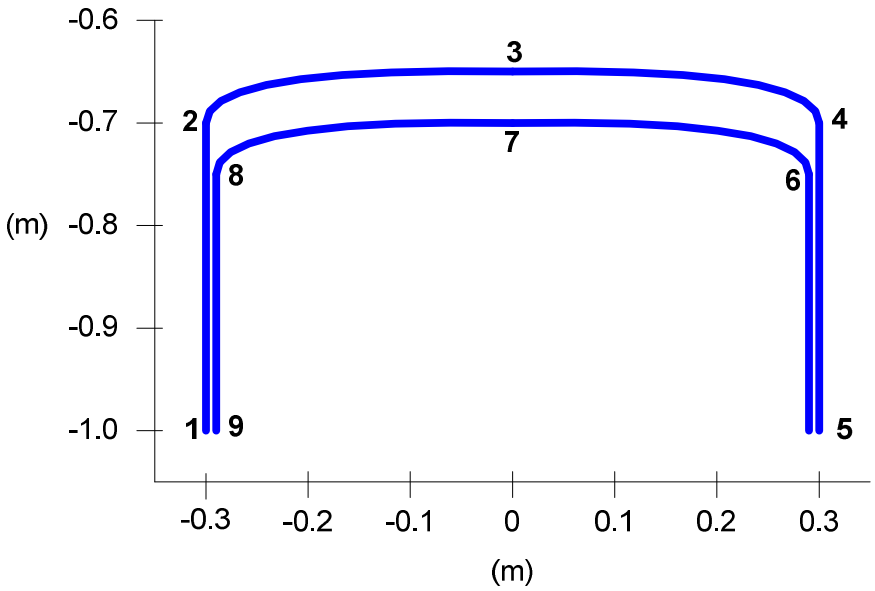

Figure 3.20 Test cycle-path. Movements follow the order from 1 through 9

When the SimMechanics™ simulation is run, a visualisation of the manipulator can be viewed showing the mechanical components moving in relation to one another under the presence of the external forces. A screen shot of this is shown in Figure 3.21. The arms are represented by simple lines, although the inertias of each component are represented in three dimensions. Running parallel to the right hand side arm is the additional stabilising arm offset by a fixed amount. The gripper is also represented as a triangle at the bottom of the two distal arms.

Table 3.2 Default parameters used in sample simulation. Values obtained from RML Engineering.

Parameter	**Default Setting**
Base length (separation of servo motor actuators)	0.3 m
Proximal (upper) arm length	0.36 m
Distal (lower) arm length	0.88 m
End-effector length	0.01 m
Proximal arm mass	3.5 kg
Distal arm mass	2 kg
End-effector mass	35 kg
Arm ID (Internal diameter)	0.01 m
Arm OD (Outer diameter)	0.02 m
Minimum angle between proximal arm and +Y-axis	43°
Maximum angle between proximal arm and +Y-axis	164°
Minimum internal angle between proximal arm and distal arm	43°
Maximum internal angle between proximal arm and distal arm	134°
Minimum internal angle between distal arms	48°
Maximum internal angle between distal arms	71°
Pick/Place Dwell Time	0.2 s

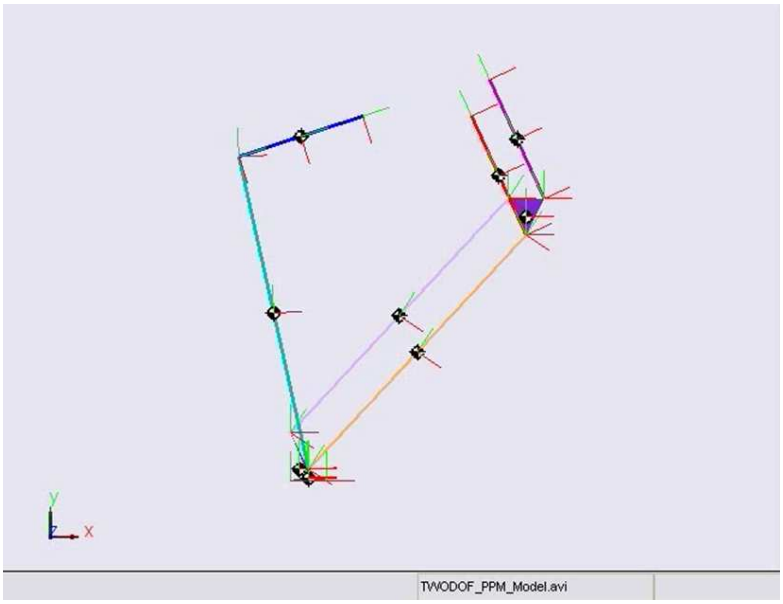

Figure 3.21 Screenshot of the SimMechanics™ simulation being run.

Figure 3.22, Figure 3.23 and Figure 3.24 show the motor positions, velocity and acceleration during the simulation. The motors start and finish with zero velocity, as well as having a stationary period of 0.2 s in the middle of the cycle. This pause in the cycle is to represent the time taken for the end-effector to 'pick' or 'place' the handled object. In Figure 3.24 it can be noted that the motor acceleration has abrupt changes in values and does not accurately represent the performance ability of a real motor. This is due to a limitation of the cubic spline trajectory planning method (see Section 4.1.4) that is used. It will be shown in Section 4.4 that this is of insignificant consequence and that the simplified motor characteristic is still valid for the level of model fidelity required in this project.

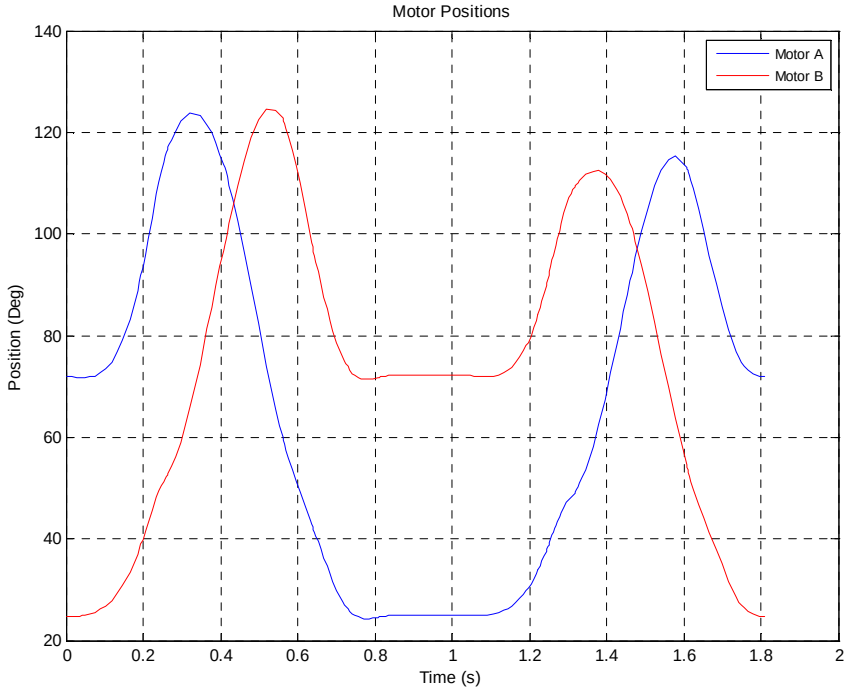

Figure 3.22 Simulated output of the motors' positions over the sample path-cycle.

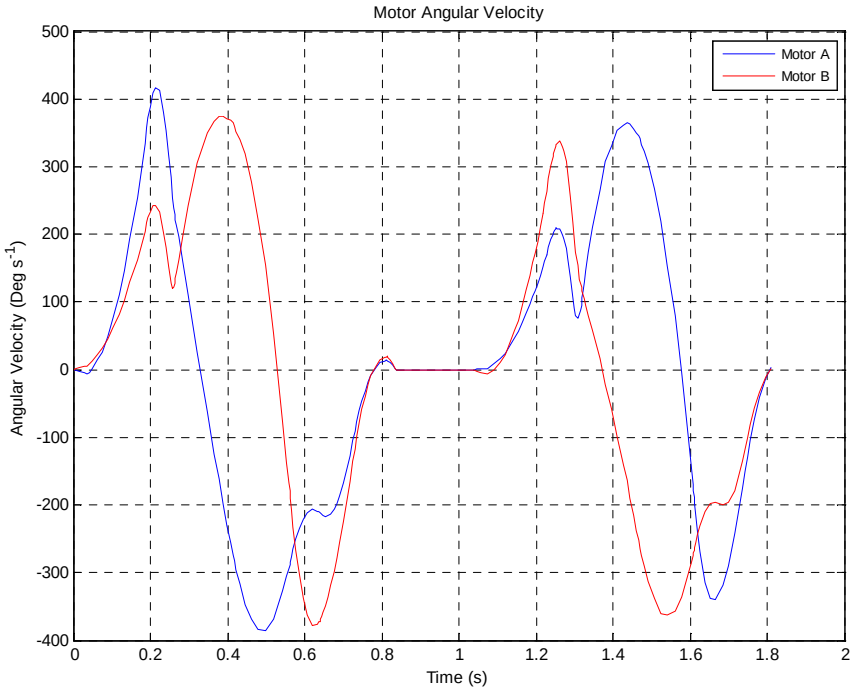

Figure 3.23 Simulated output of the motors' angular velocity over the sample path-cycle.

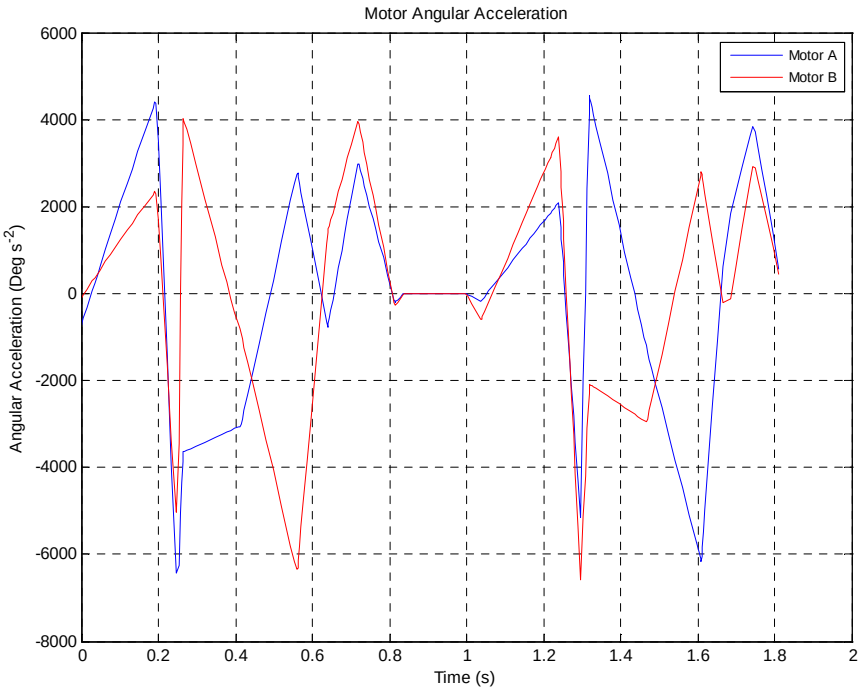

Figure 3.24 Simulated output of the motors' angular acceleration over the sample path-cycle.

The torque required to move the manipulator's actuated proximal arms in the profiles shown above in Figure 3.22 to Figure 3.24, is presented in Figure 3.25. The SimMechanics™ simulation engine takes into account the highly coupled nature of the parallel mechanism when producing this result. The sharp changes in torque, similar to the acceleration pattern found in Figure 3.24, are again the result of the trajectory planning method and can be assumed accurate enough for the simulation task at this stage in the thesis.

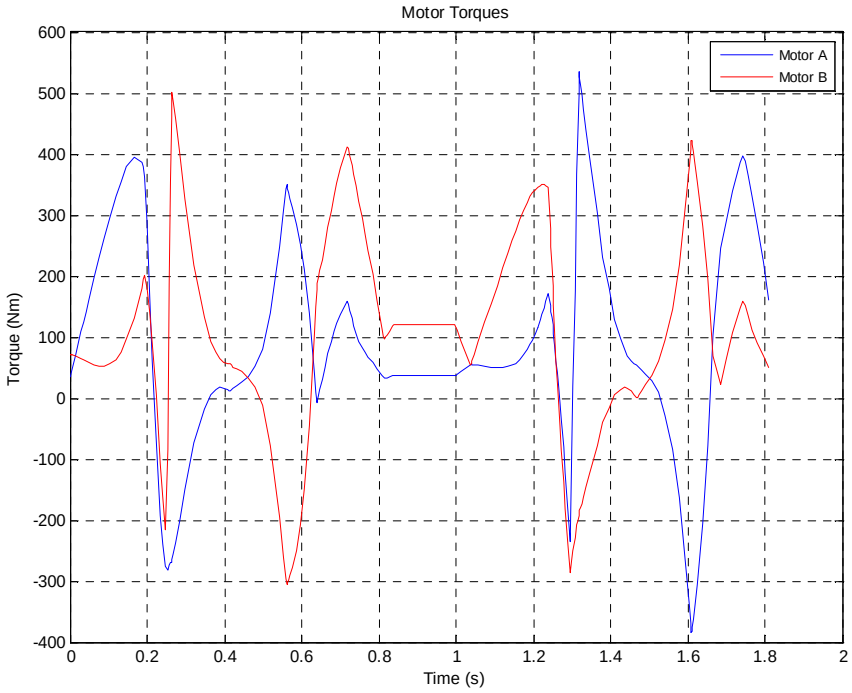

Figure 3.25 Simulated output of the motors' torque over the sample path-cycle.

Figure 3.26, Figure 3.27 and Figure 3.28 show the end-effector's (TCP) position, velocity and acceleration during the simulation. Each graph has been separated into separate (X, Y) Cartesian coordinates. In the velocity and accelerations, Figure 3.27 and Figure 3.28 respectively, an additional line has been plotted representing the summation of the X and Y components of velocity and acceleration. This information is of particular importance when designing the tool or gripper head to ensure that it is capable of handling objects with the high speeds and accelerations produced by the manipulator.

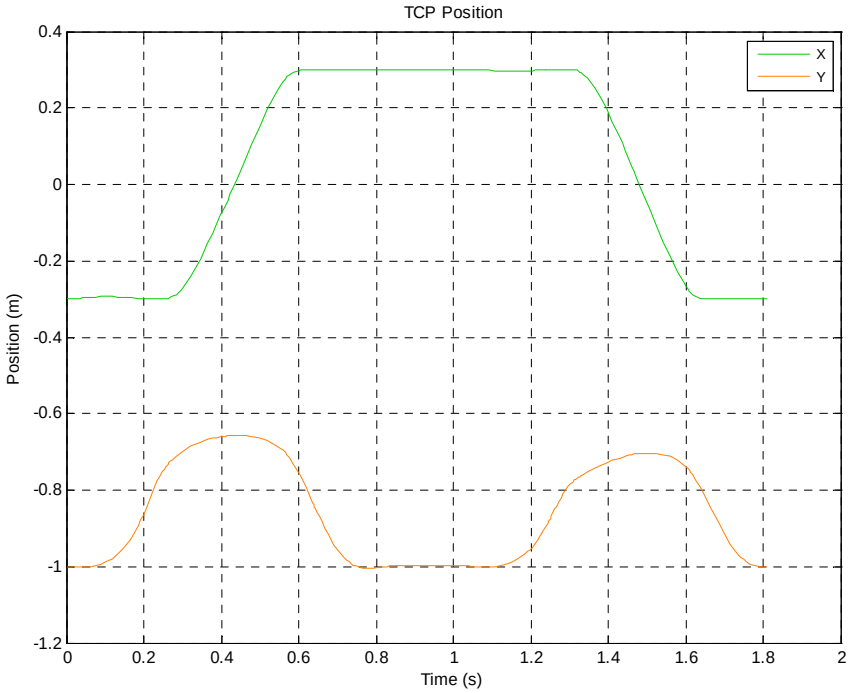

Figure 3.26 Simulated output of the end-effector's position in X and Y components over the sample path-cycle.

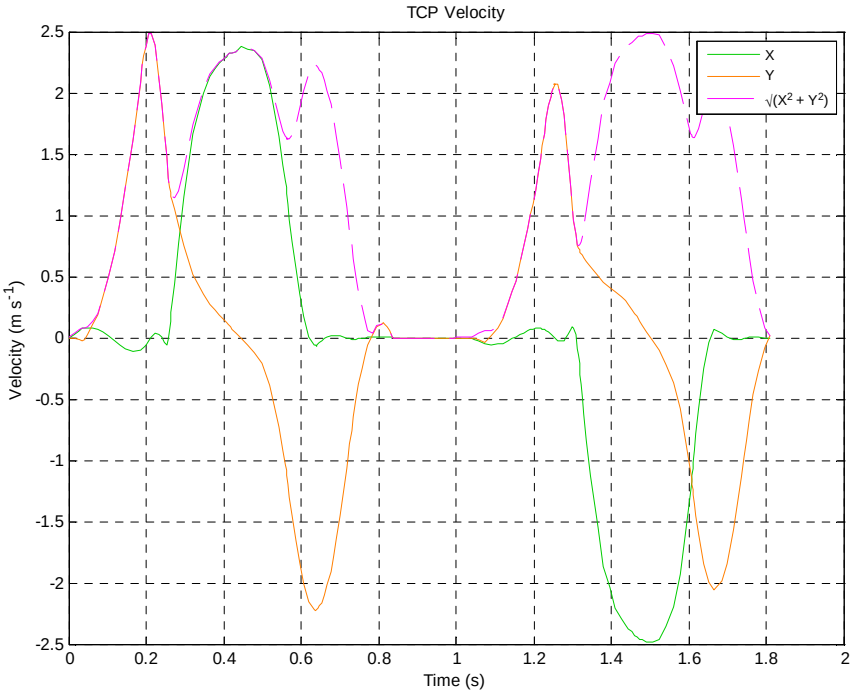

Figure 3.27 Simulated output of the end-effector's velocity in X and Y components over the sample path-cycle.

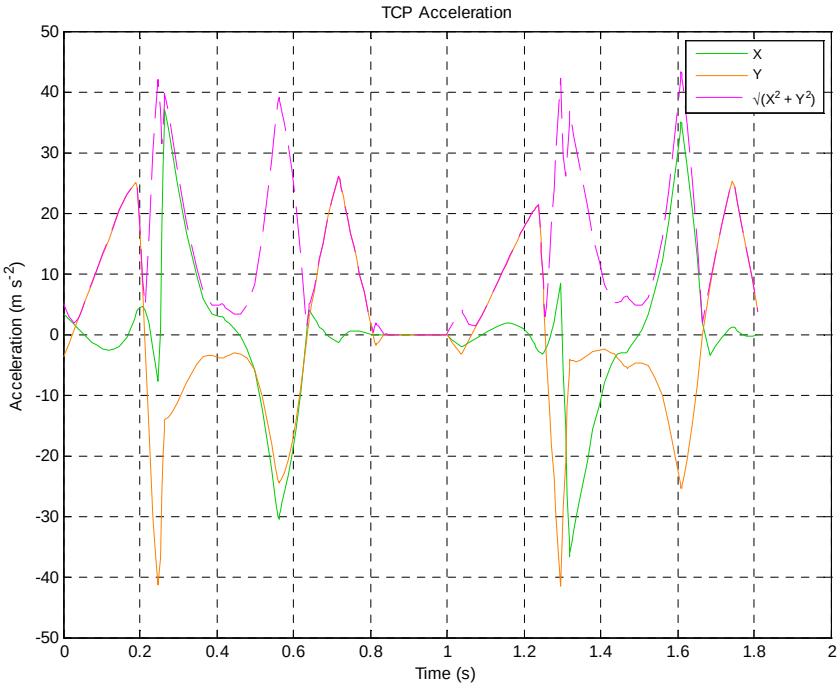

Figure 3.28 Simulated output of the end-effector's acceleration in X and Y components over the sample path-cycle.

The exact path followed by the end-effector is more clearly seen in the Cartesian plot shown in Figure 3.29. The path cycles from left to right, pauses and then returns back to the left. The data points have been plotted to show the relative position in time, with the points at the start of the cycle being plotted as a duller colour becoming progressively brighter towards the end of the cycle. Careful observation will show that the data points are unevenly dispersed throughout the path. This occurs because of how SimMechanics™ executes the simulation. The time between adjacent time segments varies depending on how much change SimMechanics™ detects in the mechanical system during the previous time segments. It can also be noted that this trajectory does not follow the desired path exactly as shown earlier in Figure 3.20. This is due to the trajectory planning method explained in the next chapter.

The simulation systems developed in this chapter produces results that can be used in later sections. The kinematic equations developed will be used in the trajectory planning process (Chapter 4), while the SimMechanics™ simulation allows the visualisation and analysis of an optimised manipulator configuration in Chapter 6.

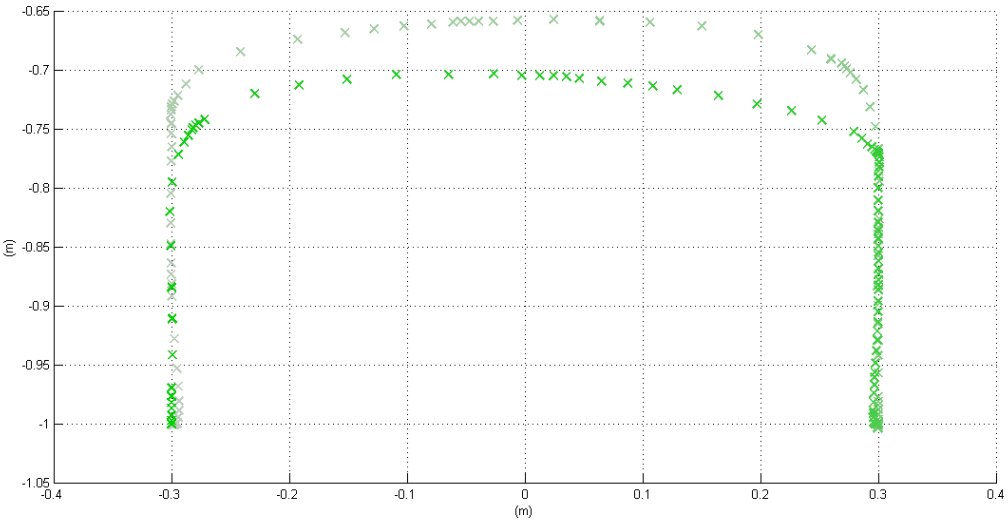

Figure 3.29 Trajectory traced by the end-effector during the SimMechanics™ simulation.

4 Trajectory Planning

In order for a manipulator to be highly productive, it must perform its task in minimal time. Many researchers have investigated methods for achieving this [23][33][37][38][45][46][48-54][67-69]. Trajectory planning techniques can be separated into off-line or on-line methods depending on their computational intensity and ability to handle new path commands on the fly. Methods that involve significant computation, such as performing an iterative optimization process, are generally too slow to be computed in real-time while simultaneously tracking the manipulator's path [26]. Conversely, on-line trajectory planning methods must be computed fast enough to avoid causing latency in the manipulator's movements. On-line methods also benefit from being able to re-compute the trajectory if obstacles are encountered or if using a vision and conveyer tracking system [26].

Many trajectory planners implemented in industrial controllers use on-line trajectory planners that only consider the kinematic limitations on the system [26]. By not considering the dynamic constraints (i.e. motor torque limits), the trajectory can be computed at sufficient speed which enables them to be on-line systems. However, by not taking into account the dynamic limitations, the manipulators are forced to underutilize their motors' performance capabilities. If the motors' maximum performance limitations are used in planning a path, saturation of the motors occurs leading to poor path tracking as the motors are not capable of producing the torque required to perform the kinematics [26]. When comparing configurations and finding an optimal solution of the 2DOFPPM, kinematic only analysis may result in solutions that do not perform well in reality.

After determining that the trajectory planner must take into account the dynamics of the system, a number of methodologies were considered. Bobrow et al. [48] and Shin and McKay [49] individually presented a method that produces a time-minimum trajectory. This method however, requires that the system's dynamic equations are known.

Two methods commonly exist for formulating the dynamic equations of a mechanical system, the Euler-Lagrangian technique and the Newton-Euler approach. These methods produce similar results but are obtained by different means. Due to the parallel structure of the 2DOFPPM, the dynamic equations are not easily obtained using either of these methods. Kim and Shin [47] developed a minimum-time path planning method in joint space using heuristics to produce a dynamic model of the manipulator. Huang

et al. [54] also used a hybrid approach to form the dynamic equations for a 2DOFPPM. The trajectory planning method used in this project considers these approaches to the problem, and where appropriate borrows their ideas to produce a method that is appropriate for this project.

4.1 Trajectory Planning Process

A range of trajectory planning methods were considered for this project. The resulting process uses an off-line, cubic spline fitment of the path in joint space, taking into account the system's kinematic limitations and an estimation of the system's dynamic limitations. One of the main objectives of this project is to enable the development of customised 2DOFPPM manipulators that are optimized for a given task. To do this a trajectory planner must be developed. While the exact path planning methodology is not crucial, it is important that comparisons between manipulator configurations are compared using the same trajectory planning process. This trajectory planning method allows for the manipulator's dynamics to be taken into account, albeit in a simplified estimated form.

The following sections present a detailed explanation of the trajectory planning process. The process is also summarised diagrammatically in Figure 4.1.

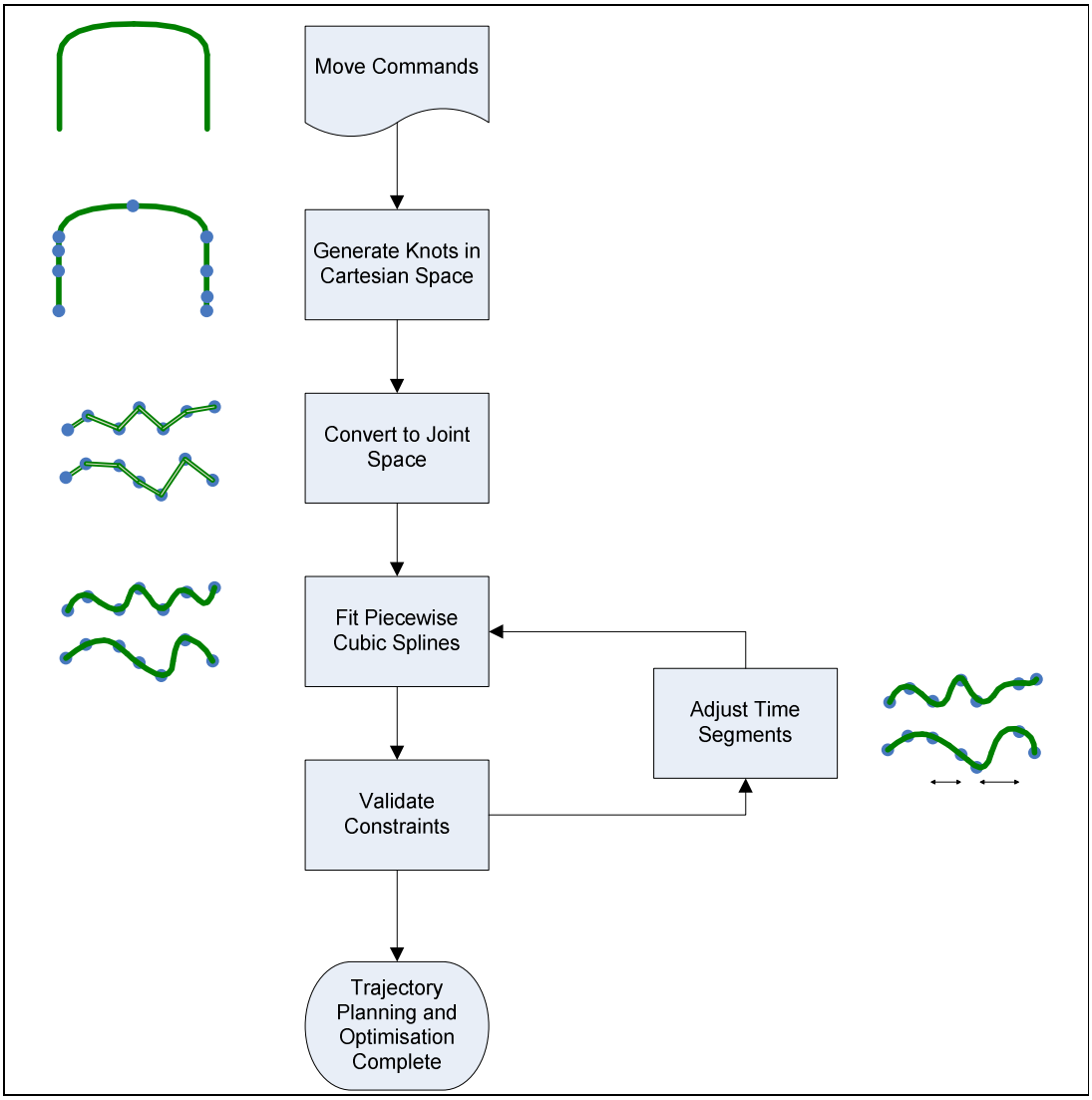

Figure 4.1 Flow diagram of the trajectory planning and optimisation process.

4.1.1 Movement Commands

The path of a manipulator can be defined in joint space, where the angles of the actuators are the defining factor, or, as is more common in industry, the path is defined in terms of the end-effectors position in the Cartesian workspace. Figure 4.2 shows the path that the end-effector is required to follow and the corresponding angle commands required to achieve that path. Clearly, when defining the movement of the manipulator, it is more intuitive to define in terms of the end-effector position in Cartesian workspace, than to be defined in terms of the motor's angular positions.

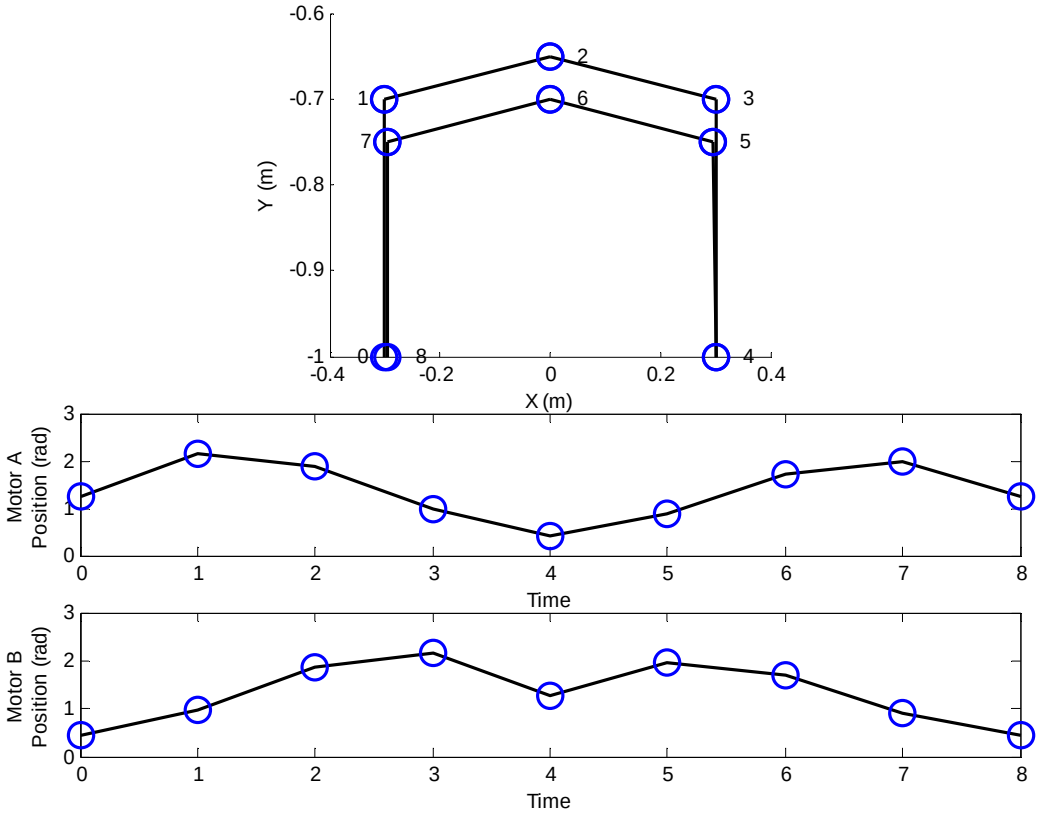

Figure 4.2 Example trajectory in Cartesian space with the corresponding motor positions required to reach each target point.

Along with defining the path in Cartesian coordinates, several other features and constraints are used to tailor the path for a particular task. Table 4.1 summarises the parameters that can be specified in this system.

Table 4.1 Path defining parameter definitions.

Parameter	Units	Description
Target	(X,Y)	The position in Cartesian coordinates where the end-effector is expected to travel to.
MoveType	['MoveJ','MoveL']	Indicates how the end-effector should move in order to reach the target.
Zone	mm	Specifies a distance from the target where the end-effector is considered close enough and can begin moving to the next target on the path.
MaxTCPSpeed	ms^{-1}	Sets a limit on the end-effector/TCP speed during the movement.
Pause	s	Optional parameter to stipulate the manipulator must pause for a period of time.

The *target* is only defined in terms of the X and Y coordinates with the Z component being omitted as the 2DOFPPM only acts in a single Z plane. The *MoveType* permits either a *linear* move (*MoveL*) or a *joint* move (*MoveJ*). A *linear* move requires the movement to be performed in a straight line between the two targets, whereas the *joint* move is a weaker constraint and allows the movement to be executed in a way that is most efficient for the joints. Figure 4.3 depicts the difference between these two moves, with the black line representing a *linear* movement during the vertical 'pick' and 'place' movements, and the green line representing a *joint* move which deviates from the direct path between targets in order to find a more efficient path for the actuated joints.

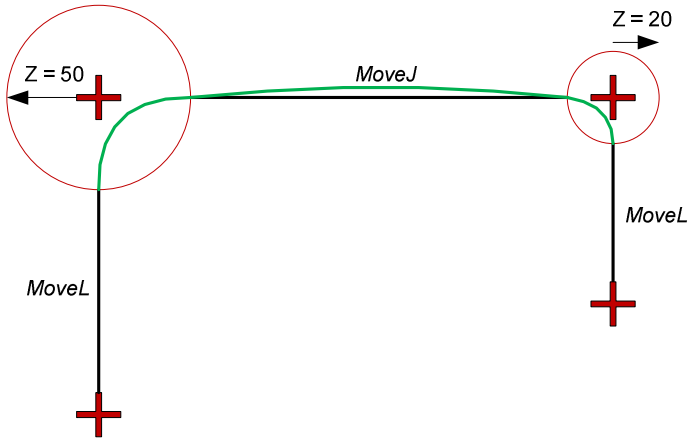

Figure 4.3 A sample path consisting of two vertical linear movements (*MoveL*) and a single joint movement (*MoveJ*). Several *targets* have a *zone* distance defined allowing a smoother trajectory on approach to the *target*.

The *zone* concept has also been employed by Lloyd and Hayward [70] where they refer to this as a *blend* between two trajectories and similarly by Macfarlane [26] as *tightness* around a quintic spline control point. A large *zone* allows a smoother, and therefore faster, path to be followed where precision movements are not required. A small, or even a zero zone, is used whenever the *target* point must be reached accurately. Figure 4.3 includes zone definitions around two of the targets, thereby facilitating a smooth arc movement on approach to the targets.

When performing a movement, the manipulator's end-effector, or gripper, may be required to move at a speed that is less than the maximum potential produced by the actuators. This may be due to limitations in the gripper's ability to hold an object or because the movement may have to cooperate with a task outside of the manipulator itself, for example, conveyor tracking. The *MaxTCPSpeed* allows the speed of the end-effector to be limited if required.

When performing pick-and-place tasks there is usually a pause at the 'pick' and 'place' positions to allow the gripper to take hold of or release the object. In industrial applications this takes the form of either a set pause time, or by waiting for a feedback signal from the system that the transition has successfully taken place. As this project is considering the manipulator in isolation from any peripheral feedback system, only the time based pause is considered. This is an optional parameter on any move command. When included, the trajectory is formed with the manipulator decelerating to a stationary position at the destination *target*. When omitted, the *target* is treated as a 'fly-by' point with continuous end-effector velocity and acceleration maintained through the *target*.

4.1.2 Formulate Knots

Although the *targets* represent the general path the end-effector must follow, the inclusion of the *MoveType*, *zone* and *pause* data associated with moving to these targets, transforms the path. This altered path will follow the targets approximately, but will do so in a way that is most efficient for the actuators yet still satisfies these movement constraints.

Knots are essentially control points for the fitment of a spline. At this point in the trajectory planning process these points are still defined in Cartesian space, but will be transformed later into joint space to allow the actual fitment of the cubic spline trajectories. If any move commands have a pause associated with them, then the set of movement commands on either side of the paused target are considered independent of each other and will be fitted with separate piecewise splines.

Additional to being a coordinate in space, knots also contain properties allowing the specification of an angular velocity. The trajectory planning process sets the manipulator's velocity to zero at the first and last targets in each set of movement command sequences. This ensures the manipulator is stationary when it reaches the final knot in a particular move sequence.

Knots are created along the path in order to provide control points for fitting a spline. These knots are formed by fitting a straight line between targets, where the line intersects the zone of the next target a knot is placed. This is shown in Figure 4.4. The path moves from left to right, with knots being placed on the approaching side of a target's zone.

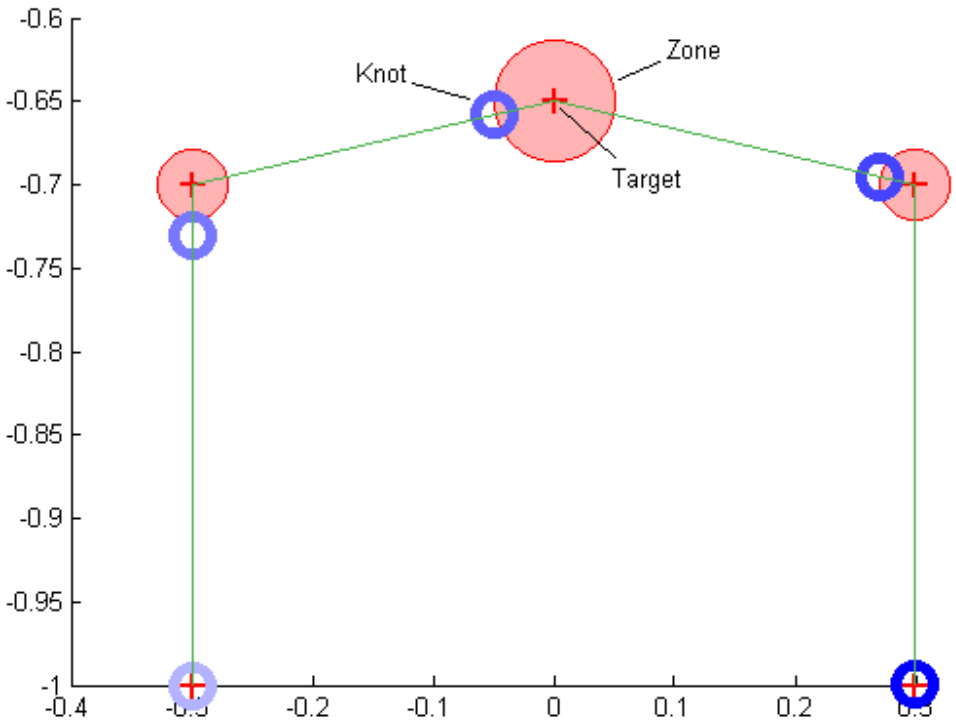

Figure 4.4 Generation of knots by taking a straight line between targets. Where the line intersects with the zone a knot is formed.

As well as the example presented in Figure 4.4, additional knots are also introduced along *linear* movements to ensure a near-linear trajectory is achieved by the end-effector. How this is achieved is discussed in Section 4.2.

4.1.3 Cartesian to Joint Space Conversion

The movement commands define the path in Cartesian space because this is more intuitive to the robotic programmer. However, the manipulator is better controlled in the joint space as this allows a trajectory that is smooth and optimal for the motor actuators to be formed. Therefore the *knots* that were defined in Cartesian space are converted to joint space using the inverse kinematic equations developed in Section 3.1.2. Figure 4.5 highlights the motor positions required at each of the *knots*.

From this point on in the trajectory planning process, the joint space becomes the standard frame of reference. The Cartesian space is only used again to validate the end-effector's speed and ensure that it is under the *MaxTCPSpeed* prescribed for each movement.

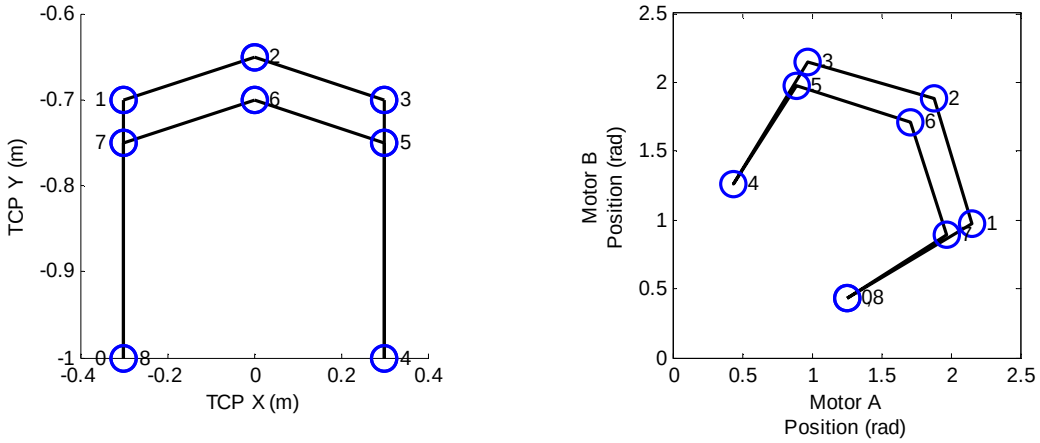

Figure 4.5 Knots defined in Cartesian space (left) are converted into joint space coordinates (right). Numbering indicates order of knots.

4.1.4 Cubic Spline Fitment

With the knots defined in joint space, splines can be fitted between them to form a smooth trajectory for each of the motors. These splines define the motion of the joint/motor with respect to time. A number of researchers have used piecewise cubic polynomials to form a smooth trajectory [23][33][69]. Cubic polynomials allow for continuous velocity and acceleration throughout the path.

In order for time-dependent splines to be fitted between the *knots*, a time value must be assigned to each knot. The exact time-spacing between knots is not crucial at this stage in the trajectory planning, however, the path-time (time travelled along the path) must continually increase at each *knot* in the order they are to be traversed through. In the optimization step of the trajectory planner the time between knots will be altered to achieve the shortest overall cycle-time. It is, however, helpful if the path-time at each knot is approximated to begin with. This is accomplished by considering the distance travelled between knots by the TCP in Cartesian space, and dividing it by the maximum velocity allowed for that move (as defined earlier in Section 4.1.1 on movement commands). This is outlined in Equation (4.1).

$$t_{K_i} = t_{K_{i-1}} + \frac{d_{K_i - K_{i-1}}}{v_{max_{K_i}}} \tag{4.1}$$

where $v_{max_{K_i}}$ is the maximum TCP velocity allowed while travelling to knot K_i, $d_{K_i - K_{i-1}}$ is the Cartesian distance between the previous knot, K_{i-1}, and the destination knot, K_i. This is shown in Equation (4.2).

$$d_{K_i - K_{i-1}} = \sqrt{\left(x_{K_i} - x_{K_{i-1}}\right)^2 + \left(y_{K_i} - y_{K_{i-1}}\right)^2} \tag{4.2}$$

If there are *k knots* in the movement, including the starting and finishing *knot*, there will be *k-1* segments for which a cubic polynomial must be fitted to each. To ensure the path is smooth over the entire movement, the 3rd order polynomials describing the position of each motor must be such that their 2nd order derivative (acceleration profile) be continuous where the segments are joined together at the knots. Figure 4.6 shows the position, velocity and acceleration profiles of a path where matching the derivatives of the cubic polynomial have been neglected and is only continuous in the position and velocity aspect of the profile. This can be contrasted with Figure 4.7, where the cubic polynomials describing the position of the motor are continually differentiable to the 2nd order of acceleration.

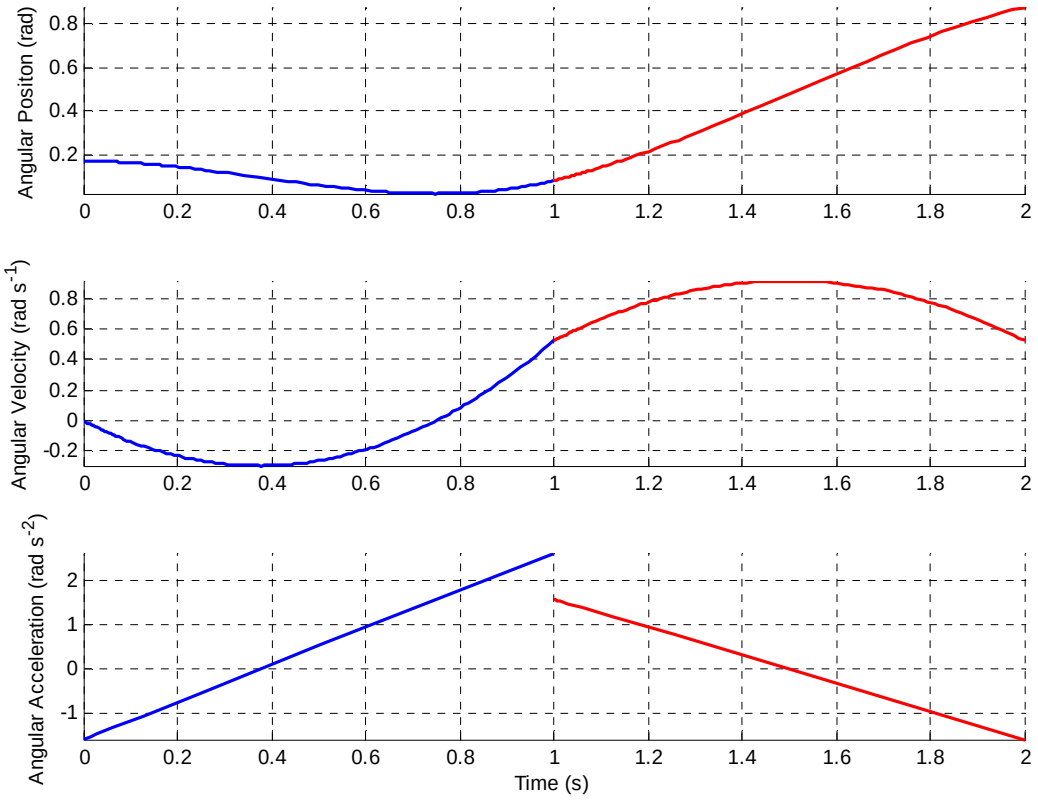

Figure 4.6 Position, velocity and acceleration of a discontinuous profile formed by two piecewise cubic polynomials between three knots.

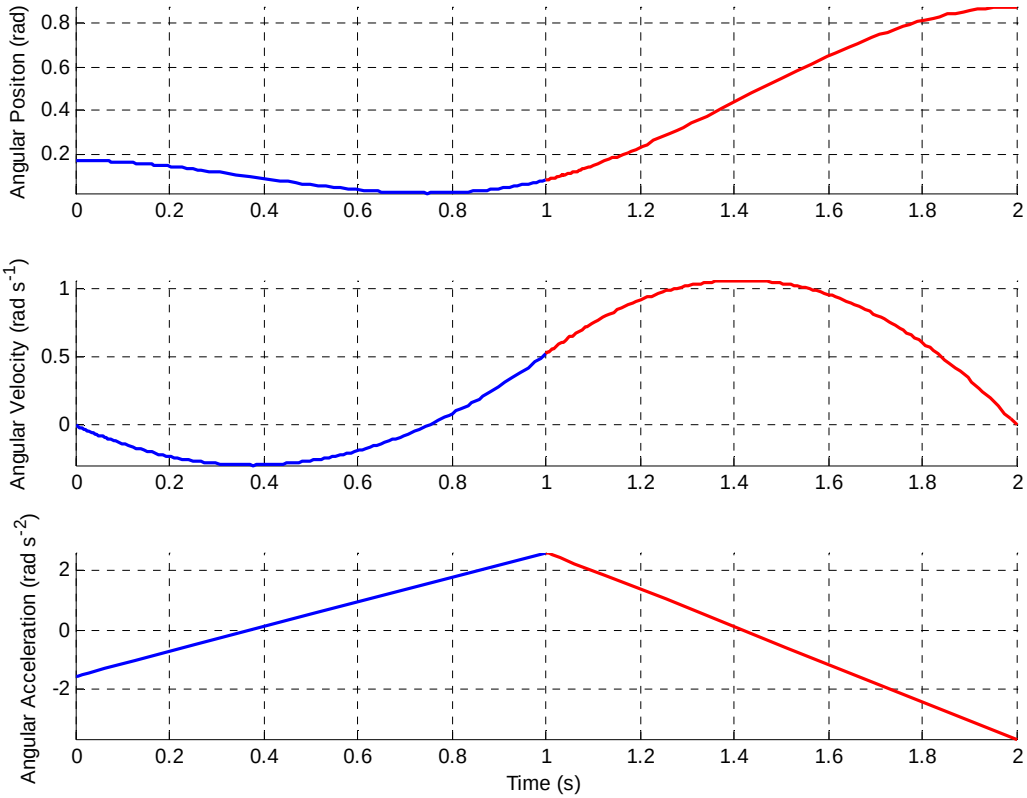

Figure 4.7 Position, velocity and acceleration of a continuous profile formed by two piecewise cubic polynomials between three knots.

To ensure the cubic splines have C^2 continuity, that is, the 2nd order derivative is continuous at the *knots*, several constraints must be placed on finding the coefficients to the cubic polynomials. Equation (4.3) is the general form for the time dependent cubic polynomial describing the angular position of the motor, θ. The more general form is shown in Equation (4.4), where the time at the previous *knot*, t_{Ki-1}, is subtracted from the current path-time, t, to get the time since the previous *knot*. There are (k-1) polynomials to represent each of the path segments between the *knots*.

$$\theta_i(t) = a_0^{(i)} + a_1^{(i)}(t) + a_2^{(i)}(t)^2 + a_3^{(i)}(t)^3 \tag{4.3}$$

$$\theta_i(t) = a_0^{(i)} + a_1^{(i)}\left(t - t_{K_{i-1}}\right) + a_2^{(i)}\left(t - t_{K_{i-1}}\right)^2 + a_3^{(i)}\left(t - t_{K_{i-1}}\right)^3 \tag{4.4}$$

for $t_{K_{i-1}} \leq t \geq t_{K_i}$

As there exist 4 unknown coefficients for each of the *k*-1 polynomials, 4(*k*-1) equations using these unknowns are needed to find the coefficients. These equations can be obtained through the constraints on the system. Equations (4.5) and (4.6) state that the starting position is known and the initial angular velocity is zero. The position of the motor at the end of a given segment, *i*-1, must be equal to the position obtained from the start of the next segment, *i*, as shown in Equations (4.7) and (4.8). Equation (4.9) states that the velocity at the end of a segment must be the same as the velocity at the following segment. Similarly the acceleration profiles between segments must be continuous at each knot as shown in Equation (4.10). Equations (4.11) and (4.12) are similar to the starting conditions in that they constrain the final position to that which is known and the final velocity to zero.

$$a_0^{(1)} = \theta(0) = \theta_{K_0} \tag{4.5}$$

$$a_1^{(1)} = \omega(0) = \omega_{K_0} = 0 \tag{4.6}$$

$$a_0^{(i)} + a_1^{(i)}\left(t_{K_i} - t_{K_{i-1}}\right) + a_2^{(i)}\left(t_{K_i} - t_{K_{i-1}}\right)^2 + a_3^{(i)}\left(t_{K_i} - t_{K_{i-1}}\right)^3 = \theta\left(t_{K_i}\right) \tag{4.7}$$

$$a_0^{(i+1)} = \theta\left(t_{K_i}\right) \tag{4.8}$$

$$a_1^{(i)} + 2a_2^{(i)}\left(t_{K_{i-1}}\right) + 3a_3^{(i)}\left(t_{K_{i-1}}\right)^2 = a_1^{(i+1)}(t_{K_i}) \tag{4.9}$$

$$2a_2^{(i)} + 6a_3^{(i)}\left(t_{K_{i-1}}\right) = 2a_2^{(i+1)}(t_{K_i}) \tag{4.10}$$

$$a_0^{(i_f)} + a_1^{(i_f)}\left(t_{K_{i_f}}\right) + a_2^{(i_f)}\left(t_{K_{i_f}}\right)^2 + a_3^{(i_f)}\left(t_{K_{i_f}}\right)^3 = \theta(t_{K_{i_f}}) \tag{4.11}$$

$$a_1^{(i_f)} + 2a_2^{(i_f)}\left(t_{K_{i_f}}\right) + 3a_3^{(i_f)}\left(t_{K_{i_f}}\right)^2 = \omega\left(t_{K_{i_f}}\right) = 0 \tag{4.12}$$

When the constraint equations are solved simultaneously, as shown in Equation (4.13), the polynomial coefficients are resolved. Equation (4.13) shows the systems of equations for solving 2 splines between three knots. θ_0 and ω_0 are the position and velocity constraints of the first *knot*. θ_1, ω_1 and α_1 are the position, velocity and acceleration of the spline at the middle *knot*. θ_2 and ω_2 are the position and velocity constraints of the final *knot*. t_0, t_1 and t_2 are the time values at each of the *knots*. This example is easily expanded out to accommodate more *knots*. The result is a set of (*k*-1) 3rd order polynomials to describe the path travelled between *k knots*. This set of cubic polynomials ensures continuous velocity and acceleration over the entire path, with stationary starting and ending points.

$$\begin{bmatrix} 1 & t_0 & t_0^2 & t_0^3 & 0 & 0 & 0 & 0 \\ 0 & 1 & 2t_0 & 3t_0^2 & 0 & 0 & 0 & 0 \\ 1 & t_1 & t_1^2 & t_1^3 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 1 & 2t_1 & 3t_1^2 & 0 & -1 & 0 & 0 \\ 0 & 0 & 2 & 6t_1 & 0 & 0 & -2 & 0 \\ 0 & 0 & 0 & 0 & 1 & t_2 & t_2^2 & t_2^3 \\ 0 & 0 & 0 & 0 & 0 & 1 & 2t_2 & 3t_2^2 \end{bmatrix} \begin{bmatrix} a_0^1 \\ a_1^1 \\ a_2^1 \\ a_3^1 \\ a_0^2 \\ a_1^2 \\ a_2^2 \\ a_3^2 \end{bmatrix} = \begin{bmatrix} \theta_0 \\ \omega_0 \\ \theta_1 \\ \theta_1 \\ \omega_1 \\ \alpha_1 \\ \theta_2 \\ \omega_2 \end{bmatrix} = \begin{bmatrix} \theta_0 \\ 0 \\ \theta_1 \\ \theta_1 \\ \omega_1 \\ \alpha_1 \\ \theta_2 \\ 0 \end{bmatrix} \tag{4.13}$$

Figure 4.8 shows the position, velocity and acceleration profiles of a sample motor's cycle-path. The individual cubic splines are highlighted by plotting adjacent splines with alternating colours. There is a break of 0.2 seconds in the middle of the profile to account for a 'pick' or 'place' action to occur. The motor profile is stationary during this time.

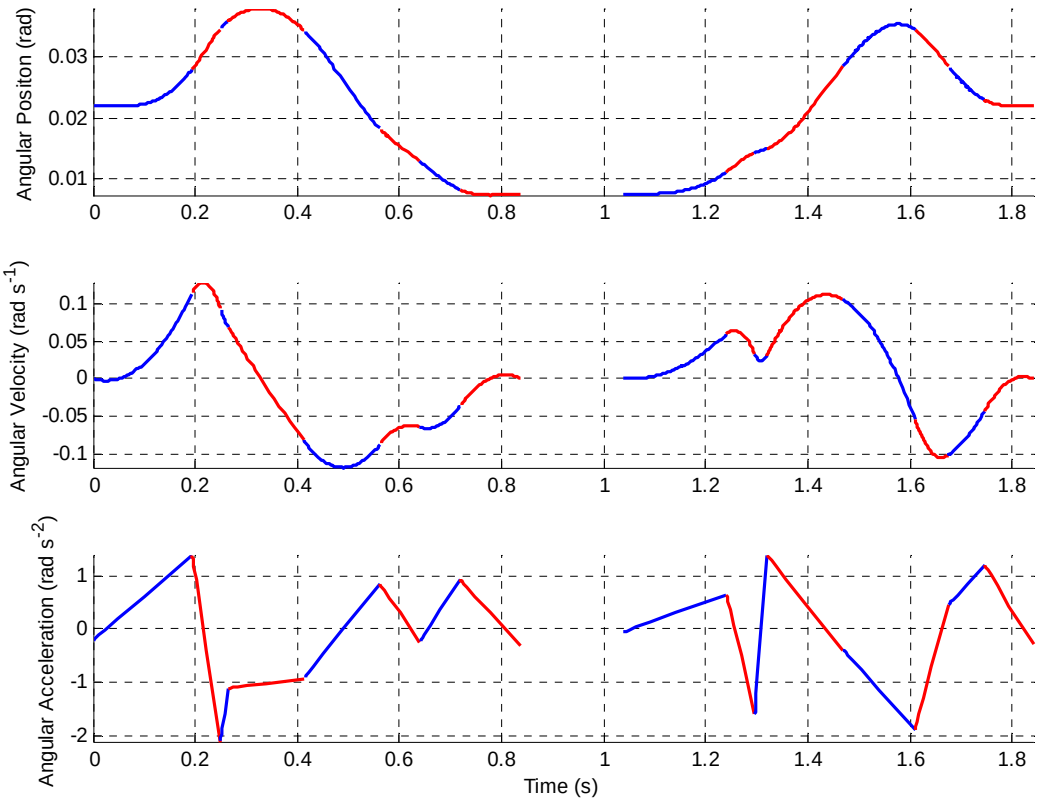

Figure 4.8 Position, velocity and acceleration of a continuous profile formed by piecewise cubic polynomials. Alternating colours differentiate individual polynomials.

4.1.5 Validation against Constraints

Once a trajectory has been developed in the form of a cubic polynomial, it is then validated against a set of constraints. These constraints, listed in Table 4.2, cover the limitations of the motor's angular velocity, acceleration, jerk and torque, as well as limitations placed on the end-effector's TCP velocity. This process must be done to ensure the trajectories developed do not exceed the capabilities of the manipulator. The trajectory can also be evaluated to see if it is maximising its capabilities throughout each path segment. If a path segment either exceeds the capabilities of the manipulator or does not come close enough to maximizing the performance available, the path is modified as shown in the next Section, 4.1.6.

Table 4.2 Constraints on trajectories.

Minimum Value	Parameter Evaluated	Maximum Value
$-\omega_{max}$	ω (Motor Angular Velocity)	ω_{max}
$-\alpha_{max}$	α (Motor Angular Acceleration)	α_{max}
$-J_{max}$	J (Motor Angular Jerk)	J_{max}
$-\tau_{max}$	$\tau_{estimated}$ (Estimated Motor Torque)	τ_{max}
	$\lvert TCP_{velocity} \rvert$ (TCP speed)	TCP_{max_speed}

While the motor's velocity, acceleration and jerk can be obtained directly from the cubic polynomial trajectory description, and the TCP speed can be known through inverse kinematics, the torque requirements of the motor are not as easily resolved. This is due to the highly coupled non-linear dynamics found in parallel mechanisms. The torque required of one motor is dependent on the torque provided by the other motor. While it is possible to obtain the torque requirements, the complex mathematics involved does not lend itself well to the task of trajectory planning. Instead, a simple estimate is made using some assumptions about the system's mechanics. These assumptions are listed below:

- The gripper's mass and inertia properties are assumed to include the mass and inertia of any carried load.
- The mass of the proximal arms are located as point masses about the centre of the length of the proximal arms.
- The mass of the distal arms are located at the point where they attach to the proximal arms.
- Half of the mass of the gripper and half of the mass of the distal crank are carried by either proximal arm and are located as point masses at the end of the proximal arms.
- The mass of the proximal stabilizing arm is located as a point mass about the centre of the length of the 'B' proximal arm.
- The mass of the distal stabilizing arm is located as a point mass at the end of the 'B' proximal arm.
- The mass of the proximal crank is located as a point mass at the end of the 'B' proximal arm.

These assumptions are highlighted in diagrammatic form in Figure 4.9.

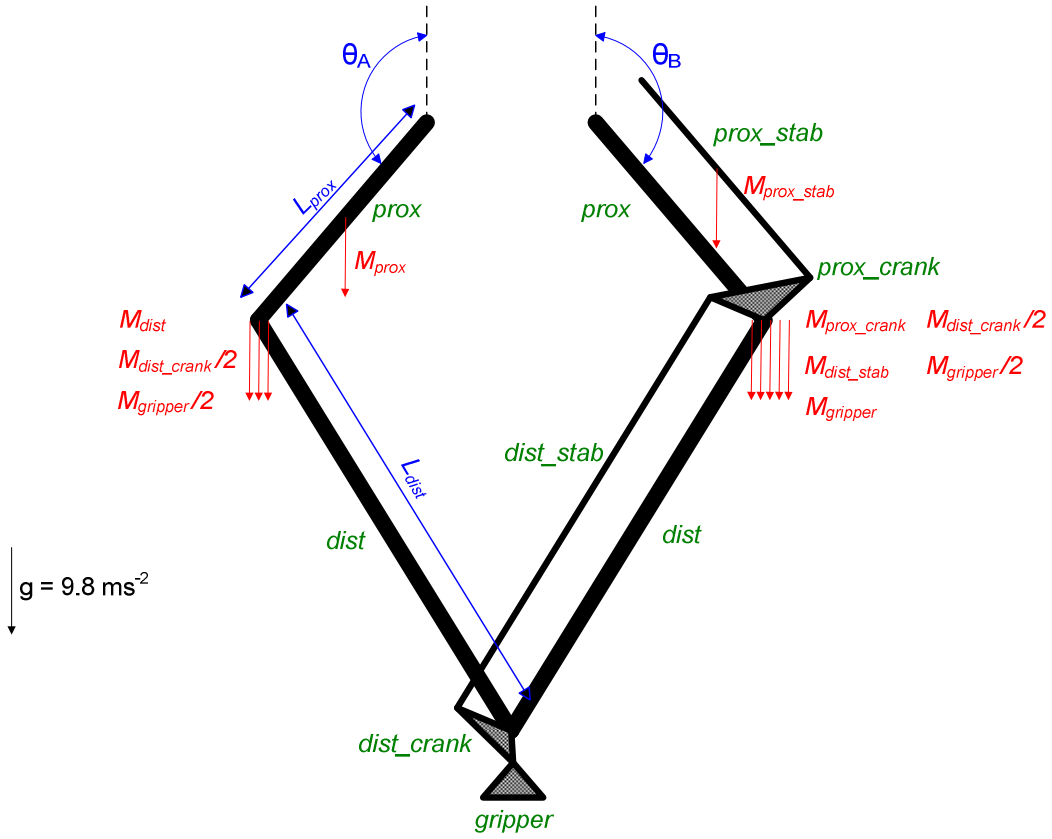

Figure 4.9 Diagramtic view of the assumptions made for torque estimation. Red components represent locations of point mass'. Blue represent distance components. Green labels the manipulators components.

The equations used to estimate the torque required from each motor are represented in (4.14) through to (4.19). The calculations for the 'A' and 'B' motors are different due to the inclusion of the stabilizing arms alongside the 'B' proximal and distal arms.

The moment of inertia coupled to motor 'A' can be estimated as:

$$I_A = M_{prox} * \left(\frac{L_{prox}}{2}\right)^2 + \left(M_{dist} + \frac{M_{dist\_crank}}{2} + \frac{M_{gripper}}{2}\right) * L_{prox}^2 \tag{4.14}$$

where M_{prox} and L_{prox} are the mass and length of the proximal arm, M_{dist} is the mass of the distal arm, M_{dist_crank} is the mass of the crank arm connected at the base of the distal arm, and $M_{gripper}$ is the mass of the gripper. The torque acting on motor 'A' due to gravity is estimated as:

$$T_{A\_gravity} = M_{prox} * \left(\frac{L_{prox}}{2}\right) * \sin(\theta_A) * g + \left(M_{dist} + \frac{M_{dist_{crank}}}{2} + \frac{M_{gripper}}{2}\right) * L_{prox} * \sin(\theta_A) * g \quad (4.15)$$

where θ_A is the angle of the proximal arm from vertical, g is the gravity vector of 9.8 ms^{-2}. By adding the torque due to gravity to the torque due to the angular acceleration of the inertia of the arms (α_A), the overall torque requirement can be calculated as:

$$T_{A\_total} = I_A * \alpha_A + T_{A\_gravity} \quad (4.16)$$

The moment of inertia coupled to motor 'B' can be estimated as:

$$I_B = \left(M_{prox} + M_{prox\_stab}\right) * \left(\frac{L_{prox}}{2}\right)^2 + \left(M_{dist} + M_{dist\_stab} + M_{prox\_crank} + \frac{M_{dist\_crank}}{2} + \frac{M_{gripper}}{2}\right) * L_{prox}^2 \quad (4.17)$$

where M_{prox_stab} and M_{dist_stab} are the masses of the proximal and distal stabiliser arms, M_{prox_crank} is the mass of the crank arm connected at the end of the proximal 'B' arm. The torque acting on motor 'B' due to gravity is estimated as:

$$T_{B\_gravity} = \left(M_{prox} + M_{prox\_stab}\right) * \left(\frac{L_{prox}}{2}\right) * \sin(\theta_B) * g + \left(M_{dist} + M_{dist\_stab} + M_{prox\_crank} + \frac{M_{dist\_crank}}{2} + \frac{M_{gripper}}{2}\right) * L_{prox} * \sin(\theta_B) * g \quad (4.18)$$

where θ_B is the angle of the proximal arm from vertical. By adding the torque due to gravity to the torque due to the angular acceleration of the inertia of the arms (α_B), the overall torque requirement can be calculated as:

$$T_{B\_total} = I_B * \alpha_B + T_{B\_gravity} \quad (4.19)$$

The accuracy of the torque estimation can be seen in Figure 4.10 where it is compared to the simulation results from SimMechanics™ for a sample trajectory. The estimated torque tracks a similar profile to the SimMechanics™ torque calculation, but does vary, particularly during large peaks in the graph. The

greatest weakness of the estimation method is when considering the mass of the components near the end-effector. By assuming these masses to be located at the ends of the proximal arms, an error is introduced. This error grows as the relative mass of the end-effector increases. The example shown in Figure 4.10 uses a heavy 35 kg gripper head. The difference in the motors' torque profiles results in a positional error of the end-effector of 0.029 m by the end of the path. However, if the gripper was to only weigh 5 kg the torque error would be less, as shown in Figure 4.11. In this case, the positional error of the end-effector shrinks to 0.006 m. Therefore, for the purposes of this project, the method of estimating motor torques is valid.

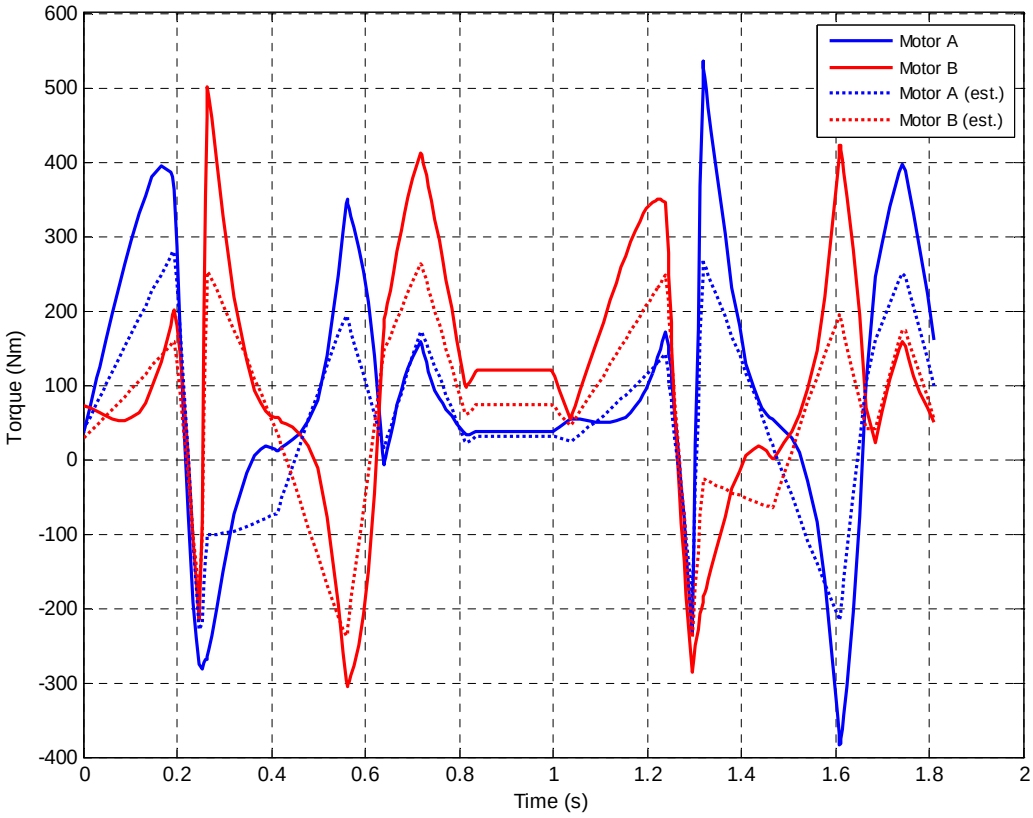

Figure 4.10 Estimated torque profiles compared to SimMechanics™ calculated torque profiles. 35 kg gripper used.

The torque estimation method has a tendency to underestimate the actual torque required, as gathered from the SimMechanics™ simulation. Therefore, when maximising a configuration's path cycle-time, the available torque of the motors is reduced slightly (~10 %) to account for the underestimation.

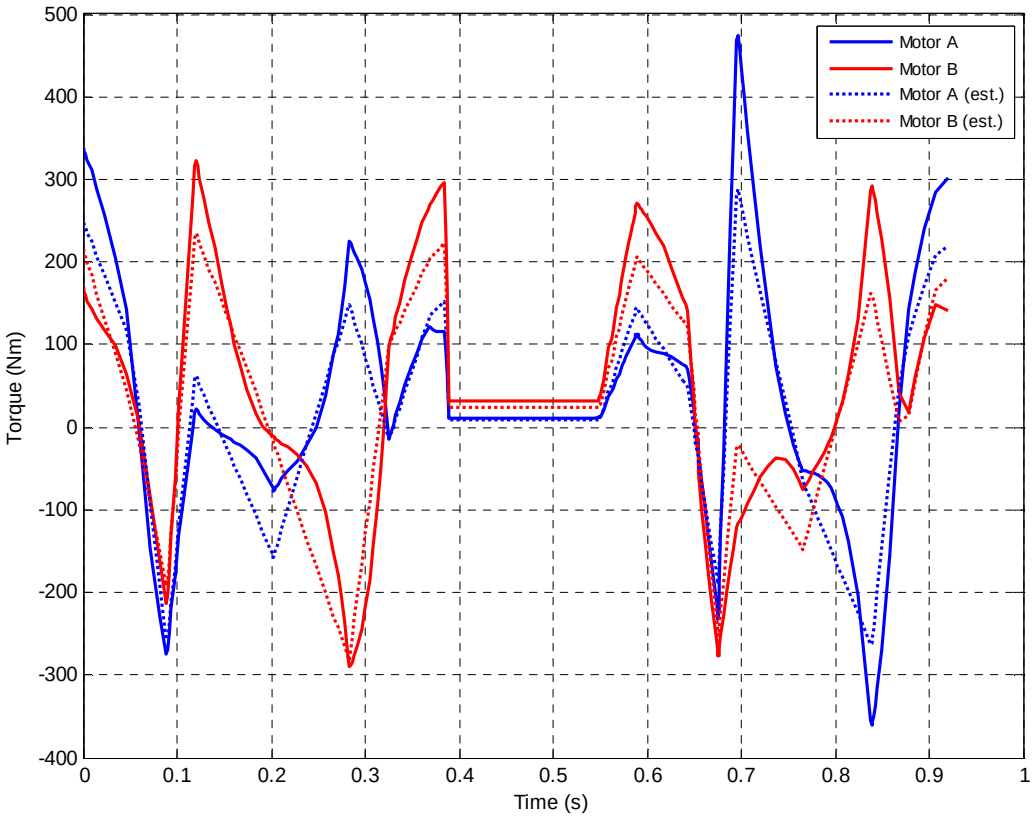

Figure 4.11 Estimated torque profiles compared to SimMechanics™ calculated torque profiles. 5 kg gripper used.

4.1.6 Altering Time Segments

As mentioned in the previous section, the time segments between knots on a path are increased or decreased depending on how the trajectory compares to the motor and end-effector constraints. If a parameter exceeds a constraint during the segment, the time segment is increased and the trajectory is recalculated from the cubic spline fitment stage in Section 4.1.4. Similarly, if no parameter is close to the constraint, the time segment can be shortened and a new spline fitted. Two variables were found heuristically for this problem. These include the amount by which the time segment is expanded or contracted, and how close a parameter must be to its constrained limit in order for it to be considered 'maximised'.

The method used is based on Nelder and Meads flexible polyhedron search method [34]. This iterative optimization approach has been used as a technique for altering time segments in piecewise polynomial trajectory planning by several researchers [23][33][69]. The algorithm that was finally used to analyse and alter the time segments is shown below in Figure 4.12. The Matlab® implementation of this procedure can be found in Appendix F (refer Figure F.46 through to Figure F.54).

```
procedure optimise time segments
begin
    let
    ς be the set of constraints
    ρ be the values of the constrained parameters
    д is an ageing factor (~100)
    Ψ is the initial acceptance threshold <1 (~0.8)

    initialise time segments t
    n = 1

    while ∃ t !optimised do
        plan path
        for each path segment ί do
            check constraints
            if ∃ ρί > ςί then
                find (ρί/ςί)max
                increase ti by (ρί/ςί)max
            else if ∄ (ρί/ςί) > Ψ(n+д)/д then
                find (ρί/ςί)max
                reduce ti by (ρί/ςί)max
            end
        end
        n = n + 1
    end
end
```

Figure 4.12 Pseudo code for optimising the time segments between knots on a path.

Each path segment is analysed separately. The performance of the path segments are compared to the constraints. A segment must never be too short as to allow a constraint to be violated, but determining how close the path segment can be to that limit is not easily achieved. A threshold, Ψ, is required to establish when a time segment is near enough to optimal. Through experimentation, an initial threshold value of 80 % has been found to provide a fast converging and near optimal time for each segment. That is, at least one of the parameters must be within 80 % of its constraint. Table 4.3 shows several iterations of the expansion and retraction of a sample path's time segments. For the path with 5 knots (4 segments), 6 iterations were required until a near-time-optimal path was found. To further expedite the optimization process, the threshold is lowered as a function of the number of iterations. This is shown in the pseudo code Figure 4.12, where an ageing factor д exponentially weakens the threshold.

Table 4.3 Time values for 4 path segments (between 5 knots) over 6 optimisation iterations.

Iteration	Segment 1	Segment 2	Segment 3	Segment 4
1	0.027	0.028	0.032	0.033
2	0.574	0.359	0.442	0.614
3	0.383	0.226	0.276	0.416
4	0.278	0.148	0.180	0.309
5	0.229	0.108	0.129	0.262
6	0.229	0.092	0.106	0.262

All actuated joint trajectories must be optimised simultaneously. In the case of the 2DOFPPM, both motor trajectories need to be considered concurrently. This is to ensure that both motors reach each knot at the same time. Therefore, when altering time segments, the parameters and constraints of both motors need to be considered before deciding how much to increase or decrease the time period.

4.1.7 Storing of Path Data

Once the trajectories have been optimized and a set of cubic polynomials have been obtained, the data needs to be structured in a way suitable for the SimMechanics™ simulation environment to process. This requires the input data to be stored in a file. The file is formatted the following way. The first row of cells contains time values. The second, third and fourth rows contain motor position, velocity and acceleration values at the corresponding time values. For the purposes of this project, taking recordings of data at 50 ms time intervals has proved accurate enough for the purposes of evaluating the kinematics and dynamics of the system.

4.2 Interpolation of Knots for Linear Movements

In order to control the end-effector along a linear movement, a unique method has been developed that still allows the use of cubic splines defined in joint space. By placing extra knots along the straight line between two targets the path is constrained to pass through each of those knots. Experimentation was carried out to determine the effect of additional knots on the linearity of the path travelled and the length of time taken to perform the move. In the following graphical examples it is assumed that a *linear* move is desired for the vertical ‘pick’ or ‘place’ actions and a *joint* move used for the ‘horizontal’ transition above the ‘pick’ and ‘place’ points.

Figure 4.13 does not include any additional knots during the vertical movements. This results in a trajectory being formed that does not represent a *linear* movement. The movement is the fastest possible path, within the constraints of the manipulator, which passes through each point, starting and finishing with zero velocity. This results in a benchmark cycle-time of 0.690 s.

If a single additional knot is introduced halfway along the *linear* movements, the trajectory becomes significantly closer to the desired path as shown in Figure 4.14. By introducing a single knot on each of the two linear movements, the cycle-time increases slightly to 0.716 s.

Having seen the benefits of adding a single extra knot to the *linear* movements, it is logical to enquire about the effects of adding multiple knots. Figure 4.15 shows a path with five additional knots along the *linear* moves. This results in a path with highly linear vertical movements but at the cost of raising the cycle-time to 1.033 s. This significant increase in cycle-time is detrimental to the overall performance of the manipulator. By introducing too many knots along the path, the motor joints are forced to switch direction frequently. As the system is maximizing the torque capabilities of the motors, the motors' torque step response may not be adequate to switch fast enough from maximum torque in one direction to maximum torque in the opposite direction. Therefore, introducing too many knots is seen as detrimental to the performance of the manipulator and a middle ground should be found that provides a suitably linear movement in a fast time that can be tracked by the motor joints.

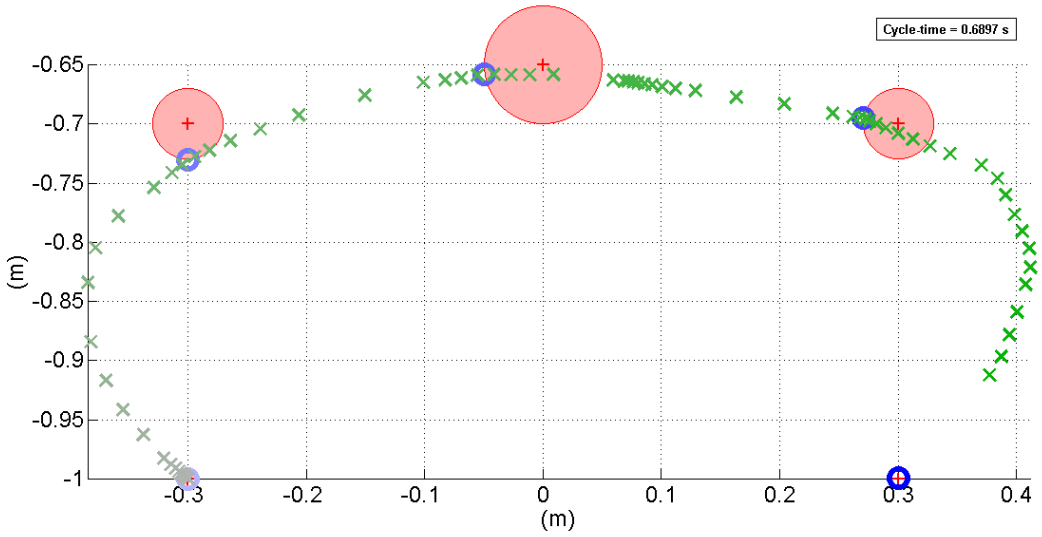

Figure 4.13 Trajectory with no linear constraints

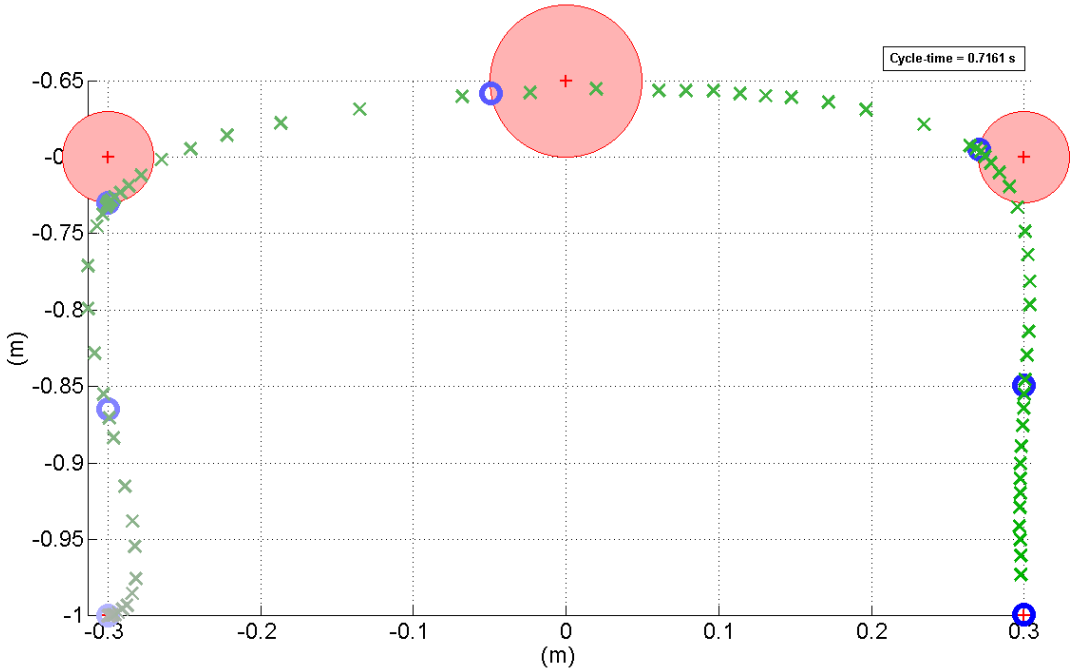

Figure 4.14 Trajectory with a single additional knot for linear movements

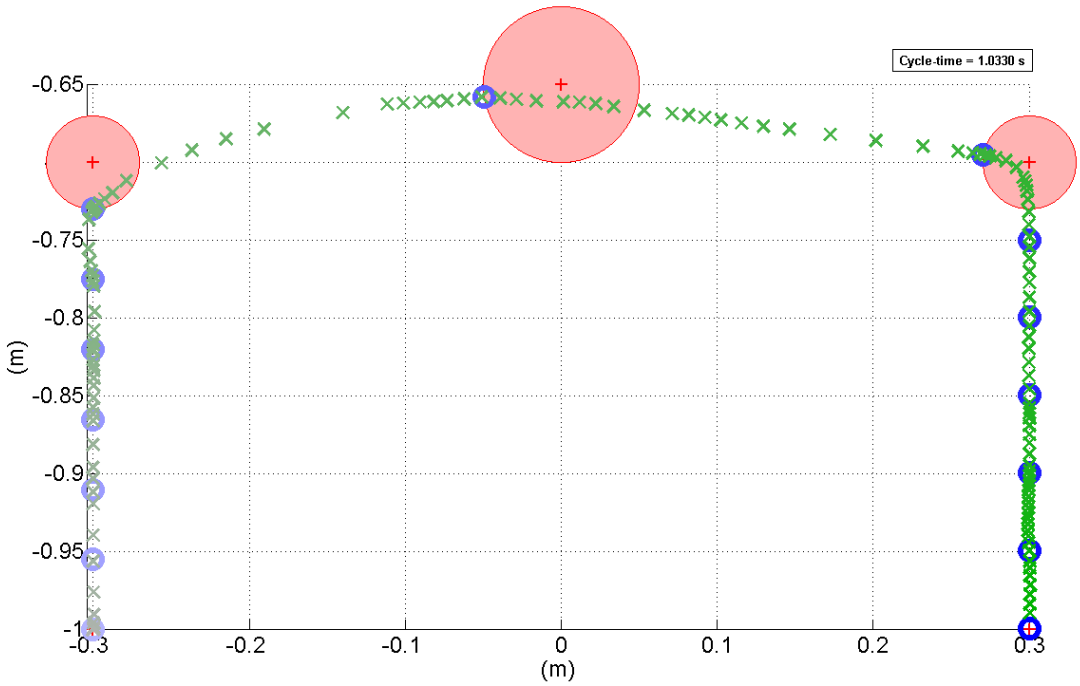

Figure 4.15 Trajectory with many additional knots for linear movement

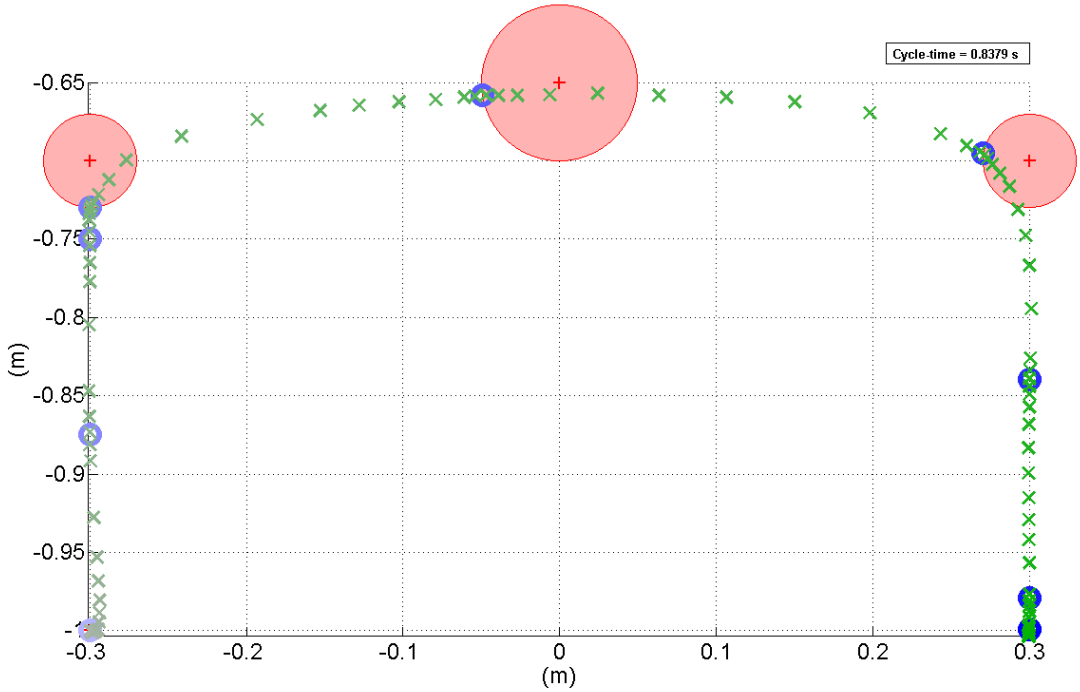

Figure 4.16 Trajectory with an additional knot halfway through linear movement and another positioned close to destination target

While introducing a single knot in Figure 4.14 improved the linearity of the movement compared to having no additional knots (Figure 4.13), it still deviates from the linear path somewhat. A solution was found by introducing a second knot near the knot at the destination target. As shown in Figure 4.16, this minimized the straight line divergence while keeping the cycle-time to 0.8379 s. This method was formulated by the inclusion of two additional parameters to a path's definition, *LinearErrorFactor* and *LastLinearTargetDistance*.

The *LinearErrorFactor* is a value, measured in metres, representing how far along a linear movement a knot must be placed. For the example path shown, a value of 0.2 m was used. This means that for a linear move of less than 0.2 m, no additional knot would be introduced. For a linear move of 0.5 m, two additional knots would be included.

The *LastLinearTargetDistance*, also measured in metres, represents the distance back from the knot at the edge of the destination target. In the example a value of 0.02 m was used. Therefore a knot was placed at the destination target of the linear move, and another placed a further 0.02 m back along the path.

As with any method of fitting splines in joint space, problems can occur when considered from the end-effector's point of view in Cartesian space. Figure 4.17 presents a path where the *LastLinearTargetDistance* is too small, causing the cubic spline fitments to result in a path that 'loops' back on itself. The trajectory planning method found the fastest trajectory for passing through both knots, within the constraints of the motors, to be a looping action. This occurred due to the knots being spaced too close together. This highlights a possible deficiency in the algorithm as trajectories like this are undesirable. It is therefore important to check parameters, like *LastLinearTargetDistance,* and view the simulated path to ensure the final path is valid.

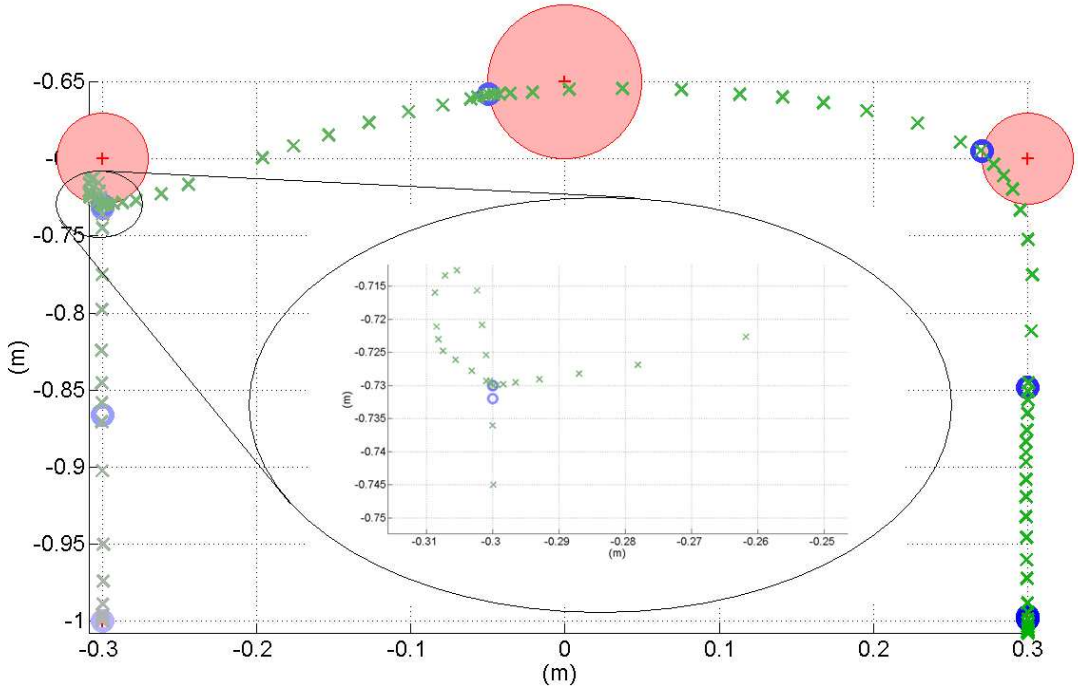

Figure 4.17 An example of the problem caused by fitment of the splines in joint space resulting in the Cartesian path looping back on itself.

4.3 B-splines, 3rd, 5th and Higher Order Polynomial Fitting

When deciding on the type of spline fitting method to be used, a number of options were considered. This section briefly details the options of using B-splines and a range of polynomials to fit between the knots.

The use of B-splines as an interpolation path planning method was popular in the past [37][38]. This was due to their easy and fast mathematical manipulation. However, B-splines do not actually pass through the control points (knots), but rather are 'pulled' towards them as shown in Figure 4.18. It is for this

reason that they are not commonly used now. The failure of a trajectory to pass through specific points in the path, renders it unsuitable for many applications.

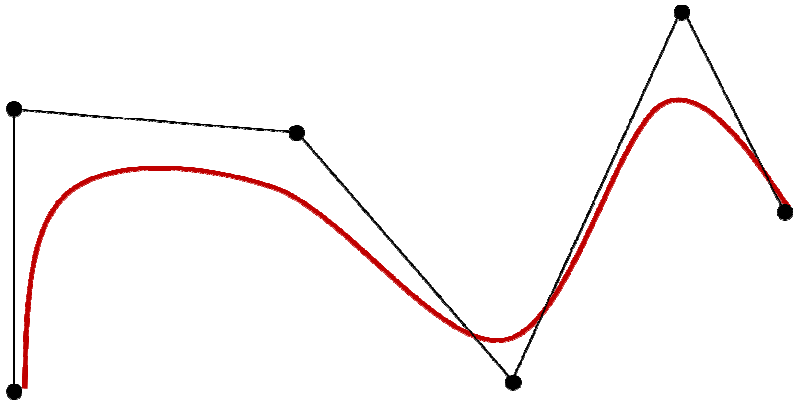

Figure 4.18 B-spline example. The red spline is 'pulled' towards the black control points.

Polynomial spline fitting between knots is a commonly used approach to trajectory planning [23][30][33][36][52][54][69][71] . The 3rd order polynomials, as used in this thesis, are the simplest of the polynomial methods. Interpolating cubic polynomials between knots allows the trajectory to be continuous in both velocity and acceleration. However, it is sometimes desirable to also be continuous in the jerk component of the trajectory [26]. This is where the 5th order, or quintic, polynomials are used. Additional constraints are put on the polynomials to allow a solution to be found. Starting and final accelerations are set to zero, as well as the constraint of continuous jerk between adjacent splines. The introduction of the extra coefficients and constraints with quintic polynomial increases the computation required, but allows for the path to be constrained within the motors' jerk limits.

An alternative method is to use a single high-order polynomial that passes through all the knots. If there are *k* knots, with constraints placed on the initial and final velocities, the polynomial must be of order *k+1* with *k+2* coefficients. A path consisting of 3 knots therefore is represented by a 4th order polynomial providing continuous velocity, acceleration and jerk. A path consisting of 4 knots is represented by a 5th order polynomial providing continuous velocity, acceleration, jerk and snap (that is, the 4th derivative of position). It can be easily seen that with a path consisting of a large number of knots, the polynomial has a large number of unknown coefficients. Solving these coefficients becomes increasingly challenging and time consuming. Therefore, using the method of a single high-order polynomial is not viable in a trajectory planning system where an unknown, and potentially large, number of knots exist.

As mentioned earlier in this chapter, the exact trajectory planning methodology is not important provided it allows a fair and realistic comparison between different manipulator configurations. This means that a trajectory must represent the capabilities of the manipulator accurately while taking into account motor constraints. As with any simulation, there will always be short-comings when compared to reality. The use of quintic and higher-order polynomials have the advantage of increased model fidelity over the cubic polynomial method, but at the cost of extra computation. The cubic polynomial fitting method was chosen as it provided a trajectory constrained to a high enough degree of accuracy for comparing manipulator configurations, while being easily computed.

4.4 Managing Discontinuous Jerk

The comparison of cubic polynomial spline fitting to higher-order polynomial fitting showed limitations in the fidelity of the trajectory produced. The cubic polynomials resulted in discontinuities of jerk at the knots. Motors are not able to produce the instantaneous change in jerk or have an infinite torque step-response. A number of researchers have successfully used piecewise cubic polynomials for trajectory planning in industrial manipulators [23][33][52][69]. Despite these researchers being satisfied with the performance, an experiment was conducted to see the variation in end-effector trajectory if the joint trajectories were subjected to a low pass filter. To do this, the acceleration profile was put through a low-pass Butterworth filter. The resulting signal was then integrated using the trapezoidal numerical method to achieve the new velocity profile, and integrated again to produce the new filtered position profile. Filtering out the high frequencies within the signal in this manner, effectively places limits on the jerk and higher derivates of motor position.

Figure 4.19 presents the position, velocity and acceleration profiles for the two motors before and after the filtering and integration process. The acceleration profile has had the sharp changes smoothed which better represents the capabilities of a real motor. As can be seen in the position profile, a positional error has been introduced. When the trajectory profile is examined in Cartesian space, as shown in Figure 4.20, the end-effector no-longer passes through all the knots. However, the error is small enough for the project's requirements. This experimentation supports the use of piecewise cubic polynomials in trajectory planning, and has enough fidelity to compare and contrast different configurations.

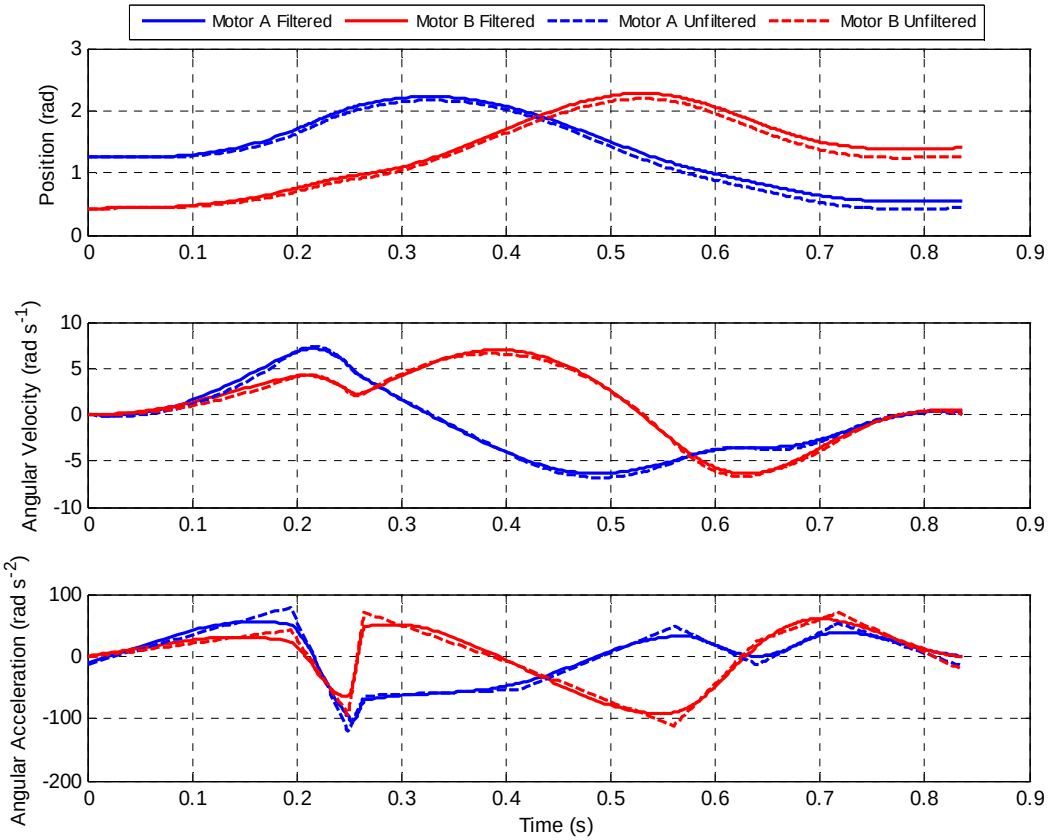

Figure 4.19 Motor position, velocity and acceleration commands before and after low-pass filtering.

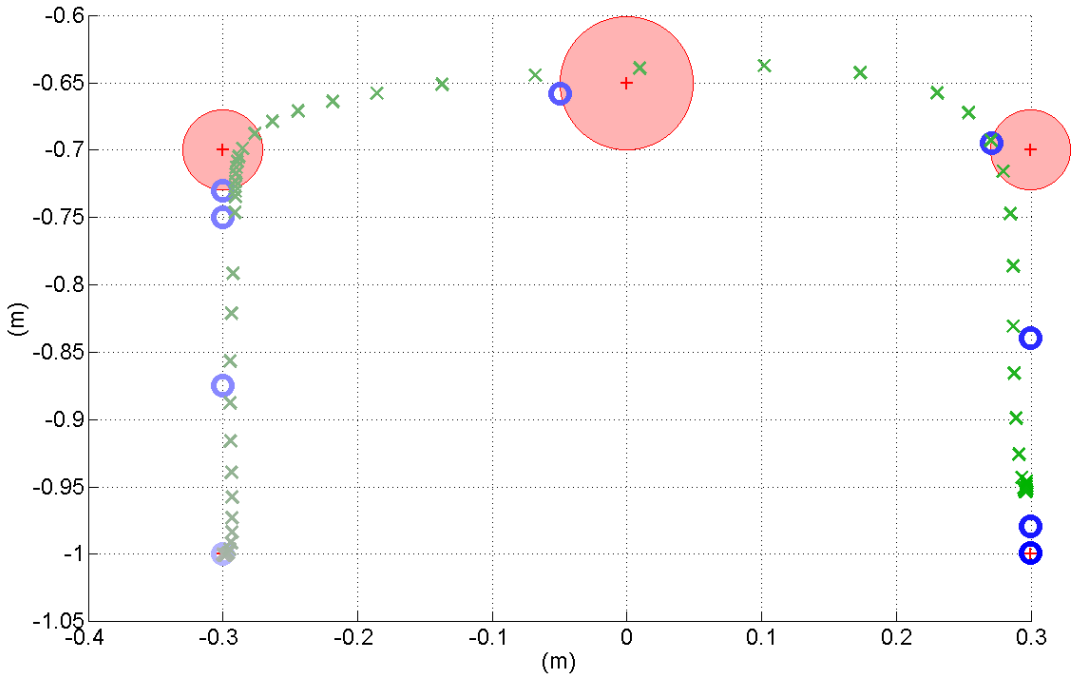

Figure 4.20 Trajectory using the filtered position, velocity and acceleration commands.

5 Dimensional Performance Analysis

The simulation of the manipulator presented in Chapter 3, using the trajectory planning method presented in Chapter 4, provides insight into the performance of the manipulator. The simulation enables the limits of the manipulator to be explored and examined in detail. If a pick-and-place application is known, say that of moving a known load through a pre-determined path, there may exist a 2DOFPPM manipulator that has the optimal dimensions for performing that task.

In this chapter, the 2DOFPPM dimensions are considered as parameters for optimisation. Motor and dimensional constraints are specified in order to limit the optimisation process. A database is created to store simulation results during the optimisation procedure. The search space of possible manipulator configurations is examined for a particular task. Knowing the shape of the search space, the possibility of applying optimisation algorithms to find the fastest configuration is discussed.

5.1 Constraints and Parameters

The 2DOFPPM has a number of parameters that can be altered. In this project, the selection of motors and the lengths of four major dimensional parameters are considered as variables to be optimised for achieving the best performance. These four dimensions are:

- Proximal arm length
- Distal arm length
- Separation distance of the motors
- Height of motors above the workspace

The dimensions relating to the positioning of the stabiliser arm components have only minimal effect on the performance of the manipulator and to minimise computation in the optimisation process, these values shall be considered fixed. The dimensions to be optimised are shown in Figure 5.1. The workspace height is defined as a distance by which the Y components of each target in the workspace are raised (if positive) or lowered (if negative).

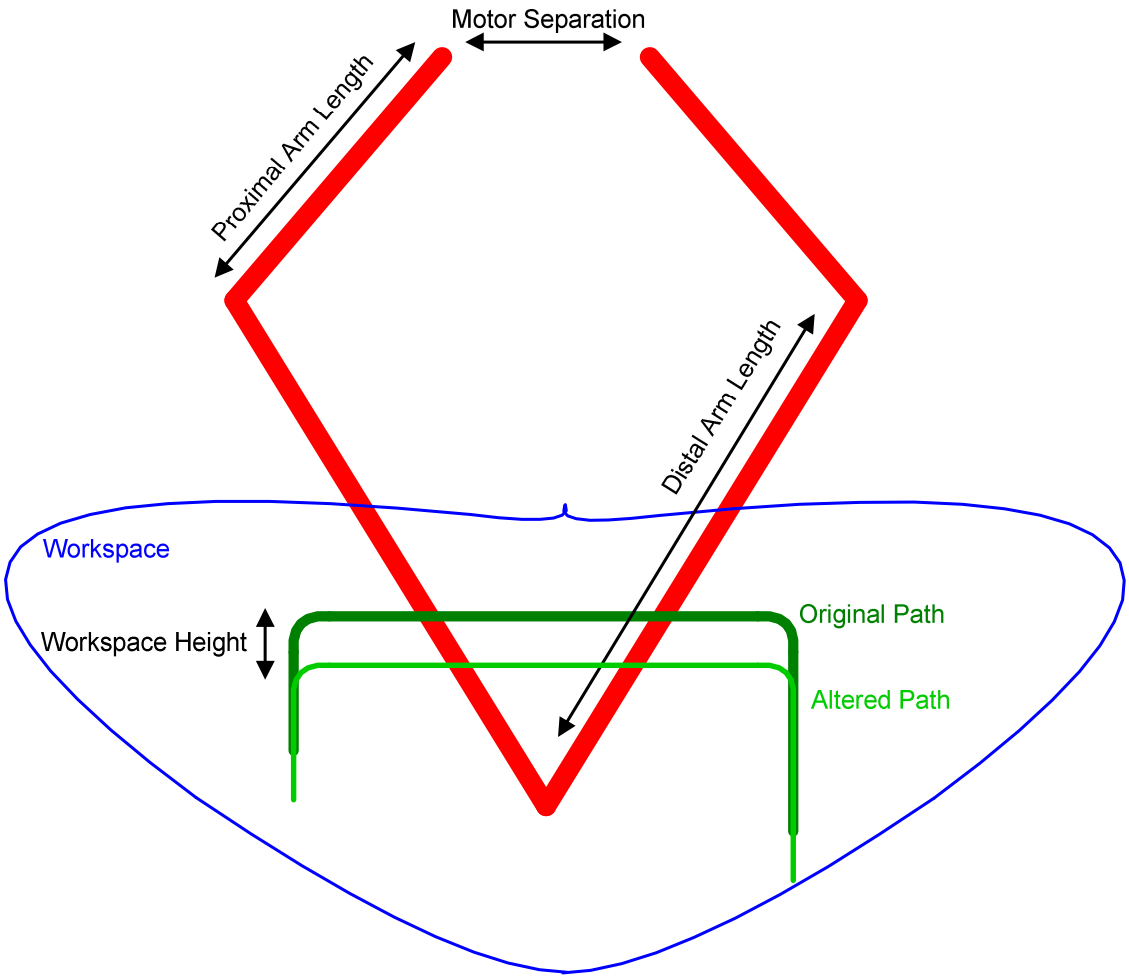

Figure 5.1 Diagram of the four dimensions to be optimised

In addition to the dimensions above, the choice of motors is also included as a variable to be optimised. A database table is used to store data associated with various motors. This allows only motors that are currently available in the marketplace to be selected, rather than assuming there is access to an ideal motor. Motors have been classified by their maximum torque, angular velocity, angular acceleration and angular jerk properties.

5.2 Results Storage

The optimising process generates a large amount of data relating to individual trajectory planning and simulations. To safely store the generated data, a database was developed. By using a database and not temporary memory storage such as RAM, more data can be stored in a permanent state. This is also useful for accessing at a future date without having to re-run the entire optimisation process.

MySQL™ was chosen as the database platform as it is free to use under the GNU GPL licensing agreement [72] and the existence of an interface for data transaction between Matlab® and MySQL™ [73]. Five tables were created in MySQL™ to store the simulation data. An Entity-Relationship Diagram (ERD) of the database tables is shown in Figure 5.2. The following sections describe each of the tables individually.

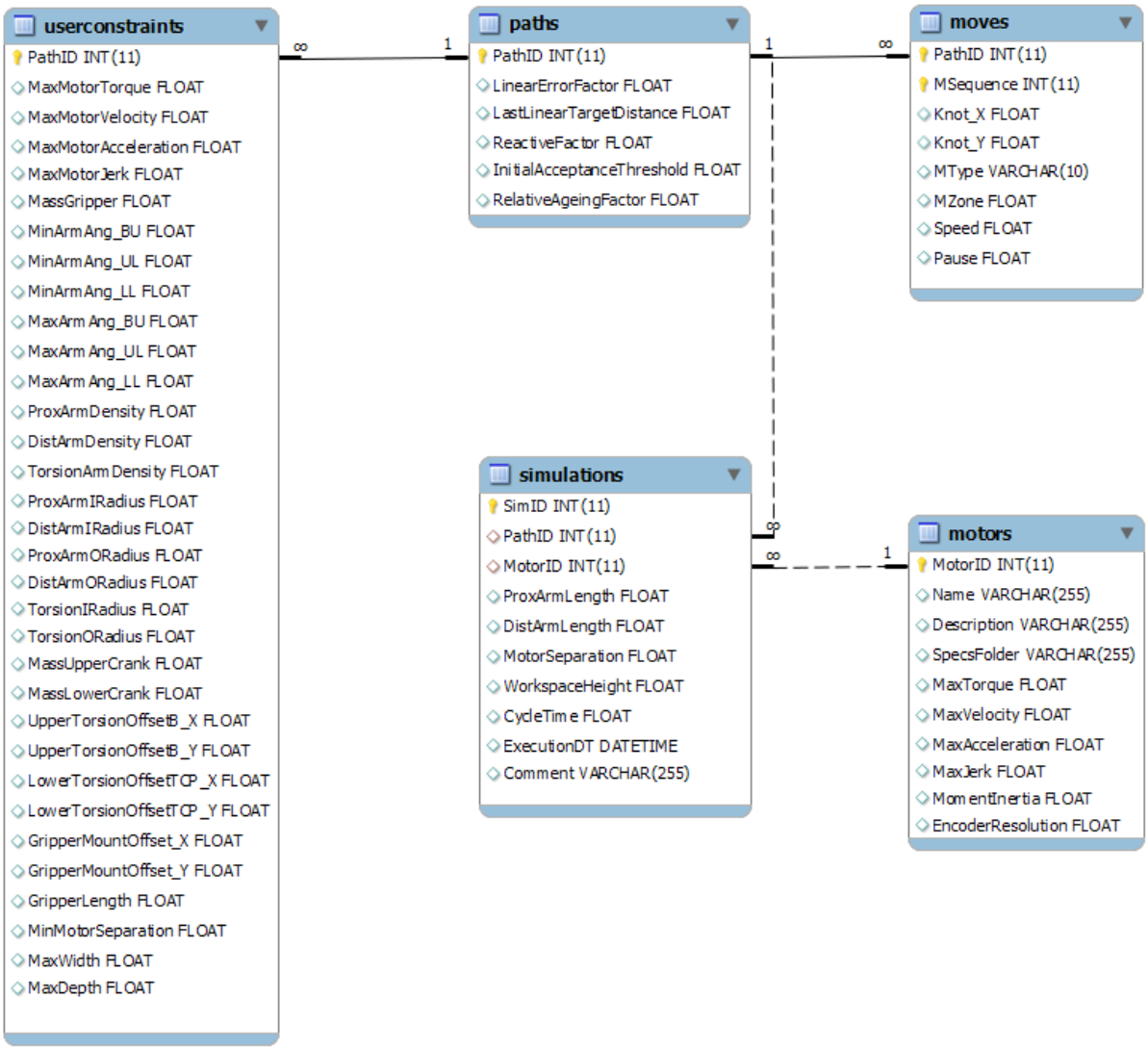

Figure 5.2 ERD diagram of the MySQL database schema

5.2.1 Paths

Whenever an optimisation process is started, the path, for which the optimisation will take place is stored in the database. A new *PathID* is assigned to uniquely identify the path. The path is stored between two tables, *paths* and *moves*. A single entry is inserted into the *paths's* table which contains parameters relating to the trajectory planning process. The *PathID* is the primary key. The actual movement commands are stored separately in the *moves* table. This separation is due to a path consisting of more than one movement. So to ensure a normalised database, a separate table was created to store the movement commands.

5.2.2 Moves

The movement commands that help define the path are stored in the *moves* table. The primary key is *PathId*, which is also a foreign key relating the entry back to the *paths* table. A second key is used to ensure that each path has a sequence of moves that can be easily identified. *MSequence* marks each move with an increasing integer in the order which the movements take place. The *moves* table contains attributes that define the *target* in Cartesian coordinates, *MoveType*, *zone*, and the maximum TCP speed for the move. A *pause* attribute also exists to allow the definition of a temporary pause in the cycle where a 'pick' or 'place' movement is programmed.

5.2.3 Userconstraints

The *userconstraints* table groups together all the constraints imposed on the optimisation by the user of the software system. Every *path* has an associated set of *userconstraints*. The *userconstraints* specifies the maximum permissible motor parameters. The table also contains dimensional constraints including the maximum and minimum angles allowed of the joints, the inner and outer radii of the arm components, and the offset dimensions of the stabiliser arm and gripper element. Limits on the overall maximum width and depth of the manipulator are recorded in this table. Gripper mass and the density properties of the arms are also stored here.

The *PathId* is the primary key for the table and is also a foreign key linking the set of *userconstraints* to the same path in the *paths* table.

5.2.4 Simulations

For a given path defined in the *paths* table, there may be numerous simulations. As the optimisation process requires multiple manipulator configurations to be simulated a separate table, *simulations*, is used to store the five variable parameters (proximal and distal arm lengths, motor separation distance, workspace height and the motor used) as well as the cycle-time achieved for the path. The *MotorID* is a

foreign key linking to a specific motor in the *motors* table. The time the simulation took place is also stored as a timestamp. The *comment* attribute is included in the table to allow additional identification of the optimisation process used when the simulation was executed.

5.2.5 Motors

The *motors* table stores data for a range of motors. *MotorID* is the primary key and is used to identify the motor in the *simulations* table. Each motor has a *name* and *description* attribute as well as a file path to a specification document. The inclusion of specification documents was added to allow easy lookup for technical details of a particular motor. The motor's limits are included, namely the maximum torque and angular velocity, acceleration and jerk. Additional details of the motor's moment of inertia and encoder resolution are optional parameters to be stored.

5.3 Search Space

Before attempting to solve an optimisation problem it is useful to gain insight into the search space of possible solutions. In the problem presented here, the search space is a set of four dimensional parameters of the 2DOFPPM, and the optimisation goal is to find the shortest cycle-time. Therefore, to find the search space, every possible permutation of the four dimensions must be considered. As the parameters being altered are length dimensions and therefore are continuous with an infinite number of possible permutations, the cycle-time must be evaluated at discrete distances between some limiting bounds for each of the four parameters.

The same cycle-path that was used in Chapter 3 (refer Figure 3.20), is used to demonstrate the optimisation methods in this chapter. In order to find the search space for the path, limits were placed on each dimension parameter as shown in Equations (5.1) through to (5.12).

$$l_{base_{min}} \leq l_{base} \leq l_{base_{max}} \tag{5.1}$$

$$l_{base_{min}} = MinMotorSeparation \tag{5.2}$$

$$l_{base_{max}} = 0.9 * MaxWidth \tag{5.3}$$

where $MaxWidth$ is a user defined parameter specifying the maximum width of the manipulator as shown in Figure 5.3. Setting a maximum width is useful as the space where the manipulator is to be installed is often limited. For the sample path, $MaxWidth$ is set at 1.5 m as this is a typical size

constraint for a manipulator performing the given path-cycle. A further limitation is imposed by restricting l_{base} to 90 % of $MaxWidth$. This is done as l_{base} cannot take up the full length of $MaxWidth$ as that would leave no room for the proximal arms.

The length of $MinMotorSeparation$ is a user defined parameter specifying the minimum separation distance of the motors. This constraint allows the physical dimensions of the motor or gearbox housings to be accounted for. For the sample path, this is set at 0.01 m to allow room for nominally sized gearboxes.

$$l_{prox_{min}} \leq l_{prox} \leq l_{prox_{max}} \tag{5.4}$$

$$l_{prox_{min}} = 0 \tag{5.5}$$

$$l_{prox_{max}} = \frac{MaxWidth}{2} \tag{5.6}$$

$$l_{dist_{min}} \leq l_{dist} \leq l_{dist_{max}} \tag{5.7}$$

$$l_{dist_{min}} = MinMotorSeparation \tag{5.8}$$

$$l_{dist_{max}} = \sqrt{MaxDepth^2 + \left(\frac{l_{base_{max}}}{2}\right)^2} \tag{5.9}$$

where $MaxDepth$ is a user defined parameter specifying the maximum length of the manipulator measured from the motor base to the end-effector, while the proximal arms are hanging down in the Y-plane as shown in Figure 5.3. Similar to the $MaxWidth$ parameter, $MaxDepth$ is implemented to account for any constraints on the space available for installing the manipulator. For the sample path this is set at 1.5 m, again to account for a typical size constraint on a manipulator executing the dimensions of the sample path. In Equation (5.6), $l_{prox_{max}}$ is limited to half of $MaxWidth$ as with two proximal arms of this length the $MaxWidth$ constraint would be reached, even with a l_{base} length of zero. Equation (5.9) is obtained by considering the Pythagoras triangle formed by $MaxDepth$ and half l_{base} as the proximal arm length approaches zero.

$$WSHeight_{min} \leq WSHeight \leq WSHeight_{max} \tag{5.10}$$

$$WSHeight_{min} = -0.3 * MaxDepth \tag{5.11}$$

$$WSHeight_{max} = 0.3 * MaxDepth \tag{5.12}$$

where $WSHeight$ is the height which the workspace is raised relative to the original programmed movement coordinates. It is expected that the coordinates are programmed with the manipulator mounted in a position where it can reach all the targets. This parameter allows for small changes to be made to the positioning of the manipulator. As such, a value of 30 % of the $MaxDepth$ parameter was considered sufficient variation to encompass the optimal workspace height.

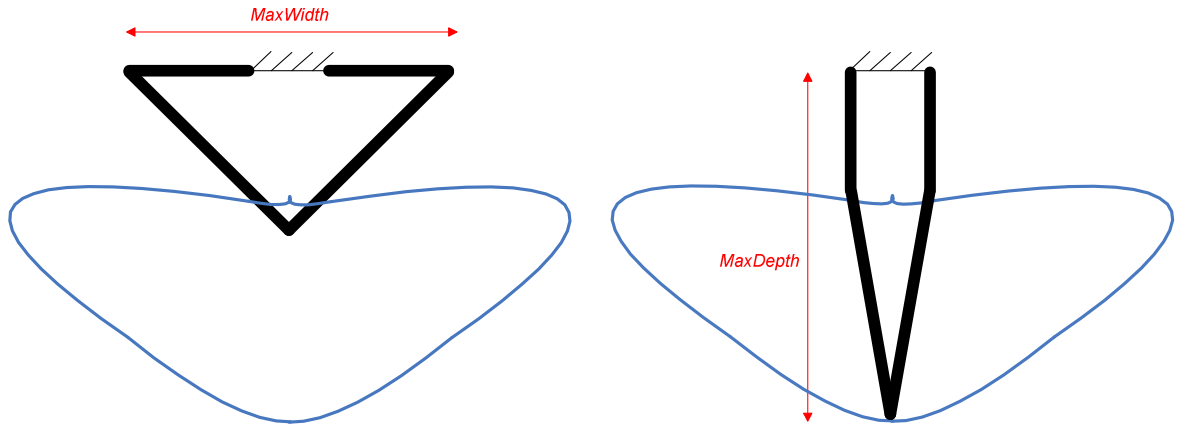

Figure 5.3 MaxWidth and MaxDepth parameters are defined by the user to limit the search space. They correspond to the dimensions in this diagram.

The dimensions being altered are continuous, therefore there are an infinite number of possible combinations despite the boundary conditions stated above. To limit the number of dimension combinations simulated, each dimension is divided into discrete values. For this project, the separation distance of the motors (l_{base}), proximal arm length (l_{prox}) and distal arm length (l_{dist}) were divided into 50 discrete values, evenly spaced between the upper and lower bounds of each parameter. The workspace height ($WSHeight$) was divided into 10 discrete values, evenly spaced between its upper and lower limits. Evaluating each of the possible combinations provides an accurate view of the solution space but is granular enough to be computed in a realistic time frame. By dividing the dimensions to this level, there exist 1.25 million combinations ($50^3 * 10$) to be explored.

At this point it useful to note that evaluating the cycle-time for every combination of the dimensions does not need to run through SimMechanics™. As the trajectory planning process calculates the position, velocity and acceleration profiles of the motors over time, the cycle-time is therefore determined at this stage. So where cycle-time is the only performance criteria being analysed, SimMechanics™ does not add any value and can consequently be omitted to save processing time. SimMechanics™ can then be used to review any particular configuration of interest at a later point in time. For example, the configuration with the fastest cycle-time after the trajectory planning process can be examined in detail in the SimMechanics™ simulation to look closer at joint torques or the end-effector's acceleration profile.

The search space for finding the optimal dimensions of the 2DOFPPM for traversing the sample path was then generated. Figure 5.4 shows the cycle-time plotted against three of the dimensions, l_{base}, l_{prox}, l_{dist}. The red data points represent configurations with the fastest cycle-time, whilst the blue represent the slowest. Due to limitations of graphing multiple parameters at once, the workspace height ($WSHeight$) data is lost within the graph. To view the effects that all four dimensions have on the path's cycle-time, Figure 5.5 shows 10 graphs at each of the 10 workspace heights.

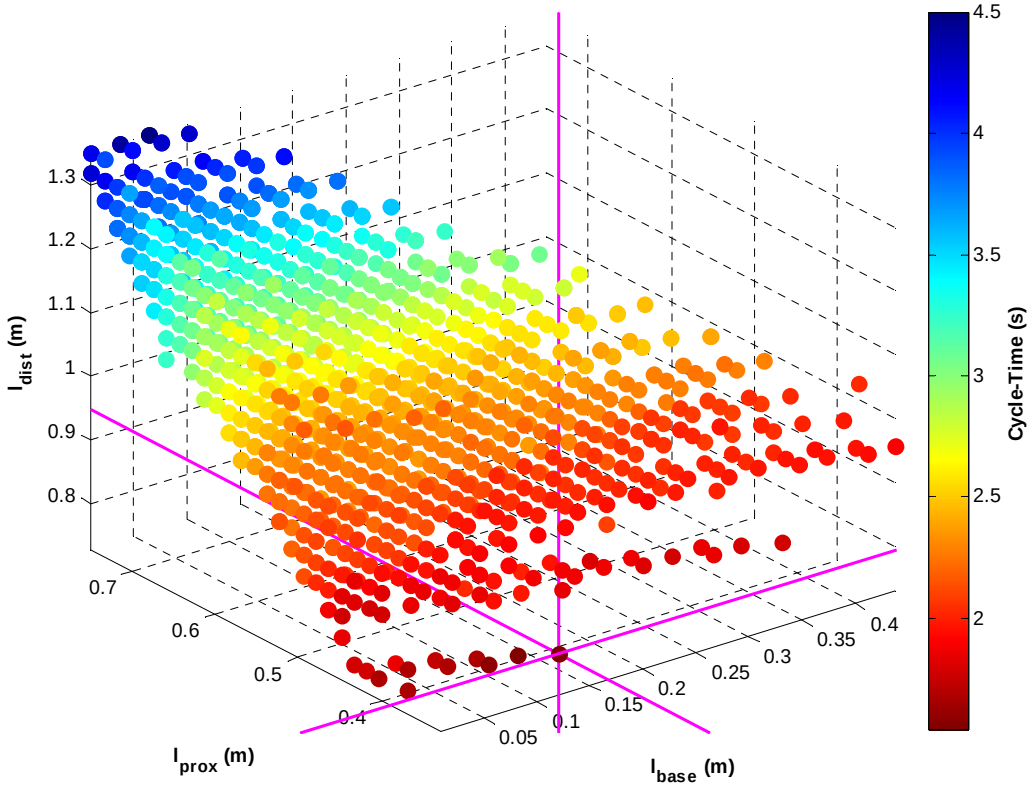

Figure 5.4 Graph of the search space for the sample path. The proximal and distal arm lengths and motor separation distance are plotted with the colours representing the cycle-time. The intersecting pink lines show the location of the minimum cycle-time.

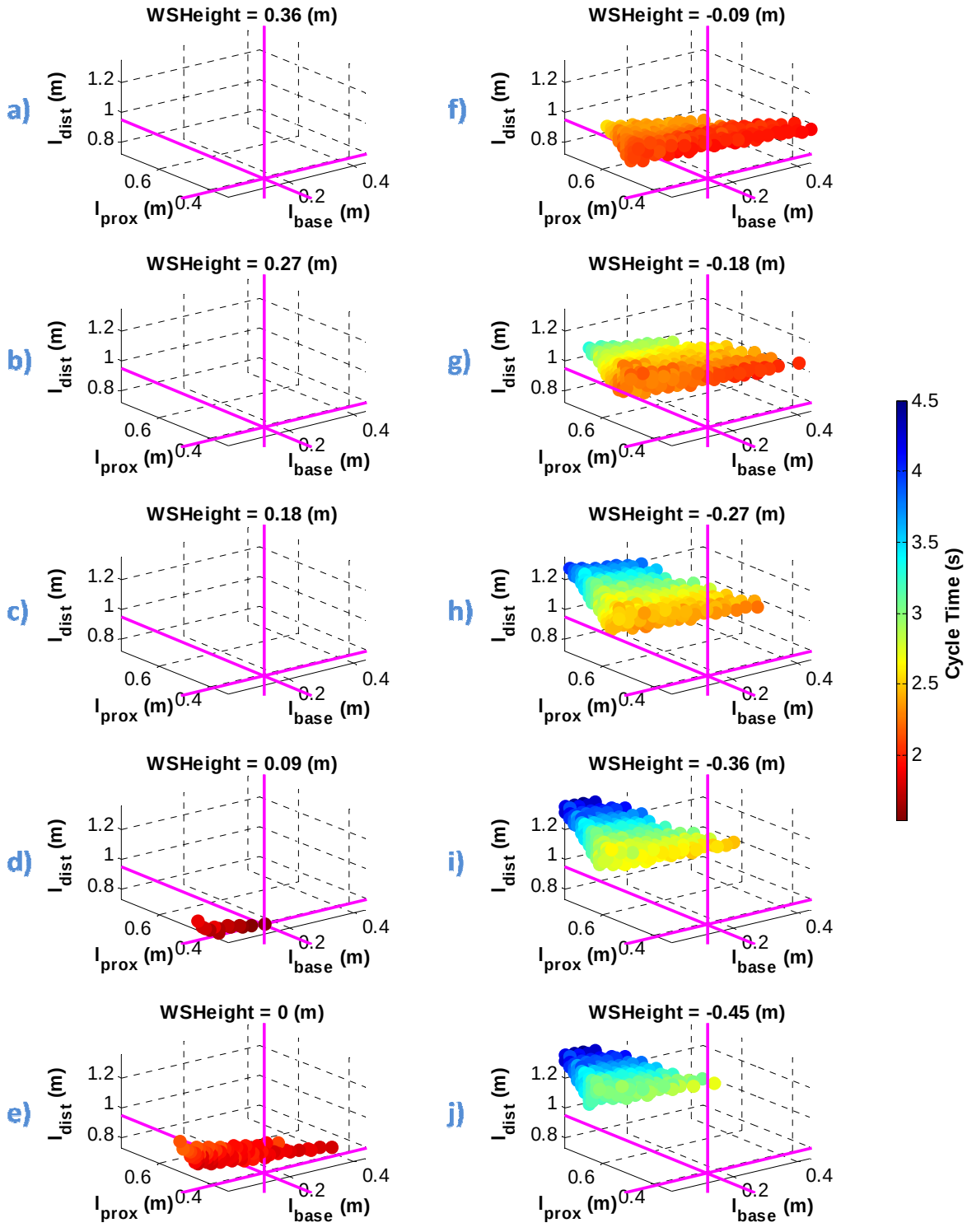

Figure 5.5 Search space for sample path. Each graph represents a different workspace height, starting from a high workspace in the a) to a low workspace in j).

From Figure 5.4 and Figure 5.5, it can be seen that for the sample path, the fastest cycle-times are achieved with relatively short proximal and distal arms, a small separation distance of the motors, and a slight rise in the workspace height. It is also noted that the optimal solution lies on the border of the search space, that is, the manipulator is only just able to reach all points on the sample path. Figure 5.5a through to Figure 5.5c show that no valid configurations exist when the workspace is raised too high.

The generation of the search surface was performed on a single computer (refer Appendix B). With the simulation optimised for speed, it took 18 hours to evaluate the 1.25 million possible combinations to this level of accuracy.

5.4 Optimisation Overview

While a coarse but complete view of the solution space can be generated in a matter of hours, as shown in the previous section, it may be possible to find a near-optimal solution faster. The optimisation task is to find the configuration of four dimensions of the 2DOFPPM that leads to the fastest cycle-time. In terms of optimisation problems, this is a simple problem and therefore simple optimisation techniques shall be considered.

There are an ever-growing number of optimisation algorithms available and comparisons between techniques are common. Prügel-Bennett [74] compared the performance of a Hill Climber, Stochastic Hill Climber and a Genetic Algorithm for a toy problem with a similar search space. Mitchell et al. [75] analyses the performances of a Hill Climber and a Genetic Algorithm to find under what conditions each algorithm is superior. Garg and Kumar [76] compare the performances of a Genetic Algorithm to Simulated Annealing as applied to manipulator path planning. These are only a few of many such comparisons between optimising techniques.

For simple optimisation problems like this there are four main techniques commonly used. These are:

- Random Restart Hill Climber
- Stochastic Hill Climber
- Simulated Annealing
- Genetic Algorithm

A comparison of these four techniques, as applied to finding the optimal dimensional configuration of a manipulator, is considered important and valid. The next chapter implements and evaluates these algorithms.

It should be noted that a near-optimal solution is being considered instead of a truly optimal solution as the optimisation techniques cannot be guaranteed to find the absolute best solution, but rather a solution that is near optimal. A near-optimal solution is sufficient, as sub-millisecond improvements to the cycle-time are insignificant given the estimation process required to generate the trajectory. There is also no benefit in optimising arm lengths beyond the degree of precision capable of the fabrication process.

6 Optimisation Methodologies

Over the following sections, four optimising techniques are implemented and their performances compared. The sample path used in Chapter 3 (refer Figure 3.20) and Chapter 5 is employed again to evaluate the techniques. All optimisations are performed on a single computer. The specifications of this computer can be found in Appendix BAppendix F.

The key performance indicators for each technique are:

- Minimum path cycle-time achieved.
- Computation time to reach 'optimisation'.

The minimum path cycle-time is the overall time to traverse the sample path as calculated by the trajectory planner. The point where each method is considered to reach 'optimisation' will vary due the individual process. However, a comparison will be made between the optimisation techniques to determine which method finds a suitably fast cycle-time with the least amount of computation.

Each technique is run multiple times to allow statistical evaluations to be performed. Where possible, the techniques have been given the same starting conditions. For example, the number of iterations for restarting the hill climber is used again as the number of stochastic hill climber starting attempts. Each technique also has a number of parameters that need to be tuned to maximise the technique's performance. In most cases the parameters are tuned by evaluating the performance over a number of runs. This allows the performance of the parameters to be fairly evaluated in a statistical manner. Due to the large processing time of evaluating some of the parameters, simple empirical testing was done to tune these parameters. The method for tuning each parameter is documented in each of the following sections.

6.1.1 Random Restart Hill Climber

After random search techniques, a Hill Climber is the simplest of optimising algorithms. Hill climbing methods are popular due to the simplicity of implementing them. All that is required is an evaluation function for which a measure of fitness can be obtained and the ability to select other solutions around the current solution (that is, the *neighbourhood*). In the case of optimising the 2DOFPPM dimensional configuration for achieving the fastest cycle-time for a given path, the evaluation function is the cycle-

time and neighbouring solutions are configurations close to the current configuration that vary slightly in the optimised dimension(s).

It should be noted that formally, Hill Climbing methods seek to achieve a *maximum*. In this project, the *minimum* cycle-time is the objective. However, the technique to minimise remains fundamentally the same as the maximisation method and as such, the term 'Hill Climber' will be used even though the opposite effect is desirable. Sometimes the minimisation method is referred to as *gradient descent*, but this term will not be used in this work.

A Hill Climber starts by randomly selecting a solution and evaluating its performance against a fitness function. The neighbouring solutions are then considered and their performance evaluated. If a neighbouring solution is found to perform better than the first solution, the neighbourhood of that solution is evaluated. This iterative process continues until a solution is found that performs better than all of its neighbouring solutions.

A Hill Climbing method works well when there are no local optima in the search space, only the global optima. When looking at the search space in the previous section (refer Figure 5.5), it could be assumed that this is the case in this project (that is, the cycle-time is minimised as the dimensions tend towards short proximal and distal arms, a small separation distance of the motors, and a slight rise in the workspace height). However, when a single instance of a Hill Climber is run, it finds itself stuck in a local optimum, unable to get out and reach the desired global optimum. This is due to the search surface containing shallow *troughs* and low *ridges* that create local optima. After some consideration, it was decided that the most likely cause of these local optima is the iterative trajectory planning process. Because the trajectory planner iteratively increases and decreases the path time between knots, a near optimal trajectory is generated. How close to truly optimal each trajectory is depends on the process and some configurations may be closer to optimal than others. This results in some configurations being considered slightly less favourable than their *neighbours*, even though they may in fact be slightly better if the trajectory planner produced a truly optimal trajectory.

To apply a Hill Climbing method to a search space containing local optima, as is the case here, it is common to use a *Random Restart Hill Climber* (*RRHC*). A RRHC differs from a standard Hill Climber by selecting more than one starting solution. This has the effect of producing hill climbers at multiple starting points in the search space. Each Hill Climber is allowed to find its own (local) optimum. By this method, a greater area of the search space is covered, increasing the likelihood of finding the global optimum. However the RRHC method cannot be guaranteed to find the global optimum.

Figure 6.1 shows the RRHC implemented in Matlab® code. The overall process is implemented for a number of iterations (restarts) in the form of a coded for-loop (lines 11-67). In each iteration a motor is selected from a database (line 14) and a random 2DOFPPM configuration is chosen within some constraints (line 19). A path is then compiled (line 25) using the *CompilePath()* method developed earlier in the trajectory planning section (refer Section 4.1 and Figure F.7 through to Figure F.11 in Appendix F). The path is then stored in the database (line 32) and the configuration and its cycle-time are considered to be the 'best' so far (lines 35-36). The neighbouring configurations are then found based on the parameter *StepSize* (lines 43-44). A trajectory is generated for each of the neighbouring configurations (line 49) and the results stored in the database (line 50). The cycle-times of the neighbouring solutions are compared to the current solution (lines 58-59) and if a better solution is found, it then becomes the 'best' configuration (line 62) and the process is repeated. If several solutions are better than the current solution, the best solution is chosen. If no neighbour improves the cycle-time, then a local minimum has been found, the while-loop (lines 38-66) is exited and the iteration stops.

When selecting the neighbouring configurations in this problem, 30 configurations are chosen. These 30 configurations are the result of the four dimensions being altered. Each dimension could remain unchanged or be increased or decreased by the *StepSize* amount. The solution that remains unchanged in all dimensions is rejected as that is the current solution. The code for this method, *SelectNeighbouringConfig()*, is included in Appendix F (refer Figure F.72).

The RRHC, contains two variables that require tuning. The first is the parameter named *StepSize* (lines 1 & 44). The *StepSize* determines the distance away from the current configuration to examine its *neighbourhood*. To tune the *StepSize*, several values were considered and tested by performing 90 runs of the *RRHC* optimising method using each. The other parameter to determine is how many random restart iterations are required to sufficiently cover the search space. In the Matlab® code this is referred to as *TermCond.Iterations* (line 11). This was also tuned by running the RRHC multiple times and considering the performance of the method as the number of iterations increased. Both parameters were tuned simultaneously by running 90 RRHCs for several values of *StepSize* meanwhile recording the performance relative to the number of restart iterations.

```
function OptimiseConfigurationHC(CP,TermCond,UConstraints,StepSize)
% Uses a random restart hill climber to narrow on a time-minimum configuration
% VARIABLES:
% CP - Cycle Path class containing geometric details of the path
% TermCond - Termination Condition class detailing conditions of terminating process
% UConstraints - User Constraints class
% StepSize - size of steps (in m) to evaluate neighbouring configurations

StorePathsUserConstraintsSQL(CP,UConstraints);  % Store path and user constraint data

for i=1:TermCond.Iterations     % Run Hill Climber for a number of iterations

    % Select 'random' motor details from database
    [motorID,newPPC] = SelectMotor(CP.PPC,UConstraints);

    CP.PPC = newPPC;    % Assign Path Planning Constraints (PPC) of motor to Cycle Path(CP)

    % Select random configuration that reaches all move targets
    config = SelectRandomConfig(CP.Moves,motorID,UConstraints);

    try
        % Compile path using Configuration and Path Planning Constraints (PPC)
        % Path Planning Results (ppr) are returned along with positional and zone data
        % about targets
        [Targets_XYZ,ppr] = CompilePath(CP.Moves,config,CP.PPC);
    catch exception
        % Skip to next iteration if exception occurs due to config unable to meet targets
        continue;
    end

    % Store results of path planning in database
    StoreSimulationsSQL(config,CP.PPC,ppr,CP.ID,i);

    local = false;  % Set flag indicating whether a local minima has been found
    minCycleTime = ppr.PathA(size(ppr.PathA,1)).EndTime; % Set best cycletime acheived
    bestConfig = config;

    while local == false % Loop until local minima has been found
        clear neighboursPPR;        % Clear variables
        clear neighboursConfig;     % Clear variables

        % Select configurations around the best configuration so far
        neighboursConfig = ...
            SelectNeighbouringConfig(bestConfig,CP.Moves,motorID,UConstraints,StepSize);

        for j=1:size(neighboursConfig,2)
            % Compile Paths using each of the neighbouring configurations(neighboursConfig)
            % Store results in database, and save Path Planning Results (ppr) in an array
            [Targets_XYZ,ppr] = CompilePath(CP.Moves,neighboursConfig(j),CP.PPC);
            StoreSimulationsSQL(neighboursConfig(j),CP.PPC,ppr,CP.ID,i);
            neighboursPPR(j)=ppr;
        end

        local = true;   % set flag - will be reset if not local
        for j=1:size(neighboursPPR,2)
            % Compare results of each neighbouring configuration. Replace bestConfig with
            % neighbour if faster cycletime is found
            if neighboursPPR(j).PathA(size(neighboursPPR(j).PathA,1)).EndTime ...
                                                                    < minCycleTime
                minCycleTime = ...
                        neighboursPPR(j).PathA(size(neighboursPPR(j).PathA,1)).EndTime;
                bestConfig = neighboursConfig(j);
                local = false;
            end
        end
    end
end
end
```

Figure 6.1 Matlab® Code of the RRHC Optimising Method

Four different values of step size were chosen. These distances are shown in Table 6.1 and are measured in metres. Alongside each is the relative length of the step size when compared to the width (0.6 m) and height (0.3 m) of the sample path (refer Figure 3.20). The comparison to the path dimensions is shown to give an indication of the appropriate *StepSize* should a significantly different path be optimised using the technique outlined in this research.

Table 6.1 StepSizes evaluated and their relative path dimensions

StepSize (m)	Percentage of path width (0.6 m)	Percentage of path height (0.3 m)
0.01	1.67 %	3.33 %
0.02	3.33 %	6.67 %
0.05	8.33 %	16.67 %
0.1	16.67 %	33.33 %

The results of the 4 x 90 runs of the RRHC have been summarised in the following figures. Figure 6.2 shows a histogram distribution of the minimum cycle-time achieved by each of the *StepSizes*. *StepSizes* of 0.01 and 0.02 m are seen to perform better than the larger distances of 0.05 and 0.1 m over 100 restart iterations. The mean, standard deviation and median cycle-times of each *StepSize* is shown in Table 6.2.

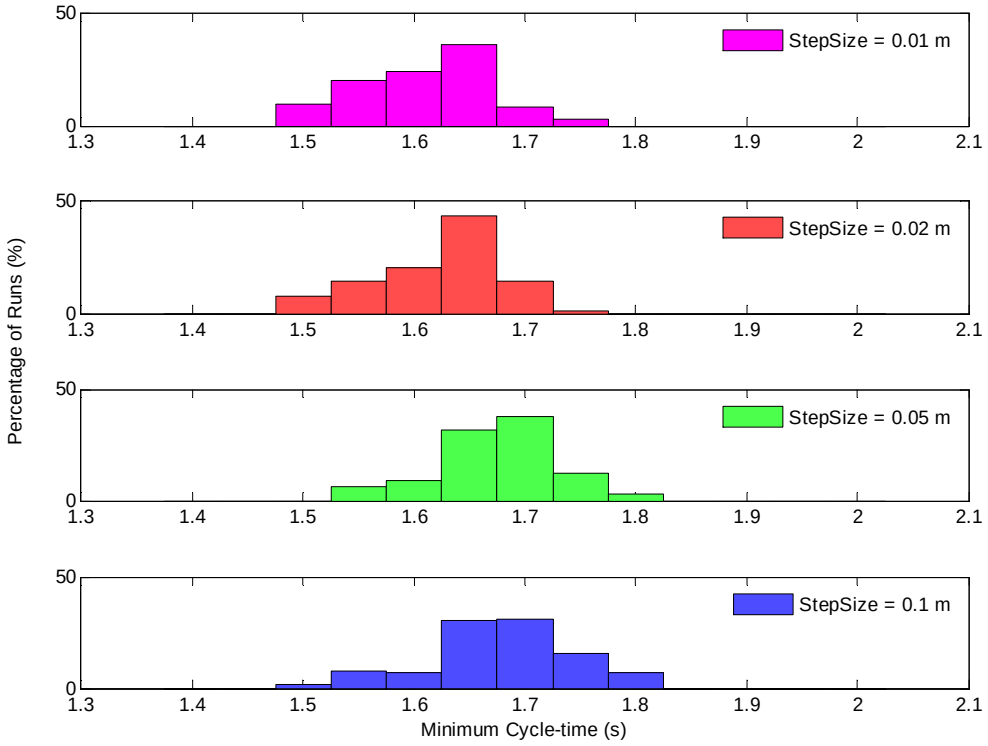

Figure 6.2 Normalised histogram of minimum cycle-time achieved by four different StepSizes using the RRHC method after 100 restart iterations. Based on 90 individual runs.

Table 6.2 Mean, standard deviation and median minimum cycle-times for four different *StepSizes*

StepSize (m)	Mean Minimum Cycle-time (s)	Minimum Cycle-time Standard Deviation (s)	Median Minimum Cycle-time (s)
0.01	1.64	0.06	1.65
0.02	1.65	0.05	1.66
0.05	1.70	0.06	1.70
0.1	1.71	0.07	1.71

While the mean and median give an indication into which *StepSize* is best to use for this project, the performance of each *StepSize* is better analysed by applying a proven statistical comparison technique known as the Wilcoxon-Mann-Whitney (WMW) two-sample rank-sum test [77] [78] (also known as a U-test). The WMW is a non-parametric method used to test whether two independent samples of observations are of equal value in a statistical sense (that is, is one *StepSize* better than another). As the median minimum cycle-time achieved by *StepSize* = 0.01 is the best of the four distances considered, the WMW method will be used to compare the significance of this result to the other three *StepSizes*. The results in Table 6.3 show that the null hypothesis is rejected for *StepSizes* = 0.05 and 0.1, but is confirmed for StepSize = 0.02. This means that the *StepSize* = 0.01 is statistically better than *StepSizes* = 0.05 and 0.1, but there is no significant difference when compared to *StepSize* = 0.02. These results were obtained using the standard 95 % confidence interval.

Table 6.3 Wilcoxon-Mann-Whitney test results comparing *StepSize* = 0.01 to the other *StepSizes*

StepSize (m)	Rejection of Null-Hypothesis	p-Value
0.02	0	0.189
0.05	1	7.73×10^{-11}
0.1	1	3.37×10^{-12}

While only the *StepSize* has been analysed so far, the number of random restart iterations is of equal importance to the RRHC algorithm. Figure 6.3 shows the mean minimum cycle-time achieved versus the number of random restart iterations for each of the *StepSizes*. The greatest improvement is seen within the first ten iterations with the average minimum cycle-time reducing by 0.15 s. The rate of improvement declines as the number of iterations increases, but even after 100 iterations, all four

StepSizes continue to improve the mean minimum cycle-time, albeit slowly. *StepSizes* of 0.01 and 0.02 m not only find better configurations of the 2DOFPPM, but also achieve them with less restart iterations. The mean minimum cycle-time was at 1.7 s after approximately 30 iterations for *StepSizes* of 0.01 and 0.02m, whereas it took on average 100 restart iterations for StepSizes 0.05 and 0.1 m.

Figure 6.4 shows histogram distributions of the minimum cycle-time for a *StepSize* of 0.02 m at intervals of 25, 50, 75 and 100 random restart iterations. This shows a very dispersed distribution when only a few restart iterations are used, as is the case with 25 random restarts. As the number of restarts is increased, the minimum cycle-time achieved by the RRHC becomes more consistent (that is, a narrower distribution) and centres on approximately 1.65 s as shown by the histograms of the 75 and 100 restart iterations. Even after 100 restart iterations, there is still variation with the RRHC sometimes achieving cycle-times as low as 1.5 s and other times only managing to optimise to 1.75 s.

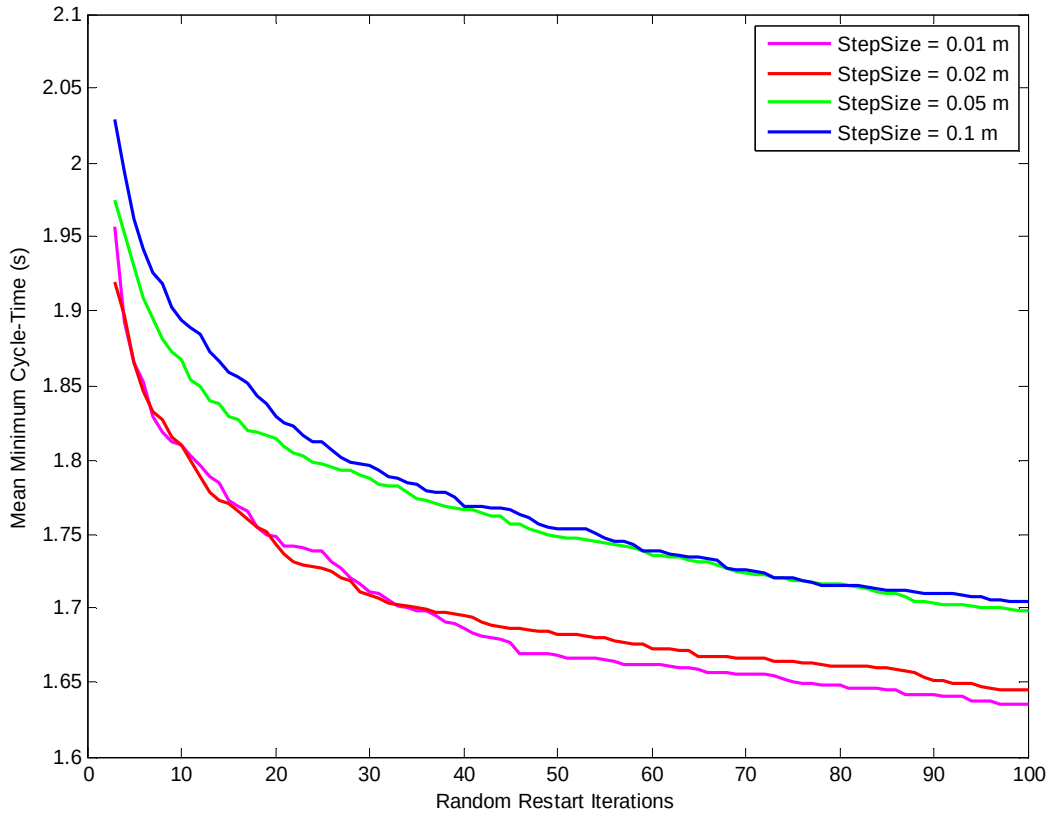

Figure 6.3 Mean Minimum Cycle-time versus the number of Random Restart Iterations for four StepSizes

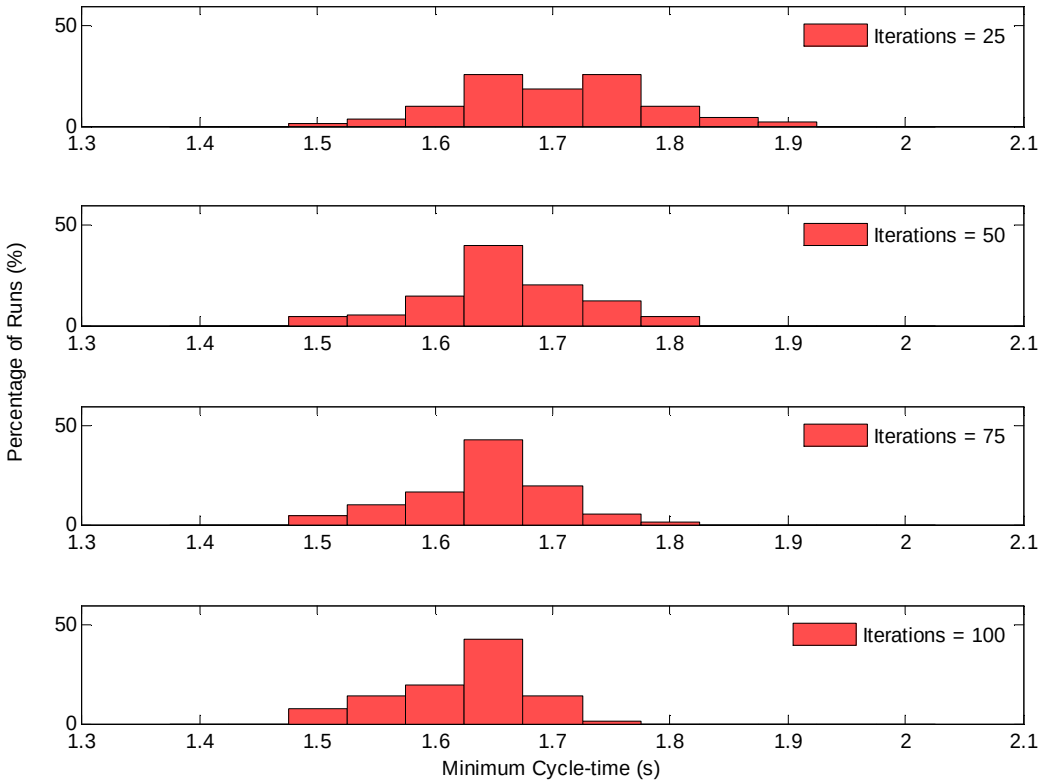

Figure 6.4 Normalised histograms of minimum cycle-time achieved by the RRHC method with a *StepSize* of 0.02 m, after 25, 50, 75, 100 restart iterations. Based on 90 individual runs.

As with any optimising algorithm, performance is also measured in how long it takes to reach 'optimisation'. So far *StepSizes* of 0.01 and 0.02 m have shown to achieve better minimum cycle-times than 0.05 and 0.1 m. However, the time taken for each RRHC to achieve its 'optimal' state varies as shown by the box and whisker plot in Figure 6.5. For this plot, 100 random restart iterations have been used. It can be seen that as the *StepSize* increases, the time to reach an 'optimal' solution is reduced. This is expected, given that a larger *StepSize* will cover the search space faster by taking larger 'steps' at each iteration in the optimisation process.

The variation in the time taken to 'optimise' the 2DOFPPM dimensions is visible in Figure 6.6, where the four *StepSizes* are again compared. The mean minimum cycle-time achieved by the RRHC is plotted against the length of time the algorithm is run for. Once more, 100 random restart iterations are used for each instance of the RRHC. It is observed that larger *StepSizes* result in faster converging algorithms. The 0.1 m *StepSize* completes its optimisation within 300 seconds, the 0.05 m *StepSize* takes close to 1000 seconds, the 0.02 m *StepSize* on average takes 4200 seconds, and the 0.01 m *StepSize* requires 5000 seconds to reach its optimised state. It can also be noted that when the 0.1 m *StepSize*

optimisation is completed, on average it has found a better solution than the other *StepSizes* after the same length of time.

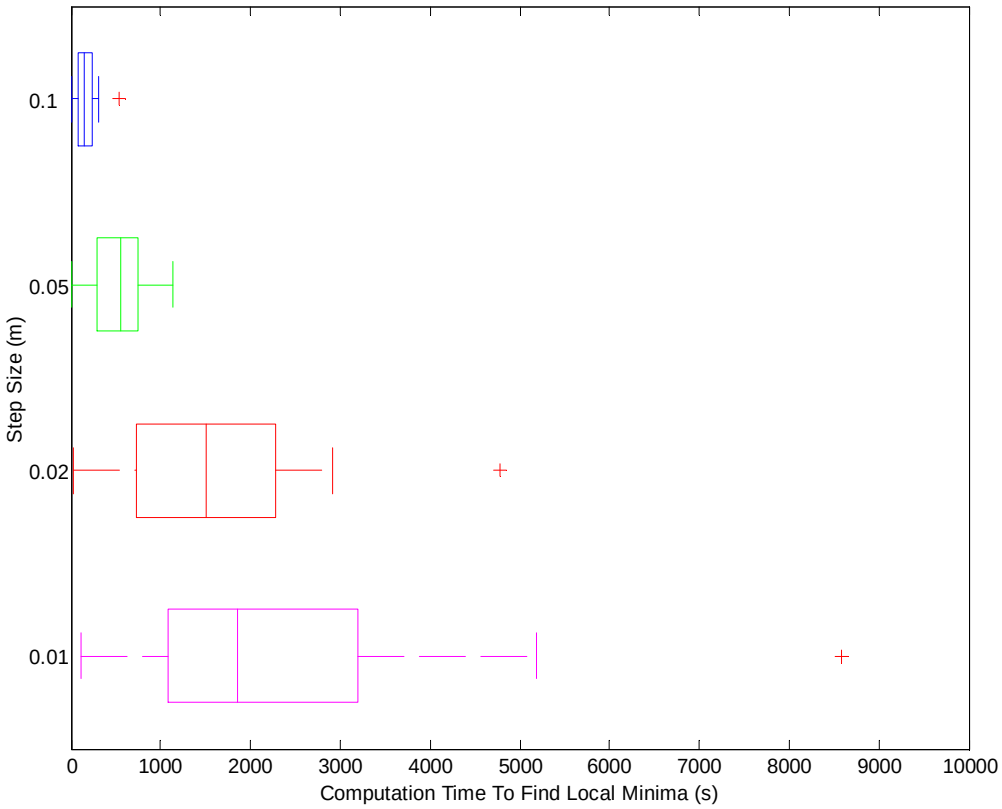

Figure 6.5 Box plot of the computation time required for each RRHC to find its minimum cycle-time. Graph shows separate box plots for each *StepSize*.

This section has discussed the use of the RRHC optimising method as applied to finding the optimal dimensions of the 2DOFPPM. Several parameters have been evaluated to achieve optimal performance from the algorithm. The first of these is the *StepSize* distance used to determine the space to neighbouring solutions. The second is the number of random restart iterations required to sufficiently explore the search space. *StepSizes* of 0.01 and 0.02 m were shown to outperform 0.05 and 0.1 m, but after 100 iterations there was little to distinguish between 0.01 and 0.02 m. The difference in the computation time required to find the minimum cycle-time, as shown in Figure 6.5, demonstrates that the *StepSize* of 0.02 m makes it the preferred choice as it takes significantly less time to reach its optimised state. The number of random restart iterations required has been set at 100 as the graph in Figure 6.3 shows the improvement in the minimum cycle-time reaching a plateau around this number.

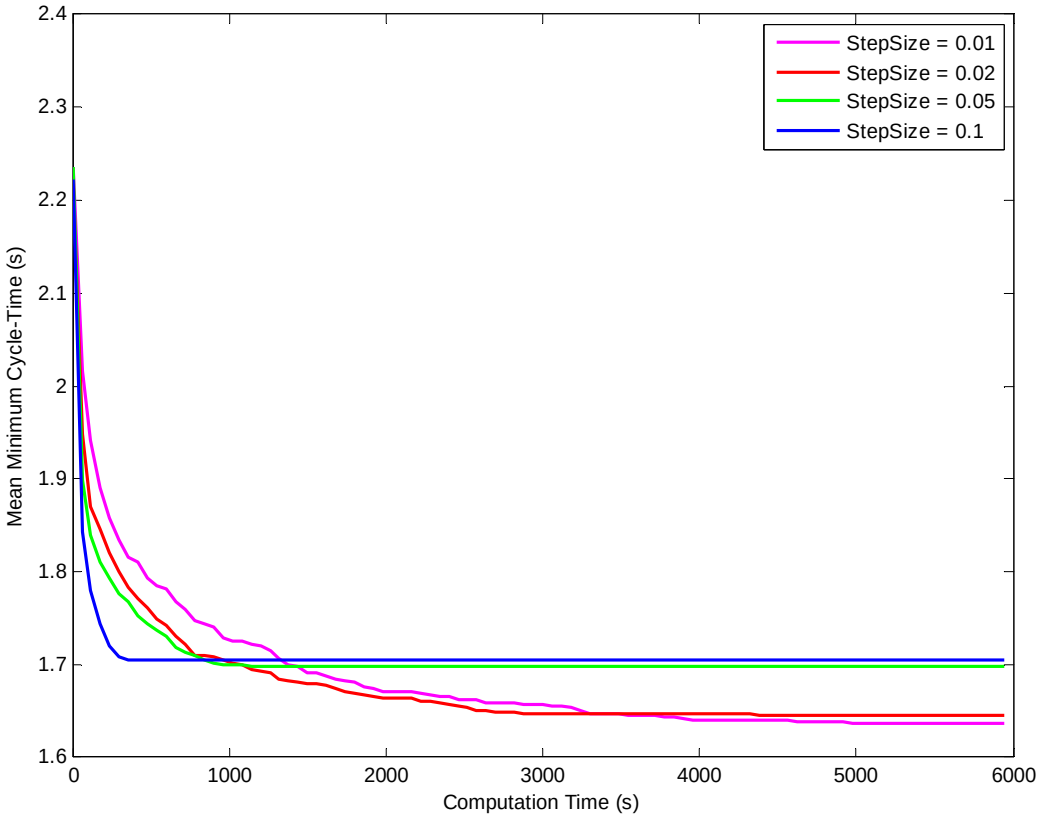

Figure 6.6 Computation time versus mean minimum cycle-time for four *StepSizes*

6.1.2 Stochastic Hill Climber

The RRHC technique suffered from becoming stuck in local optima. While the concept of using multiple restart positions helped to solve this, there are alternative methods that could potentially perform better. One of these is the *Stochastic Hill Climber* (*SHC*). The SHC varies from a standard Hill Climber in two ways:

- Rather than checking all solutions in the neighbourhood of the current solution and then selecting the best one, the SHC only selects one neighbour at random for evaluation.
- The method of selecting this new neighbour is probabilistic based on the relative performance of the current solution and the neighbour.

The SHC gets its name from the fact that the selection process is now stochastic rather than based on the absolute difference in performance of the solutions. This process now allows a 'weaker' solution (that is, a solution with a slower cycle-time) to be selected over a 'stronger' solution based on some probability. The function used to determine this probability is stated in Equation (5.13).

$$P = \frac{1}{1 + e^{\frac{(N_{CT} - C_{CT})}{T}}} \tag{5.13}$$

where P is the probability of selection, N_{CT} and C_{CT} are the cycle-times of the neighbouring and current solution respectively, and T is a constant that determines the shape of the selection probability profile.

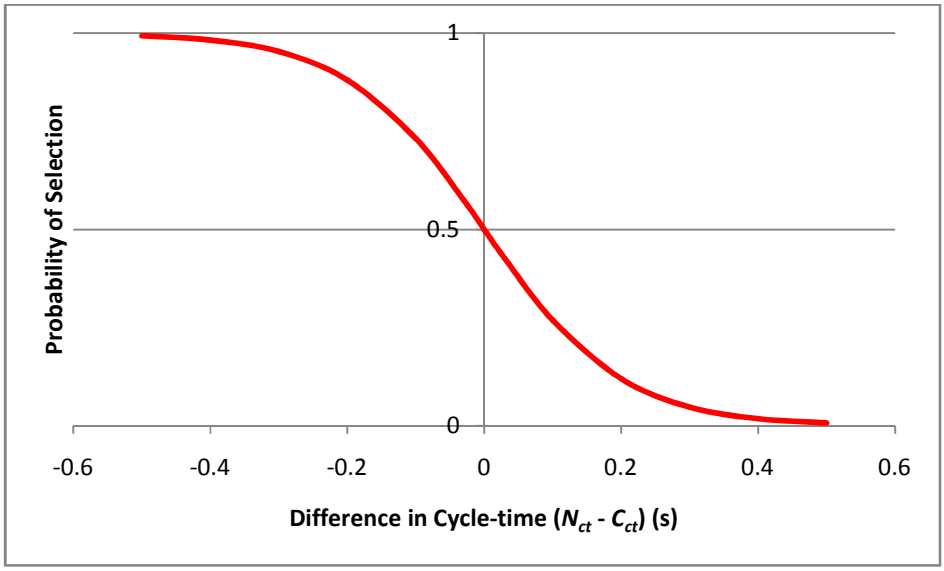

Figure 6.7 Example selection probability profile for a SHC

The probability profile resulting from the function can be seen in Figure 6.7 where the probability of acceptance is plotted against the difference in cycle-times of the two solutions being compared. As can be seen, the probability of selection is greater if the neighbouring solution is better than the current solution (that is, a negative difference in cycle-time). However, the probability of selecting a neighbouring solution with a slower cycle-time also exists. As the difference in cycle-times increases, the probability of selection approaches that of a traditional Hill Climber. This stochastic approach allows the SHC to escape from local optima.

The Matlab® code used to execute the SHC is presented in Figure 6.8. The implementation is very similar to the RRHC (cf. Figure 6.1) but varies in that only one neighbour is selected at random with its path compiled (lines 42-45), and the selection process is now probabilistic (line 59). By only selecting and evaluating one neighbour there is less redundant computation which will be shown to lead to a faster optimising algorithm.

```
function OptimiseConfigurationSHC(CP,TermCond,UConstraints,StepSize,MaxAttempts,T)
% Uses a random restart stochastic hill climber to narrow on time-minimum configuration
% VARIABLES:
% CP - Cycle Path class containing geometric details of the path
% TermCond - Termination Condition class detailing conditions of terminating process
% UConstraints - User Constraints class
% StepSize - size of steps (in m) to evaluate neighbouring configurations
% MaxAttempts - the number of attempts before deciding current iteration is complete
% T - constant in algorithm that affects probability of selection

StorePathsUserConstraintsSQL(CP,UConstraints);  % Store path and user constraint data
for i=1:TermCond.Iterations     % Run Stochastic Hill Climber for a number of iterations
    % Select 'random' motor details from database
    [motorID,newPPC] = SelectMotor(CP.PPC,UConstraints);
    CP.PPC = newPPC;   % Assign Path Planning Constraints (PPC) of motor to Cycle Path(CP)
    % Select random configuration that reaches all move targets
    config = SelectRandomConfig(CP.Moves,motorID,UConstraints);
    try
        % Compile path using Configuration and Path Planning Constraints (PPC)
        % Path Planning Results (ppr) are returned along with positional and zone data
        % about targets
        [Targets_XYZ,ppr] = CompilePath(CP.Moves,config,CP.PPC);
    catch exception
        % Skip to next iteration if exception occurs due to config unable to meet targets
        continue;
    end
    % Store results of path planning in database
    StoreSimulationsSQL(config,CP.PPC,ppr,CP.ID,i);
    local = false;  % Set flag indicating whether a local minima has been found
    minCycleTime = ppr.PathA(size(ppr.PathA,1)).EndTime; % Set best cycletime acheived
    bestConfig = config;                % Set the best Configuration
    currentConfig = config;             % Set the current Configuration
    currentCycleTime = minCycleTime;    % Set cycletime acheived by currentConfig
    while attempts < MaxAttempts    % Loop for a set number of attempts
        clear neighboursPPR;            % Clear variables
        clear neighboursConfig;         % Clear variables
        clear selectedNeighbourConfig;  % Clear variables
        clear selectedNeighbourPPR;     % Clear variables
        % Select configurations around the currentConfig
        neighboursConfig = SelectNeighbouringConfig(...
                                currentConfig,CP.Moves,motorID,UConstraints,StepSize);
        % Select a random neighbour
        randIndex = randperm(numel(neighboursConfig));
        selectedNeighbourConfig = neighboursConfig(randIndex(1,1));
        % Evaluate the selected neighbour by compiling a path
        [Targets_XYZ,ppr] = CompilePath(CP.Moves,selectedNeighbourConfig,CP.PPC);
        %Store results of path planning in database
        StoreSimulationsSQL(selectedNeighbourConfig,CP.PPC,ppr,CP.ID,i);
        selectedNeighbourPPR = ppr;
        selectedNeighbourCycleTime = ...
                selectedNeighbourPPR.PathA(size(selectedNeighbourPPR.PathA,1)).EndTime;
        % Check if it is the best, save if it is
        if selectedNeighbourCycleTime < minCycleTime
            minCycleTime = selectedNeighbourCycleTime;
            bestConfig = selectedNeighbourPPR;
        end
        % Determine probability of selection based on cycle time
        diff = selectedNeighbourCycleTime - currentCycleTime;
        probOfSelection = 1/(1+exp((selectedNeighbourCycleTime - currentCycleTime)/T));
        myRand = rand(1);   % Select neighbouring config based on probability
        if myRand < probOfSelection
            currentConfig = selectedNeighbourConfig;
            currentCycleTime = selectedNeighbourPPR.PathA(...
                                            size(selectedNeighbourPPR.PathA,1)).EndTime;
        end
    end
end
end
```

Figure 6.8 Matlab® Code of the SHC Optimising Method

The SHC optimising method has several parameters that need to be tuned to maximise the performance of the algorithm. Firstly, the *StepSize* (lines 1 & 40), as was also used in the RRHC, determines how far to look for neighbouring solutions. As the SHC and RRHC algorithms apply the *StepSize* in similar ways, the *StepSize* value that was tuned for the RRHC will again be used for the SHC. This also simplifies the parameter tuning process, as there are fewer parameter combinations to be evaluated. As shall be seen in the following algorithms, the number of parameter combinations can become large. Making valid assumptions such as this become necessary to limit the evaluation time. The SHC *StepSize* parameter is set as 0.02 m.

The second parameter to be tuned is the *T* value (lines 1 & 59) used in Equation (5.13). This sets the shape of the probability profile. A low *T* value produces a profile approaching that of the RRHC, whereas a high value approaches a random search. Figure 6.9 shows the selection probability profiles generated for several values of *T*. This section analyses which of those profiles is best suited to the problem of optimising the 2DOFPPM dimensions to achieve the fastest cycle-time.

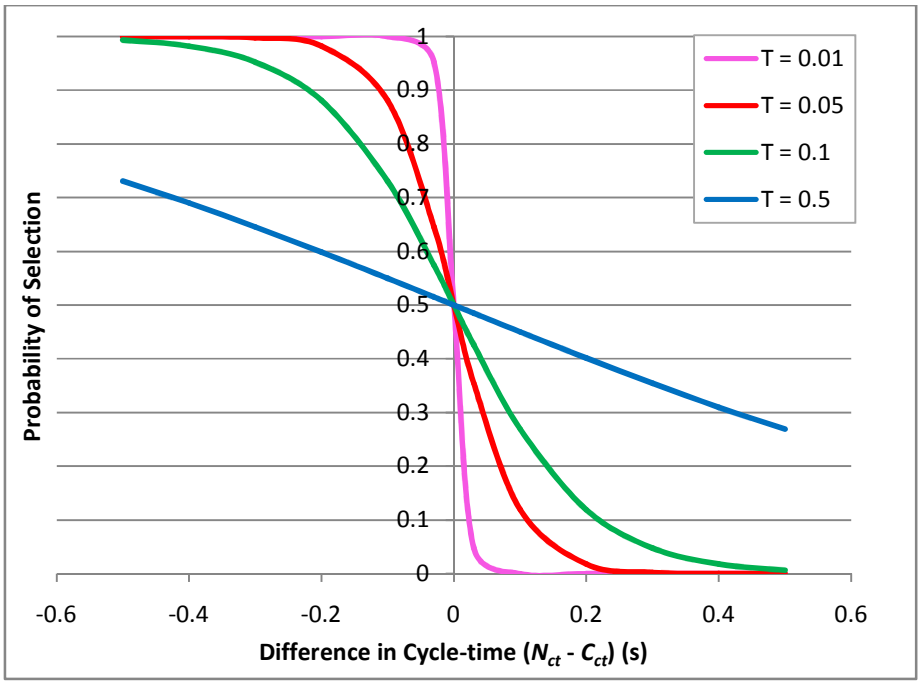

Figure 6.9 Selection probability profiles of four T constants for a SHC

The final parameter to be tuned is the *MaxAttempts* (line 1 & 34). Unlike the RRHC, the SHC has no obvious termination condition. The RRHC is terminated once it has found a solution surrounded by less optimal neighbours. This may result in terminating at a local optimum. The SHC seeks to avoid this by

stochastically selecting a worse neighbour, even if all the neighbours are less optimal. Therefore, the parameter *MaxAttempts* sets the number of iterations that the algorithm will perform. This section determines the value for *MaxAttempts* for this research problem.

In order to evaluate the best values for *T* and *MaxAttempts*, 4 x 150 runs of the SHC were performed. This consisted of 150 runs using each of the four *T* values being examined (0.01, 0.05, 0.1, and 0.5). The performance of the SHC was also monitored in relation to the number of iteration attempts. This provided data to evaluate the best value for *MaxAttempts*. The results of the SHC evaluation are summarised in the following figures. Figure 6.10 shows a histogram distribution of the minimum cycle-time achieved using each of the four *T* constants. A *T* value of 0.05 is seen to perform better than the others after 5000 iterations. The mean, standard deviation and median minimum cycle-times for each *T* value are presented in Table 6.4.

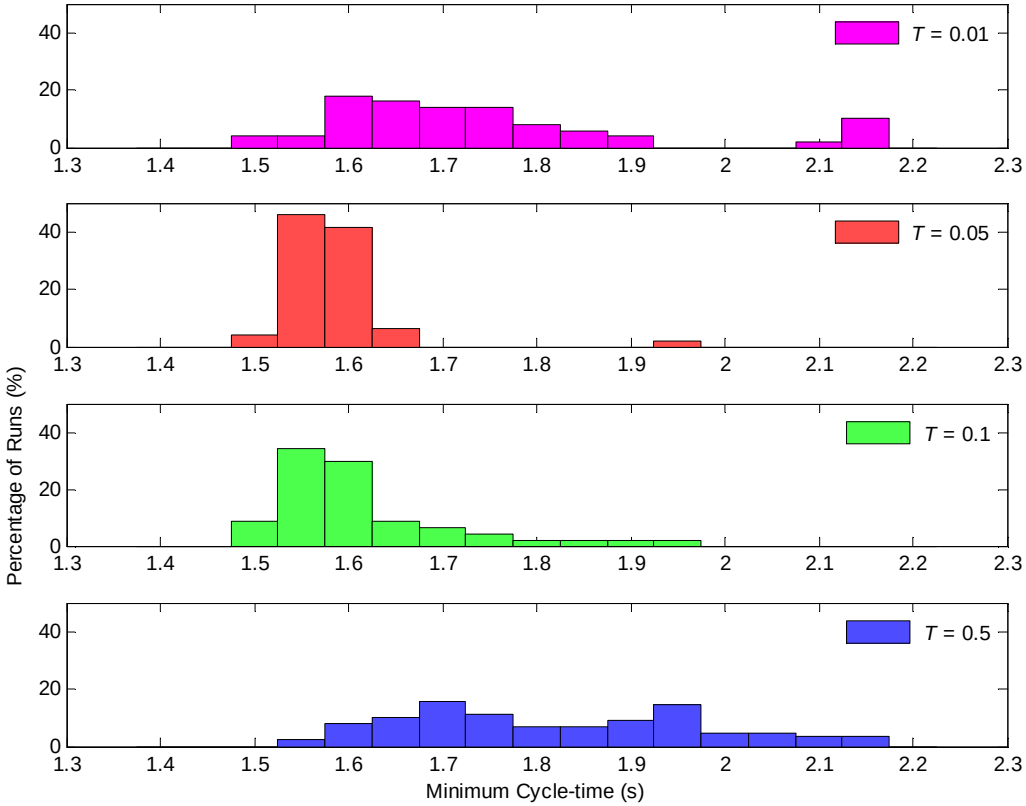

Figure 6.10 Normalised histogram of minimum cycle-time achieved by four different T values using the SHC method after 5000 iterations. Based on 150 individual runs.

Table 6.4 Mean and Median minimum cycle-times for four different values of *T*

T	Mean Minimum Cycle-time (s)	Minimum Cycle-time Standard Deviation (s)	Median Minimum Cycle-time (s)
0.01	1.79	0.22	1.73
0.05	1.61	0.07	1.60
0.1	1.64	0.10	1.61
0.5	1.85	0.17	1.81

While the mean and median minimum cycle-times for a *T* value of 0.05 are shorter than the other three values of *T*, it is useful to validate this statistically using the Wilcoxon-Mann-Whitney U-test, as done with the *StepSizes* of the RRHC. The optimisation results achieved with a *T* value of 0.05 are compared to the results obtained using the other three *T* values. The U-test results can be seen in Table 6.5, where the null hypothesis is rejected for *T* = 0.01 and 0.5 with a 95 % confidence interval. However there is no statistically significant difference with the minimum cycle-time achieved by *T* = 0.1 and *T* = 0.05.

Table 6.5 Wilcoxon-Mann-Whitney test results comparing *T* = 0.05 to the other values of *T*

T	Rejection of Null-Hypothesis	p-Value
0.01	1	1.83×10^{-10}
0.1	0	0.240
0.5	1	5.60×10^{-18}

So far the performance of the SHC with different values of *T* has only considered the results after 5000 iterations (that is, *MaxAttempts* = 5000). The histogram plots in Figure 6.11 show the distribution of the minimum cycle-times after 100, 1000 and 5000 attempts. The median minimum cycle-time is evaluated for each value of *T* at 100, 1000 and 5000 attempts, and is shown in Table 6.6. After 100 attempts, a *T* value of 0.1 is found to give the lowest average minimum cycle-time. However, 0.05 is found to produce the lowest mean minimum cycle-time after both 1000 and 5000 attempts. The significance of these results is evaluated using the Wilcoxon-Mann-Whitney U-test with a 95 % confidence interval. The results of this test are shown in Table 6.7. After 100 attempts, there is no difference in the minimum cycle-time achieved using T values 0.01, 0.05 and 0.1. Using a T value of 0.5 performs worse than the other three values tested. After both 1000 and 5000 attempts the T value of 0.05 proves better than

0.01 and 0.5, but there is no significant difference in the performance when compared to a T value of 0.1. Therefore, T can be set to either 0.05 or 0.1 for best results of the SHC based on the values tested.

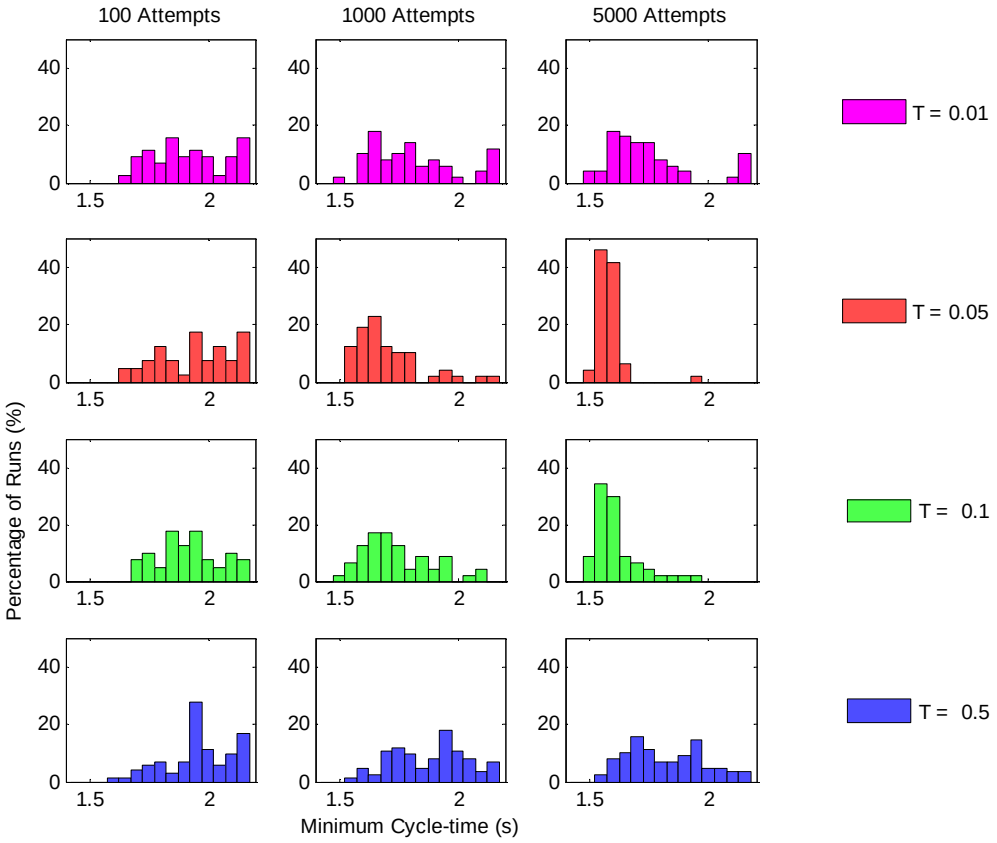

Figure 6.11 Normalised histograms of minimum cycle-times using four T values after 100, 1000 and 5000 attempts

Table 6.6 Median Minimum cycle-times for different *T* values after different number of attempts

T	Median Minimum Cycle-Time (s)		
	100 Attempts	1000 Attempts	5000 Attempts
0.01	1.98	1.82	1.73
0.05	2.01	1.69	1.60
0.1	1.97	1.74	1.61
0.5	2.02	1.96	1.81

Table 6.7 Wilcoxon-Mann-Whitney test results. Comparing *T* = 0.1 to the other values of *T* for 100 attempts, and *T* = 0.05 to the other values of *T* for 1000 and 5000 attempts.

T	100 Attempts (*T* = 0.1)		1000 Attempts (*T* = 0.05)		5000 Attempts (*T* = 0.05)	
	Rejection of Null-Hypothesis	p-Value	Rejection of Null-Hypothesis	p-Value	Rejection of Null-Hypothesis	p-Value
0.01	0	0.915	1	6.18×10^{-4}	1	1.83×10^{-10}
0.05	0	0.501	N/A	N/A	N/A	N/A
0.1	N/A	N/A	0	0.156	0	0.240
0.5	1	0.030	1	7.50×10^{-10}	1	5.60×10^{-18}

The other parameter to be tuned in the SHC algorithm is *MaxAttempts*. It is important to look at the minimum cycle-time achieved as a function of the attempts required. This can give an indication of how many iteration attempts are required until a near optimal solution is expected to be found. Figure 6.12 shows that for all values of *T*, the greatest improvement in finding a solution with a minimum cycle-time, is achieved within the first 500 iteration attempts. The mean minimum cycle-time continues to improve, but at a slowing rate, right up until 5000 attempts. However, a value of 2500 attempts is a suitable compromise for the SHC algorithm to find a configuration that produces a near minimum cycle-time.

Figure 6.13 shows the computation time required for the SHC to achieve a given mean minimum cycle-time. Four lines are plotted for each of the *T* values tested. It can be seen that by 2500 seconds, all of the *T* values have come close to reaching an 'optimal' solution. *T* values of 0.05 and 0.1 easily outperform the values of 0.01 and 0.5.

The SHC is a modification of the RRHC which introduces a probability to the selection process. This allows the SHC to escape from local optima. Several parameters that affect the performance of the SHC have been tuned in this section. Firstly, the *StepSize* value is chosen to be the same as that of the RRHC and is set at 0.2 m. Secondly, the value *T* sets the shape of the probability selection curve. Four values were tested and a value of 0.05 was chosen to provide the fastest convergence on the optimal 2DOFPPM dimensions. Finally, as the SHC has no obvious terminating condition, a value had to be set to limit the number of iteration attempts. 2500 attempts were shown to provide a suitable number of iterations to converge on a near optimal set of dimensions. Therefore the parameter *MaxAttempts* has been set at 2500.

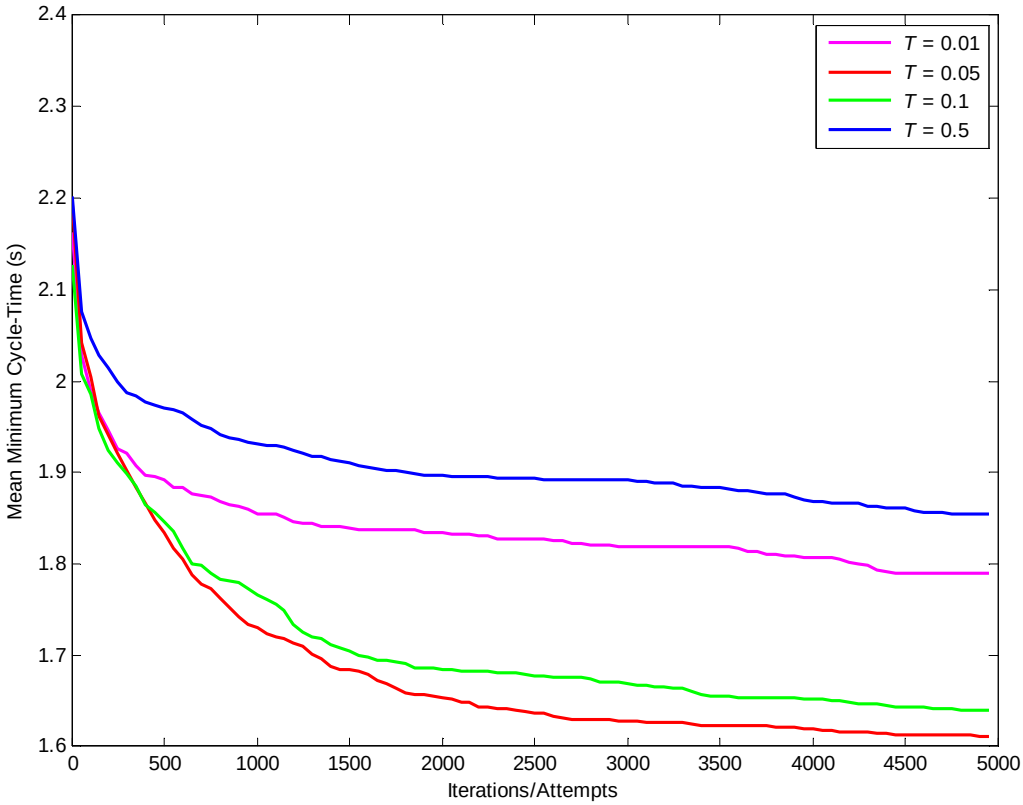

Figure 6.12 Mean minimum cycle-time achieved relative to the number of iteration attempts for four values of T

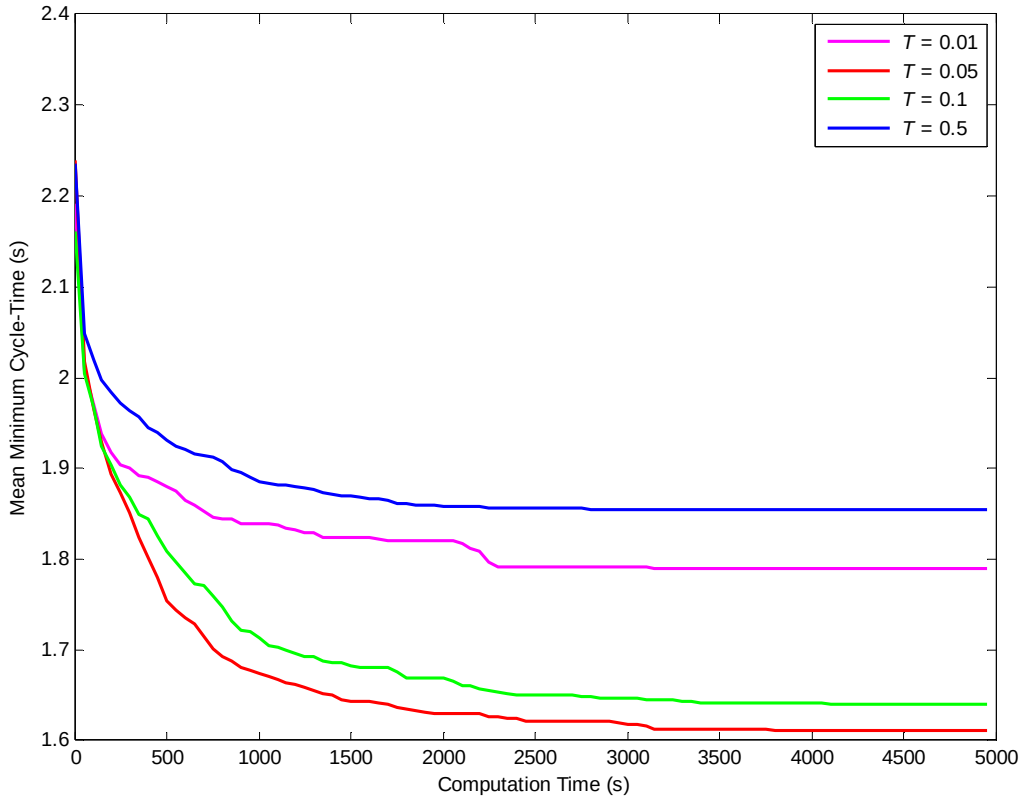

Figure 6.13 Computation time versus mean minimum cycle-time for four *T* values of the SHC

6.1.3 Simulated Annealing

The SHC algorithm can be modified to produce an algorithm known as *Simulated Annealing* (*SA*). SA differs from SHC in two ways:

- The main difference is that the parameter *T* is varied during the optimising process. *T* starts out large and is reduced over a number of iterations.
- Unlike the SHC, SA always accepts solutions if they are better than the current solution.

SA gets its name from an analogy to the thermodynamics process of slowing cooling a crystal so that it forms in a state of lowest energy. In the same way, the SA algorithm slowly 'cools' the value of *T* so that the algorithm finds the lowest value of a minimisation problem. With an initially high value of *T*, the SA has a high 'energy' state and the search method is closer to a random search technique. As *T* is reduced, the optimisation process becomes closer to a standard Hill Climber. Figure 6.14 shows an example of the selection probability profile of a SA, as applied to optimising the 2DOFPPM for minimum cycle-time. It can be noted that a better solution (that is, one with a lower cycle-time) is always selected with a probability of 1. Also, the probability of selecting a weaker solution starts out greater but is reduced over successive iterations as the value of *T* is reduced. Near the end of the optimising process the SA selection profile becomes close to that of a Hill Climber.

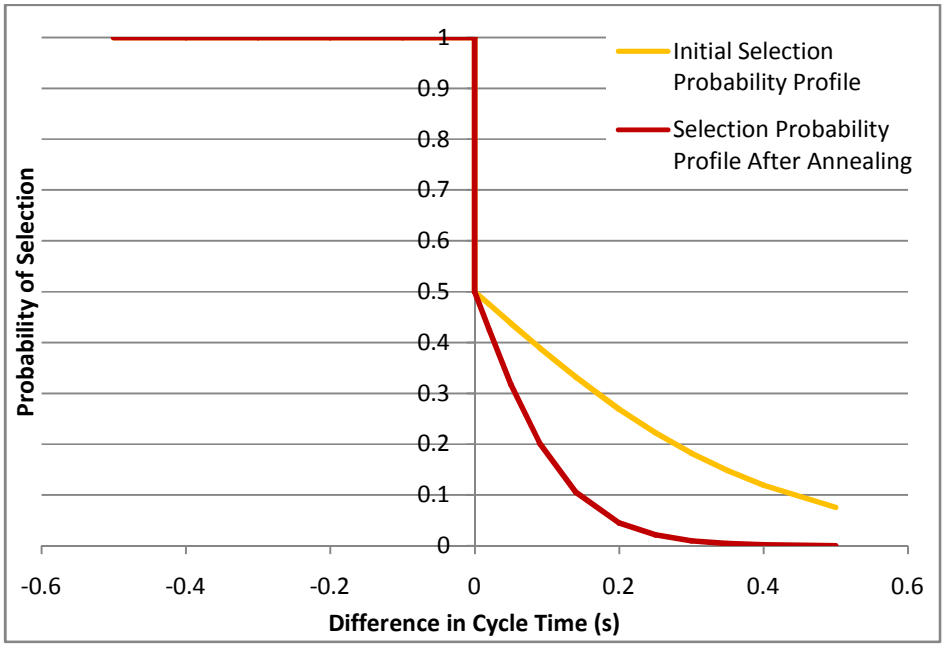

Figure 6.14 SA selection probability profile before and after annealing

Equation (5.14) is used to determine the probability of selection. This is the same equation as that used in the SHC, but rather than *T* being constant, it is reduced as the iterations increase. *T* is attenuated after a set number of iterations using the formula shown in Equation (5.15).

$$P = \frac{1}{1 + e^{\frac{(N_{CT} - C_{CT})}{T}}} \tag{5.14}$$

$$T = T * T_{attenuation} \tag{5.15}$$

where P is the probability of selection, N_{CT} and C_{CT} are the cycle-times of the neighbouring and current solution respectively, T is a constant that determines the shape of the selection probability profile and $0 < T_{attenuation} < 1$ is a value to attenuate the value of T over time.

The Matlab® code developed to execute the SA is presented in Figure 6.15 and Figure 6.16. This implementation is similar to the SHC (refer Figure 6.8). The main differences are that there are now two coded loops; the inner loop (lines 53-95) which behaves like the SHC's loop and an outer loop (lines 50-98) which alters the value of *T* at each iteration (line 96). The other variation is that the selection process is now conditional on whether the selected neighbour's cycle-time is faster or slower than the current solution's cycle-time (lines 82-93).

Once again, the *StepSize* (lines 1 & 61) obtained from tuning the RRHC is used as the *StepSize* for the SA. This is because the techniques do not vary in this regard. Therefore, the *StepSize* is 0.02 m. The number of iterations of the inner loop is controlled by the parameter *MaxAttempts1* (lines 1 & 53). Several values were chosen, and the resulting performances evaluated as presented in this section. The values being tested for *MaxAttempts1* are 200, 500 and 2000. After some enumerative testing, the outer loop parameter *MaxAttempts2* (lines 2 & 50) was set at 5. This provided enough 'cooling' for the SA to work effectively.

Optimisation runs were also made with lower values of *MaxAttempts1* (10 and 40), but these did not provide enough iterations for the optimisation process to complete. The results using these values of *MaxAttempts1* are included in Appendix D.

```
function OptimiseConfigurationSA(CP,TermCond,UConstraints,StepSize,MaxAttempts1,...
                                                              MaxAttempts2,T,Attenuation)
% Uses a random restart hill climber with simulated annealing to narrow on a
%  time-minimum configuration
% VARIABLES:
% CP - Cycle Path class containing geometric details of the path
% TermCond - Termination Condition class detailing conditions of terminating process
% UConstraints - User Constraints class
% StepSize - size of steps (in m) to evaluate neighbouring configurations
% MaxAttempts1 - maximum number of attempts/iterations in the inner loop of algorithm
%                   before 'cooling' takes place
% MaxAttempts2 - maximum number of attempts/iterations of the outer loop in algorithm.
%                   The number of 'cooling' steps taking place
% T - constant in algorithm that affects probability of selection
% Attenuation - the 'cooling' factor reducing the probability of selecting a less optimal
%               configuration as time goes on

% Store path and user constraint data
StorePathsUserConstraintsSQL(CP,UConstraints);

for i=1:TermCond.Iterations     % Run Simulated Annealer for a number of iterations

    % Select 'random' motor details from database
    [motorID,newPPC] = SelectMotor(CP.PPC,UConstraints);

    CP.PPC = newPPC;   % Assign Path Planning Constraints (PPC) of motor to Cycle Path(CP)

    % Select random configuration that reaches all move targets
    config = SelectRandomConfig(CP.Moves,motorID,UConstraints);

    try
        % Compile path using Configuration and Path Planning Constraints (PPC)
        % Path Planning Results (ppr) are returned along with positional and zone data
        % about targets
        [Targets_XYZ,ppr] = CompilePath(CP.Moves,config,CP.PPC);
    catch exception
        % Skip to next iteration if exception occurs due to config unable to meet targets
        continue;
    end

    % Store results of path planning in database
    StoreSimulationsSQL(config,CP.PPC,ppr,CP.ID,i);

    minCycleTime = ppr.PathA(size(ppr.PathA,1)).EndTime; % Set best cycletime acheived
    bestConfig = config;                % Set the best Configuration
    currentConfig = config;             % Set the current Configuration
    currentCycleTime = minCycleTime;    % Set cycletime acheived by currentConfig

    attempts2 = 0;  % Reset counter
    while attempts2 < MaxAttempts2

        attempts1 = 0;  % Reset counter
        while attempts1 < MaxAttempts1
            clear neighboursPPR;            % Clear variables
            clear neighboursConfig;         % Clear variables
            clear selectedNeighbourConfig;  % Clear variables
            clear selectedNeighbourPPR;     % Clear variables

            % Select configurations around the currentConfig
            neighboursConfig = SelectNeighbouringConfig(...
                                    currentConfig,CP.Moves,motorID,UConstraints,StepSize);

            % Select a random neighbour
            randIndex = randperm(numel(neighboursConfig));
            selectedNeighbourConfig = neighboursConfig(randIndex(1,1));

```

Figure 6.15 Matlab® Code of the SA Optimising Method (Part 1/2)

```
67              % Evaluate the selected neighbour
68              [Targets_XYZ,ppr] = CompilePath(CP.Moves,selectedNeighbourConfig,CP.PPC);
69
70              StoreSimulationsSQL(selectedNeighbourConfig,CP.PPC,ppr,CP.ID,i);
71              selectedNeighbourPPR = ppr;
72              selectedNeighbourCycleTime = ...
73                  selectedNeighbourPPR.PathA(size(selectedNeighbourPPR.PathA,1)).EndTime;
74
75              % Check if it is the best, save if it is
76              if selectedNeighbourCycleTime < minCycleTime
77                  minCycleTime = selectedNeighbourCycleTime;
78                  bestConfig = selectedNeighbourPPR;
79              else
80
81              % check if it is better than the current config/cycletime
82              if selectedNeighbourCycleTime < minCycleTime
83                  % Replace currentConfig with neighbour
84                  currentConfig = selectedNeighbourConfig;
85                  currentCycleTime = selectedNeighbourCycleTime;
86              else
87                  % Determine probability of selection based on cycletime and the ...
88                  %  'cooling' process
89                  probOfSelection = ...
90                            1/(1+exp((selectedNeighbourCycleTime - currentCycleTime)/T));
91
92                  % Select neighbouring config based on probability
93                  myRand = rand(1);
94                  if myRand < probOfSelection
95                      currentConfig = selectedNeighbourConfig;
96                      currentCycleTime = selectedNeighbourPPR.PathA(...
97                                              size(selectedNeighbourPPR.PathA,1)).EndTime;
92                  end
93              end
94              attempts1 = attempts1+1;
95          end
96          T=Attenuation*T;    % Reduce T by an amount over time ('cooling')
97          attempts2 = attempts2+1;
98          end
99      end
100 end
```

Figure 6.16 Matlab® Code of the SA Optimising Method (Part 2/2)

Three values of *T* (lines 2 & 90) were chosen and analysed with three different attenuation rates, $T_{attenuation}$ (lines 2 & 96). The values of *T* were 0.05, 0.2 and 0.5. The three attenuation rates were 0.7, 0.8 and 0.9. Figure 6.17 shows the selection probability profiles over time for the nine possible combinations of these values. The first column shows the initial selection probability, the central column shows the probability of selection after some cooling has taken place, and the final column shows the final selection probability profiles at the end of the SA optimisation process.

In order to tune the SA parameters, 100 runs were made for each of the 27 combinations of *MaxAttempts1* (= 200, 500, 2000), *T* (= 0.05, 0.2, 0.5) and $T_{attenuation}$ (= 0.7, 0.8, 0.9). The results were then statistically analysed to find the best combination. These results are presented in the following pages.

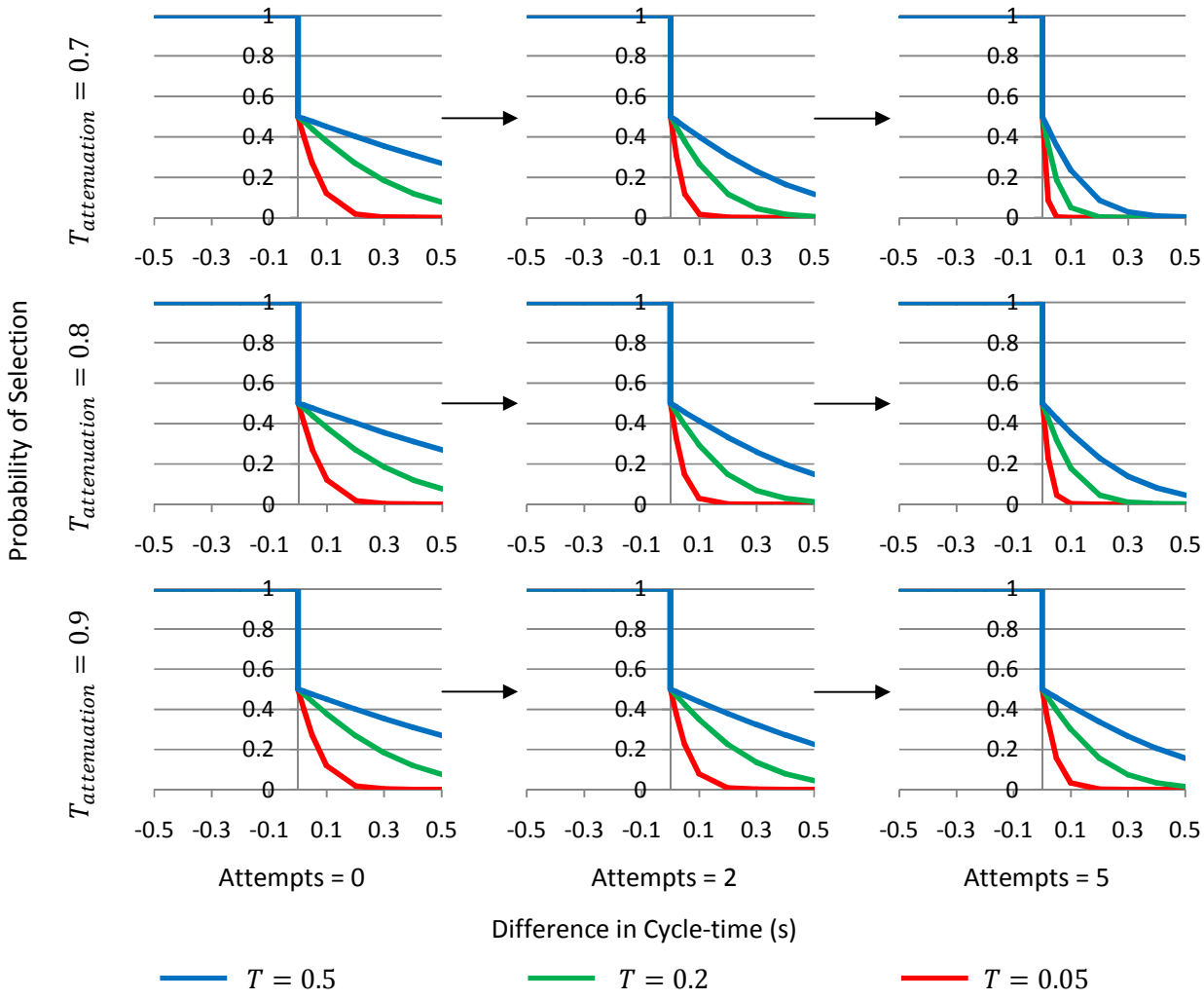

Figure 6.17 Selection probability profiles for three values of *T* at three different attenuation rates over time

Figure 6.18 shows the distributions of minimum cycle-times from 100 runs of the SA algorithm using a *MaxAttempts1* value of 200 and nine different combinations of T and $T_{attenuation}$. The means, standard deviations and medians of these results are summarised in Table 6.8. It can be seen that there is little variation in the minimum cycle-time due to the different values of T and $T_{attenuation}$. This is supported by the Wilcoxon-Mann-Whitney test results in Table 6.9 which compares the median minimum cycle-time of the SA algorithm, using $T = 0.05$ and $T_{attenuation} = 0.9$, to each of the other eight parameter settings. The test shows that, in all but two of the eight other parameter combinations, there is no statistical difference in the performance of using $T = 0.05$ and $T_{attenuation} = 0.9$ as parameters in the SA algorithm.

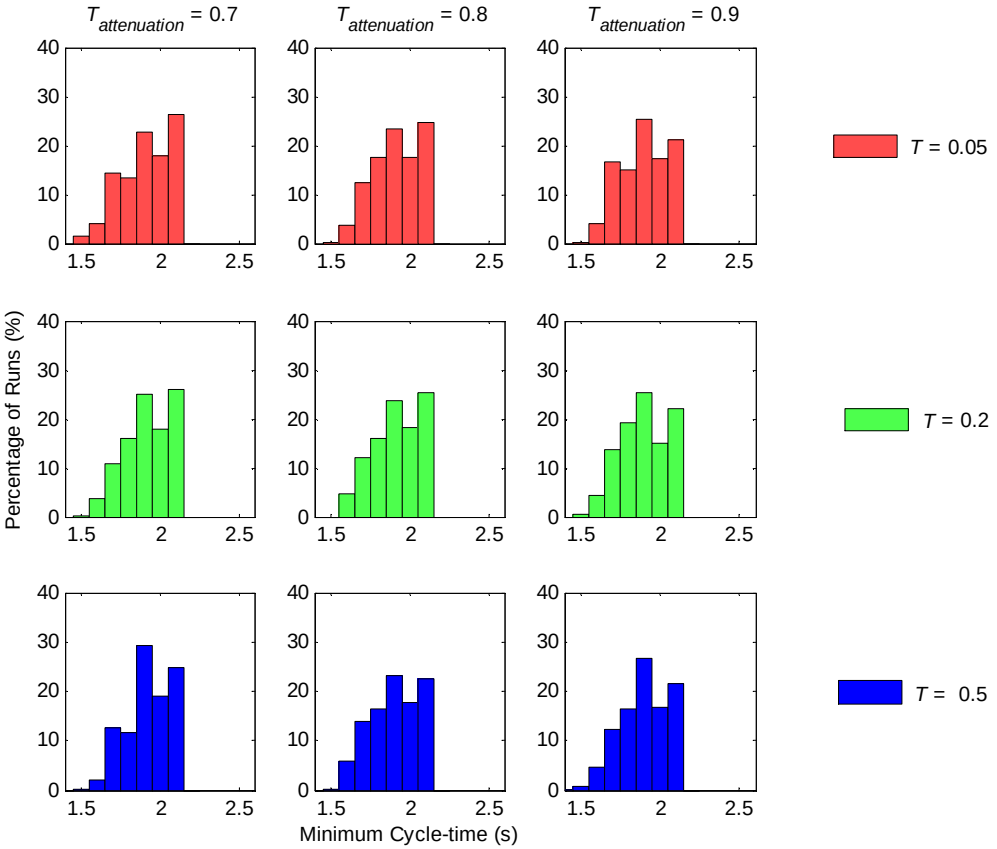

Figure 6.18 Normalised histograms of minimum cycle-times for three T values with three Tattenuation rates. MaxAttempts1 = 200.

Table 6.8 Mean (μ), standard deviation (σ) and median (M) minimum cycle-times for *MaxAttempts1* = 200

T	$T_{attenuation}$ = 0.7			$T_{attenuation}$ = 0.8			$T_{attenuation}$ = 0.9		
	μ (s)	σ (s)	M (s)	μ (s)	σ (s)	M (s)	μ (s)	σ (s)	M (s)
0.05	2.02	0.19	2.01	2.01	0.18	2.00	2.00	0.19	1.99
0.2	2.02	0.18	2.01	2.02	0.19	2.02	1.99	0.18	1.99
0.5	2.03	0.18	2.00	2.01	0.18	2.00	2.01	0.19	2.00

Figure 6.19 shows the distributions of minimum cycle-times from 100 runs of the SA algorithm using a *MaxAttempts1* value of 500 and nine different combinations of *T* and $T_{attenuation}$. As was the result when using 200 as the *MaxAttempts1* value, there is little variation in the minimum cycle-time distribution due to the different values of *T* and $T_{attenuation}$, as seen in Table 6.10. The Wilcoxon-Mann-Whitney test results in

Table 6.11 compare the median minimum cycle-time of the SA algorithm, using T = 0.05 and $T_{attenuation}$ = 0.7, to each of the other eight parameter settings. The test shows that, in five of the eight other parameter combinations, there is no statistical difference in the performance of using T = 0.05 and $T_{attenuation}$ = 0.7 as parameters in the SA algorithm.

Table 6.9 Wilcoxon-Mann-Whitney test results. Comparing T = 0.05, $T_{attenuation}$ = 0.9 to the other combinations of values tested with *MaxAttempts1* = 200

T	$T_{attenuation}$ = 0.7		$T_{attenuation}$ = 0.8		$T_{attenuation}$ = 0.9	
	Rejection of Null-Hypothesis	p-Value	Rejection of Null-Hypothesis	p-Value	Rejection of Null-Hypothesis	p-Value
0.05	1	0.044	0	0.294	N/A	N/A
0.2	0	0.062	0	0.053	0	0.988
0.5	1	0.024	0	0.513	0	0.316

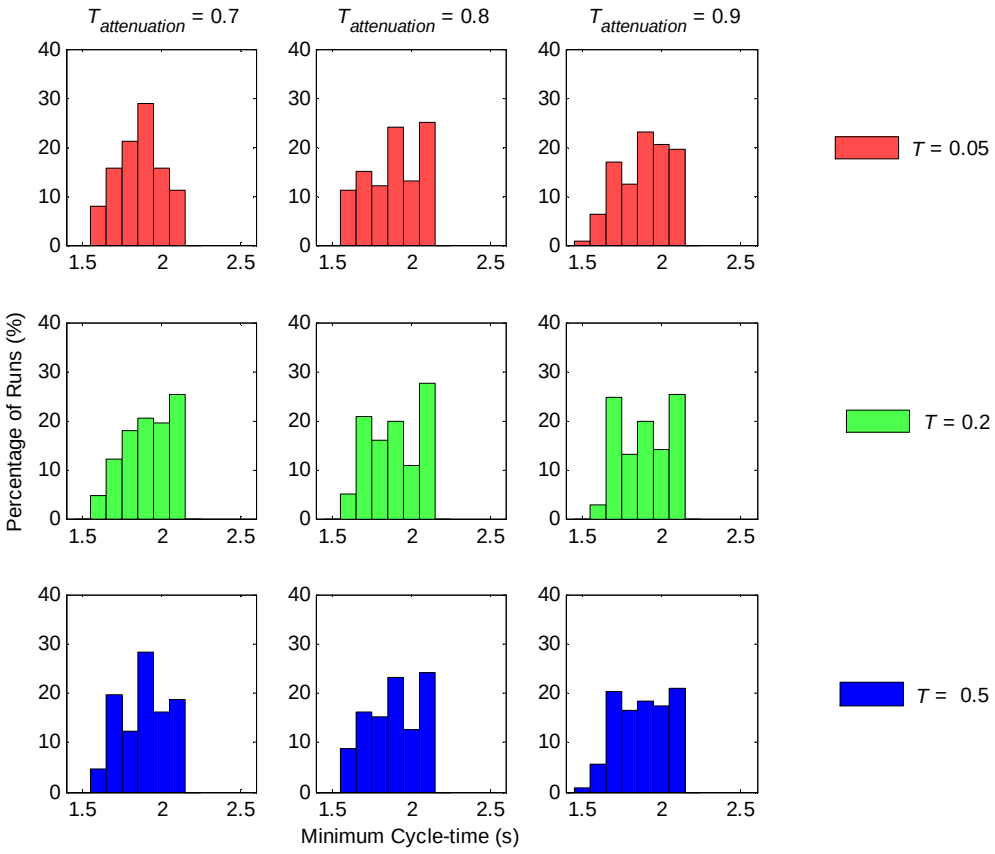

Figure 6.19 Normalised histograms of minimum cycle-times for three T values with three Tattenuation rates. MaxAttempts1 = 500.

Table 6.10 Mean (μ), standard deviation (σ) and median (M) minimum cycle-times for *MaxAttempts1* = 500

T	$T_{attenuation}$ = 0.7			$T_{attenuation}$ = 0.8			$T_{attenuation}$ = 0.9		
	μ (s)	σ (s)	M (s)	μ (s)	σ (s)	M (s)	μ (s)	σ (s)	M (s)
0.05	1.95	0.17	1.96	2.01	0.22	2.0	1.99	0.18	1.99
0.2	2.00	0.18	2.00	2.00	0.20	2.0	1.97	0.18	1.98
0.5	1.99	0.18	1.99	1.98	0.19	1.99	1.98	0.19	1.99

Table 6.11 Wilcoxon-Mann-Whitney test results. Comparing *T* = 0.05, $T_{attenuation}$ = 0.7 to the other combinations of values tested with *MaxAttempts1* = 500

T	$T_{attenuation}$ = 0.7		$T_{attenuation}$ = 0.8		$T_{attenuation}$ = 0.9	
	Rejection of Null-Hypothesis	p-Value	Rejection of Null-Hypothesis	p-Value	Rejection of Null-Hypothesis	p-Value
0.05	N/A	N/A	1	0.041	0	0.093
0.2	1	0.014	1	0.044	0	0.273
0.5	0	0.054	0	0.200	0	0.205

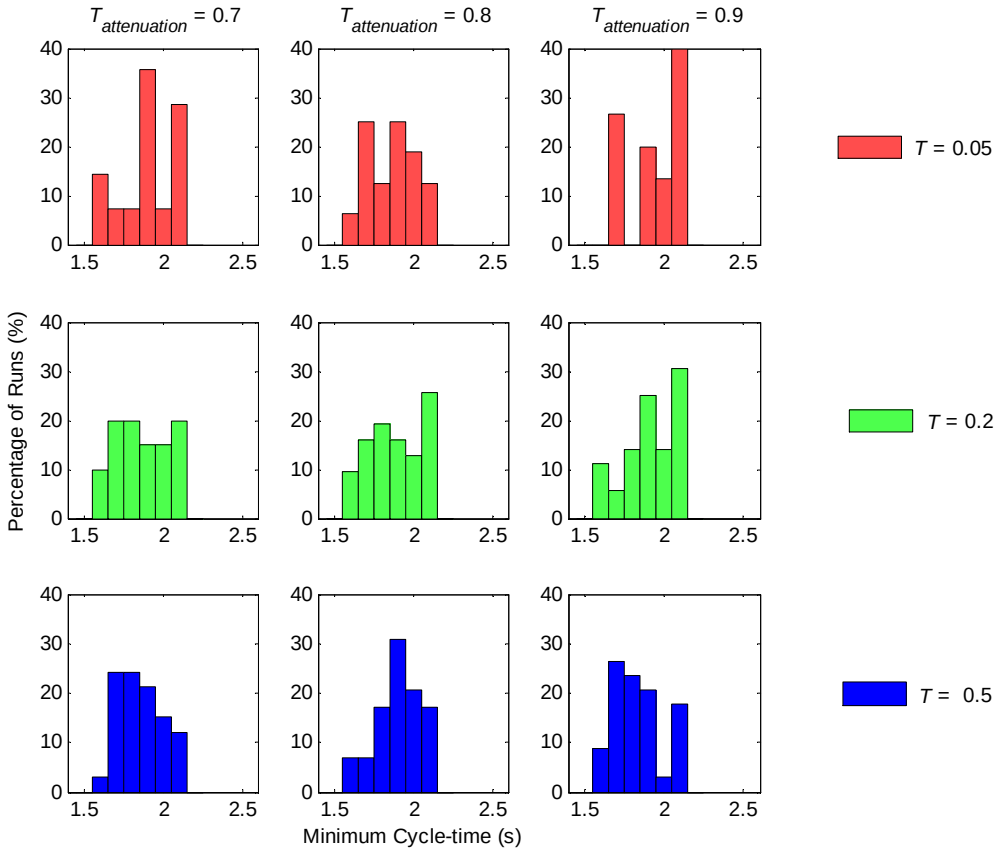

Figure 6.20 Normalised histograms of minimum cycle-times for three T values with three Tattenuation rates. MaxAttempts1 = 2000.

Figure 6.20 shows the distributions of minimum cycle-times from 100 runs of the SA algorithm using a *MaxAttempts1* value of 2000 and nine different combinations of T and $T_{attenuation}$. While not as uniform as the results when using values of 200 and 500 for *MaxAttempts1*, there is still little variation in the minimum cycle-time distribution due to the different values of T and $T_{attenuation}$. The mean, standard deviation and median results are summarised in Table 6.12. The Wilcoxon-Mann-Whitney test results in Table 6.13 compare the median minimum cycle-time of the SA algorithm, using T = 0.05 and $T_{attenuation}$ = 0.7, to each of the other eight parameter settings. The test shows that the combination of T and $T_{attenuation}$ producing the lowest median cycle-time, are statistically better than four of the other eight, T and $T_{attenuation}$ combinations.

Table 6.12 Mean (μ), standard deviation (σ) and median (M) minimum cycle-times for *MaxAttempts1* = 2000

T	$T_{attenuation}$ = 0.7			$T_{attenuation}$ = 0.8			$T_{attenuation}$ = 0.9		
	μ (s)	σ (s)	M (s)	μ (s)	σ (s)	M (s)	μ (s)	σ (s)	M (s)
0.05	2.05	0.24	1.98	1.93	0.17	1.97	2.04	0.21	2.09

0.2	1.97	0.20	1.97	1.96	0.19	1.99	1.99	0.17	2.00
0.5	1.91	0.17	1.92	2.00	0.16	2.00	1.89	0.16	1.86

Table 6.13 Wilcoxon-Mann-Whitney test results. Comparing T = 0.5, $T_{attenuation}$ = 0.9 to the other combinations of values tested with *MaxAttempts1* = 2000

T	$T_{attenuation}$ = 0.7		$T_{attenuation}$ = 0.8		$T_{attenuation}$ = 0.9	
	Rejection of Null-Hypothesis	p-Value	Rejection of Null-Hypothesis	p-Value	Rejection of Null-Hypothesis	p-Value
0.05	1	0.021	0	0.473	1	0.016
0.2	0	0.152	0	0.103	1	0.018
0.5	0	0.451	1	0.006	N/A	N/A

The results above compare the effect the parameters, T and $T_{attenuation}$, have on the minimum cycle-time. For each of the three MaxAttempts1 values there was little difference in the minimum cycle-time performance due to the values of T and $T_{attenuation}$. Another important aspect in evaluating an algorithm is how long it takes to perform the optimisation. Figure 6.21 through to Figure 6.23 show the mean minimum cycle-time achieved relative to the length of time the optimisation was run for. Each graph plots the nine combinations of T and $T_{attenuation}$.

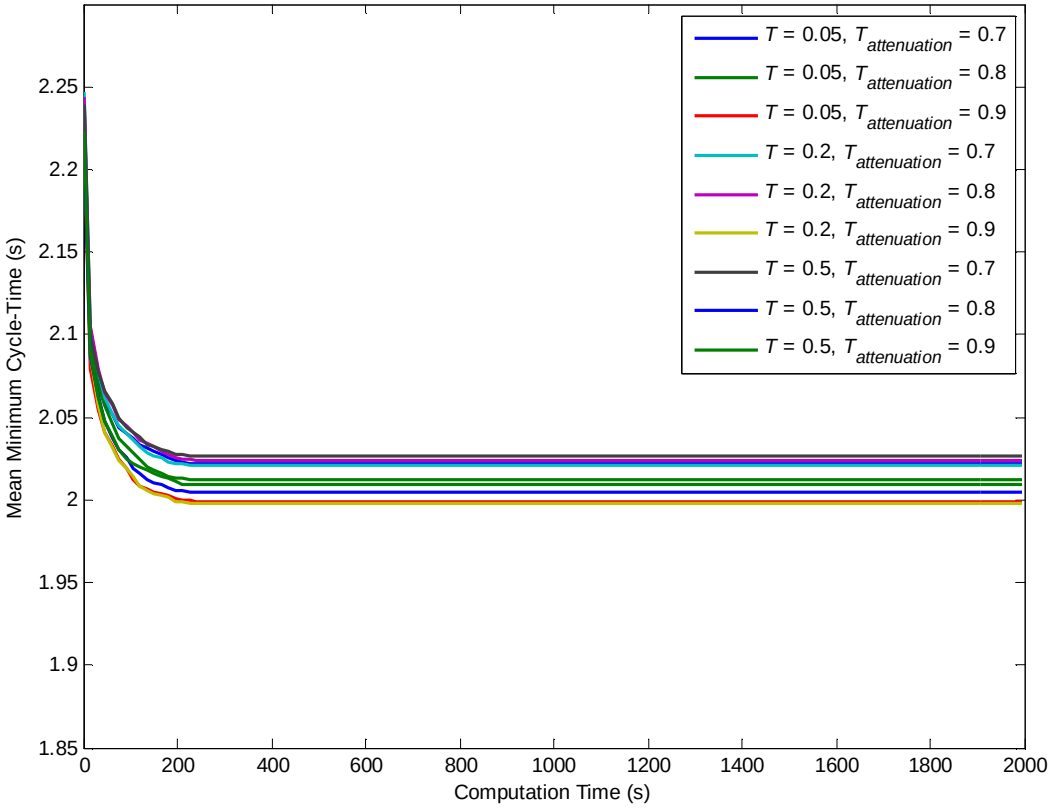

Figure 6.21 Mean minimum cycle-time versus computation time with *MaxAttempts1* = 200 for nine combinations of *T* and $T_{attenuation}$.

Figure 6.21 presents the mean minimum cycle-time achieved by the SA, using a value of 200 for *MaxAttempts1*, over the time taken during computation. Nine combinations of *T* and $T_{attenuation}$ are plotted. There is very little difference between the performances of the different parameter combinations. The 'optimised' cycle-time of between 2 and 2.05 seconds is reached after approximately 200 seconds of computation time, when *MaxAttempts1* is set at 200.

In comparison, Figure 6.22 shows the same data but when *MaxAttempts1* is set at 500. In this case, it takes approximately 550 seconds of computation to reach an 'optimised' state. Unlike the situation when *MaxAttempts1* was set at 200, for *MaxAttempts1* being 500 there is a clearly better performing parameter combination of $T = 0.05$ and $T_{attenuation} = 0.7$.

Figure 6.23 represents the SA computation time performance but with *MaxAttempts1* being set at 2000. The time taken for each parameter set to reach its mean minimum cycle-time varies between 300 and 1400 seconds. The mean minimum cycle-time reached by each parameter also varies greatly between different combinations of *T* and $T_{attenuation}$.

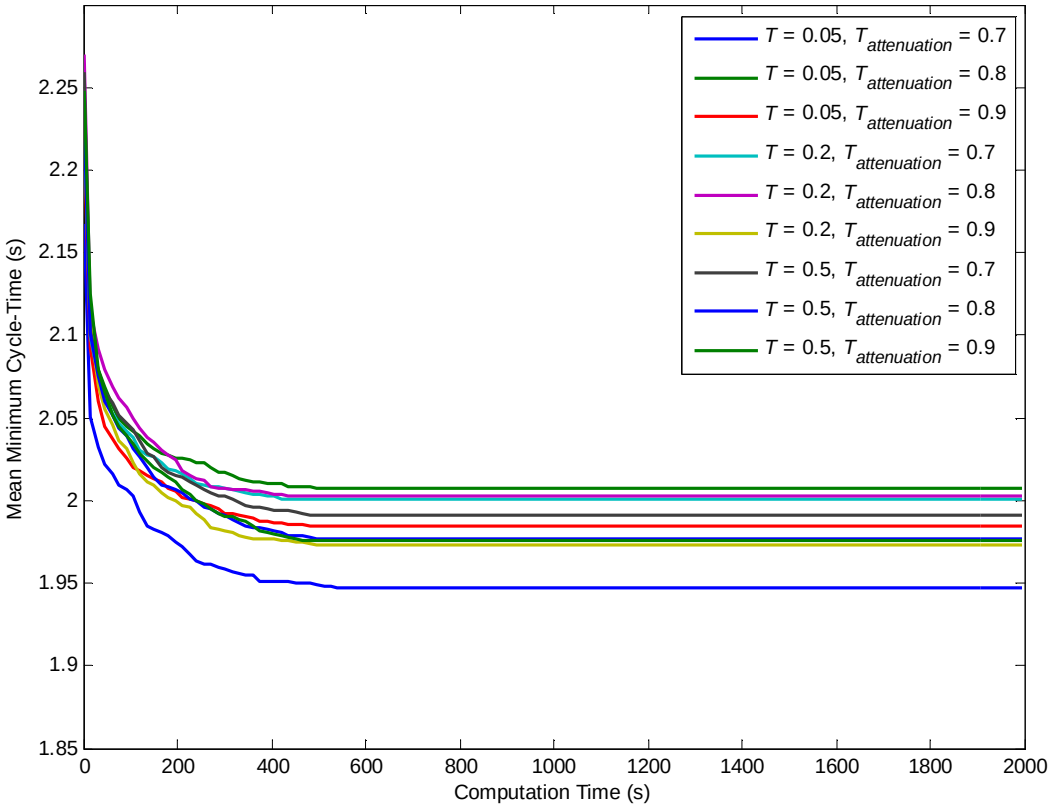

Figure 6.22 Mean minimum cycle-time versus computation time with *MaxAttempts1* = 500 for nine combinations of *T* and $T_{attenuation}$.

The SA technique of optimisation always accepts a better solution (configuration), and accepts a weaker solution with some probability relative to the strength of the solution. The probability of selecting a weaker solution decreases, or 'cools', as the algorithm progresses.

There are five parameters used in the SA. The first is the *StepSize*, like the RRHC and SHC algorithms, this determines how far away to look for neighbouring solutions. This is set at 0.02 m based on the results from the RRHC analysis in Section 6.1.1. The other parameters are specific to the SA algorithm. *MaxAttempts2* was found by enumerative testing and set at 5. *MaxAttempts1*, *T* and $T_{attenuation}$ have been analysed by statistical methods and the results presented here. The values of *T* and $T_{attenuation}$ tended to have little effect on the performance of the algorithm as applied to this research problem. Therefore, the focus is turned to the parameter *MaxAttempts1*. When *MaxAttempts1* was higher (2000) a better solution (lower cycle-time) was found. However, this required a longer processing time. These results are due to *MaxAttempts1*, along with *MaxAttempts2*, dictating the number of solutions examined. The time taken for the SA to reach 'optimisation' is relatively short given the algorithm is seeking to find the optimal dimensions of a manipulator, which may take a number of weeks to fully

design. It is for this reason that the value of 2000 is the preferred choice for *MaxAttempts1* out of the three parameter values examined. With *MaxAttempts1* set at 2000, the combination of T = 0.5 and $T_{attenuation}$ = 0.9 is chosen as they find the fastest cycle-time as shown in Figure 6.23 and Table 6.13.

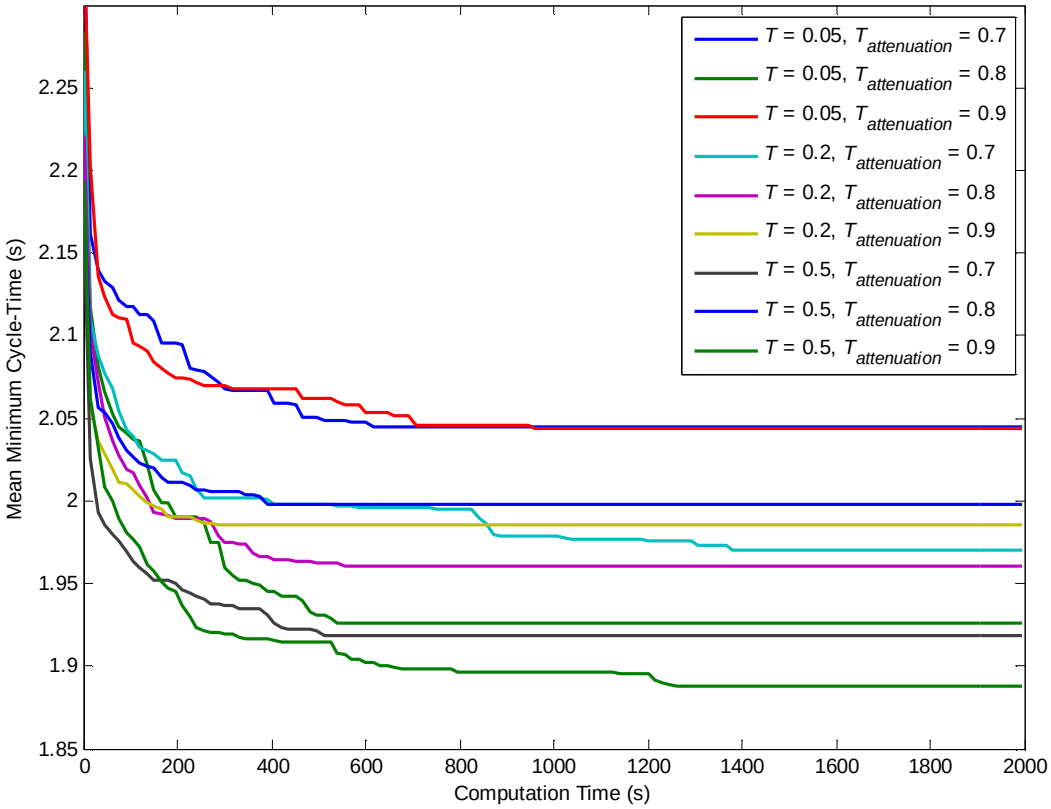

Figure 6.23 Mean minimum cycle-time versus computation time with *MaxAttempts1* = 2000 for nine combinations of *T* and $T_{attenuation}$.

6.1.4 Genetic Algorithm

The three methods presented so far (RRHC, SHC and SA) are all based around the concept of searching neighbouring solutions of a randomly selected solution, in the hope of iteratively making improvements. An alternative is to use an evolutionary approach which aims to produce better solutions by 'breeding' good solutions together. This is modelled on the way organisms adapt and improve in the biological world. The *Genetic Algorithm* (*GA*) is a popular algorithm for implementing an evolutionary optimising technique.

The GA begins by randomly producing a number of samples (in the case of the 2DOFPPM these samples are different dimensional configurations). This group of samples is known as the *population*. The GA performs a number of evolutionary cycles, also known as *generations*. During each generation the

fitness of each individual in the population is evaluated (for the 2DOFPPM this is the path cycle-time). Once the fitness of each sample in the population is evaluated, the population goes through a selection process which chooses a number of samples. The samples are selected based on a probability relating to their fitness. The unselected samples are discarded which reduces the population. To rebuild the population to its full size, *reproduction* occurs. Reproduction consists of the selected samples being 'mated' with each other, through a stochastic process, to produce offspring. The offspring are new samples that consist of traits found in their two parents (*crossover*) and possibly some random variation (*mutation*). These offspring are then placed into the population and another generation begins.

Figure 6.24 shows the first segment of the Matlab® code used to execute the GA. As in the previous methods, the data relating to the path and the constraints on the manipulator are stored in the database (line 14). Where the previous methods selected a single configuration at random, the GA selects a number of random configurations which is known as a *population* (lines 21-34). For each of these dimensional configurations, a motor and its properties are also selected from the database (line 25).

Once the population is initialised it undergoes a number of iterations, or generations, which alter the population in the hope of producing improved configurations (lines 27-244). After a number of generations, where the GA has reproduced new configurations by 'breeding' other configurations in the population, the population can lose diversity. In terms of the biological analogy, this is due to inbreeding. The result of this loss in diversity means the GA is no longer searching in the global search space but effectively becomes stuck in a local minimum. One of the ways to counter this effect is implemented in the second half of the code in Figure 6.24 (lines 52-68). If the difference between the minimum and maximum cycle-times of the population is less than a 0.2 s, 10 % of the inbred population are replaced by new configurations selected randomly from the search space (line 60). These new configurations add diversity to the 'gene' pool and allow the GA to continue to find improved solutions.

```
function OptimiseConfigurationGA(CP,TermCond,UConstraints,PopSize,...
                SelectionSize,MutationRate,MutationAmount)
% Uses a Genetic Algorithm to narrow on time-minimum configuration
% VARIABLES:
% CP - Cycle Path class containing geometric details of the path
% TermCond - Termination Condition class detailing conditions of terminating process
% UConstraints - User Constraints class
% PopSize - Number of individuals in GA population
% SelectionSize - Number of individuals selected for breeding
% MutationRate - Probability of mutation occuring in child (%)
% MutationAmount - The amount of mutation to occur in child (%)

% Store path and user constraint data
StorePathsUserConstraintsSQL(CP,UConstraints);
population = repmat(Configuration,PopSize,1);
popPPC = repmat(PPConstraints,PopSize,1);
popFitness = zeros(PopSize,1);
popCycleTime = zeros(PopSize,1);
popMotorID = zeros(PopSize,1);

%% INITIALISATION - Initialise population by selecting random configurations

for p=1:PopSize
    % Select 'random' motor details from database
    [motorID,newPPC] = SelectMotor(CP.PPC,UConstraints);
    CP.PPC = newPPC; % Assign Path Planning Constraints (PPC) of motor to Cycle Path(CP)

    % Select random configuration that reaches all move targets
    config = SelectRandomConfig(CP.Moves,motorID,UConstraints);

    population(p) = config;
    popPPC(p) = CP.PPC;
    popMotorID(p) = motorID;
end

% Perform GA for a set number of evolution cycles
for i=1:TermCond.Iterations
    % Check if popCycleTimes are too similar and replace some with random configurations
    if i >1
        minct = 500;
        maxct = 0;
        for p=1:PopSize
            if popCycleTime(p) < 5000
               if popCycleTime(p) < minct
                   minct = popCycleTime(p);
               end
               if popCycleTime(p) > maxct
                   maxct = popCycleTime(p);
               end
            end
        end
        if maxct-minct < 0.2   % Population is too inbred!
            % Replace 10% of inbred population with random individuals
            for rp = 1:floor(PopSize/10)
                % Select 'random' motor details from database
                [motorID,newPPC] = SelectMotor(CP.PPC,UConstraints);
                % Assign Path Planning Constraints (PPC) of motor to Cycle Path (CP)
                CP.PPC = newPPC;
                % Select random configuration that reaches all move targets
                config = SelectRandomConfig(CP.Moves,motorID,UConstraints);

                % Randomly select an individual from population for replacement
                replaceP = ceil(PopSize*rand(1));
                population(replaceP) = config;
                popPPC(replaceP) = CP.PPC;
                popMotorID(replaceP) = motorID;
            end
        end
    end
```

Figure 6.24 Matlab® Code of the GA Optimising Method (Part 1/4)

After the population is initialised and an iterative loop is entered into, each configuration is evaluated to determine its fitness. As shown in Figure 6.25 (line 83), this involves planning a trajectory using the *CompilePath()* method (refer Figure F.7 - Figure F.11 in Appendix F) and storing the resulting path in the database (line 85). To limit the amount of computation, the algorithm first checks if the exact same configuration has previously been analysed, and if it has, the previously calculated cycle-time result is returned (line 80). Once the cycle-time for each configuration in the database has been obtained, the fitness of each solution is set as being the inverse of the cycle-time cubed (line 95). This function gave an exponentially increasing fitness to the configurations with faster cycle-times.

Following the evaluation of each configuration, the GA performs a selection process on the population. The code in the lower half of Figure 6.25 shows how the GA selects a percentage of the population based on the fitness of each configuration (lines 98-132). The configurations with higher fitness have a higher probability of being selected (line 124).

With a percentage of the population already selected, Figure 6.26 shows how the GA performs a reproduction action to produce new configurations from the selected configurations. A new population is formed (line 137) and the selected configurations are automatically inserted into it (lines 140-144). To fill the remainder of the new population, the selected configurations are chosen randomly in pairs (lines 154-162) and 'bred' to form new configurations. This breeding process is done in the form of taking some 'genes' (in the case of the 2DOFPPM, this refers to the four dimensions being optimised) from one of the two 'parent' configurations and combining it with the 'genes' of the other 'parent' configuration (lines 164-178). The result is a 'child' configuration that has dimensions from both of the two selected configurations.

While the new configuration is different from both of its 'parent' configurations, the GA also introduces some mutation to help keep diversity in the population. This is documented in Figure 6.27 (lines 180-212). Each of the four dimensions of the new 'child' configuration is, with some probability (*MutationRate*), subject to being altered in this way (lines 181, 189, 197 & 205). The amount that it is altered is set by a parameter called *MutationAmount*.

The coded GA also attempts to keep diversity in the population by ensuring that any new configuration introduced by the reproduction process is unique. This is shown in the bottom half of Figure 6.27 (lines 218-238).

```
70
71
72  %% EVALUATION - Evaluate the performance of each individual in population
73
74      for p=1:PopSize
75          config = population(p);
76          ppc = popPPC(p);
77          try
78              %Check if config has already been simulated,return cycletime if it exists in
79              % database
80              ct = CheckConfigExists(CP.ID,config);
81              if isempty(ct)
82                  % Evaluate the selected individual by compiling a path
83                  [Targets_XYZ,ppr] = CompilePath(CP.Moves,config,ppc);
84                   %Store results of path planning in database
85                  StoreSimulationsSQL(config,ppc,ppr,CP.ID,i);
86
87                  popCycleTime(p) = ppr.PathA(size(ppr.PathA,1)).EndTime;
88              else
89                  popCycleTime(p) = ct;
90              end
91          catch exception
92              %If error occurs give individual poor cycletime so will be repaced nextcycle
93              popCycleTime(p) = 99999;
94          end
95          popFitness(p) = 1/(popCycleTime(p)^3);%Fitness equals inverse of cycletime cubed
96      end
97
98  %% SELECTION - Select sub population from population for breeding based on fitness
99
100     sumFitness = 0;
101     for p=1:PopSize
102        sumFitness = sumFitness + popFitness(p);
103     end
104     % Assign selection probability to each individual in population based on fitness
105     popProb = zeros(PopSize,1);
106     for p=1:PopSize
107        popProb(p) = popFitness(p)/sumFitness;
108     end
109     selectionProb = zeros(PopSize,1);
110     sumProb = 0;
111     for p=1:PopSize
112        selectionProb(p) = sumProb + popProb(p);
113        sumProb = selectionProb(p);
114     end
115
116     % Select a number (SelectionSize) of the population for breeding
117     selectedPop = repmat(Configuration,SelectionSize,1);
118     selectedPopPPC = repmat(PPConstraints,SelectionSize,1);
119     selectedPopMotorID = zeros(SelectionSize,1);
120     selectedPopCycleTime = zeros(SelectionSize,1);
121     for s=1:SelectionSize
122         randnum = rand(1);
123         for p=1:PopSize
124             if selectionProb(p) > randnum
125                 selectedPop(s) = population(p);
126                 selectedPopPPC(s) = popPPC(p);
127                 selectedPopMotorID(s) = popMotorID(p);
128                 selectedPopCycleTime(s) = popCycleTime(p);
129                 break;
130             end
131         end
132     end
133
```

Figure 6.25 Matlab® Code of the GA Optimising Method (Part 2/4)

```
%% REPRODUCTION

    % Add selected parents to new population
    newPopulation = repmat(Configuration,PopSize,1);
    newPopPPC = repmat(PPConstraints,PopSize,1);
    newPopMotorID = zeros(PopSize,1);
    for s=1:SelectionSize
        newPopulation(s) = selectedPop(s);
        newPopPPC(s) = selectedPopPPC(s);
        newPopMotorID(s) = selectedPopMotorID(s);
    end

    % Generate children to fill rest of new population
    for p=SelectionSize:PopSize
        reachable = false;
        unique = true;
        while reachable == false && unique == true;
            % CROSSOVER - children configuration dimensions are a random number between
            % their two parents

            % select two random parents from selectedPop
            randnum1 = ceil(rand(1)*SelectionSize);
            randnum2 = ceil(rand(1)*SelectionSize);
            parent1Config = selectedPop(randnum1);
            parent2Config = selectedPop(randnum2);
            parent1ppc = selectedPopPPC(randnum1);
            parent2ppc = selectedPopPPC(randnum2);
            parent1motorID = selectedPopMotorID(randnum1);
            parent2motorID = selectedPopMotorID(randnum2);

            minlb = min([parent1Config.LengthBase parent2Config.LengthBase]);
            maxlb = max([parent1Config.LengthBase parent2Config.LengthBase]);
            lb = minlb+(maxlb-minlb)*rand(1);   % Set childs base length

            minll = min([parent1Config.LengthLower parent2Config.LengthLower]);
            maxll = min([parent1Config.LengthLower parent2Config.LengthLower]);
            ll = minll+(maxll-minll)*rand(1);   % Set childs distal arm length

            minlu = min([parent1Config.LengthUpper parent2Config.LengthUpper]);
            maxlu = min([parent1Config.LengthUpper parent2Config.LengthUpper]);
            lu = minlu+(maxlu-minlu)*rand(1);   % Set childs proximal arm length

            minwh = min([parent1Config.WorkspaceHeight parent2Config.WorkspaceHeight]);
            maxwh = min([parent1Config.WorkspaceHeight parent2Config.WorkspaceHeight]);
            wh = minwh+(maxwh-minwh)*rand(1);   % Set childs workspace height

```

Figure 6.26 Matlab® Code of the GA Optimising Method (Part 3/4)

```
            % MUTATION - with some probability alter childs dimension
            if rand(1) < MutationRate
                if rand(1) <0.5
                    MutAmount = MutationAmount;
                else
                    MutAmount = -MutationAmount;
                end
                lb = lb * (1+MutAmount);
            end
            if rand(1) < MutationRate
                if rand(1) <0.5
                    MutAmount = MutationAmount;
                else
                    MutAmount = -MutationAmount;
                end
                lu = lu * (1+MutAmount);
            end
            if rand(1) < MutationRate
                if rand(1) <0.5
                    MutAmount = MutationAmount;
                else
                    MutAmount = -MutationAmount;
                end
                ll = ll * (1+MutAmount);
            end
            if rand(1) < MutationRate
                if rand(1) <0.5
                    MutAmount = MutationAmount;
                else
                    MutAmount = -MutationAmount;
                end
               wh = wh * (1+MutAmount);
          end

          % Generate new configuration based on childs dimensions
          [config, reachable] = ...
                  CalculateConfig(lb,lu,ll,wh,CP.Moves,parent1motorID,UConstraints);

          % Check configuration is unique in population
          for pp=1:PopSize
              existingConfig = newPopulation(pp);
              try
                  if config.LengthBase == existingConfig.LengthBase ...
                          && config.LengthUpper == existingConfig.LengthUpper ...
                          && config.LengthLower == existingConfig.LengthLower ...
                          && config.WorkspaceHeight == existingConfig.WorkspaceHeight
                      unique = false;
                      break;
                  end
              catch exception
              end
          end

          % Add child to population if it is unique and can acheive the desired path
          if unique == true && reachable == true
              newPopulation(p) = config;
              newPopPPC(p) = parent1ppc;
              newPopMotorID(p) = parent1motorID;
          end
      end
  end
  population = newPopulation;
  popPPC = newPopPPC;
  popMotorID = newPopMotorID;
end
end
```

Figure 6.27 Matlab® Code of the GA Optimising Method (Part 4/4)

The GA code has a number of parameters. The choosing of these parameters is critical to the performance of the algorithm. Some of the parameters have been tuned through initial empirical methods, whereas some have been more thoroughly evaluated. The parameters *MutationRate* (lines 2 & 181-205) and *MutationAmount* (lines 2 & 183-211) were chosen through trial and error until suitable values were found. The *MutationRate* determines the chance of each dimension of a newly generated configuration being altered. The amount that the dimension is altered is set by the *MutationAmount*. The tuned values for these parameters are 25 % for the *MutationRate* and 10 % alteration by the *MutiationAmount*.

There are three other parameters that affect the performance of the GA. Firstly, the *Population* (lines 1, 23 and throughout the algorithm) states the number of individual configurations maintained throughout the algorithm. Secondly, the *Selection Rate* (lines 2 & 121) establishes the percentage of the *Population* that will be selected to be in the population for the next generation and to reproduce. The third parameter is the number of *Iterations* (line 37) that the GA will perform until it determines it has found an 'optimum' configuration.

In order to find good values for these three parameters, a number of optimisation runs were performed using three different *Population* values (30, 50, 100), with three different *Selection Rates* (30 %, 60 %, 80 %). Each of the nine parameter combinations was run for 300 iterations to compare the performance over time. To statistically validate the performance of each permutation, 75 runs were done with every parameter combination.

The distributions of the minimum cycle-time achieved with each parameter combination are shown by the histograms in Figure 6.28. The means, standard deviations and medians of this same data are shown in Table 6.14. The best performing combination, with a median minimum cycle-time of 1.59 seconds, was a *Population* size of 100 with 80 % of the configurations being selected for reproduction as given by the *Selection Rate*.

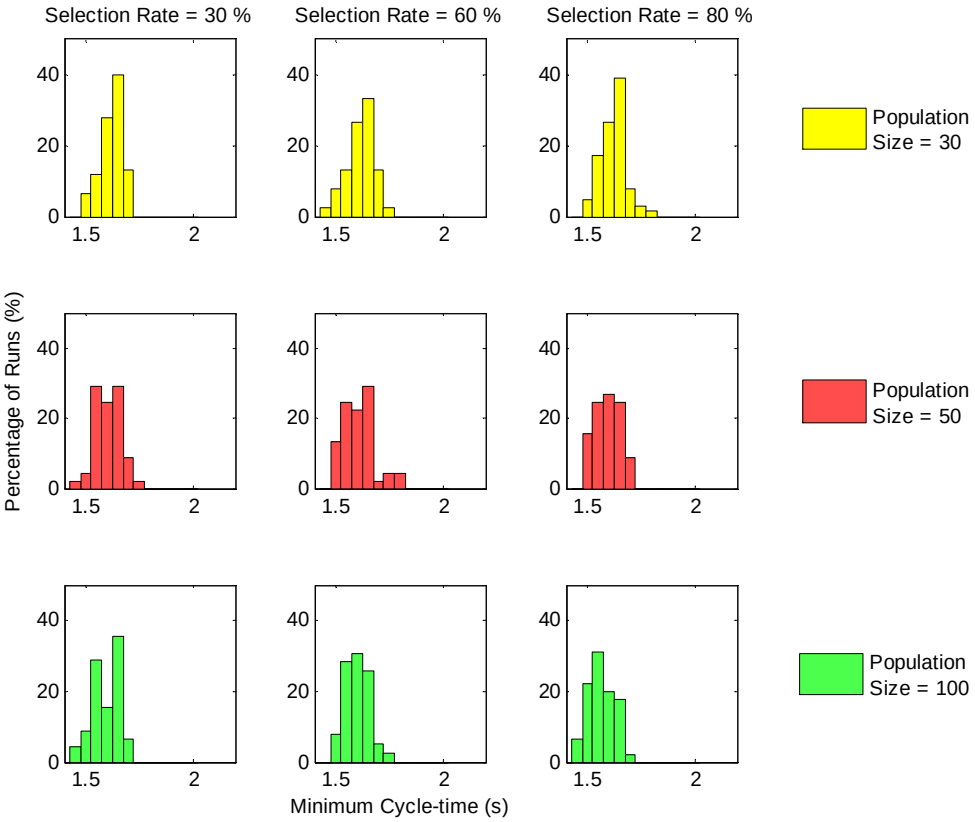

Figure 6.28 Normalised histograms of minimum cycle-time achieved by the GA method with a mutation rate of 25 %, mutation amount of 5 % using population sizes of 30, 50 and 100 with selection rates of 30 %, 60 % and 80 %. Results are based on 75 individual runs.

Table 6.14 Mean (μ), standard deviation (σ) and median (M) minimum cycle-times with varying population and selection size

Population Size	Selection Rate = 30 %			Selection Rate = 60 %			Selection Rate = 80 %		
	μ (s)	σ (s)	M (s)	μ (s)	σ (s)	M (s)	μ (s)	σ (s)	M (s)
30	1.65	0.05	1.66	1.64	0.06	1.65	1.65	0.06	1.65
50	1.63	0.06	1.62	1.63	0.08	1.62	1.62	0.06	1.63
100	1.62	0.07	1.64	1.63	0.06	1.62	1.59	0.06	1.59

To validate this result, the Wilcoxon-Mann-Whitney test was performed, comparing the combination of *Population* = 100 and *Selection Rate* = 80 % to the other eight parameter combinations tested. The results, presented in Table 6.15, show that the null hypothesis was rejected in all eight cases. This means that the best parameter combination performs significantly better than the other combinations tested.

Table 6.15 Wilcoxon-Mann-Whitney test results. Comparing a population size of 100 and selection rate of 80 % to the other combinations of values tested

Population Size	Selection Rate = 30 %		Selection Rate = 60 %		Selection Rate = 80 %	
	Rejection of Null-Hypothesis	p-Value	Rejection of Null-Hypothesis	p-Value	Rejection of Null-Hypothesis	p-Value
30	1	7.66×10^{-7}	1	3.53×10^{-5}	1	5.73×10^{-6}
50	1	4.27×10^{-4}	1	0.012	1	0.034
100	1	0.015	1	0.007	N/A	N/A

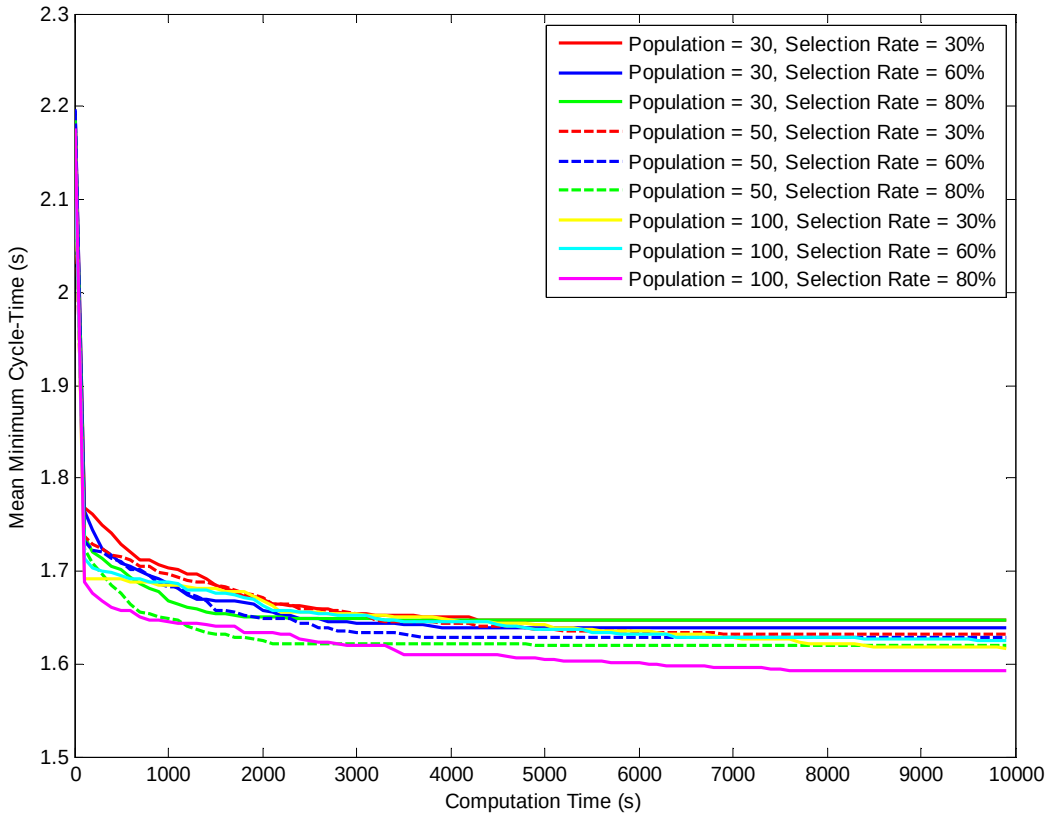

Figure 6.29 Computation time versus mean minimum cycle-time for nine combinations of population size and selection rate

The average computation time to reach 'optimisation' was faster for smaller population sizes as shown in Figure 6.29. This is easily explained by the fact that fewer configurations had to be analysed for the smaller populations. It can be noted however, that the combination of *Population* = 100 and *Selection Rate* = 80 %, almost always found a better solution (that is, a lower cycle-time), for the duration of the process than the other parameter combinations.

The third parameter to be tuned was the number of evolution *Iterations*. Figure 6.30 plots the mean minimum cycle-time over the number of *Iterations*. It can be seen that the greatest improvement in the minimum cycle-time occurs during the early iterations. After 300 iterations, the average performance increase is minimal. However, improvement can still be seen after 300 iterations, therefore indicating that the GA has the potential to continue optimising with a greater number of iterations.

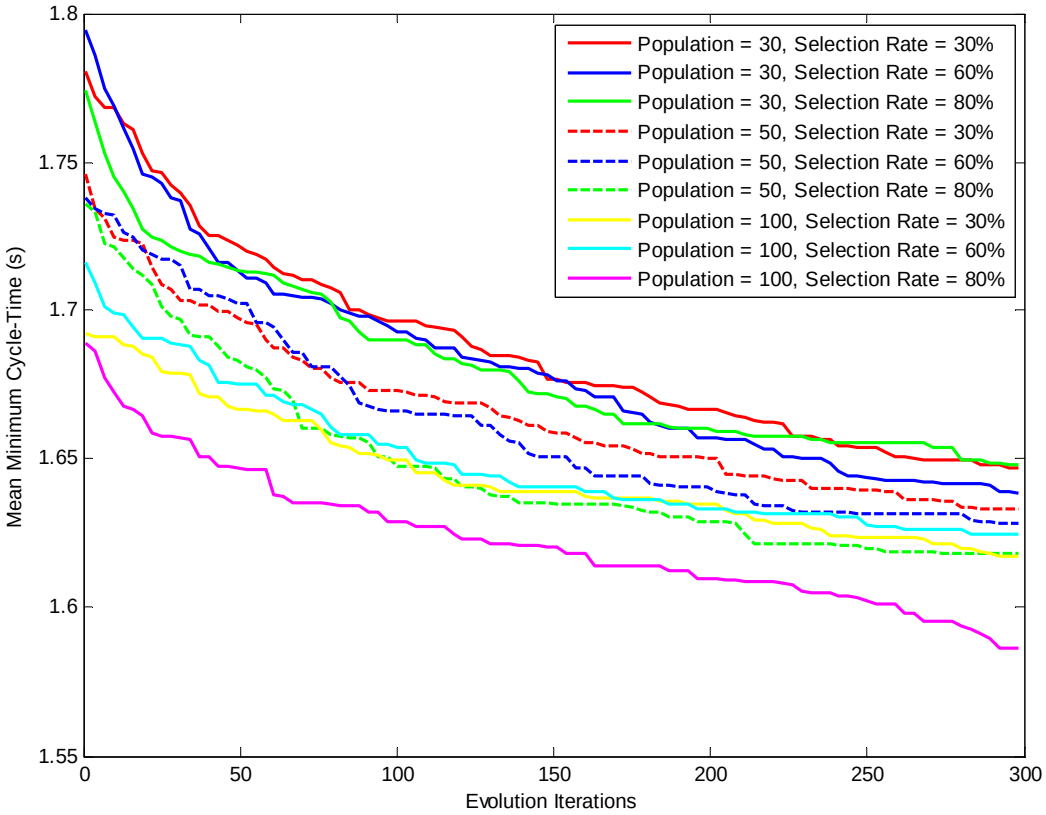

Figure 6.30 Number of evolution iterations/generations versus the mean minimum cycle-time for nine combinations of population size and selection rate

The GA method provides an evolutionary approach to optimising the 2DOFPPM's dimensions to achieve the fastest cycle-time for a given path. This section has presented the code used to implement the algorithm in Matlab®, as well as discussed and tuned the parameters required to maximise the GA's capability. Initial empirical testing led to values of 25 % and 10 % for the *MutationRate* and *MutationAmount*, respectively. The lowest minimum cycle-time was shown to be achieved using a *Population* size of 100 and a *Selection Rate* of 80 %. 300 *Iterations* has been chosen as the number of evolutionary generations to be used by the GA. While the GA still shows signs of improvements past 300 iterations, the improvement is minimal and the time taken to achieve those iterations is far greater than the time taken by the other algorithms being considered.

6.1.5 Comparison

Four optimisation techniques have been implemented to find the best dimensional configuration for the 2DOFPPM. Each technique has several parameters that had to be tuned to achieve the best performance from the algorithm. Table 6.16 presents the four optimising techniques along with a summary of the tuned parameter values associated with each method.

Table 6.16 Summary of parameter values for the optimising algorithms

Algorithm	**Parameter**	**Value**
RRHC	*StepSize*	0.02 m
	Iterations	100
SHC	*StepSize*	0.02 m
	T	0.05
	MaxAttempts	2500
SA	*StepSize*	0.02 m
	T	0.5
	$T_{attenuation}$	0.9
	MaxAttempts1	2000
	MaxAttempts2	5
GA	*Population* Size	100
	Selection Rate	80 %
	MutationRate	25 %
	MutationAmount	10 %

In order to compare the relative performance of the four optimisation methods several graphs have been plotted and a statistical evaluation undertaken. The first of these graphs is found in Figure 6.31, which plots a histogram of the minimum cycle-time achieved by each of the methods. These histograms represent the results of each technique using the best tuned parameters from the previous sections. Along with the histogram distributions, Table 6.17 summarises the mean and median minimum cycle-times of each method. It is noted that the RRHC and SHC perform relatively well with mean cycle-times

of 1.65 and 1.64 seconds respectively. The distribution of the RRHC is near normal, while the SHC is more heavily weighted towards finding lower cycle-times. The SA process resulted in a wide spread of minimum cycle-times and a mean minimum cycle-time of 1.89 seconds. The large variation in results as well as a higher mean minimum cycle-time shows that the SA failed to optimise as well as the other techniques. The GA achieved the lowest mean minimum cycle-time of the four techniques with a time of 1.59 seconds. The distribution is near normal and does not suffer from any outliers.

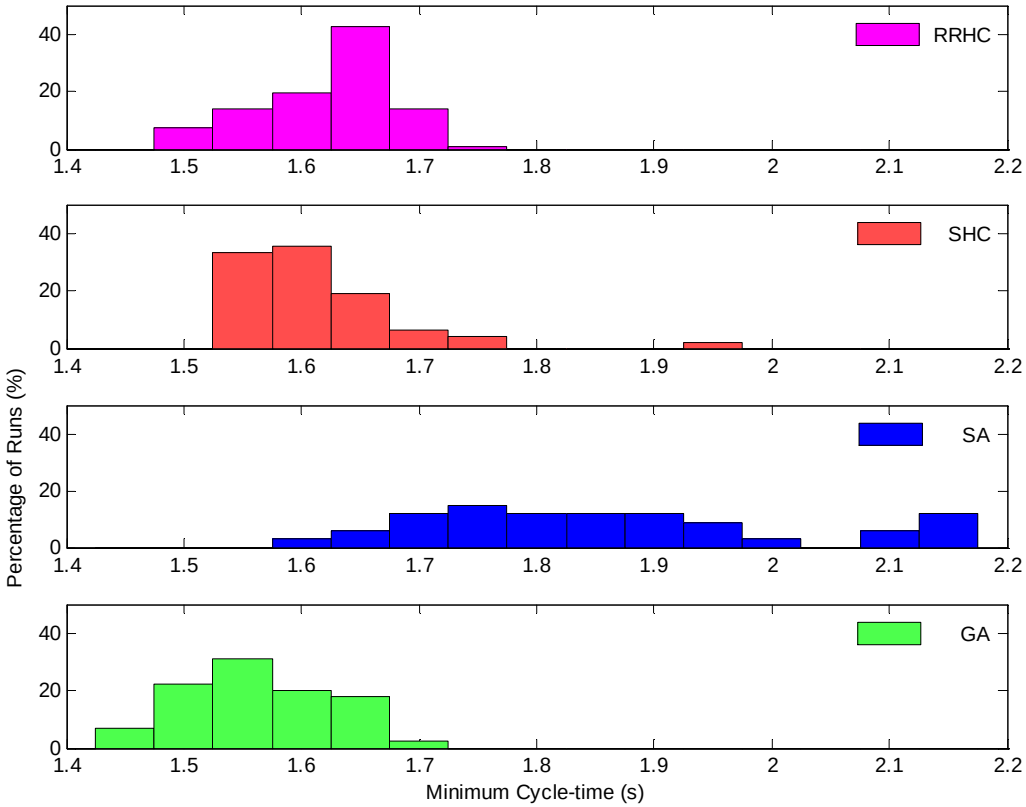

Figure 6.31 Normalised histograms of minimum cycle-time achieved by the RRHC, SHC, SA and GA optimisation methods

Table 6.17 Mean and median minimum cycle-times achieved by the RRHC, SHC, SA and GA optimisation methods

Algorithm	Mean Minimum Cycle-time (s)	Median Minimum Cycle-time (s)
RRHC	1.65	1.66
SHC	1.64	1.62
SA	1.89	1.86
GA	1.59	1.59

While the mean and median give an indication into which optimising algorithm is best suited for this project, a statistical analysis of the minimum cycle-times is needed. The Wilcoxon-Mann-Whitney test is used to compare the algorithm with the lowest median (that is, the GA) with the other three algorithms. The results of this test are shown in Table 6.18 where the null-hypothesis is rejected for all of the three distributions with a 95 % confidence level. This proves that, based on the sample of results collected, the GA is the best of the four algorithms at finding the 2DOFPPM configuration that can achieve the lowest minimum cycle-time for a given path.

Table 6.18 Wilcoxon-Mann-Whitney test results. Comparing the GA to the RRHC, SHC and SA optimisation methods

Algorithm	Rejection of Null-Hypothesis	p-Value
RRHC	1	5.82×10^{-7}
SHC	1	0.002
SA	1	9.99×10^{-14}

Up until this point in the comparison of optimisation methods, only the end result of the algorithm has been considered. Figure 6.32 shows a comparison of the computation time taken by the four techniques. The mean minimum cycle-time is plotted as a function of the computation time. It can be seen that the SA technique terminates first in less than 1500 seconds, but even during that time it never outperformed the other techniques. The SHC finished in just under 2000 seconds, and found a near optimal solution in a third less time the RRHC. The RRHC took 3000 seconds to terminate, and the average minimum cycle-time had plateaued near this time. The GA required almost 8000 seconds to finish, but on average had outperformed the other techniques in the first 1000 seconds.

Four optimising techniques have been implemented to find the best dimensional configuration of the 2DOFPPM for achieving the fastest cycle-time over a given path. Of these four techniques, the GA finds the configuration giving the lowest mean cycle-time. While the GA takes the longest to complete its optimisation process, on average it never performs worse than any of the other algorithms over any given time frame. It is therefore, that the GA is the recommended choice in algorithms when optimising the dimensions of the 2DOFPPM.

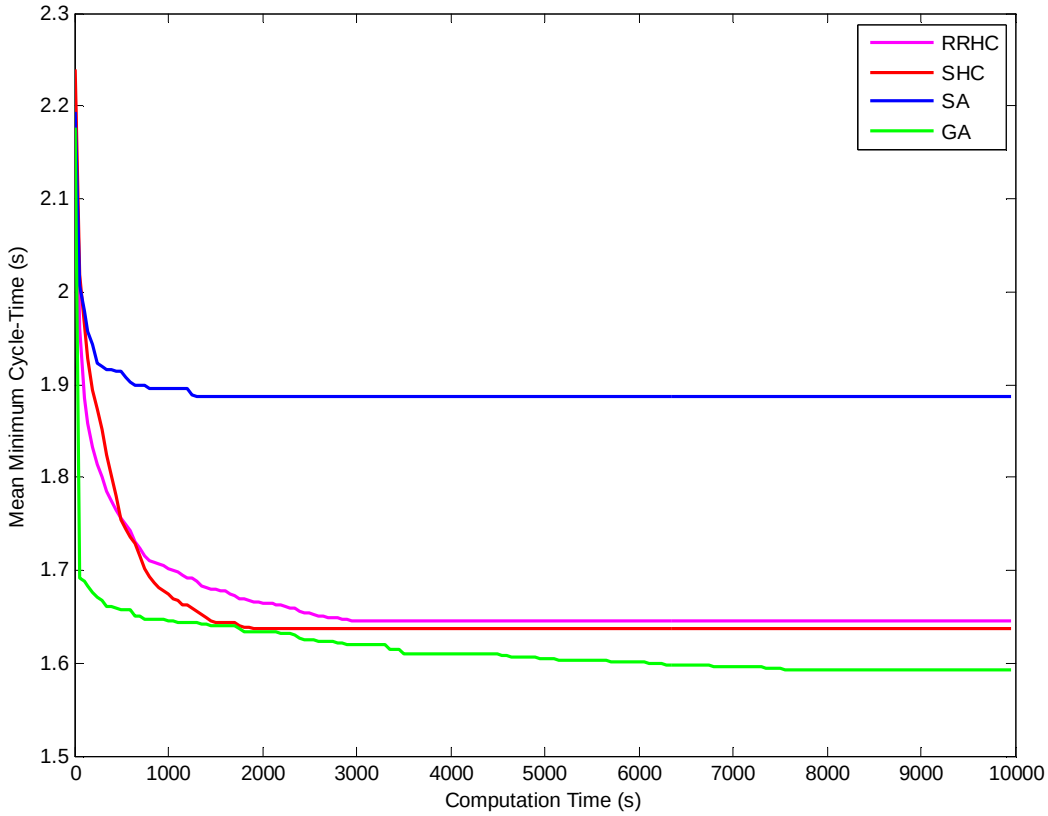

Figure 6.32 Computation time versus mean minimum cycle-time for the RRHC, SHC, SA and GA optimisation methods

6.2 Selecting a Configuration

The optimising algorithms presented in this chapter seek to find the configuration giving the fastest cycle-time. This provides the designers of the 2DOFPPM with a useful tool to find near optimal dimensions for maximising the productivity of the manipulator. It is important to realise that this tool, while useful, should not be used in isolation. If the dimensions of the optimised configuration are used without regard for other design considerations, the customised 2DOFPPM may fail to perform its task. An example of this is highlighted by taking the optimal dimensions and running a SimMechanics™ simulation. Figure 6.33 shows the trajectory and reachable workspace of a near optimal 2DOFPPM configuration as found by the GA in Section 6.1.4. It can be seen that the reachable workspace of this 2DOFPPM only just encompasses the trajectory followed by the end-effector. If there was a slight change in the design constraints (for example, a widening of the pick and place positions) the 'optimised' configuration would no longer be able to reach all the targets, thus rendering it unsuitable for the task.

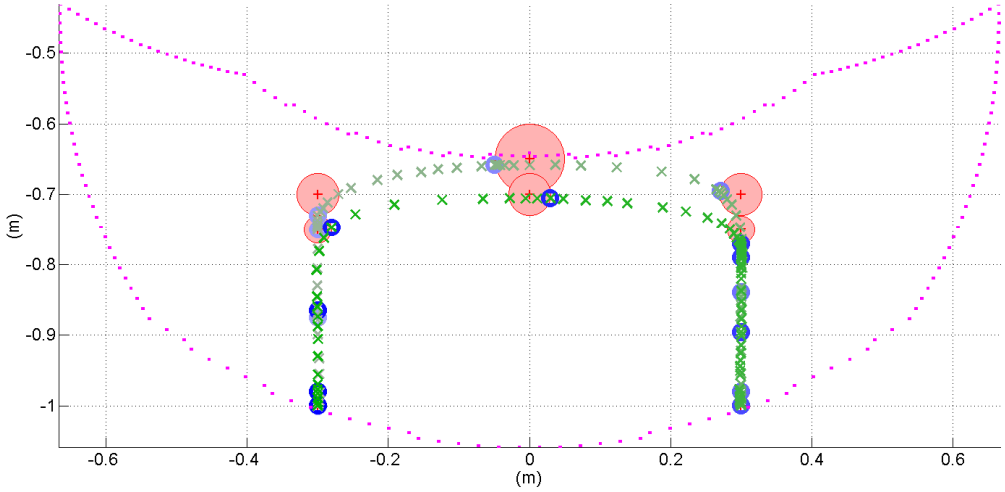

Figure 6.33 Trajectory and workspace of the optimised 2DOFPPM configuration resulting from the GA

To avoid this situation, two options are available for the designer. The first is to test the optimised 2DOFPPM in SimMechanics™ on a variety of slightly modified paths. The second option is to select a less optimal configuration that has a larger workspace with greater room for changes in the path.

7 Conclusion and Recommendations

7.1 Conclusion

This thesis has studied the simulation and optimisation of the 2DOFPPM on behalf of RML Engineering Ltd. With the continuing pressure to increase throughput in factories, development in product handling robotic manipulators is of great importance. Many industrial pick-and-place manipulators perform the same cyclic movement for the lifetime of the robot. RML Engineering Ltd. wanted to investigate developing 2DOFPPM pick-and-place manipulators that are customised for individual applications. While a standard manipulator design would be made, the dimensions of the mechanism would be changeable to provide a configuration that is optimised for a specific task. This thesis develops a simulation model of the 2DOFPPM and produces software systems to allow the optimisation of the manipulator's dimensions.

The 2DOFPPM configuration is a simple parallel manipulator design that is capable of performing high speed translational movements in a single plane of motion. The parallel architecture grants it a highly stiff structure with great positional accuracy characteristics. The leverages obtained by the manipulator's construction provide high velocities and accelerations of the end-effector, thus leading to fast product handling cycles.

Initially, the kinematics of the 2DOFPPM were presented and used to analyse the reachable workspace of the manipulator. The effects that joint limits and relative manipulator dimensions have on the shape and size of the workspace were examined. This provides a method of tuning the manipulator dimensions to achieve a workspace with a robust and useful shape.

A model of the system was developed using SimMechanics™. This model takes motor input commands and simulates the movement of the mechanism's bodies under the actuation of the motors. The forces and torques acting on the joints were measured along with the velocity and acceleration components experienced by the end-effector. The SimMechanics™ simulation model allows detailed analysis of the dynamic performance of a 2DOFPPM design prior to physical fabrication of the device.

To provide meaningful input into the simulation model a trajectory planner was developed. The trajectory planner was required to minimise the time taken for the manipulator to traverse the path as it would later be used in part of the manipulator configuration optimisation.

The trajectory planner takes a set of movement commands that describes the path the end-effector must traverse through. The movement commands not only state the positions in Cartesian space that the end-effector must move to and from, but also enable additional parameters to be specified that alter the shape and/or the speed of the movements. The trajectory planner then converts the set of Cartesian movement instructions into joint space commands for the motor actuators to follow. Piecewise cubic polynomial splines are fitted between knots in joint space. This provides continuous velocity and acceleration profiles for the motors to follow. The travel time allocated between knots is iteratively altered as the algorithm seeks to maximise the kinematic and dynamic capabilities of the manipulator at all stages in the trajectory. The algorithm developed generates a near time minimum trajectory.

The trajectory planner was then used in the process of optimising the manipulator for a given path. By varying the dimensions of the manipulator, the minimum path cycle-time achievable also changes. Four key dimensions were used as parameters for optimisation, the proximal and distal arm lengths, the separation distance of the motors and the height of the manipulator above the workspace. The effect each dimension has on the path cycle-time was examined by plotting the solution space, that is, a coarse view of all the possible combinations of the four dimensions.

While the solution space gave an indication into the optimal manipulator configuration, it was proposed that an optimising algorithm may be able to find the best configuration faster. Four optimising algorithms were implemented. These were the RRHC, SHC, SA and GA techniques. Given a sample path and a set of manipulator constraints, each algorithm was set to find the best dimensional configuration for the manipulator. Every technique had parameters that required tuning to maximise its performance. The parameters were tuned either through initial empirical testing or via running each algorithm multiple times with different parameter values and performing statistical comparisons. Once all parameters of every algorithm were sufficiently tuned, multiple runs of each algorithm were performed to enable fair statistical evaluation of the methods.

The GA was the best performing algorithm, on average finding a configuration that could achieve a faster cycle-time than the other techniques. The SHC and RRHC had similar performance characteristics to one another, although the SHC converged on a near optimal solution faster. The SA technique failed to converge consistently. A Wilcoxon-Mann-Whitney test was undertaken to statistically compare the significance of the individual optimising technique's results. The outcome of this test showed that the GA was the best performing algorithm with a 95 % confidence interval.

This thesis has presented RML Engineering Ltd. with several tools to assist in the design of customised 2DOFPPMs. By using a GA to optimise the manipulators dimensions, a fast cycle-time can be achieved which in turn leads to increased productivity. While knowing the optimal dimensions is useful, it cannot be used in isolation as the risk of designing a manipulator so specific that any changes to the pick-and-place path render it unusable. Therefore, the SimMechanics™ simulation model is used to test the optimised design under a range of conditions. This validates both the performance and reachable workspace of the manipulator configuration.

7.2 Industry Review

The following is a review from the project's industry supervisor, Daryl Joyce, at RML Engineering Ltd. The full statement is included in Appendix C.

> *"This project has provided RML Engineering with further knowledge and tools to continue our development of the customisable packaging robot placement module. The robotic simulation will assist us in our mechanical design, while at the same time being a useful tool for showing the robot's performance to potential customers. One of the key advantages to this packaging robot will be the capacity to optimise the dimensions to achieve a faster cycle rate than current standardised manipulators. The optimisation methods developed as part of this project will allow us to achieve this. Overall the project has been fruitful in providing us with software tools and a greater knowledge of robotic manipulators."*

7.3 Future Work

This thesis has provided tools that enable the development, analysis and optimisation of the 2DOFPPM mechanism. While the results of the thesis are of significant use to RML Engineering Ltd., there remain a number of improvements to be explored. These, along with a number of research directions, are discussed below.

Simulation Model:

(1) The SimMechanics™ simulation could be extended to include a model of the motors. This could be achieved using another Matlab® toolbox called SimElectronics™. By modelling the motors a more detailed view of the system could be obtained.

(2) Developing an on-line trajectory planner within the simulation model with a feedback loop to control the motors in 'real-time'. By integrating the trajectory planner with the low level control system a wider view of the system could be simulated.

(3) With the introduction of recommendations (1) and (2), the impact any controller or motor inaccuracies had on the positional error of the end-effector could be evaluated.

(4) Analysing the effects vibration has on the end-effector's positional accuracy by introducing non-rigid bodies into the model. This would provide the model with a high degree of fidelity which is useful in the final stages of the design process.

Trajectory Planning:

(5) An improved trajectory planning algorithm that calculates the path faster while achieving truly time minimum trajectories would assist in the final stages of optimising the 2DOFPPM configuration.

Optimisation Methods:

(6) While several optimising techniques have been applied to finding the optimal dimensions of the 2DOFPPM, the ever growing field of optimisation means that other methods may exist that perform better than those tested within this thesis. One of these approaches is to use a hybrid algorithm. For example, combining a genetic algorithm with a hill climber.

(7) The optimisation in this thesis has focused on minimising the path's cycle-time. While this is commonly the most important performance measure for pick-and-place manipulators, other details are also of some value. Multi-objective optimisation techniques could be employed to maximise end-effector positional accuracy while at the same time minimising the cycle-time.

(8) The four major dimensions of the 2DOFPPM, along with the selection of the motors, have been considered as variables to be optimised. These contribute to being the major factors determining the minimum cycle-time achieved by the manipulator. However, there are other aspects that also affect the result, including the less significant dimensions and the density and volume of the materials used to fabricate the arms. A wider optimisation could be performed that takes into account these other factors.

The investigation of these topics would further add to the study of the 2DOFPPM and optimisation of its parameters.

7.4 Summary

The 2DOFPPM mechanism has been studied in this thesis. A simulation model has been developed using SimMechanics™. This model provides insight into the dynamic performance of the manipulator under the actuation of motor torques and external forces. To assist in this, a trajectory planner has been developed that provides a near time-minimum trajectory.

The ability to customise the manipulator for a specific task has been identified as valuable to increasing productivity. Several optimising algorithms have been implemented to tune the dimensions until the best configuration is found. The most successful of these techniques is the Genetic Algorithm.

This work was undertaken for RML Engineering Ltd. The company is now using the software tools developed within this thesis to optimise and analyse the 2DOFPPM for specific industrial applications.

References

References used in this thesis are listed in order of citation with IEEE formatting.

[1] Pester Pac Automation. [Online]. www.pester.com

[2] Jean-Pierre Merlet, *Parallel Robots*. Dordrecht, The Netherlands: Kluwer Academic Publishers, 2000.

[3] DFA Media. Drives & Controls. [Online]. http://www.drives.co.uk/features.asp?id=6

[4] T.S. Mruthyunjaya Bhaskar Dasguptaa, "The Stewart platform manipulator: a review," 1998.

[5] S. Lemieux, J.P. Merlet C.M. Gosselin, "A new architecture of planar three-degree-of-freedom parallel," , Minneapolis, 1996.

[6] Simon Henein, Ivo Magnani, Reymond Clavel Eric Pernette, "Design of parallel robots in microrobotics," vol. 15, 1997.

[7] Reymond Clavel Peter Vischer, "A Novel 3-DoF Parallel Wrist Mechanism," vol. 19, no. 1, 2000.

[8] Vincent Nabat, Olivier Company, Sebastian Krut, Francois Pierrot Cedric Baradat, "Par2: a Spatial Mechanism for Fast Planar, 2-dof, Pick-and-Place Applications," *Proceedings of the Second International Workshop on Fundamental Issues and Future Directions for Parallel Mechanisms and Manipulators*, 2008.

[9] V. E. Gough, "Contribution to discussion of papers on research in Automotive stability, control and tyre performance," *Proc. Auto. Div.*, 1956.

[10] D. Stewart, "A platform with 6 degrees of freedom," *Aircraft Engineering and Aerospace Technology*, vol. 38, no. 4, 1966.

[11] R. Clavel, "DELTA, a fast robot with parallel geometry," *International Symposium on Industrial Robots*, 1988.

[12] Jiangping Mei, Xueman Zhao, Derek G. Chetwynd Tian Huang, "A Method for Estimating Servomotor Parameters of a Parallel Robot for Rapid Pick-and-Place Operations," *Journal of Mechanical Design*, vol. 127, 2005.

[13] Zhanxian Li, Meng Li, Derek G. Chetwynd, Clement M. Gosselin Tian Huang, "Conceptual Design and Dimensional Synthesis of a Novel 2-DOF Translational Parallel Robot for Pick-and-Place Operations," vol. 126, May 2004.

[14] Meng Li, Zhanxian Li, Derek G. Chetwynd, David J. Whitehouse Tian Huang, "Optimal Kinematic Design of 2-DOF Parallel Manipulators With Well-Shaped Workspace Bounded by a Specified Conditioning Index," *IEEE Transactions on Robotics and Automation*, vol. 20, no. 3, 2004.

[15] D. G. Chetwynd, J. P. Mei, X. M. Zhao T. Huang, "Tolerance Design of a 2-DOF Overconstrained Translational Parallel Robot," *IEEE Transactions on Robotics*, vol. 22, no. 1, 2006.

[16] Yong-Sheng Zhao, Hong-Rui Wang Feng Gao Xiao-qiu Zhang, "A Physical Model of the Solution Space and the Altas of the Reachable Workspace for 2-DOF Parallel Planar Manipulators," *Mechanical Machine Theory*, vol. 31, no. 2, 1996.

[17] XinJun Liu, William A. Gruver Feng Gao, "Performance Evaluation of Two-Degree-Of-Freedom Planar Parallel Robots," *Mechanical Machine Theory*, vol. 33, no. 6, 1998.

[18] Zhiyong Yang, Jiangping Mei, Tian Huang Haihong Li, "A New Dynamic Index of Parallel Robots with Flexible Links," *IEEE Industrial Conference on Industrial Technology*, 2008.

[19] Zhiyong Yang, Tian Huang, Jiangping Mei Haihong Li, "Dynamics and Optimization of a 2-Dof Parallel Robot with Flexible Links," *7th World Congress on Intelligent Control and Automation*, 2008.

[20] J. G. Rendon-Sanchez J. J. Cervantes-Sanchez, "A simplified approach for obtaining the workspace of a class of 2-dof planar parallel manipulators," *Mechanism and Machine Theory*, vol. 34, 1999.

[21] Vistrian Maties, Radu Balan Sergiu-Dan Stan, "Optimisation of a 2 DOF Micro Parallel Robot Using Genetic Algorithms," in *Frontiers in Evolutionary Robotics*. Cluj-Napoca, Romania: I-Tech Education and Publishing, Vienna, Austria, 2008.

[22] W. L. Cleghorn, J. K. Mills G. Piras, "Dynamic finite-element analysis of a planar high-speed, high-precision parallel manipulator with flexible links," *Mechanism and Machine Theory*, vol. 40, 2005.

[23] Xianmin Zhang, Jinqing Zhan Junfeng Hu, "Trajectory Planning of a Novel 2-DoF High-Speed Planar

Parallel Manipulator," 2008.

[24] C. Gosselin and J. Angeles, "A Global Performance Index for the Kinematic Optimization of Robotic Manipulators," *Journal of Mechanical Design*, vol. 113, no. 3, 1991.

[25] Xin-Jun Liu, Jinsong Wang, Kun-Ku Oh, and Jongwon Kim, "A New Approach to the Design of a DELTA Robot with a Desired Workspace," *Journal of Intelligent and Robotic Systems*, vol. 39, 2004.

[26] S. Macfarlane, "On-Line Smooth Trajectory Planning for Manipulators," 1999.

[27] V. Zanotto A. Gasparetto, "A new method for smooth trajectory planning," *Mechanism and Machine Theory*, vol. 42, 2007.

[28] C. S. Lin J. Y. S. Luh, "Optimum Path Planning for Mechanical Manipulators," *Journal of Dynamic Systems, Measurement, and Control*, vol. 103, no. 2, 1981.

[29] Man Zhihong, *Robotics - Second Edition*. Jurong, Singapore: Prentice Hall, 2005.

[30] Seth Hutchinson, M. Vidyasagar Mark W. Spong, *Robot Modeling and Control*. Hoboken, NJ, USA: John Wiley & Sons, 2006.

[31] R. P. C. Paul, *Modelling, trajectory calculation and servoing of a computer controlled arm*., 1972.

[32] R. A. Finkel, "Constructing and Debugging Manipulator Programs," Stanford, California, 1976.

[33] Po-rong Chang, J. Y. S. Luh Chun-shin Lin, "Formulation and Optimisation of Cubic Polynomial Joint Trajectories for Industrial Robots," *IEEE Transactions on Automatic Control*, vol. AC-28, 1983.

[34] R. Mead J. A. Nelder, "A Simplex Method for Function Minimisation," *The Computer Journal*, 1965.

[35] Keith L. Doty Sujeet Chand, "On-Line Polynomial Trajectories for Robot Manipulators," *The International Journal of Robotics Research*, vol. 4, no. 2, 1985.

[36] A. Grabos M. Boryga, "Planning of manipulator motion trajectory with higher-degree polynomial use," vol. 44, 2009.

[37] Rajnikant V. Patel Stuart E. Thompson, "Formulation of Joint Trajectories for Industrial Robots Using B-Splines," *IEEE Transactions on Industrial Electronics*, vol. 34, 1987.

[38] Jong-gin Horng Chi-hsu Wang, "Constrained Minimum-Time Path Planning for Robot Manipulators Via Virtual Knots of the Cubic B-Spline Functions," *IEEE Transactions on Automatic Control*, vol. 35, 1990.

[39] B. Roth M. E. Kahn, "The Near-Minimum-Time Control Of Open-Loop Articulated Kinematic Chains," *Journal of Dynamic Systems, Measurement, and Control* , vol. 93, no. 3, 1971.

[40] L. S. Pontryagin and R. V. Gamkrelidze, *The Mathematical Theory of Optimal Processes*.: Gordon and Breach Science Publishers, 1987.

[41] V. Rajan, "Minimum Time Trajectory Planning," *IEEE International Conference on Robotics and Automation*, 1985.

[42] Lino Guzella, Stephan A. R. Hepner, Christopher H. Onder Hans P. Geering, "Time-Optimal Motions of Robots in Assembly Tasks," *Proceedings of 24th Conference on Decision and Control*, 1985.

[43] Alan A. Desrochers Yaobin Chen, "Time-optimal control of two-degree of freedom robot arms," *IEEE Transactions on Robotics and Automation*, vol. 6, no. 3, 1990.

[44] C.S., Chang, P.R., Luh, J.Y.S. Lin, "Formulation and Optimization of Cubic Polynomial Joint Trajectories for Industrial Robots," *IEEE Transactions on Automatic Control*, vol. AC-28, 1983.

[45] E. Red, "A dynamic Optimal Trajectory Generator for Cartesian Path Following," *Robotica*, vol. 18, 2000.

[46] H.H., Potts, R.B. Tan, "Minimum Time Trajectory Planner for the Discrete Dynamic Robot Model with Dynamic Constraints," *IEEE Journal of Robotics and Automation*, vol. 4, 1988.

[47] K.G. Shin B.K. Kim, "Minimum-Time Path Planning for Robot Arms and Their Dynamics," *IEEE Transactions on Systems, Man, and Cybernetics*, vol. SMC-15, 1985.

[48] S. Dubowsky, J.S. Gibson J.E. Bobrow, "Time-Optimal Control of Robotic Manipulators Along

Specified Paths," *The International Journal of Robotics Research*, vol. 3, 1985.

[49] Neil D. McKay Kang. G. Shin, "Minimum-Time Control of Robotic Manipualtors with Geometric Path Constraints," *IEEE Transactions of Automatic Control*, vol. AC-30, 1985.

[50] L. Zlajpah, "On Time Optimal Path Control of Manipulators with Bounded Joint Velocities and Torques," , Ljubljana, Slovenia, 1996.

[51] E.A. Croft D. Constantinescu, "Smooth and Time-Optimal Trajectory Planning for Industrial Manipulators along Specified Paths," 2000.

[52] Gordon I. Dodds Bailin Cao, "Time-Optimal and Smooth Joint Path Generation for Robot Manipulators," , Coventry, UK, 1994.

[53] A.S. White, R. Gill J.V. Miro, "On-line Time-optimal Algorithm for Manipulator Trajectory Planning," *Proceedings of the European Control Conference*, 1997.

[54] P.F. Wang, J.P. Mei, X.M. Zhao, D.G. Chetwynd T. Huang, "Time Minimum Trajectory Planning of a 2-DOF Translational Parallel Robot for Pick-and-place Operations," *Annals of the CIRP*, vol. 56, 2007.

[55] Jean-Pierre Merlet, "Determination of the Orientation Workspace of Parallel Manipulators," *Journal of Intelligent and Robotic Systems*, vol. 13, no. 2, 1995.

[56] Kuanchih Wang, Zvi S. Roth Hanqi Zhuang, "Optimal selection of measurement configurations for robot calibration using simulated annealing," *IEEE International Conference on Robotics and Automation*, 1994.

[57] Jan Korst Emile Aarts, *Simulated Annealing and Boltzmann Machines*.: John Wiley & Sons, 1989.

[58] Vistrian Maties, Radu Balan Sergiu-Dan Stan, "Optimization of workspace of a 2 DOF parallel minirobot using Genetic Algorithms and Simulated Annealing optimization methods," *IEEE Workshop on Advanced Robotics and Its Social Impacts*, 2007.

[59] Radu Balan, Vistrian Maties Sergiu Stan, "Optimization of the workspace of PKM with 2 DOF," *IEEE International Conference on Automation Science and Engineering*, 2006.

[60] P. Bidaud O. Chocron, "Genetic Design of 3D Modular Manipulators," *IEEE International Conference on Robotics and Automation*, 1997.

[61] John T. Feddema, "Kinematically Optimal Robot Placement for Minimum Time Coordinate Motion," *IEEE International Conference on Robotics and Automation*, vol. 4, 1996.

[62] M. A. Pashkevich A. P. Pashkevich, "Multiobjective optimisation of robot location in a workcell using genetic algorithms," *UKACC International Conference on Control*, vol. 1, 1998.

[63] K.-D. Bouzakis, D. Sagris, G. Mansour S. Mitsi, "Determination of optimum robot base location considering discrete end-effector positions by means of hybrid genetic algorithm," *Robotics and Computer-Integrated Manufacturing*, vol. 24, 2008.

[64] Edmund Taylor Whittaker and William McCrea, *A Treatise on the Analytical Dynamics of Particles and Rigid Bodies*.: Cambridge University Press, 1988.

[65] ABB Robotics, IRB360 Flexpicker Product Manual, 2008.

[66] Jae-Won Lee, Hyuk-Jin Lee Wenbin Deng, "Kinematics Simulation and Control of a New 2 DOF Parallel Mechanism Based on Matlab/SimMechanics," 2009.

[67] F.C. Park, A. Sideris J.E. Bobrow, "Recent Advances on the Algorithmic Optimization of Robot Motion," in *Fast Motions in Biomechanics and Robotics*. Heidelberg, Germany: Springer, 2006.

[68] Amir Khajepour Saeed Behzadipour, "Time-Optimal Trajectory Planning in Cable-Based Manipulators," *IEEE Transactions on Robotics*, vol. 22, 2006.

[69] Gordon I. Dodds, George W. Irwin Bailin Cao, "Time-Optimal and Smooth Constrained Path Planning for Robot Manipulators," 1994.

[70] V. Hayward J. Lloyd, "Trajectory Generation for Sensor-Driven and Time-Varying Tasks," *International Journal of Robotics Research*, vol. 12, 1993.

[71] Man Zhihong, *Robotics*. Singapore: Prentice Hall, 2005.

[72] MySQL. [Online]. http://www.mysql.com/about/legal/licensing/oem/

[73] Robert Almgren, MySQL client for Matlab.

[74] Adam Prügel-Bennett, "When a genetic algorithm outperforms hill-climbing," *Theoretical Computer Science*, vol. 320, no. 1, 2004.

[75] John H. Holland, Stephanie Forrest Melanie Mitchell, "When Will a Genetic Algorithm Outperform Hill Climbing?," *Advances in Neural Information Processing Systems*, vol. 6, 1993.

[76] Manish Kumar Devendra P. Garg, "Optimization techniques applied to multiple manipulators for path planning and torque minimization," *Engineering Applications of Artificial Intelligence*, vol. 15, no. 3-4, 2002.

[77] Frank Wilcoxon, "Individual Comparisons by Ranking Methods," *Biometrics Bulletin*, vol. 1, no. 6, 1945.

[78] H B Mann and D R Whitney, "On a Test of Whether one of Two Random Variables is Stochastically Larger than the Other," *The Annals of Mathematical Statistics*, vol. 18, no. 1, 1947.

Appendix A Simulation Parameters

Table A.1 Default parameters used in sample simulation in Chapter 3. Values obtained from RML Engineering.

Parameter	Default Setting
Base length (separation of servo motor actuators)	0.3 m
Proximal (upper) arm length	0.36 m
Distal (lower) arm length	0.88 m
End-effector length	0.01 m
Proximal arm mass	3.5 kg
Distal arm mass	2 kg
End-effector mass	35 kg
Proximal (upper) stabiliser arm mass	0.2 kg
Distal (lower) stabiliser arm mass	0.3 kg
Proximal (upper) crank arm mass	0.2 kg
Distal (lower) crank arm mass	0.2 kg
Arm ID (Internal diameter)	0.01 m
Arm OD (Outer diameter)	0.02 m
Stabiliser Arm ID (Internal diameter)	0.005 m
Stabiliser Arm OD (Outer diameter)	0.01 m
End-effector mount offset (from joining of distal arms) (X)	0 m
End-effector mount offset (from joining of distal arms) (Y)	-0.02 m
Proximal (upper) stabiliser arm offset from motor B (X)	0.05 m
Proximal (upper) stabiliser arm offset from motor B (Y)	0.1 m
Distal (lower) stabiliser arm offset from end-effector (X)	-0.05 m
Distal (lower) stabiliser arm offset from end-effector (Y)	-0.1 m
Minimum angle between proximal arm and +Y-axis	43°
Maximum angle between proximal arm and +Y-axis	164°
Minimum internal angle between proximal arm and distal arm	43°
Maximum internal angle between proximal arm and distal arm	134°
Minimum internal angle between distal arms	48°
Maximum internal angle between distal arms	71°
Pick/Place Dwell Time	0.2 s

Appendix B Computer Specifications

Table B.1 Specifications of the computer used to perform all computations in this thesis.

Operating System	32-bit Microsoft Windows XP Professional Version 2002 Service Pack 3
Processor	Intel(R) Core(TM)2 Cuo CPU E8400 @ 3.00GHz 2.99 GHz
RAM	3.21 GB

Appendix C Industry Review

rml Automation Ltd
P.O Box 20414
66 Norris Ave, Te Rapa
HAMILTON 3241
New Zealand

28 February 2011

Mr Ben Haughey
C/- Victoria University of Wellington
Engineering Computer Science
P.O Box 600
Wellington, 6140

Dear Ben,

Re: Product Placement Module – Ben Haughey thesis review 26/01/2011

This project has provided rml Engineering with further knowledge and tools to continue our development of the customisable packaging robot placement module.

The robotic simulation will assist us in our mechanical design, while at the same time being a useful tool for showing the robotic modules performance to potential customers.

One of the key advantages to this packaging robot will be the capacity to optimise the dimensions to achieve a faster cycle rate than current standardised manipulators. The optimisation methods developed as part of this project will allow us to achieve this.

Overall the project has been fruitful in providing us with software tools and a greater knowledge of robotic manipulators.

Kind regards,

DARYL JOYCE
General Manager | rml Automation Ltd
Part of the rml Engineering group of companies

www.rmlengineering.com

Figure C.1 Industry feedback from RML Engineering Ltd.

Appendix D Simulated Annealing Additional Results

The following figures and tables are the results of using values of 10 and 40 for the parameter *MaxAttempts1* in the SA algorithm (refer Section 6.1.3). All the other parameters remain the same as found in the body of the thesis.

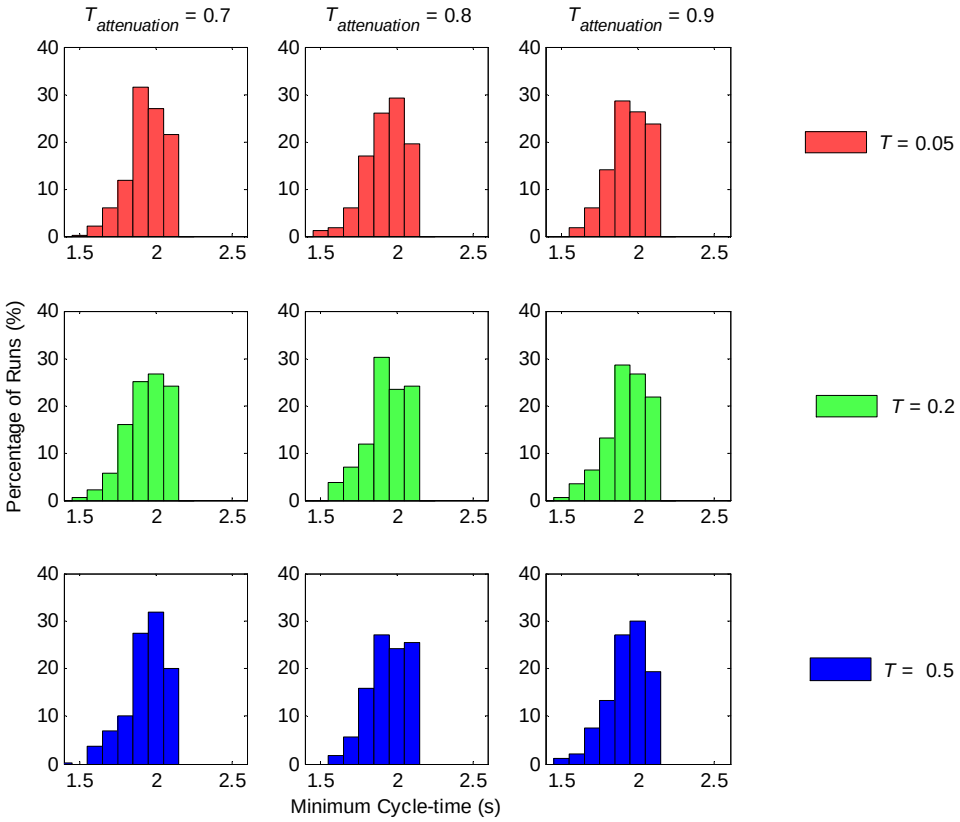

Figure D.1 Normalised histograms of minimum cycle-times for three *T* values with three $T_{attenuation}$ rates. *MaxAttempts1* = 10.

Table D.1 Mean (μ), standard deviation (σ) and median (M) minimum cycle-times for *MaxAttempts1* = 10

T	$T_{attenuation}$ = 0.7			$T_{attenuation}$ = 0.8			$T_{attenuation}$ = 0.9		
	μ (s)	σ (s)	M (s)	μ (s)	σ (s)	M (s)	μ (s)	σ (s)	M (s)
0.05	2.121	0.211	2.108	2.135	0.224	2.130	2.141	0.224	2.145
0.2	2.132	0.216	2.125	2.136	0.226	2.130	2.143	0.234	2.141
0.5	2.130	0.229	2.104	2.146	0.221	2.157	2.137	0.231	2.117

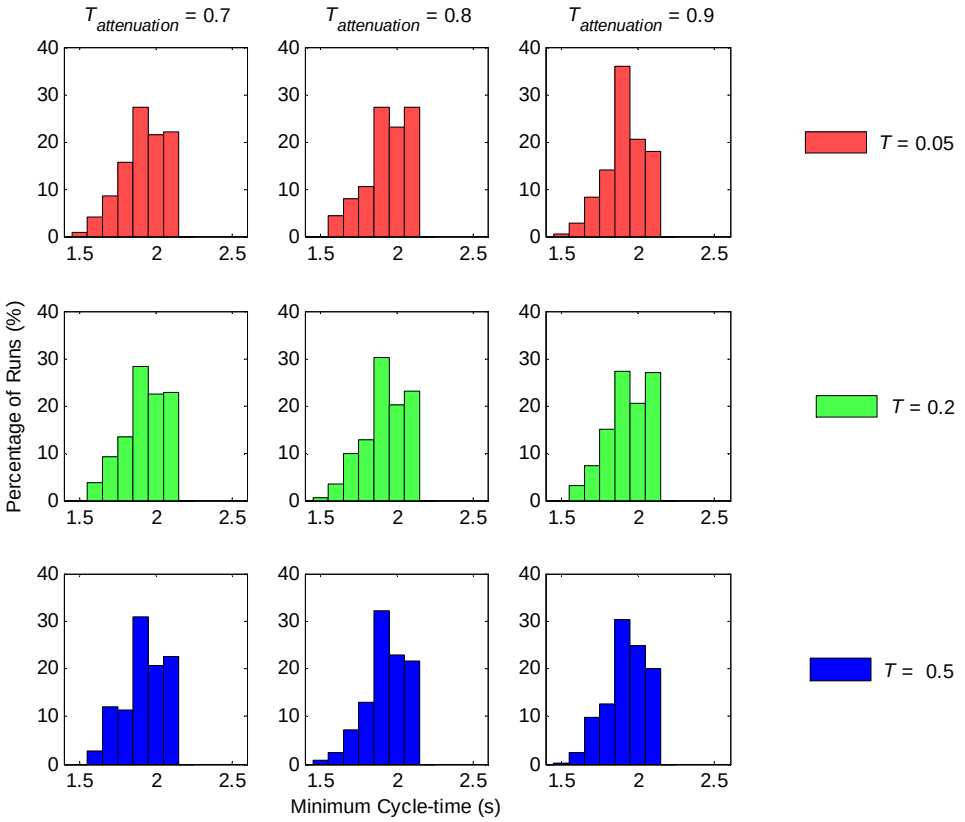

Figure D.2 Normalised histograms of minimum cycle-times for three *T* values with three $T_{attenuation}$ rates. *MaxAttempts1* = 40.

Table D.2 Mean (μ), standard deviation (σ) and median (M) minimum cycle-times for *MaxAttempts1* = 40

T	$T_{attenuation}$ = 0.7			$T_{attenuation}$ = 0.8			$T_{attenuation}$ = 0.9		
	μ (s)	σ (s)	M (s)	μ (s)	σ (s)	M (s)	μ (s)	σ (s)	M (s)
0.05	2.068	0.198	2.046	2.081	0.185	2.086	2.065	0.196	2.023
0.2	2.074	0.191	2.062	2.067	0.196	2.057	2.082	0.190	2.094
0.5	2.069	0.189	2.056	2.073	0.184	2.045	2.066	0.188	2.037

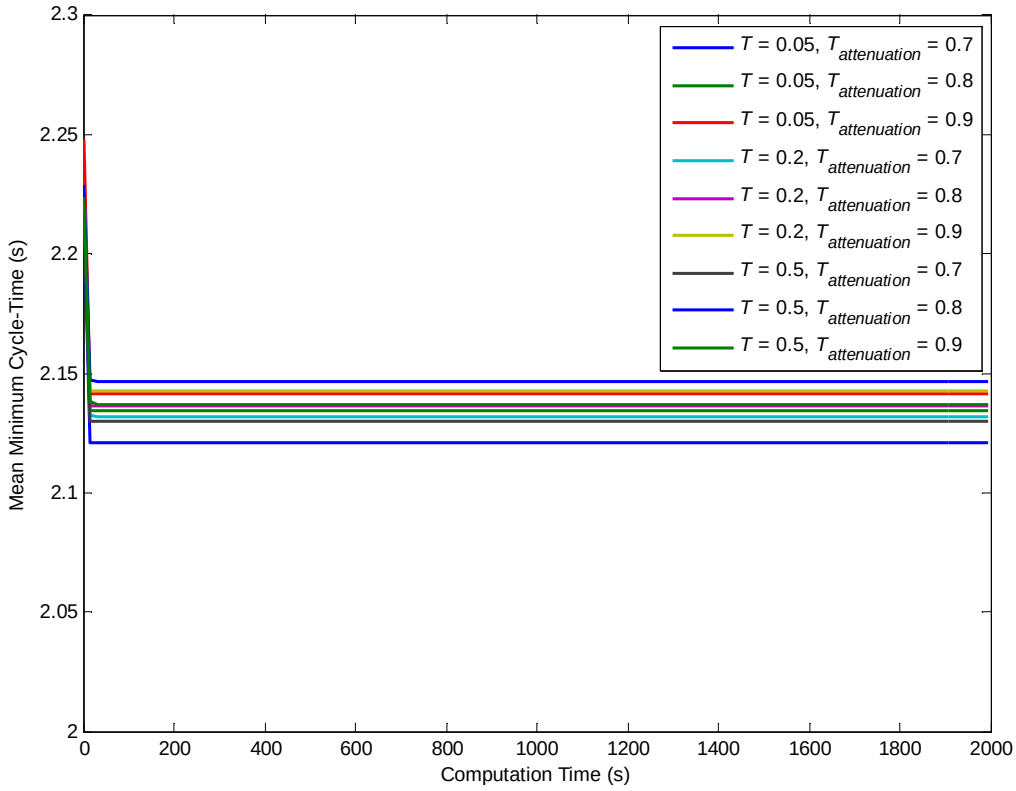

Figure D.3 Mean minimum cycle-time versus computation time with *MaxAttempts1* = 10 for nine combinations of *T* and $T_{attenuation}$.

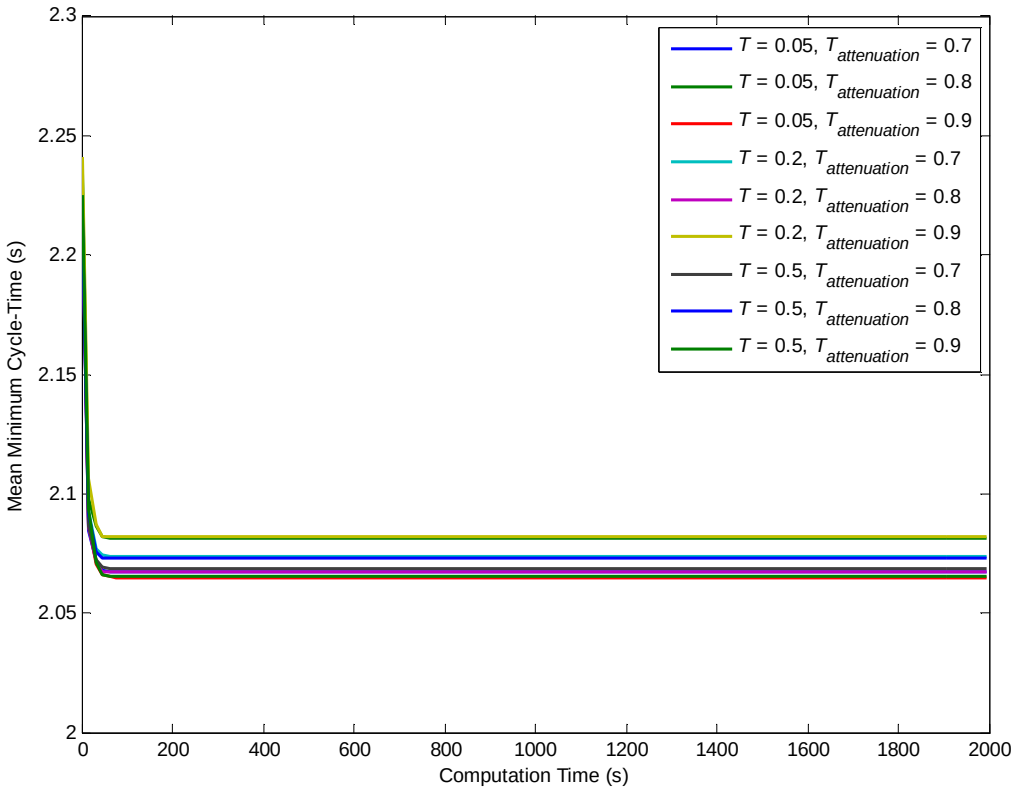

Figure D.4 Mean minimum cycle-time versus computation time with *MaxAttempts1* = 40 for nine combinations of *T* and $T_{attenuation}$.

Appendix E SQL Code

```
CREATE DATABASE IF NOT EXISTS matlab_2dofppm;

DROP TABLE matlab_2dofppm.Simulations IF EXISTS;
DROP TABLE matlab_2dofppm.Motors IF EXISTS;
DROP TABLE matlab_2dofppm.UserConstraints IF EXISTS;
DROP TABLE matlab_2dofppm.Moves IF EXISTS;
DROP TABLE matlab_2dofppm.Paths IF EXISTS;

CREATE TABLE matlab_2dofppm.Paths(
  PathID INT,
  LinearErrorFactor FLOAT,
  LastLinearTargetDistance FLOAT,
  ReactiveFactor FLOAT,
  InitialAcceptanceThreshold FLOAT,
  RelativeAgeingFactor FLOAT,
  AttemptedConfigurations INT,
  PRIMARY KEY (PathID)
);

CREATE TABLE matlab_2dofppm.UserConstraints(
  PathID INT,
  MaxMotorTorque FLOAT,
  MaxMotorVelocity FLOAT,
  MaxMotorAcceleration FLOAT,
  MaxMotorJerk FLOAT,
  MassGripper FLOAT,
  MinArmAng_BU FLOAT,
  MinArmAng_UL FLOAT,
  MinArmAng_LL FLOAT,
  MaxArmAng_BU FLOAT,
  MaxArmAng_UL FLOAT,
  MaxArmAng_LL FLOAT,
  ProxArmDensity FLOAT,
  DistArmDensity FLOAT,
  TorsionArmDensity FLOAT,
  ProxArmIRadius FLOAT,
  DistArmIRadius FLOAT,
  ProxArmORadius FLOAT,
  DistArmORadius FLOAT,
  TorsionIRadius FLOAT,
  TorsionORadius FLOAT,
  MassUpperCrank FLOAT,
  MassLowerCrank FLOAT,
  UpperTorsionOffsetB_X FLOAT,
  UpperTorsionOffsetB_Y FLOAT,
  LowerTorsionOffsetTCP_X FLOAT,
  LowerTorsionOffsetTCP_Y FLOAT,
  GripperMountOffset_X FLOAT,
  GripperMountOffset_Y FLOAT,
  GripperLength FLOAT,
  MinMotorSeparation FLOAT,
  MaxWidth FLOAT,
  MaxDepth FLOAT,
  PRIMARY KEY (PathID),
  FOREIGN KEY (PathID) REFERENCES matlab_2dofppm.Paths(PathID)
);
```

Figure E.1 Create SQL Database and Tables Script (Part 1/2)

```
CREATE TABLE matlab_2dofppm.Moves(
  PathID INT,
  MSequence INT,
  Knot_X FLOAT,
  Knot_Y FLOAT,
  MType VARCHAR(10),
  MZone FLOAT,
  Speed FLOAT,
  Pause FLOAT,
  PRIMARY KEY (PathID,MSequence),
  FOREIGN KEY (PathID) REFERENCES matlab_2dofppm.Paths(PathID)
);

CREATE TABLE matlab_2dofppm.Motors(
  MotorID INT,
  Name VARCHAR(255),
  Description VARCHAR(255),
  SpecsFolder VARCHAR(255),
  MaxTorque FLOAT,
  MaxVelocity FLOAT,
  MaxAcceleration FLOAT,
  MaxJerk FLOAT,
  MomentInertia FLOAT,
  EncoderResolution FLOAT,
  PRIMARY KEY (MotorID)
);

CREATE TABLE matlab_2dofppm.Simulations(
  SimID INT,
  ProxArmLength FLOAT,
  DistArmLength FLOAT,
  MotorSeparation FLOAT,
  WorkspaceHeight FLOAT,
  MotorID INT,
  CycleTime FLOAT,
  ExecutionDT DATETIME,
  PathID INT,
  Comment VARCHAR(255),
  Comment2 VARCHAR (255),
  Iteration INT,
  Attempts1 INT,
  PRIMARY KEY (SimID),
  FOREIGN KEY (PathID) REFERENCES matlab_2dofppm.Paths(PathID),
  FOREIGN KEY (MotorID) REFERENCES matlab_2dofppm.Motors(MotorID)
);
```

Figure E.2 Create SQL Database and Tables Script (Part 2/2)

Appendix F Matlab® Code

The following figures contain all the Matlab® code used in this thesis. Methods, scripts and classes are listed in alphabetical order.

```
function [config, reachable] = CalculateConfig(...
                    LengthBase,LengthUpper,LengthLower,WorkspaceHeight,Moves,MotorID,uc)
% Calculates the configuration's parameters needed for SimMechanics model
% VARIABLES:
% LengthBase - Length of the base/separation of motors
% LengthUpper - Length of upper/proximal arms
% LengthLower - Length of lower/distal arms
% WorkspaceHeight - Height of workspace
% Moves - Class containing path move commands
% MotorID - Integer identifying with motor in database
% uc - UserConstraints class

config = Configuration;     % Create new instance of a Configuration

reachable = false;  % set flag

%% Fixed Parameters

config.LengthBase = LengthBase;
config.LengthUpper = LengthUpper;
config.LengthLower = LengthLower;
config.WorkspaceHeight = WorkspaceHeight;

config.MotorID = MotorID;

config.MassUpper = ThickWalledTubeMass(uc.ProxArmIRadius,uc.ProxArmORadius,...
                                       config.LengthUpper,uc.ProxArmDensity);
config.MassLower = ThickWalledTubeMass(uc.DistArmIRadius,uc.DistArmORadius,...
                                       config.LengthLower,uc.DistArmDensity);
config.MassGripper = uc.MassGripper;
config.MassUpperTorsion = ThickWalledTubeMass(uc.TorsionIRadius,uc.TorsionORadius,...
                                              config.LengthUpper,uc.TorsionArmDensity);
config.MassLowerTorsion = ThickWalledTubeMass(uc.TorsionIRadius,uc.TorsionORadius,...
                                              config.LengthLower,uc.TorsionArmDensity);
config.MassUpperCrank = uc.MassUpperCrank;
config.MassLowerCrank = uc.MassLowerCrank;

config.GripperMountOffset_X = uc.GripperMountOffset_X;
config.GripperMountOffset_Y = uc.GripperMountOffset_Y;
config.GripperLength = uc.GripperLength;

config.UpperTorsionOffsetB_X = uc.UpperTorsionOffsetB_X;
config.UpperTorsionOffsetB_Y = uc.UpperTorsionOffsetB_Y;
config.LowerTorsionOffsetTCP_X = uc.LowerTorsionOffsetTCP_X;
config.LowerTorsionOffsetTCP_Y = uc.LowerTorsionOffsetTCP_Y;

config.InRadiusArms = uc.ProxArmIRadius;
config.OutRadiusArms = uc.DistArmIRadius;
config.InRadiusTorsion = uc.TorsionIRadius;
config.OutRadiusTorsion = uc.TorsionORadius;
…
```

Figure F.1 CalculateConfig Function (Part 1/4)

```
…

config.InertiaUpper =ThickWalledTubeInertia(config.InRadiusArms,config.OutRadiusArms,...
                                            config.LengthUpper,config.MassUpper);
config.InertiaLower =ThickWalledTubeInertia(config.InRadiusArms,config.OutRadiusArms,...
                                            config.LengthLower,config.MassLower);
config.InertiaGripper = ThickWalledTubeInertia(config.InRadiusArms,...
                        config.OutRadiusArms,config.GripperLength,config.MassGripper);
config.InertiaUpperTorsion = ThickWalledTubeInertia(config.InRadiusTorsion,...
                             config.OutRadiusTorsion,config.LengthUpper,...
                             config.MassUpperTorsion);
config.InertiaLowerTorsion = ThickWalledTubeInertia(config.InRadiusTorsion,...
                             config.OutRadiusTorsion,config.LengthLower,...
                             config.MassLowerTorsion);
config.InertiaUpperCrank = ThickWalledTubeInertia(config.InRadiusTorsion,...
                           config.OutRadiusTorsion,...
                           sqrt(config.UpperTorsionOffsetB_X^2 ...
                           +config.UpperTorsionOffsetB_X^2),config.MassUpperCrank);
config.InertiaLowerCrank = ThickWalledTubeInertia(config.InRadiusTorsion,...
                           config.OutRadiusTorsion,...
                           sqrt(config.LowerTorsionOffsetTCP_X^2 ...
                           +config.LowerTorsionOffsetTCP_X^2),config.MassLowerCrank);

config.MinUpperArmAngle = uc.MinArmAng_BU;
config.MaxUpperArmAngle = uc.MaxArmAng_BU;
config.Min1_2ArmAngle = uc.MinArmAng_UL;
config.Max1_2ArmAngle = uc.MaxArmAng_UL;
config.MinLowerArmAngle = uc.MinArmAng_LL;
config.MaxLowerArmAngle = uc.MaxArmAng_LL;
…
```

Figure F.2 CalculateConfig Function (Part 2/4)

```
…

%% Internally Computed Parameters

config.ThetaAstart = d2r(180);
config.ThetaBstart = d2r(180);

[ aBaseX,aBaseY,bBaseX,bBaseY,ajX,ajY,bjX,bjY,tcpX,tcpY,error,errorMsg ] = ...
    Direct_2DOF_PPM(config.ThetaAstart,config.ThetaBstart,config.LengthBase, ...
    config.LengthUpper,config.LengthLower,config.Min1_2ArmAngle, ...
    config.Max1_2ArmAngle,config.MinLowerArmAngle,config.MaxLowerArmAngle );

if error > 1
    reachable = false;
    errorMsg
    return;
end

config.CS1_UpperA = [0, 0, 0];
config.CS1_LowerA = [0, 0, 0];
config.CS1_UpperB = [0, 0, 0];
config.CS1_LowerB = [0, 0, 0];
config.CS1_Gripper = [0, 0, 0];
config.CS1_UpperTorsion = [0, 0, 0];
config.CS1_UpperCrank = [0, 0, 0];
config.CS1_LowerTorsion = [0, 0, 0];
config.CS1_LowerCrank = [0, 0, 0];

config.CS2_UpperA = [ajX-aBaseX,ajY-aBaseY,0];
config.CS2_LowerA = [tcpX-ajX,tcpY-ajY,0];
config.CS2_UpperB = [bjX-bBaseX,bjY-bBaseY,0];
config.CS2_LowerB = [tcpX-bjX,tcpY-bjY,0];
config.CS2_Gripper = [0,-config.GripperLength,0];
config.CS2_UpperTorsion = [bjX-bBaseX,bjY-bBaseY,0];
config.CS2_UpperCrank = [config.UpperTorsionOffsetB_X, config.UpperTorsionOffsetB_Y, 0];
config.CS2_LowerTorsion = [tcpX-bjX,tcpY-bjY,0];
config.CS2_LowerCrank=[config.LowerTorsionOffsetTCP_X,config.LowerTorsionOffsetTCP_Y,0];

config.CS3_LowerB = [0 0 0];
config.CS3_UpperB = [0 0 0];
config.CS3_Gripper = [-0.05,0,0];
config.CS3_UpperCrank=[config.LowerTorsionOffsetTCP_X,config.LowerTorsionOffsetTCP_Y,0];
config.CS3_LowerCrank = [config.GripperMountOffset_X,config.GripperMountOffset_Y,0];

config.CS4_Gripper = [0.05,0,0];

config.CG_UpperA = [(ajX-aBaseX)/2,(ajY-aBaseY)/2,0];
config.CG_LowerA = [(tcpX-ajX)/2,(tcpY-ajY)/2,0];
config.CG_UpperB = [(bjX-bBaseX)/2,(bjY-bBaseY)/2,0];
config.CG_LowerB = [(tcpX-bjX)/2,(tcpY-bjY)/2,0];
config.CG_Gripper = [0,-config.GripperLength/2,0];
config.CG_UpperTorsion = [(bjX-bBaseX)/2,(bjY-bBaseY)/2,0];
config.CG_UpperCrank = [(config.UpperTorsionOffsetB_X+ ...
                         config.LowerTorsionOffsetTCP_X)/2, ...
                         config.UpperTorsionOffsetB_Y/2,0];
config.CG_LowerTorsion = [(tcpX-bjX)/2,(tcpY-bjY)/2,0];
config.CG_LowerCrank = [config.LowerTorsionOffsetTCP_X/2,0,0];
…
```

Figure F.3 CalculateConfig Function (Part 3/4)

```
…

config.OrientCG_UpperA = [0,0,0];
config.OrientCG_LowerA = [0,0,0];
config.OrientCG_UpperB = [0,0,0];
config.OrientCG_LowerB = [0,0,0];
config.OrientCG_Gripper = [0,0,0];
config.OrientCG_UpperTorsion = [0,0,0];
config.OrientCG_UpperCrank = [0,0,0];
config.OrientCG_LowerTorsion = [0,0,0];
config.OrientCG_LowerCrank = [0,0,0];

config.Gpoint_1 = [aBaseX, aBaseY, 0];
config.Gpoint_2 = [bBaseX, bBaseY, 0];
config.Gpoint_3 = [bBaseX+config.UpperTorsionOffsetB_X, ...
                   bBaseY+config.UpperTorsionOffsetB_Y,0];

[ thetaA, thetaB, error, errorMsg ] = Inverse_2DOF_PPM(tcpX,tcpY,config.LengthBase,...
                                                       config.LengthUpper,config.LengthLower);
if error ~=0
    reachable = false;
    errorMsg
    return;
end
config.ThetaA_IC = mod(thetaA + config.ThetaAstart,pi);
config.ThetaB_IC = mod(thetaB + config.ThetaBstart,pi);

%% Check reachability

reachable = CheckReachability(Moves,config);

%% Check dimensions are within user constraints
if reachable == true % only test if already passed reachability test
    if config.LengthBase < uc.MinMotorSeparation
        reachable = false;
    elseif config.LengthBase > 0.9*uc.MaxWidth
        reachable = false;
    elseif config.LengthUpper > (uc.MaxWidth - config.LengthBase)/2
        reachable = false;
    elseif config.LengthLower < config.LengthBase
        reachable = false;
    elseif config.LengthLower > sqrt((uc.MaxDepth - config.LengthUpper)^2 + ...
                                     (config.LengthBase/2)^2)
        reachable = false;
    end
end

end
```

Figure F.4 CalculateConfig Function (Part 4/4)

```
function cycletime = CheckConfigExists(pathID,config)
% Checks if Configuration has already been tested for this path.
% Returns an empty matrix if doesn't exist otherwise returns cycletime
% VARIABLES:
% pathID - ID for the current path being optimised
% config - Instance of Configuration class

    % Open database connection
    mysql('open','localhost:3306','root','mysql');
    mysql('use matlab_2dofppm');

    % Query searches for an exact matching of configuration parameters up to 4 decimal
    %  places accurate
    query = ['SELECT cycletime FROM simulations '...
            'WHERE PathID = "',num2str(pathID),'" '...
            'AND ROUND(proxarmlength,4) = ROUND("',num2str(config.LengthUpper),'",4) '...
            'AND ROUND(distarmlength,4) = ROUND("',num2str(config.LengthLower),'",4) '...
            'AND ROUND(motorseparation,4) = ROUND("',num2str(config.LengthBase),'",4) '...
            'AND ROUND(workspaceheight,4) = '...
            'ROUND("',num2str(config.WorkspaceHeight),'",4) '...
            'Limit 1 '
            ];

    cycletime = mysql(query);

    mysql('close')
end
```

Figure F.5 CheckConfigExists Function

```
function reachable = CheckReachability(Moves,config)
% Checks to see if all targets can be reached by the configuration
% VARIABLES:
% Moves - Instance of Moves class
% config - Instance of Configuration class

    reachable = true;   % set flag

    % Loop through each move and use inverse kinematics to check target can
    % be reached by configuration
    for m=1:size(Moves,2)
        knotX = Moves(m).Target.Knot.X;
        knotY = Moves(m).Target.Knot.Y + config.WorkspaceHeight;

        [thetaA,thetaB,error,errorMsg] = Inverse_2DOF_PPM(knotX,knotY,...
                                config.LengthBase,config.LengthUpper,config.LengthLower);

        if error ~= 0
            reachable = false;
            break
        end

        [tcpX,tcpY,error,errorMsg] = Direct_2DOF_PPM(thetaA,thetaB,config.LengthBase,...
                                        config.LengthUpper,config.LengthLower,...
                                        config.Min1_2ArmAngle,config.Max1_2ArmAngle,...
                                        config.MinLowerArmAngle,config.MaxLowerArmAngle);
        if error ~= 0
            reachable = false;
            break   % return from function if error occurs as it indicates its unreachable
        end
    end
end
```

Figure F.6 CheckReachability Function

```
function ppr = CompilePath(Moves,Config,PPC)
% Compiles a path for the 2DOFPPM Config based on the user specifed Move
% commands while keeping within the PPC (Path Planning Constraints)
% VARIABLES:
% Moves - Instance of the Moves class
% Config - Instance of the Configuration class
% PPC - Instance of the PPConstraints class (Path Planning Constraints)
% RETURNS:
% ppr - Instance of the PPResults class (Path Planning Results)

% Give Targets (within Moves) a PathTime to start with, based on the moves
% max Velocity constraint and the distance between the knots.
for m=1:(size(Moves,2)-1)
    Xc = Moves(m).Target.Knot.X;
    Yc = Moves(m).Target.Knot.Y;
    Xn = Moves(m+1).Target.Knot.X;
    Yn = Moves(m+1).Target.Knot.Y;
    dist = sqrt((Xn-Xc)^2+(Yn-Yc)^2);
    vel = Moves(m+1).Velocity;
    Moves(m+1).Target.PathTime = Moves(m).Target.PathTime + dist/vel;
end

Targets = repmat(Target,1,1);

%Formulate Knots from Targets
for m=1:(size(Moves,2)-1)
    current_move = Moves(m);
    next_move = Moves(m+1);
    current_target = Moves(m).Target;
    next_target = Moves(m+1).Target;
    next_MaxVel = Moves(m+1).Velocity;  %get max TCP velocity permitted during move

    if (m==1)   %then add first knot
        Targets(end)=current_target;
    end
…
```

Figure F.7 CompilePath Function (Part 1/5)

```
…
if strcmp(next_move.MoveType,'MoveL')  %then create additional knots inbetween targets
        kc = current_target.Knot;
        kn = next_target.Knot;
        Xc = kc.X;
        Yc = kc.Y;
        Tc = current_target.PathTime;
        Xn = kn.X;
        Yn = kn.Y;
        Tn = next_target.PathTime;
        Zn = next_move.Zone * 10^-3;  %zone data is defined in mm, therefore we scale

        % X value at edge of zone at next target
        if Xn >= Xc
            Xz = Xn-Zn*sin(atan(abs(Xn-Xc)/abs(Yn-Yc)));
        else
            Xz = Xn+Zn*sin(atan(abs(Xn-Xc)/abs(Yn-Yc)));
        end
        % Y value at edge of zone at next target
        if Yn >= Yc
            Yz = Yn-Zn*cos(atan(abs(Xn-Xc)/abs(Yn-Yc)));
        else
            Yz = Yn+Zn*cos(atan(abs(Xn-Xc)/abs(Yn-Yc)));
        end
        % Time value at edge of zone at next target
        Tz = Tn - (Tn-Tc)*Zn/sqrt((Xn-Xc)^2+(Yn-Yc)^2);
        % X value at last target before target at edge of zone
        if Xz >= Xc
            Xl = Xn-(Zn+PPC.LastLinearTargetDistance)*sin(atan(abs(Xn-Xc)/abs(Yn-Yc)));
        else
            Xl = Xn+(Zn+PPC.LastLinearTargetDistance)*sin(atan(abs(Xn-Xc)/abs(Yn-Yc)));
        end
        % Y value at last target before target at edge of zone
        if Yz >= Yc
            Yl = Yn-(Zn+PPC.LastLinearTargetDistance)*cos(atan(abs(Xn-Xc)/abs(Yn-Yc)));
        else
            Yl = Yn+(Zn+PPC.LastLinearTargetDistance)*cos(atan(abs(Xn-Xc)/abs(Yn-Yc)));
        end
        % Time value at edge of zone at next target
        Tl = Tn - (Tn-Tc)*(Zn+PPC.LastLinearTargetDistance)/sqrt((Xn-Xc)^2+(Yn-Yc)^2);
        %calculate number of steps/extra knots required in linear move
        LinearSteps = ceil(sqrt((Xl-Xc)^2+(Yl-Yc)^2)/PPC.LinearErrorFactor);

        for i=1:LinearSteps
            Xi = ((Xl-Xc)/LinearSteps*i)+Xc;
            Yi = ((Yl-Yc)/LinearSteps*i)+Yc;
            Ti = ((Tl-Tc)/LinearSteps*i)+Tc;
            k = Knot(Xi,Yi);
            t = Target(k,Ti,next_MaxVel);
            Targets(end+1)=t;
        end

        %finally add knot/target at edge of zone
        k = Knot(Xz,Yz);
        t = Target(k,Tz,next_MaxVel);
        Targets(end+1)=t;
…
```

Figure F.8 CompilePath Function (Part 2/5)

```
…
    elseif strcmp(next_move.MoveType, 'MoveJ') %then we can just add knots at targets
            kc = current_target.Knot;
            kn = next_target.Knot;
            Xc = kc.X;
            Yc = kc.Y;
            Tc = current_target.PathTime;
            Xn = kn.X;
            Yn = kn.Y;
            Tn = next_target.PathTime;
            Zn = next_move.Zone * 10^-3;

            % X value at edge of zone at next target
            if Xn >= Xc
                Xz = Xn-Zn*sin(atan(abs(Xn-Xc)/abs(Yn-Yc)));
            else
                Xz = Xn+Zn*sin(atan(abs(Xn-Xc)/abs(Yn-Yc)));
            end
            % Y value at edge of zone at next target
            if Yn >= Yc
                Yz = Yn-Zn*cos(atan(abs(Xn-Xc)/abs(Yn-Yc)));
            else
                Yz = Yn+Zn*cos(atan(abs(Xn-Xc)/abs(Yn-Yc)));
            end
            % Time value at edge of zone at next target
            Tz = Tn - (Tn-Tc)*Zn/sqrt((Xn-Xc)^2+(Yn-Yc)^2);

            k = Knot(Xz,Yz);
            t = Target(k,Tz,next_MaxVel);
            Targets(end+1)=t;
    end

    %add extra target for pause if move command has one
    if (next_move.Pause > 0)
        t=Targets(end);
        t.Knot.Omega_A = 0;
        t.Knot.Omega_B = 0;
        Targets(end)=t;
        tn=t;
        tn.PathTime = tn.PathTime + next_move.Pause;
        tn.Knot.Omega_A = 0;
        tn.Knot.Omega_B = 0;
        Targets(end+1)=tn;
        % increase time on following targets
        for n=m+1:(size(Moves,2))
            Moves(n).Target.PathTime = Moves(n).Target.PathTime+next_move.Pause;
        end
    end
end

% Assign zero velocity to first and last Knots
kf = Targets(1).Knot;
kf.Omega_A = 0;
kf.Omega_B = 0;
Targets(1).Knot = kf;
kl = Targets(end).Knot;
kl.Omega_A = 0;
kl.Omega_B = 0;
Targets(end).Knot = kl;

Targets1 = repmat(Target,1,0);
TargetsNew = repmat(Target,1,0);
pa = repmat(PathSegment,1,0);
pb = repmat(PathSegment,1,0);
…
```

Figure F.9 CompilePath Function (Part 3/5)

```
…

for t=1:(size(Targets,2)-1)
   if ((Targets(t).Knot.X == Targets(t+1).Knot.X) && ...
            (Targets(t).Knot.Y == Targets(t+1).Knot.Y))

       % collate last target that belongs to this particular sector/action
       Targets1(end+1) = Targets(t);

       [pa2,pb2,Targets2] = PathGenerator(Targets1,Config,PPC);

        %add targets for this sector/action to our collection of Targets
        for t2=1:size(Targets2,2)
            TargetsNew(end+1) = Targets2(t2);
        end
        %add paths for this sector/action to our collection of pa and pb
        for p2=1:size(pa2,1)
            if isempty(pa)
                pa(end,1) = pa2(p2);
                pb(end,1) = pb2(p2);
            else
                pa(end+1,1) = pa2(p2);
                pb(end+1,1) = pb2(p2);
            end
        end

        % As the PathTime has most likely changed, alter the remaining Targets PathTime
        %  so that it is continuous with the targets in TargetsNew
        intersectingTargetNum = size(TargetsNew,2);
        pathTimeDiff = TargetsNew(intersectingTargetNum).PathTime - ...
                                                Targets(intersectingTargetNum).PathTime;
        for tr=size(TargetsNew,2):size(Targets,2)
            Targets(tr).PathTime = Targets(tr).PathTime + pathTimeDiff;
        end

        % Clear this 'Targets1' because starting to collate from scratch new targets that
        %  will be used for a separate sector/action
        clear Targets1;
        Targets1 = repmat(Target,1,0);
   else
       % Collate targets that belong to this particular sector/action
       Targets1(end+1) = Targets(t);
   end
end
    %do this once more with the final targets from Targets1
    Targets1(end+1) = Targets(end);   %add final target
    kl = Targets1(end).Knot;
    kl.Omega_A = 0;
    kl.Omega_B = 0;
    Targets1(end).Knot = kl;

    [pa2,pb2,Targets2] = PathGenerator(Targets1,Config,PPC);

    %add targets for this sector/action to our collection of Targets
    for t2=1:size(Targets2,2)
        TargetsNew(end+1) = Targets2(t2);
    end
    %add paths for this sector/action to our collection of pa and pb
    for p2=1:size(pa2,1)
        pa(end+1,1) = pa2(p2);
        pb(end+1,1) = pb2(p2);
    end
…
```

Figure F.10 CompilePath Function (Part 4/5)

```
…

%% Path planning is now complete. The following processes the path for storage in files.

% get the number of interpolated readings for path
segmentIntPoints = 5;   % Take 5 samples along each path segment
intPoints = size(pa,1)*segmentIntPoints;

thetasA = zeros(intPoints,1);
omegasA = zeros(intPoints,1);
alphasA = zeros(intPoints,1);
thetasB = zeros(intPoints,1);
omegasB = zeros(intPoints,1);
alphasB = zeros(intPoints,1);

time = zeros(intPoints,1);
n = 1;

for s=1:size(pa,1)
    psA = pa(s);
    psB = pb(s);

    stepsize = (psA.EndTime-psA.StartTime)/segmentIntPoints;

    for t=psA.StartTime+0.0001:stepsize:psA.EndTime
        %NB: psB start and end times are the same as psA's
        thetaA = psA.getTheta(t);
        thetasA(n) = thetaA;
        omegaA = psA.getOmega(t);
        omegasA(n) = omegaA;
        alphaA = psA.getAlpha(t);
        alphasA(n) = alphaA;
        thetaB = psB.getTheta(t);
        thetasB(n) = thetaB;
        omegaB = psB.getOmega(t);
        omegasB(n) = omegaB;
        alphaB = psB.getAlpha(t);
        alphasB(n) = alphaB;
        time(n) = t;
        n = n+1;
    end
end

% remove trailing zeros from the pva results using deblank method
warning('off','MATLAB:deblank:NonStringInput');  %turn off warning
pvaA = deblank([time, thetasA, omegasA, alphasA]');
pvaB= deblank([time, thetasB, omegasB, alphasB]');

% produces a stop (1) command at the end of pva's to stop SimMechanics simulation
sControl = zeros(size(pvaA,2),1);
sControl(end) = 1;
SimControl = [deblank(time')',sControl]';

% save pva's and SimControl to .mat files for use in SimMechanis simulation
save(strcat(pwd,'\PG_Outputs\SimControl.mat'),'SimControl');
save(strcat(pwd,'\PG_Outputs\pvaA.mat'),'pvaA');
save(strcat(pwd,'\PG_Outputs\pvaB.mat'),'pvaB');
save(strcat(pwd,'\PG_Outputs\Knots_TXY.mat'),'Knots_TXY');

ppr = PPResults;
ppr.PathA = pa;
ppr.PathB = pb;
ppr.Knots = Knots_TXY;

end
```

Figure F.11 CompilePath Function (Part 5/5)

```
classdef Configuration
% Contains parameters defining the physical configuration of the manipulator.
% Also referred to as 'mvar' (model variable) in some methods.

properties
    MassUpper               % Mass of the upper/proximal arm
    MassLower               % Mass of the lower/distal arm
    MassGripper             % Mass of the gripper
    MassUpperTorsion        % Mass of the upper torsion bar
    MassLowerTorsion        % Mass of the lower torsion bar
    MassUpperCrank          % Mass of the upper crank arm
    MassLowerCrank          % Mass of the lower crank arm
    LengthBase              % Distance between the centers of the two motors
    LengthUpper             % Length of the upper/proximal arm
    LengthLower             % Length of the lower/distal arm
    GripperMountOffset_X    % Offset from bottom revolute joint where the gripper mounts(X)
    GripperMountOffset_Y    % Offset from bottom revolute joint where the gripper mounts(Y)
    GripperLength           % Length of the gripper

    WorkspaceHeight         % Height from motors to heighest knot

    MotorID                 % Id for motor type used

    UpperTorsionOffsetB_X   %Offset from centerof motorB for base point of stabiliserarm(X)
    UpperTorsionOffsetB_Y   %Offset from centerof motorB for base point of stabiliserarm(Y)
    LowerTorsionOffsetTCP_X % Offset from center of 'TCP' for lower torsion bar (X)
    LowerTorsionOffsetTCP_Y % Offset from center of 'TCP' for lower torsion bar (Y)

    InRadiusArms            % Inner radius of the tubular arms
    OutRadiusArms           % Outer radius of the tubular arms
    InRadiusTorsion         % Inner radius of the tubular torsion bars
    OutRadiusTorsion        % Outer radius of the tubular torsion bars

    InertiaUpper            % Inertia of the upper/proximal arm
    InertiaLower            % Inertia of the lower/distal arm
    InertiaGripper          % Inertia of the gripper
    InertiaUpperTorsion     % Inertia of the upper torsion bar
    InertiaLowerTorsion     % Inertia of the lower torsion bar
    InertiaUpperCrank       % Inertia of the upper crank arm
    InertiaLowerCrank       % Inertia of the lower crank arm

    MinUpperArmAngle        % Minimum angle allowed between upper arm and vertical
    MaxUpperArmAngle        % Minimum angle allowed between upper arm and vertical
    Min1_2ArmAngle          % Minimum angle allowed between upper-lower arms
    Max1_2ArmAngle          % Maximum angle allowed between upper-lower arms
    MinLowerArmAngle        % Minimum angle allowed between lower-lower arms
    MaxLowerArmAngle        % Maximum angle allowed between lower-lower arms

    ThetaAstart             % Starting angle between +Y axis and left upper arm
    ThetaBstart             % Starting angle between +Y axis and right upper arm

    CS1_UpperA              % Coordinate system 1 on the upper/proximal A arm
    CS1_LowerA              % Coordinate system 1 on the lower/distal A arm
    CS1_UpperB              % Coordinate system 1 on the upper/proximal B arm
    CS1_LowerB              % Coordinate system 1 on the lower/distal B arm
    CS1_Gripper             % Coordinate system 1 on the gripper
    CS1_UpperTorsion        % Coordinate system 1 on the upper torsion bar
    CS1_UpperCrank          % Coordinate system 1 on the upper crank arm
    CS1_LowerTorsion        % Coordinate system 1 on the lower torsion bar
    CS1_LowerCrank          % Coordinate system 1 on the lower crank arm
…
```

Figure F.12 Configuration Class (Part 1/2)

```
…
    CS2_UpperA              % Coordinate system 2 on the upper/proximal A arm
    CS2_LowerA              % Coordinate system 2 on the lower/distal A arm
    CS2_UpperB              % Coordinate system 2 on the upper/proximal B arm
    CS2_LowerB              % Coordinate system 2 on the lower/distal B arm
    CS2_Gripper             % Coordinate system 2 on the gripper
    CS2_UpperTorsion        % Coordinate system 2 on the upper torsion bar
    CS2_UpperCrank          % Coordinate system 2 on the upper crank arm
    CS2_LowerTorsion        % Coordinate system 2 on the lower torsion bar
    CS2_LowerCrank          % Coordinate system 2 on the lower crank arm

    CS3_LowerB              % Coordinate system 3 on the lower/distal A arm
    CS3_UpperB              % Coordinate system 3 on the lower/distal B arm
    CS3_Gripper             % Coordinate system 3 on the gripper
    CS3_UpperCrank          % Coordinate system 3 on the upper crank arm
    CS3_LowerCrank          % Coordinate system 3 on the lower crank arm

    CS4_Gripper             % Coordinate system 4 on the gripper

                            % CoG = Center of Gravity
    CG_UpperA               % CoG coordinate system on the upper/proximal A arm
    CG_LowerA               % CoG coordinate system on the lower/distal A arm
    CG_UpperB               % CoG coordinate system on the upper/proximal B arm
    CG_LowerB               % CoG coordinate system on the lower/distal B arm
    CG_Gripper              % CoG coordinate system on the gripper
    CG_UpperTorsion         % CoG coordinate system on the upper torsion bar
    CG_UpperCrank           % CoG coordinate system on the upper crank arm
    CG_LowerTorsion         % CoG coordinate system on the lower torsion bar
    CG_LowerCrank           % CoG coordinate system on the lower crank arm

    OrientCG_UpperA         %Orientation of CoG coordinate system on the upper/proximalAarm
    OrientCG_LowerA         %Orientation of CoG coordinate system on the lower/distal A arm
    OrientCG_UpperB         %Orientation of CoG coordinate system on the upper/proximalBarm
    OrientCG_LowerB         %Orientation of CoG coordinate system on the lower/distal B arm
    OrientCG_Gripper        % Orientation of CoG coordinate system on the gripper
    OrientCG_UpperTorsion   % Orientation of CoG coordinate system on the upper torsion bar
    OrientCG_UpperCrank     % Orientation of CoG coordinate system on the upper crank arm
    OrientCG_LowerTorsion   % Orientation of CoG coordinate system on the lower torsion bar
    OrientCG_LowerCrank     % Orientation of CoG coordinate system on the lower crank arm

    Gpoint_1                % Ground point 1
    Gpoint_2                % Ground point 2
    Gpoint_3                % Ground point 3

    ThetaA_IC               % Initial condition for theta position on motor A
    ThetaB_IC               % Initial condition for theta position on motor B
end

end
```

Figure F.13 Configuration Class (Part 2/2)

```
classdef CyclePath
% Contains constraints and move commands for a single cycle of a path

properties
    ID          % ID to uniquely identify each cycle path
    Moves       % Moves associated with this path
    PPC         % Path Planning Constraints (PPConstraints) for this path
end

end
```

Figure F.14 CyclePath Class

```
function r = d2r(d)
%#eml
    r=d/180*pi;     % Converts degrees to radians
end
```

Figure F.15 d2r (Degrees to Radians) Function

```
function [ tcpX, tcpY, error, errorMsg ] = ...
    Direct_2DOF_PPM( thetaA, thetaB, LengthBase,LengthUpper,LengthLower,...
        min1_2ArmAngle,max1_2ArmAngle,minLowerArmAngle,maxLowerArmAngle )
% Direct_2DOF_PPM takes the angles of the two upper arms (wrt the +Y axis) of a
% 2DOFPPM and outputs the coordinates of the TCP (tool center point)
% VARIABLES:
% thetaA - angle of motor A from +Y-axis (radians)
% thetaB - angle of motor B from +Y-axis (radians)
% LengthBase - length of base / separation of motors (m)
% LengthUpper - length of upper/proximal arm (m)
% LengthLower - length of lower/distal arm (m)
% min1_2ArmAngle - minimum allowable acute angle between proximal and distal arms(radians)
% max1_2ArmAngle - maximum allowable acute angle between proximal and distal arms(radians)
% minLowerArmAngle - minimum allowable acute angle between distal arms(radians)
% maxLowerArmAngle - maximum allowable acute angle between distal arms(radians)
% RETURNS:
% tcpX - X co-ordinate of the TCP/end-effector
% tcpY - Y co-ordinate of the TCP/end-effector
% error - value indicating an error (0 = no error)
% errorMsg - message associated with an error

D=LengthBase;           % Length of base
aBaseX = -D/2;          % X component of lhs of base
aBaseY = 0;             % Y component of lhs of base
bBaseX = D/2;           % X component of rhs of base
bBaseY = 0;             % Y component of rhs of base
a1=LengthUpper;         % Left Upper Arm
a2=LengthLower;         % Left Lower Arm
b1=LengthUpper;         % Right Upper Arm
b2=LengthLower;         % Right Lower Arm

error = 0;              % Notify an error exists by setting to 1
errorMsg = 'null';      % Details about error

ajX = ((-D/2)-a1*sin(pi-thetaA));       % X component of lhs arm joint
ajY = (-a1*cos(pi-thetaA));             % Y component of lhs arm joint
bjX = ((D/2)+b1*sin(pi-thetaB));        % X component of rhs arm joint
bjY = (-b1*cos(pi-thetaB));             % Y component of rhs arm joint

k = sqrt((bjX-ajX)^2+(abs(bjY-ajY))^2); % Distance between lhs & rhs joints

i = (a2^2-b2^2+k^2)/(2*k);

h = sqrt(a2^2-i^2);

mX = ajX + (i*(bjX-ajX))/k;
mY = ajY + (i*(bjY-ajY))/k;

tcpX = mX + (h*(bjY-ajY))/k;            % X component of TCP
tcpY = mY - (h*(bjX-ajX))/k;            % Y component of TCP


% Check lower arms still reach, else throw an error - added a 1% tollerance to allow for
%  calculation errors
if ((sqrt((ajX-tcpX)^2+(ajY-tcpY)^2)>a2*1.01)||(sqrt((bjX-tcpX)^2+(bjY-tcpY)^2)>b2*1.01))
    error = 2;
    errorMsg = 'Arm configuration cannot be resolved';
end

% Only permit TCP's below the base
if (tcpY > aBaseY)
    error = 2;
    errorMsg = 'TCP cannot be raised above base';
end
…
```

Figure F.16 Direct_2DOF_PPM Function (Part 1/2)

```
…

% Angle between upper left arm and lower left arm
thetaAJ = -atan2(((aBaseX-ajX)*(tcpY-ajY)-(tcpX-ajX)*(aBaseY-ajY)),...
                    (aBaseX-ajX)*(tcpX-ajX)+(aBaseY-ajY)*(tcpY-ajY));

% Angle between upper right arm and lower right arm
thetaBJ = atan2(((bBaseX-bjX)*(tcpY-bjY)-(tcpX-bjX)*(bBaseY-bjY)),...
                    (bBaseX-bjX)*(tcpX-bjX)+(bBaseY-bjY)*(tcpY-bjY));

% Angle between right and left lower fore arms
thetaTCP = atan2(((bjX-tcpX)*(ajY-tcpY)-(ajX-tcpX)*(bjY-tcpY)),...
                    (bjX-tcpX)*(ajX-tcpX)+(bjY-tcpY)*(ajY-tcpY));


% Check all joint angles are within limits
if (thetaAJ < min1_2ArmAngle)
    error = 1;
    errorMsg = strcat('Interference between A arms. (',num2str(thetaAJ*180/pi),...
                        '<',num2str(min1_2ArmAngle*180/pi),')!');
elseif (thetaBJ < min1_2ArmAngle)
    error = 1;
    errorMsg = strcat('Interference between B arms. (',num2str(thetaBJ*180/pi),...
                        '<',num2str(min1_2ArmAngle*180/pi),')!');
elseif (thetaTCP < minLowerArmAngle)
    error = 1;
    errorMsg = strcat('Interference between lower arms. (',num2str(thetaTCP*180/pi),...
                        '<',num2str(minLowerArmAngle*180/pi),')!');
elseif (thetaAJ > max1_2ArmAngle)
    error = 1;
    errorMsg = strcat('Angle between A arms is too great. (',num2str(thetaAJ*180/pi),...
                        '>',num2str(max1_2ArmAngle*180/pi),')!');
elseif (thetaBJ > max1_2ArmAngle)
    error = 1;
    errorMsg = strcat('Angle between B arms is too great. (',num2str(thetaBJ*180/pi),...
                        '>',num2str(max1_2ArmAngle*180/pi),')!');
elseif (thetaTCP > maxLowerArmAngle)
    error = 1;
    errorMsg = strcat('Angle between lower arms is too great. (',...
                    num2str(thetaTCP*180/pi),'>',num2str(maxLowerArmAngle*180/pi),')!');
end

% Check all parameters are real (if complex, it indicates that the arms can not reach).
% Return error = 2 if can't reach
if (isreal(aBaseX) == false ...
   || isreal(aBaseY) == false ...
   || isreal(bBaseX) == false ...
   || isreal(bBaseY) == false ...
   || isreal(ajX) == false ...
   || isreal(ajY) == false ...
   || isreal(bjX) == false ...
   || isreal(bjY) == false ...
   || isreal(tcpX) == false ...
   || isreal(tcpY) == false)

    error = 2;
    errorMsg = strcat('Arm configuration is invalid. Cannot form closed loop.');
end
end
```

Figure F.17 Direct_2DOF_PPM Function (Part 2/2)

```
function [Velocity] = EstimateTCPVel(Knot0,Knot1,StartTime,EndTime)
% Estimates the TCP/end-effector velocity based on the time taken to travel between two
% knots
% VARIABLES:
% Knot0 - Instance of Knot class, travelling from
% Knot1 - Instance of Knot class, travelling to
% StartTime - Time at Knot0
% EndTime - Time at Knot1
% RETURNS:
% Velocity - estimated velocity

    x0 = Knot0.X;
    y0 = Knot0.Y;
    x1 = Knot1.X;
    y1 = Knot1.Y;

    dist = sqrt((x1-x0)^2+(y1-y0)^2);

    Velocity = dist/(EndTime-StartTime);
end
```

Figure F.18 EstimateTCPVel Function

```
function Torque = EstimateTorqueA(LengthUpper,Mass_upper,Mass_lower,MassGripper,...
                                  Mass_LowerCrank,theta,alpha)
% Estimates the torque requited by MotorA under a given acceleration
% VARIABLES:
% LengthUpper - Length of upper/proximal arm (m)
% Mass_upper - Mass of upper/proximal arm (kg)
% Mass_lower - Mass of lower/distal arm (kg)
% MassGripper - Mass of gripper (kg)
% Mass_LowerCrank - Mass of lower crank (kg)
% theta - angle of MotorA (rad)
% alpha - angular acceleration fo MotorA (rad/s/s)
% RETURNS:
% Torque - estimate of torque required by MotorA (Nm)

g = 9.81;   % Define gravity in SI units

% Inertia of arms acting on motor
Inertia = ((Mass_upper)*(LengthUpper/2)^2) + ...
            ((Mass_lower+Mass_LowerCrank/2+MassGripper/2)*(LengthUpper)^2);

% Torque due to gravity
T_gravity = (Mass_upper)*g*(LengthUpper/2)*sin(theta) + ...
                (Mass_lower+Mass_LowerCrank/2+MassGripper/2)*g*LengthUpper*sin(theta);

% Total torque
Torque = Inertia * alpha + T_gravity;

end
```

Figure F.19 EstimateTorqueA Function

```
function Torque = EstimateTorqueB(LengthUpper,Mass_upper,Mass_lower,MassGripper,...
                                  Mass_UpperTorsion,Mass_LowerTorsion,Mass_UpperCrank,...
                                  Mass_LowerCrank,theta,alpha)
% Estimates the torque required by MotorB under a given acceleration
% VARIABLES:
% LengthUpper - Length of upper/proximal arm (m)
% Mass_upper - Mass of upper/proximal arm (kg)
% Mass_lower - Mass of lower/distal arm (kg)
% MassGripper - Mass of gripper (kg)
% Mass_UpperTorsion - Mass of upper/proximal torsion arm (kg)
% Mass_LowerTorsion - Mass of lower/distal torsion arm (kg)
% Mass_UpperCrank - Mass of upper/proximal crank (kg)
% Mass_LowerCrank - Mass of lower/distal crank (kg)
% theta - angle of MotorB (rad)
% alpha - angular acceleration fo MotorB (rad/s/s)
% RETURNS:
% Torque - estimate of torque required by MotorA (Nm)

g=9.81;     % Define gravity in SI units

% Inertia of arms acting on motor
Inertia = ((Mass_upper+Mass_UpperTorsion)*(LengthUpper/2)^2) + ...
           ((Mass_lower+Mass_LowerTorsion+Mass_UpperCrank+Mass_LowerCrank/2+ ...
                                                    MassGripper/2)*(LengthUpper)^2);

% Torque due to gravity
T_gravity = (Mass_upper+Mass_UpperTorsion)*g*(LengthUpper/2)*sin(theta) + ...
           (Mass_lower+Mass_LowerTorsion+Mass_UpperCrank+Mass_LowerCrank/2+ ...
                                            MassGripper/2)*g*LengthUpper*sin(theta);

% Total torque
Torque = Inertia * alpha + T_gravity;
```

Figure F.20 EstimateTorqueB Function

```
function [Alpha,Time] = FindMaxAlpha(Coef,StartTime,EndTime)
% Finds near-maximum angular acceleration of motors for cubic polynomial trajectory
% VARIABLES:
% Coef - Cubic polynomial coefficients
% StartTime - Start time of trajectory sector
% EndTime - End time of trajectory sector
% RETURNS:
% Alpha - Maximum angular acceleration of trajectory sector
% Time - Time at which maximum angular acceleration occurs

Alpha = -99999999999999;
Time = StartTime;
% take 10 samples of Alpha between start and end times
for time=StartTime:(EndTime-StartTime)/10:EndTime
    t = time - StartTime;
    A = 2*Coef(3) + 6*Coef(4)*t;

    if (A > Alpha)
        Alpha = A;
        Time = time;
    end
end

end
```

Figure F.21 FindMaxAlpha Function

```
function [Jerk,Time] = FindMaxJerk(Coef,StartTime,EndTime)
% Finds near-maximum angular jerk of motors for cubic polynomial trajectory
% VARIABLES:
% Coef - Cubic polynomial coefficients
% StartTime - Start time of trajectory sector
% EndTime - End time of trajectory sector
% RETURNS:
% Jerk - Maximum angular jerk of trajectory sector
% Time - Time at which maximum angular jerk occurs


Time = StartTime;
Jerk = 6*Coef(4);

end
```

Figure F.22 FindMaxJerk Function

```
function [Omega,Time] = FindMaxOmega(Coef,StartTime,EndTime)
% Finds near-maximum angular velocity of motor for cubic polynomial trajectory
% VARIABLES:
% Coef - Cubic polynomial coefficients
% StartTime - Start time of trajectory sector
% EndTime - End time of trajectory sector
% RETURNS:
% Omega - Maximum angular velocity of trajectory sector
% Time - Time at which maximum angular velocity occurs


Omega = -99999999999999;
Time = StartTime;
% take 10 samples of Omega between start and end times
for time=StartTime:(EndTime-StartTime)/10:EndTime
    t = time - StartTime;
    O = Coef(2) + 2*Coef(3)*t + 3*Coef(4)*t^2;

    if (O > Omega)
        Omega = O;
        Time = time;
    end
end

end
```

Figure F.23 FindMaxOmega Function

```
function [Torque,Time] = FindMaxTorqueA(Coef,StartTime,EndTime,Mass_upper,Mass_lower,...
                                        MassGripper,Mass_LowerCrank,LengthUpper)
% Finds near-maximum torque of motor A for cubic polynomial trajectory
% VARIABLES:
% Coef - Cubic polynomial coefficients
% StartTime - Start time of trajectory sector (s)
% EndTime - End time of trajectory sector (s)
% Mass_upper - Mass of upper/proximal arm (kg)
% Mass_lower - Mass of lower/distal arm (kg)
% MassGripper - Mass of gripper (kg)
% Mass_LowerCrank - Mass of lower crank (kg)
% LengthUpper - Length of upper/proximal arm (m)
% RETURNS:
% Torque - Maximum torque of trajectory sector
% Time - Time at which maximum torque occurs

Torque = -99999999999999;
Time = StartTime;
% take 10 samples of estimated torque between start and end times
for time=StartTime:(EndTime-StartTime)/10:EndTime
    t = time - StartTime;
    T = EstimateTorqueA(LengthUpper,Mass_upper,Mass_lower,MassGripper,Mass_LowerCrank,...
            (Coef(1) + Coef(2)*t + Coef(3)*t^2 + Coef(4)*t^3),(2*Coef(3)+6*Coef(4)*t));

    if (T > Torque)
        Torque = T;
        Time = time;
    end
end

end
```

Figure F.24 FindMaxTorqueA Function

```
function [Torque,Time] = FindMaxTorqueB(Coef,StartTime,EndTime,Mass_upper,Mass_lower,...
                                        MassGripper,Mass_UpperTorsion,Mass_LowerTorsion,...
                                        Mass_UpperCrank,Mass_LowerCrank,LengthUpper)
% Finds near-maximum torque of motor B for cubic polynomial trajectory
% VARIABLES:
% Coef - Cubic polynomial coefficients
% StartTime - Start time of trajectory sector (s)
% EndTime - End time of trajectory sector (s)
% Mass_upper - Mass of upper/proximal arm (kg)
% Mass_lower - Mass of lower/distal arm (kg)
% MassGripper - Mass of gripper (kg)
% Mass_UpperTorsion - Mass of upper/proximal torsion arm (kg)
% Mass_LowerTorsion - Mass of lower/distal torsion arm (kg)
% Mass_UpperCrank - Mass of upper/proximal crank (kg)
% Mass_LowerCrank - Mass of lower/distal crank (kg)
% LengthUpper - Length of upper/proximal arm (m)
% RETURNS:
% Torque - Maximum torque of trajectory sector
% Time - Time at which maximum torque occurs

Torque = -99999999999999;
Time = StartTime;
% take 10 samples of estimated torque between start and end times
for time=StartTime:(EndTime-StartTime)/10:EndTime
    t = time - StartTime;
    T = EstimateTorqueB(LengthUpper,Mass_upper,Mass_lower,MassGripper,...
                        Mass_UpperTorsion,Mass_LowerTorsion,Mass_UpperCrank,...
                        Mass_LowerCrank,(Coef(1) + Coef(2)*t + Coef(3)*t^2 + ...
                                         Coef(4)*t^3),(2*Coef(3)+6*Coef(4)*t));
    if (T > Torque)
        Torque = T;
        Time = time;
    end
end

end
```

Figure F.25 FindMaxTorqueB Function

```
function [Alpha,Time] = FindMinAlpha(Coef,StartTime,EndTime)
% Finds near-minimum angular acceleration of motors for cubic polynomial trajectory
% VARIABLES:
% Coef - Cubic polynomial coefficients
% StartTime - Start time of trajectory sector
% EndTime - End time of trajectory sector
% RETURNS:
% Alpha - Minimum angular acceleration of trajectory sector
% Time - Time at which minimum angular acceleration occurs

Alpha = 99999999999999;
Time = StartTime;
% take 10 samples of Alpha between start and end times
for time=StartTime:(EndTime-StartTime)/10:EndTime
    t = time - StartTime;
    A = 2*Coef(3) + 6*Coef(4)*t;

    if (A < Alpha)
        Alpha = A;
        Time = time;
    end
end

end
```

Figure F.26 FindMinAlpha Function

```
function [Jerk,Time] = FindMinJerk(Coef,StartTime,EndTime)
% Finds near-minimum angular jerk of motors for cubic polynomial trajectory
% VARIABLES:
% Coef - Cubic polynomial coefficients
% StartTime - Start time of trajectory sector
% EndTime - End time of trajectory sector
% RETURNS:
% Jerk - Minimum angular jerk of trajectory sector
% Time - Time at which minimum angular jerk occurs

Time = StartTime;
Jerk = 6*Coef(4);

end
```

Figure F.27 FindMinJerk Function

```
function [Omega,Time] = FindMinOmega(Coef,StartTime,EndTime)
% Finds near-minimum angular velocity of motor for cubic polynomial trajectory
% VARIABLES:
% Coef - Cubic polynomial coefficients
% StartTime - Start time of trajectory sector
% EndTime - End time of trajectory sector
% RETURNS:
% Omega - Minimum angular velocity of trajectory sector
% Time - Time at which minimum angular velocity occurs

Omega = 99999999999999;
Time = StartTime;
% take 10 samples of Omega between start and end times
for time=StartTime:(EndTime-StartTime)/10:EndTime
    t = time - StartTime;
    O = Coef(2) + 2*Coef(3)*t + 3*Coef(4)*t^2;

    if (O < Omega)
        Omega = O;
        Time = time;
    end
end

end
```

Figure F.28 FindMinOmega Function

```
function [Torque,Time] = FindMinTorqueA(Coef,StartTime,EndTime,Mass_upper,Mass_lower,...
                                        MassGripper,Mass_LowerCrank,LengthUpper)
% Finds near-minimum torque of motor A for cubic polynomial trajectory
% VARIABLES:
% Coef - Cubic polynomial coefficients
% StartTime - Start time of trajectory sector (s)
% EndTime - End time of trajectory sector (s)
% Mass_upper - Mass of upper/proximal arm (kg)
% Mass_lower - Mass of lower/distal arm (kg)
% MassGripper - Mass of gripper (kg)
% Mass_LowerCrank - Mass of lower crank (kg)
% LengthUpper - Length of upper/proximal arm (m)
% RETURNS:
% Torque - Minimum torque of trajectory sector (Nm)
% Time - Time at which minimum torque occurs

Torque = 99999999999999;
Time = StartTime;
% take 10 samples of estimated torque between start and end times
for time=StartTime:(EndTime-StartTime)/10:EndTime
    t = time - StartTime;
    T = EstimateTorqueA(LengthUpper,Mass_upper,Mass_lower,MassGripper,Mass_LowerCrank,...
            (Coef(1) + Coef(2)*t + Coef(3)*t^2 + Coef(4)*t^3),(2*Coef(3)+6*Coef(4)*t));

    if (T < Torque)
        Torque = T;
        Time = time;
    end
end

end
```

Figure F.29 FindMinTorqueA Function

```
function [Torque,Time] = FindMinTorqueB(Coef,StartTime,EndTime,Mass_upper,Mass_lower,...
                                        MassGripper,Mass_UpperTorsion,Mass_LowerTorsion,...
                                        Mass_UpperCrank,Mass_LowerCrank,LengthUpper)
% Finds near-minimum torque of motor B for cubic polynomial trajectory
% VARIABLES:
% Coef - Cubic polynomial coefficients
% StartTime - Start time of trajectory sector (s)
% EndTime - End time of trajectory sector (s)
% Mass_upper - Mass of upper/proximal arm (kg)
% Mass_lower - Mass of lower/distal arm (kg)
% MassGripper - Mass of gripper (kg)
% Mass_UpperTorsion - Mass of upper/proximal torsion arm (kg)
% Mass_LowerTorsion - Mass of lower/distal torsion arm (kg)
% Mass_UpperCrank - Mass of upper/proximal crank (kg)
% Mass_LowerCrank - Mass of lower/distal crank (kg)
% LengthUpper - Length of upper/proximal arm (m)
% RETURNS:
% Torque - Minimum torque of trajectory sector
% Time - Time at which minimum torque occurs

Torque = 99999999999999;
Time = StartTime;
% take 10 samples of estimated torque between start and end times
for time=StartTime:(EndTime-StartTime)/10:EndTime
    t = time - StartTime;
    T = EstimateTorqueB(LengthUpper,Mass_upper,Mass_lower,MassGripper,...
                        Mass_UpperTorsion,Mass_LowerTorsion,Mass_UpperCrank,...
                        Mass_LowerCrank,(Coef(1) + Coef(2)*t + Coef(3)*t^2 + ...
                                         Coef(4)*t^3),(2*Coef(3)+6*Coef(4)*t));
    if (T < Torque)
        Torque = T;
        Time = time;
    end
end

end
```

Figure F.30 FindMinTorqueB Function

```
function PathID = GetNextPathID()
% Returns the next available (unused) path identifier from database

    % Connect to database
    mysql('open','localhost:3306','root','mysql')
    mysql('use matlab_2dofppm')

    PathID = mysql('SELECT IFNULL(MAX(PathID)+1,1) FROM paths');

    mysql('close')

end
```

Figure F.31 GetNextPathID Function

```
function ppc = GetPPConstraints()
% Returns an instance of the PPConstraints class containing the Path Planning Constraints

    % Create new instance of class
    ppc = PPConstraints;

    ppc.LinearErrorFactor = 0.2;
    ppc.LastLinearTargetDistance = 0.02;
    ppc.ReactiveFactor = 0.5;
    ppc.InitialAcceptanceThreshold = 0.8;
    ppc.RelativeAgeingFactor = 100;

end
```

Figure F.32 GetPPConstraints Function

```
function [ thetaA, thetaB, error, errorMsg ] = ...
                Inverse_2DOF_PPM( tcpX, tcpY, LengthBase,LengthUpper,LengthLower)
% Returns the angles required for a given (X,Y) TCP coordinate
% VARIABLES:
% tcpX - tcp/end-effector X coordinate
% tcpY - tcp/end-effector Y coordinate
% LengthBase - length of base / separation of motors (m)
% LengthUpper - length of upper/proximal arm (m)
% LengthLower - length of lower/distal arm (m)
% RETURNS:
% thetaA - angle of motor A from +Y-axis (radians)
% thetaB - angle of motor B from +Y-axis (radians)
% error - value indicating an error (0 = no error)
% errorMsg - message associated with an error

    D=LengthBase;           % Length of base
    a1=LengthUpper;         % Left Upper Arm
    a2=LengthLower;         % Left Lower Arm
    b1=LengthUpper;         % Right Upper Arm
    b2=LengthLower;         % Right Lower Arm

    error = 0;              % Set as no error
    errorMsg = 'null';      % Details about error

    aA = -2*a1*tcpY;
    aB = -2*a1*(tcpX + (D/2));
    aC = tcpX^2 + tcpY^2+(D/2)^2+a1^2-a2^2+2*(D/2)*tcpX;

    bA = -2*b1*tcpY;
    bB = -2*b1*(tcpX - (D/2));
    bC = tcpX^2 + tcpY^2+(D/2)^2+b1^2-b2^2-2*(D/2)*tcpX;

    thetaA = 2*atan((-aA-sqrt(aA^2-aC^2+aB^2))/(aC-aB));
    thetaB = 2*atan((-bA+sqrt(bA^2-bC^2+bB^2))/(bC-bB));

    thetaA = thetaA - d2r(90);  % Convert to project's conventions
    thetaB = d2r(90)-thetaB;    % Convert to project's conventions

    % Check all parameters are real (if complex, it indicates that the arms can not reach.
    % Give error = 2 if can't reach
    if (isreal(thetaA) == false || isreal(thetaB) == false)
        error = 2;
        errorMsg = strcat('Arm configuration is invalid. Cannot form closed loop.');
    end

end
```

Figure F.33 Inverse_2DOF_PPM Function

```
classdef Knot
% Defines a knot (position of end effector).
% The knot is defined both in terms of its cartesian coordinates in the
% workspace, as well as its joint coordinates in the joint space.

properties
    X        % X component of knot in cartesian coordinate
    Y        % Y component of knot in cartesian coordinate
    Theta_A  % Theta_A component of knot in joint space
    Theta_B  % Theta_B component of knot in joint space
    Omega_A  % Angular velocity of motor A at knot
    Omega_B  % Angular velocity of motor B at knot
end

methods
    % Create instance of Knot class with variables
    function k = Knot(X,Y,Theta_A,Theta_B)
        if nargin == 2  % Allow defining with only X,Y
            k.X = X;
            k.Y = Y;
        elseif nargin == 4
            k.X = X;
            k.Y = Y;
            k.Theta_A = Theta_A;
            k.Theta_B = Theta_B;
        else
        end
    end

end
end
```

Figure F.34 Knot Class

```
classdef MoveCMD
% Defines a move command (Target,MoveType,Velocity,Zone,[Pause]).

properties
Target      % Position/Orientation and PathTime
MoveType    % Either Linear or Joint move
Velocity    % Maximum velocity limit for the TCP
Zone        % Distance from Knot at which Target is considered reached thus
            %  allowing next Target to be aimed at
Pause       % Time period for manipulator to stop stationary after completing this move
end

methods
    % Create instance of MoveCMD class with variables
    function m = MoveCMD(Target,MoveType,Velocity,Zone,Pause)
        if nargin == 4  % If only 4 arguments specified (omitting Pause) set Pause = 0
            m.Target = Target;
            m.MoveType = MoveType;
            m.Velocity = Velocity;
            m.Zone = Zone;
            m.Pause = 0;
        end
        if nargin == 5
            m.Target = Target;
            m.MoveType = MoveType;
            m.Velocity = Velocity;
            m.Zone = Zone;
            m.Pause = Pause;
        end
    end
end
end
```

Figure F.35 MoveCMD Class

```
% Create new instance of UserConstraints class and set variables
uc = UserConstraints;
uc.MaxMotorTorque = 300;
uc.MaxMotorVelocity = 20;
uc.MaxMotorAcceleration = 9999;
uc.MaxMotorJerk = 999999;
uc.MassGripper = 35;
uc.MinArmAng_BU = d2r(33);
uc.MinArmAng_UL = d2r(43);
uc.MinArmAng_LL = d2r(48);
uc.MaxArmAng_BU = d2r(175);
uc.MaxArmAng_UL = d2r(134);
uc.MaxArmAng_LL = d2r(71);
uc.ProxArmDensity = 2700;
uc.DistArmDensity = 2700;
uc.TorsionArmDensity = 2700;
uc.ProxArmIRadius = 0.01;
uc.DistArmIRadius = 0.01;
uc.ProxArmORadius = 0.02;
uc.DistArmORadius = 0.02;
uc.TorsionIRadius = 0.005;
uc.TorsionORadius = 0.01;
uc.MassUpperCrank = 0.2;
uc.MassLowerCrank = 0.2;
uc.UpperTorsionOffsetB_X = 0.05;
uc.UpperTorsionOffsetB_Y = 0.1;
uc.LowerTorsionOffsetTCP_X = -0.05;
uc.LowerTorsionOffsetTCP_Y = 0.1;
uc.GripperMountOffset_X = 0;
uc.GripperMountOffset_Y = -0.02;
uc.GripperLength = 0.01;
uc.MinMotorSeparation = 0.01;
uc.MaxWidth = 1.5;
uc.MaxDepth = 2;

% Specify knots for Path
k1 = Knot(-0.3,-1);
k2 = Knot(-0.3,-0.7);
k3 = Knot(0,-0.65);
k4 = Knot(0.3,-0.7);
k5 = Knot(0.3,-1);
k6 = Knot(0.3,-0.75);
k7 = Knot(0,-0.7);
k8 = Knot(-0.3,-0.75);
k9 = Knot(-0.3,-1);

% Create Move commands from Knots
m1 = MoveCMD(Target(k1),'MoveJ',10,1);
m2 = MoveCMD(Target(k2),'MoveL',10,30);
m3 = MoveCMD(Target(k3),'MoveJ',10,50);
m4 = MoveCMD(Target(k4),'MoveJ',10,30);
m5 = MoveCMD(Target(k5),'MoveL',10,1,0.2);
m6 = MoveCMD(Target(k6),'MoveL',10,20);
m7 = MoveCMD(Target(k7),'MoveJ',10,30);
m8 = MoveCMD(Target(k8),'MoveJ',10,20);
m9 = MoveCMD(Target(k9),'MoveL',10,1);

% Create new CyclePath from Moves
cp = CyclePath;
cp.ID = GetNextPathID();
cp.Moves = [m1 m2 m3 m4 m5 m6 m7 m8 m9];
cp.PPC = GetPPConstraints();

% Set termination conditions
termcond = TerminationCondition;
termcond.CycleTime = 0.1;
termcond.Iterations = 300;
…
```

Figure F.36 OptimisationStart Script (Part 1/2)

```
…

try
    % Optimisation method specific parameters (in this case the GA)
    popSize = 50;
    selectionSize = 30;
    mutationRate = 0.25;
    mutationAmount = 0.1;

    % Run optimisation technique (in this case the GA)
    OptimiseConfigurationGA(...
                    cp,termcond,uc,popSize,selectionSize,mutationRate,mutationAmount);

catch exception     % Send email notification if excetion occurs
    send_mail_message('matlab2dofppm','ERROR: MATLAB Simulation',...
                                                    getReport(exception, 'extended'))
end
```

Figure F.37 OptimisationStart Script (Part 2/2)

```
function OptimiseConfigurationGA(CP,TermCond,UConstraints,PopSize,...
                                                    SelectionSize,MutationRate,MutationAmount)
% Uses a Genetic Algorithm to narrow on time-minimum configuration
% VARIABLES:
% CP - Cycle Path class containing geometric details of the path
% TermCond - Termination Condition class detailing conditions of terminating process
% UConstraints - User Constraints class
% PopSize - Number of individuals in GA population
% SelectionSize - Number of individuals selected for breeding
% MutationRate - Probability of mutation occuring in child (%)
% MutationAmount - The amount of mutation to occur in child (%)

% Store path and user constraint data
StorePathsUserConstraintsSQL(CP,UConstraints);
population = repmat(Configuration,PopSize,1);
popPPC = repmat(PPConstraints,PopSize,1);
popFitness = zeros(PopSize,1);
popCycleTime = zeros(PopSize,1);
popMotorID = zeros(PopSize,1);

%% INITIALISATION - Initialise population by selecting random configurations

for p=1:PopSize
    % Select 'random' motor details from database
    [motorID,newPPC] = SelectMotor(CP.PPC,UConstraints);
    CP.PPC = newPPC;    % Assign Path Planning Constraints (PPC) of motor to Cycle Path(CP)
    % Select random configuration that reaches all move targets
    config = SelectRandomConfig(CP.Moves,motorID,UConstraints);
    population(p) = config;
    popPPC(p) = CP.PPC;
    popMotorID(p) = motorID;
end

% Perform GA for a set number of evolution cycles
for i=1:TermCond.Iterations
    % Check if popCycleTimes are too similar and replace some with random configurations
    if i >1
        minct = 500;
        maxct = 0;
        for p=1:PopSize
            if popCycleTime(p) < 5000
               if popCycleTime(p) < minct
                   minct = popCycleTime(p);
               end
               if popCycleTime(p) > maxct
                   maxct = popCycleTime(p);
               end
            end
        end
        if maxct-minct < 0.2   % Population is too inbred!
            % Replace 10% of inbred population with random individuals
            for rp = 1:floor(PopSize/10)
                % Select 'random' motor details from database
                [motorID,newPPC] = SelectMotor(CP.PPC,UConstraints);
                % Assign Path Planning Constraints (PPC) of motor to Cycle Path (CP)
                CP.PPC = newPPC;
                % Select random configuration that reaches all move targets
                config = SelectRandomConfig(CP.Moves,motorID,UConstraints);

                % Randomly select an individual from population for replacement
                replaceP = ceil(PopSize*rand(1));
                population(replaceP) = config;
                popPPC(replaceP) = CP.PPC;
                popMotorID(replaceP) = motorID;
            end
        end
    end
…
```

Figure F.38 OptimiseConfigurationGA Function (Part 1/4)

```
…

%% EVALUATION - Evaluate the performance of each individual in population

    for p=1:PopSize
        config = population(p);
        ppc = popPPC(p);
        try
            % Check if config has already been simulated, return cycletime if it exists in
            % database
            ct = CheckConfigExists(CP.ID,config);
            if isempty(ct)
                % Evaluate the selected individual by compiling a path
                [Targets_XYZ,ppr] = CompilePath(CP.Moves,config,ppc);
                 %Store results of path planning in database
                StoreSimulationsSQL(config,ppc,ppr,CP.ID,i);

                popCycleTime(p) = ppr.PathA(size(ppr.PathA,1)).EndTime;
            else
                popCycleTime(p) = ct;
            end
        catch exception
            % If error occurs give individual poor cycletime so will be repaced next cycle
            popCycleTime(p) = 99999;
        end
        popFitness(p) = 1/(popCycleTime(p)^3); % Fitness equals inverse of cycletime cubed
    end

%% SELECTION - Select sub population from population for breeding based on fitness

    sumFitness = 0;
    for p=1:PopSize
       sumFitness = sumFitness + popFitness(p);
    end
    % Assign selection probability to each individual in population based on fitness
    popProb = zeros(PopSize,1);
    for p=1:PopSize
       popProb(p) = popFitness(p)/sumFitness;
    end
    selectionProb = zeros(PopSize,1);
    sumProb = 0;
    for p=1:PopSize
       selectionProb(p) = sumProb + popProb(p);
       sumProb = selectionProb(p);
    end

    % Select a number (SelectionSize) of the population for breeding
    selectedPop = repmat(Configuration,SelectionSize,1);
    selectedPopPPC = repmat(PPConstraints,SelectionSize,1);
    selectedPopMotorID = zeros(SelectionSize,1);
    selectedPopCycleTime = zeros(SelectionSize,1);
    for s=1:SelectionSize
        randnum = rand(1);
        for p=1:PopSize
            if selectionProb(p) > randnum
                selectedPop(s) = population(p);
                selectedPopPPC(s) = popPPC(p);
                selectedPopMotorID(s) = popMotorID(p);
                selectedPopCycleTime(s) = popCycleTime(p);
                break;
            end
        end
    end
…
```

Figure F.39 OptimiseConfigurationGA Function (Part 2/4)

```
…

%% REPRODUCTION

    % Add selected parents to new population
    newPopulation = repmat(Configuration,PopSize,1);
    newPopPPC = repmat(PPConstraints,PopSize,1);
    newPopMotorID = zeros(PopSize,1);
    for s=1:SelectionSize
        newPopulation(s) = selectedPop(s);
        newPopPPC(s) = selectedPopPPC(s);
        newPopMotorID(s) = selectedPopMotorID(s);
    end

    % Generate children to fill rest of new population
    for p=SelectionSize:PopSize
        reachable = false;
        unique = true;
        while reachable == false && unique == true;
            % CROSSOVER - children configuration dimensions are a random number between
            % their two parents

            % select two random parents from selectedPop
            randnum1 = ceil(rand(1)*SelectionSize);
            randnum2 = ceil(rand(1)*SelectionSize);
            parent1Config = selectedPop(randnum1);
            parent2Config = selectedPop(randnum2);
            parent1ppc = selectedPopPPC(randnum1);
            parent2ppc = selectedPopPPC(randnum2);
            parent1motorID = selectedPopMotorID(randnum1);
            parent2motorID = selectedPopMotorID(randnum2);

            minlb = min([parent1Config.LengthBase parent2Config.LengthBase]);
            maxlb = max([parent1Config.LengthBase parent2Config.LengthBase]);
            lb = minlb+(maxlb-minlb)*rand(1);   % Set childs base length

            minll = min([parent1Config.LengthLower parent2Config.LengthLower]);
            maxll = min([parent1Config.LengthLower parent2Config.LengthLower]);
            ll = minll+(maxll-minll)*rand(1);   % Set childs distal arm length

            minlu = min([parent1Config.LengthUpper parent2Config.LengthUpper]);
            maxlu = min([parent1Config.LengthUpper parent2Config.LengthUpper]);
            lu = minlu+(maxlu-minlu)*rand(1);   % Set childs proximal arm length

            minwh = min([parent1Config.WorkspaceHeight parent2Config.WorkspaceHeight]);
            maxwh = min([parent1Config.WorkspaceHeight parent2Config.WorkspaceHeight]);
            wh = minwh+(maxwh-minwh)*rand(1);   % Set childs workspace height
…
```

Figure F.40 OptimiseConfigurationGA Function (Part 3/4)

```
…
            % MUTATION - with some probability alter childs dimension
            if rand(1) < MutationRate
                if rand(1) <0.5
                    MutAmount = MutationAmount;
                else
                    MutAmount = -MutationAmount;
                end
                lb = lb * (1+MutAmount);
            end
            if rand(1) < MutationRate
                if rand(1) <0.5
                    MutAmount = MutationAmount;
                else
                    MutAmount = -MutationAmount;
                end
                lu = lu * (1+MutAmount);
            end
            if rand(1) < MutationRate
                if rand(1) <0.5
                    MutAmount = MutationAmount;
                else
                    MutAmount = -MutationAmount;
                end
                ll = ll * (1+MutAmount);
            end
            if rand(1) < MutationRate
                if rand(1) <0.5
                    MutAmount = MutationAmount;
                else
                    MutAmount = -MutationAmount;
                end
                wh = wh * (1+MutAmount);
            end

            % Generate new configuration based on childs dimensions
            [config, reachable] = ...
                    CalculateConfig(lb,lu,ll,wh,CP.Moves,parent1motorID,UConstraints);

            % Check configuration is unique in population
            for pp=1:PopSize
                existingConfig = newPopulation(pp);
                try
                    if config.LengthBase == existingConfig.LengthBase ...
                            && config.LengthUpper == existingConfig.LengthUpper ...
                            && config.LengthLower == existingConfig.LengthLower ...
                            && config.WorkspaceHeight == existingConfig.WorkspaceHeight
                        unique = false;
                        break;
                    end
                catch exception
                end
            end

            % Add child to population if it is unique and can acheive the desired path
            if unique == true && reachable == true
                newPopulation(p) = config;
                newPopPPC(p) = parent1ppc;
                newPopMotorID(p) = parent1motorID;
            end
        end
    end
    population = newPopulation;
    popPPC = newPopPPC;
    popMotorID = newPopMotorID;
end
end
```

Figure F.41 OptimiseConfigurationGA Function (Part 4/4)

```
function OptimiseConfigurationHC(CP,TermCond,UConstraints,StepSize)
% Uses a random restart hill climber to narrow on a time-minimum configuration
% VARIABLES:
% CP - Cycle Path class containing geometric details of the path
% TermCond - Termination Condition class detailing conditions of terminating process
% UConstraints - User Constraints class
% StepSize - size of steps (in m) to evaluate neighbouring configurations

StorePathsUserConstraintsSQL(CP,UConstraints);  % Store path and user constraint data

for i=1:TermCond.Iterations     % Run Hill Climber for a number of iterations

    % Select 'random' motor details from database
    [motorID,newPPC] = SelectMotor(CP.PPC,UConstraints);

    CP.PPC = newPPC;    % Assign Path Planning Constraints (PPC) of motor to Cycle Path(CP)

    % Select random configuration that reaches all move targets
    config = SelectRandomConfig(CP.Moves,motorID,UConstraints);
    try
        % Compile path using Configuration and Path Planning Constraints (PPC)
        % Path Planning Results (ppr) are returned along with positional and zone data
        % about targets
        [Targets_XYZ,ppr] = CompilePath(CP.Moves,config,CP.PPC);
    catch exception
        % Skip to next iteration if exception occurs due to config unable to meet targets
        continue;
    end

    % Store results of path planning in database
    StoreSimulationsSQL(config,CP.PPC,ppr,CP.ID,i);
    local = false;  % Set flag indicating whether a local minima has been found
    minCycleTime = ppr.PathA(size(ppr.PathA,1)).EndTime; % Set best cycletime acheived
    bestConfig = config;

    while local == false % Loop until local minima has been found
        clear neighboursPPR;        % Clear variables
        clear neighboursConfig;     % Clear variables

        % Select configurations around the best configuration so far
        neighboursConfig = ...
            SelectNeighbouringConfig(bestConfig,CP.Moves,motorID,UConstraints,StepSize);

        for j=1:size(neighboursConfig,2)
            % Compile Paths using each of the neighbouring configurations(neighboursConfig)
            % Store results in database, and save Path Planning Results (ppr) in an array
            [Targets_XYZ,ppr] = CompilePath(CP.Moves,neighboursConfig(j),CP.PPC);
            StoreSimulationsSQL(neighboursConfig(j),CP.PPC,ppr,CP.ID,i);
            neighboursPPR(j)=ppr;
        end

        local = true;   % set flag - will be reset if not local
        for j=1:size(neighboursPPR,2)
            % Compare results of each neighbouring configuration. Replace bestConfig with
            % neighbour if faster cycletime is found
            if neighboursPPR(j).PathA(size(neighboursPPR(j).PathA,1)).EndTime ...
                                                                    < minCycleTime
                minCycleTime = ...
                        neighboursPPR(j).PathA(size(neighboursPPR(j).PathA,1)).EndTime;
                bestConfig = neighboursConfig(j);
                local = false;
            end
        end
    end
end
end
```

Figure F.42 OptimiseConfigurationHC Function

```
function OptimiseConfigurationSA(CP,TermCond,UConstraints,StepSize,MaxAttempts1,...
                                                              MaxAttempts2,T,Attenuation)
% Uses a random restart hill climber with simulated annealing to narrow on a
%  time-minimum configuration
% VARIABLES:
% CP - Cycle Path class containing geometric details of the path
% TermCond - Termination Condition class detailing conditions of terminating process
% UConstraints - User Constraints class
% StepSize - size of steps (in m) to evaluate neighbouring configurations
% MaxAttempts1 - maximum number of attempts/iterations in the inner loop of algorithm
%                   before 'cooling' takes place
% MaxAttempts2 - maximum number of attempts/iterations of the outer loop in algorithm.
%                   The number of 'cooling' steps taking place
% T - constant in algorithm that affects probability of selection
% Attenuation - the 'cooling' factor reducing the probability of selecting a less optimal
%               configuration as time goes on

% Store path and user constraint data
StorePathsUserConstraintsSQL(CP,UConstraints);

for i=1:TermCond.Iterations     % Run Simulated Annealer for a number of iterations

    % Select 'random' motor details from database
    [motorID,newPPC] = SelectMotor(CP.PPC,UConstraints);

    CP.PPC = newPPC;   % Assign Path Planning Constraints (PPC) of motor to Cycle Path(CP)

    % Select random configuration that reaches all move targets
    config = SelectRandomConfig(CP.Moves,motorID,UConstraints);

    try
        % Compile path using Configuration and Path Planning Constraints (PPC)
        % Path Planning Results (ppr) are returned along with positional and zone data
        % about targets
        [Targets_XYZ,ppr] = CompilePath(CP.Moves,config,CP.PPC);
    catch exception
        % Skip to next iteration if exception occurs due to config unable to meet targets
        continue;
    end

    % Store results of path planning in database
    StoreSimulationsSQL(config,CP.PPC,ppr,CP.ID,i);

    minCycleTime = ppr.PathA(size(ppr.PathA,1)).EndTime; % Set best cycletime acheived
    bestConfig = config;                % Set the best Configuration
    currentConfig = config;             % Set the current Configuration
    currentCycleTime = minCycleTime;    % Set cycletime acheived by currentConfig
…
```

Figure F.43 OptimiseConfigurationSA Function Part (1/2)

```
…
    attempts2 = 0;  % Reset counter
    while attempts2 < MaxAttempts2

        attempts1 = 0;  % Reset counter
        while attempts1 < MaxAttempts1
            clear neighboursPPR;            % Clear variables
            clear neighboursConfig;         % Clear variables
            clear selectedNeighbourConfig;  % Clear variables
            clear selectedNeighbourPPR;     % Clear variables

            % Select configurations around the currentConfig
            neighboursConfig = SelectNeighbouringConfig(...
                                    currentConfig,CP.Moves,motorID,UConstraints,StepSize);

            % Select a random neighbour
            randIndex = randperm(numel(neighboursConfig));
            selectedNeighbourConfig = neighboursConfig(randIndex(1,1));

            % Evaluate the selected neighbour
            [Targets_XYZ,ppr] = CompilePath(CP.Moves,selectedNeighbourConfig,CP.PPC);

            StoreSimulationsSQL(selectedNeighbourConfig,CP.PPC,ppr,CP.ID,i);
            selectedNeighbourPPR = ppr;
            selectedNeighbourCycleTime = ...
                selectedNeighbourPPR.PathA(size(selectedNeighbourPPR.PathA,1)).EndTime;

            % Check if it is the best, save if it is
            if selectedNeighbourCycleTime < minCycleTime
                minCycleTime = selectedNeighbourCycleTime;
                bestConfig = selectedNeighbourPPR;
            else

            % check if it is better than the current config/cycletime
            if selectedNeighbourCycleTime < minCycleTime
                % Replace currentConfig with neighbour
                currentConfig = selectedNeighbourConfig;
                currentCycleTime = selectedNeighbourCycleTime;
            else
                % Determine probability of selection based on cycletime and the ...
                %  'cooling' process
                probOfSelection = ...
                            1/(1+exp((selectedNeighbourCycleTime - currentCycleTime)/T));

                % Select neighbouring config based on probability
                myRand = rand(1);
                if myRand < probOfSelection
                    currentConfig = selectedNeighbourConfig;
                    currentCycleTime = selectedNeighbourPPR.PathA(...
                                            size(selectedNeighbourPPR.PathA,1)).EndTime;
                end
            end
            attempts1 = attempts1+1;
        end
        T=Attenuation*T;    % Reduce T by an amount over time ('cooling')
        attempts2 = attempts2+1;
        end
    end
end
```

Figure F.44 OptimiseConfigurationSA Function Part (2/2)

```
function OptimiseConfigurationSHC(CP,TermCond,UConstraints,StepSize,MaxAttempts,T)
% Uses a random restart stochastic hill climber to narrow on time-minimum configuration
% VARIABLES:
% CP - Cycle Path class containing geometric details of the path
% TermCond - Termination Condition class detailing conditions of terminating process
% UConstraints - User Constraints class
% StepSize - size of steps (in m) to evaluate neighbouring configurations
% MaxAttempts - the number of attempts before deciding current iteration is complete
% T - constant in algorithm that affects probability of selection

StorePathsUserConstraintsSQL(CP,UConstraints);  % Store path and user constraint data
for i=1:TermCond.Iterations     % Run Stochastic Hill Climber for a number of iterations
    % Select 'random' motor details from database
    [motorID,newPPC] = SelectMotor(CP.PPC,UConstraints);
    CP.PPC = newPPC;   % Assign Path Planning Constraints (PPC) of motor to Cycle Path(CP)
    % Select random configuration that reaches all move targets
    config = SelectRandomConfig(CP.Moves,motorID,UConstraints);
    try
        % Compile path using Configuration and Path Planning Constraints (PPC)
        % Path Planning Results (ppr) are returned along with positional and zone data
        % about targets
        [Targets_XYZ,ppr] = CompilePath(CP.Moves,config,CP.PPC);
    catch exception
        % Skip to next iteration if exception occurs due to config unable to meet targets
        continue;
    end
    % Store results of path planning in database
    StoreSimulationsSQL(config,CP.PPC,ppr,CP.ID,i);
    local = false;  % Set flag indicating whether a local minima has been found
    minCycleTime = ppr.PathA(size(ppr.PathA,1)).EndTime; % Set best cycletime acheived
    bestConfig = config;                % Set the best Configuration
    currentConfig = config;             % Set the current Configuration
    currentCycleTime = minCycleTime;    % Set cycletime acheived by currentConfig
    while attempts < MaxAttempts    % Loop for a set number of attempts
        % Select configurations around the currentConfig
        neighboursConfig = SelectNeighbouringConfig(...
                                currentConfig,CP.Moves,motorID,UConstraints,StepSize);
        % Select a random neighbour
        randIndex = randperm(numel(neighboursConfig));
        selectedNeighbourConfig = neighboursConfig(randIndex(1,1));
        % Evaluate the selected neighbour by compiling a path
        [Targets_XYZ,ppr] = CompilePath(CP.Moves,selectedNeighbourConfig,CP.PPC);
        %Store results of path planning in database
        StoreSimulationsSQL(selectedNeighbourConfig,CP.PPC,ppr,CP.ID,i);
        selectedNeighbourPPR = ppr;
        selectedNeighbourCycleTime = ...
                selectedNeighbourPPR.PathA(size(selectedNeighbourPPR.PathA,1)).EndTime;
        % Check if it is the best, save if it is
        if selectedNeighbourCycleTime < minCycleTime
            minCycleTime = selectedNeighbourCycleTime;
            bestConfig = selectedNeighbourPPR;
        end
        % Determine probability of selection based on cycle time
        diff = selectedNeighbourCycleTime - currentCycleTime;
        probOfSelection = 1/(1+exp((selectedNeighbourCycleTime - currentCycleTime)/T));
        myRand = rand(1);   % Select neighbouring config based on probability
        if myRand < probOfSelection
            currentConfig = selectedNeighbourConfig;
            currentCycleTime = selectedNeighbourPPR.PathA(...
                                        size(selectedNeighbourPPR.PathA,1)).EndTime;
        end
    end
end
end
```

Figure F.45 OptimiseConfigurationSHC Function

```
function [pathA,pathB,Targets] = PathGenerator(Targets, Config, ppc)
% Creates a path trajectory for each motor, that travels through each target. The paths
% are optimised to maximise the configuration's capabilities as defined in the Path
% Planning Constraints
% VARIABLES:
% Targets - Contains the Targets that define the path in Cartesian coordinates
% Config - The 2DOFPPM configuration
% ppc - Path Planning Constraints defining the limitations of the configuration
% RETURNS:
% pathA - Path Segments for motor A
% pathB - Path Segments for motor B
% Targets - Updated Target objects

TCPOffset_X = Config.GripperMountOffset_X;
TCPOffset_Y = Config.GripperMountOffset_Y - Config.GripperLength;

% convert knots from cartesian to joint space
numKnots = size(Targets,2);
for t=1:numKnots
   knot = Targets(t).Knot;
   [knot.Theta_A,knot.Theta_B,er,ermsg] = Inverse_2DOF_PPM(knot.X-TCPOffset_X,...
            knot.Y-TCPOffset_Y,Config.LengthBase,Config.LengthUpper,Config.LengthLower);
   if (er > 0)
        error(ermsg)
   end
   %modulate angles so that theta is between 0 and pi
   knot.Theta_A = mod(Config.ThetaAstart-knot.Theta_A,pi);
   knot.Theta_B = mod(Config.ThetaBstart-knot.Theta_B,pi);
   Targets(t).Knot = knot;
end

withinConstraints = 0;      % flag
OptimisationIterations = 0; % flag

while (withinConstraints == 0)

    %% Finding coefficients for the cubic polynomial that defines each path segment
    %% between adjacent knots

    A = zeros((numKnots-1)*4);      % Initialised matrix that contains the multiples of
                                    %  the coefficients (a0,a1,a2,a3)
    wA = zeros((numKnots-1)*4,1);   % Initialised array that contains the numerical
                                    %  'answer' to A*coefficients
    B = zeros((numKnots-1)*4);      % Initialised matrix that contains the multiples of
                                    %  the coefficients (a0,a1,a2,a3)
    wB = zeros((numKnots-1)*4,1);   % Initialised array that contains the numerical
                                    %  'answer' to A*coefficients

    row = 1;    %keeps track of row in matrices of equations, each row is a new equation
 …
```

Figure F.46 PathGenerator Function (Part 1/9)

```
…
    % equations derived from first knot
        knot1 = Targets(1).Knot;

        % start position (eq.1)
            A(row,1) = 1;
            wA(row,1) = knot1.Theta_A;
            B(row,1) = 1;
            wB(row,1) = knot1.Theta_B;
            row = row + 1;  %increment row, next equation

        % start velocity (eq.2)
            A(row,2) = 1;
            wA(row,1) = knot1.Omega_A;
            B(row,2) = 1;
            wB(row,1) = knot1.Omega_B;
            row = row + 1;  %increment row, next equation

    for t=2:numKnots-1

       target0 = Targets(t-1);
       target1 = Targets(t);
       knot0 = Targets(t-1).Knot;
       knot1 = Targets(t).Knot;
       % time between previous target and current target
       tpt = target1.PathTime - target0.PathTime;

       % position as defined by previous path segment (eq.3)
           %a0km + a1km(tpk) + a2km(tpk)^2 + a3km(tpk)^3 = knot1.ThetaA;
           A(row,(t-1)*4-3+0) = 1;
           A(row,(t-1)*4-3+1) = tpt;
           A(row,(t-1)*4-3+2) = tpt^2;
           A(row,(t-1)*4-3+3) = tpt^3;
           wA(row,1) = knot1.Theta_A;
           B(row,(t-1)*4-3+0) = 1;
           B(row,(t-1)*4-3+1) = tpt;
           B(row,(t-1)*4-3+2) = tpt^2;
           B(row,(t-1)*4-3+3) = tpt^3;
           wB(row,1) = knot1.Theta_B;
           row = row + 1;  %increment row, next equation

       % position as defined by next path segment (eq.4)
           %a0k = knot1.ThetaA;
           A(row,(t)*4-3+0) = 1;
           wA(row,1) = knot1.Theta_A;
           B(row,(t)*4-3+0) = 1;
           wB(row,1) = knot1.Theta_B;
           row = row + 1;  %increment row, next equation
…
```

Figure F.47 PathGenerator Function (Part 2/9)

```
…
        % velocity
            if (isempty(knot1.Omega_A)==false)   %Omega_A is specified (eq.2)
                 %a1k = Omega_A
                 A(row,(t)*4-3+1) = 1;
                 wA(row,1) = knot1.Omega_A;

            else     % velocity as defined by previous and next path segment (eq.5)
                %0 = a1k + 2(tpk)a2k + 3(tpk^2)a3k - a1kp
                 A(row,(t-1)*4-3+1) = 1;
                 A(row,(t-1)*4-3+2) = 2*tpt;
                 A(row,(t-1)*4-3+3) = 3*tpt^2;
                 A(row,(t)*4-3+1) = -1;
                 wA(row,1) = 0;
            end

            if (isempty(knot1.Omega_B)==false)   %Omega_A is specified (eq.2)
                 %a1k = Omega_A
                 B(row,(t)*4-3+1) = 1;
                 wB(row,1) = knot1.Omega_B;

            else     % velocity as defined by previous and next path segment (eq.5)
                %0 = a1k + 2(tpk)a2k + 3(tpk^2)a3k - a1kp
                 B(row,(t-1)*4-3+1) = 1;
                 B(row,(t-1)*4-3+2) = 2*tpt;
                 B(row,(t-1)*4-3+3) = 3*tpt^2;
                 B(row,(t)*4-3+1) = -1;
                 wB(row,1) = 0;
            end
            row = row + 1;  %increment row, next equation

        % acceleration as defined by previous and next path segment (eq.6)
             %2a2k + 6a3k(tpk) - 2a2kp = 0
             A(row,(t-1)*4-3+2) = 2;
             A(row,(t-1)*4-3+3) = 6*tpt;
             A(row,(t)*4-3+2) = -2;
             wA(row,1) = 0;
             B(row,(t-1)*4-3+2) = 2;
             B(row,(t-1)*4-3+3) = 6*tpt;
             B(row,(t)*4-3+2) = -2;
             wB(row,1) = 0;
             row = row + 1;  %increment row, next equation
    end
…
```

Figure F.48 PathGenerator Function (Part 3/9)

```
…

    % equations derived from last knot
        target0 = Targets(numKnots-1);
        target1 = Targets(numKnots);
        knot0 = target0.Knot;
        knot1 = target1.Knot;
        % time between previous target and current target
        tpt = target1.PathTime - target0.PathTime;
        t = numKnots;

        % final position (eq.7)
            %a0km + a1km(tpk) + a2km(tpk)^2 + a3km(tpk)^3 = knot1.ThetaA;
            A(row,(t-1)*4-3+0) = 1;
            A(row,(t-1)*4-3+1) = tpt;
            A(row,(t-1)*4-3+2) = tpt^2;
            A(row,(t-1)*4-3+3) = tpt^3;
            wA(row,1) = knot1.Theta_A;
            B(row,(t-1)*4-3+0) = 1;
            B(row,(t-1)*4-3+1) = tpt;
            B(row,(t-1)*4-3+2) = tpt^2;
            B(row,(t-1)*4-3+3) = tpt^3;
            wB(row,1) = knot1.Theta_B;
            row = row + 1;  %increment row, next equation

        % final velocity (eq.8)
            A(row,(t-1)*4-3+1) = 1;
            A(row,(t-1)*4-3+2) = 2*tpt;
            A(row,(t-1)*4-3+3) = 3*tpt^2;
            wA(row,1) = knot1.Omega_A;
            B(row,(t-1)*4-3+1) = 1;
            B(row,(t-1)*4-3+2) = 2*tpt;
            B(row,(t-1)*4-3+3) = 3*tpt^2;
            wB(row,1) = knot1.Omega_B;

    % coefficients of the equations (a10,a11,a12,a13,a20,a21,a22,a23,...,ak0,ak1,ak2,ak3)
    a = A\wA;
    % coefficients of the equations (b10,b11,b12,b13,b20,b21,b22,b23,...,bk0,bk1,bk2,bk3)
    b = B\wB;
…
```

Figure F.49 PathGenerator Function (Part 4/9)

```
…

    % Now we have the coefficients, we can create path segments between knots
    % using the coefficients to describe the linking polynomials

    PathA = repmat(PathSegment,numKnots-1,1);   % Initialise array, data type 'PathSegment'
    PathB = repmat(PathSegment,numKnots-1,1);   % Initialise array, data type 'PathSegment'
    maxTorqueA = zeros(numKnots-1,2);       % Initialising array to store maximum torque
                                            %  for each segment of PathA
    maxTorqueB = zeros(numKnots-1,1);       % Initialising array to store maximum torque
                                            %  for each segment of PathB
    minTorqueA = zeros(numKnots-1,2);       % Initialising array to store minimum torque
                                            %  for each segment of PathA
    minTorqueB = zeros(numKnots-1,2);       % Initialising array to store minimum torque
                                            %  for each segment of PathB
    maxOmegaA = zeros(numKnots-1,2);        % Initialising array to store maximum angular
                                            %  velocity (omega) for each segment of PathA
    maxOmegaB = zeros(numKnots-1,2);        % Initialising array to store maximum angular
                                            %  velocity (omega) for each segment of PathB
    minOmegaA = zeros(numKnots-1,2);        % Initialising array to store minimum angular
                                            %  velocity (omega) for each segment of PathA
    minOmegaB = zeros(numKnots-1,2);        % Initialising array to store minimum angular
                                            %  velocity (omega) for each segment of PathB
    maxAlphaA = zeros(numKnots-1,2);        % Initialising array to store maximum angular
                                            %  acceleration(alpha)for each segment of PathA
    maxAlphaB = zeros(numKnots-1,2);        % Initialising array to store maximum angular
                                            %  acceleration(alpha)for each segment of PathB
    minAlphaA = zeros(numKnots-1,2);        % Initialising array to store minimum angular
                                            %  acceleration(alpha)for each segment of PathA
    minAlphaB = zeros(numKnots-1,2);        % Initialising array to store minimum angular
                                            %  acceleration(alpha)for each segment of PathB
    maxJerkA = zeros(numKnots-1,2);         % Initialising array to store maximum angular
                                            %  jerk (jerk) for each segment of PathA
    maxJerkB = zeros(numKnots-1,2);         % Initialising array to store maximum angular
                                            %  jerk (jerk) for each segment of PathB
    minJerkA = zeros(numKnots-1,2);         % Initialising array to store minimum angular
                                            %  jerk (jerk) for each segment of PathA
    minJerkB = zeros(numKnots-1,2);         % Initialising array to store minimum angular
                                            %  jerk (jerk) for each segment of PathB
    TCPVelocities = zeros(numKnots-1,1);    % Initialising array to store estimated TCP
                                            %  velocities achieved during each path segment
…
```

Figure F.50 PathGenerator Function (Part 5/9)

```
…
    for t=1:numKnots-1
        StartTime = Targets(t).PathTime;
        EndTime = Targets(t+1).PathTime;
        CoefA = [a((t*4-3)+0) a((t*4-3)+1) a((t*4-3)+2) a((t*4-3)+3)];
        CoefB = [b((t*4-3)+0) b((t*4-3)+1) b((t*4-3)+2) b((t*4-3)+3)];
        psA = PathSegment(CoefA,StartTime,EndTime);
        psB = PathSegment(CoefB,StartTime,EndTime);

        %find min and max torques within each PathSegment
        [maxTorqueA(t,1),maxTorqueA(t,2)] = FindMaxTorqueA(CoefA,StartTime,EndTime,...
                                Config.MassUpper,Config.MassLower,Config.MassGripper,...
                                Config.MassLowerCrank,Config.LengthUpper);
        [minTorqueA(t,1),minTorqueA(t,2)] = FindMinTorqueA(CoefA,StartTime,EndTime,...
                                Config.MassUpper,Config.MassLower,Config.MassGripper,...
                                Config.MassLowerCrank,Config.LengthUpper);
        [maxTorqueB(t,1),maxTorqueB(t,2)] = FindMaxTorqueB(CoefB,StartTime,EndTime,...
                                Config.MassUpper,Config.MassLower,Config.MassGripper,....
                                Config.MassUpperTorsion,Config.MassLowerTorsion,...
                                Config.MassUpperCrank,Config.MassLowerCrank,...
                                Config.LengthUpper);
        [minTorqueB(t,1),minTorqueB(t,2)] = FindMinTorqueB(CoefB,StartTime,EndTime,...
                                Config.MassUpper,Config.MassLower,Config.MassGripper,...
                                Config.MassUpperTorsion,Config.MassLowerTorsion,...
                                Config.MassUpperCrank,Config.MassLowerCrank,...
                                Config.LengthUpper);

        %find min and max angular velocity within each PathSegment
        [maxOmegaA(t,1),maxOmegaA(t,2)] = FindMaxOmega(CoefA,StartTime,EndTime);
        [minOmegaA(t,1),minOmegaA(t,2)] = FindMinOmega(CoefA,StartTime,EndTime);
        [maxOmegaB(t,1),maxOmegaB(t,2)] = FindMaxOmega(CoefB,StartTime,EndTime);
        [minOmegaB(t,1),minOmegaB(t,2)] = FindMinOmega(CoefB,StartTime,EndTime);

        %find min and max angular acceleration within each PathSegment
        [maxAlphaA(t,1),maxAlphaA(t,2)] = FindMaxAlpha(CoefA,StartTime,EndTime);
        [minAlphaA(t,1),minAlphaA(t,2)] = FindMinAlpha(CoefA,StartTime,EndTime);
        [maxAlphaB(t,1),maxAlphaB(t,2)] = FindMaxAlpha(CoefB,StartTime,EndTime);
        [minAlphaB(t,1),minAlphaB(t,2)] = FindMinAlpha(CoefB,StartTime,EndTime);

        %find min and max angular velocity within each PathSegment
        [maxJerkA(t,1),maxJerkA(t,2)] = FindMaxJerk(CoefA,StartTime,EndTime);
        [minJerkA(t,1),minJerkA(t,2)] = FindMinJerk(CoefA,StartTime,EndTime);
        [maxJerkB(t,1),maxJerkB(t,2)] = FindMaxJerk(CoefB,StartTime,EndTime);
        [minJerkB(t,1),minJerkB(t,2)] = FindMinJerk(CoefB,StartTime,EndTime);

        %estimate TCP Velocities for each PathSegment
        [TCPVelocities(t,1)] = ...
                    EstimateTCPVel(Targets(t).Knot,Targets(t+1).Knot,StartTime,EndTime);

        %add path segment into path
        PathA(t)=psA;
        PathB(t)=psB;
    end
…
```

Figure F.51 PathGenerator Function (Part 6/9)

```
…

    %% Check constraints against actual values, and modify time periods as neccessary

    withinConstraints = 1; %reset flag
    for t=1:numKnots-1
        if ((Targets(t).Knot.X == Targets(t+1).Knot.X) && ...
                                                (Targets(t).Knot.Y == Targets(t+1).Knot.Y))
            %do nothing - don't change time as it is a user specified pause
        else
            %determine greatest absolute torque reached in segment
            if (abs(maxTorqueA(t,1)) > abs(minTorqueA(t,1)))
                mTorqueA = abs(maxTorqueA(t,1));
            else
                mTorqueA = abs(minTorqueA(t,1));
            end
            if (abs(maxTorqueB(t,1)) > abs(minTorqueB(t,1)))
                mTorqueB = abs(maxTorqueB(t,1));
            else
                mTorqueB = abs(minTorqueB(t,1));
            end
            if (mTorqueA > mTorqueB)
                mTorque = mTorqueA;
            else
                mTorque = mTorqueB;
            end

            %determine greatest absolute angular velocity (Omega) reached in segment
            if (abs(maxOmegaA(t,1)) > abs(minOmegaA(t,1)))
                mOmegaA = abs(maxOmegaA(t,1));
            else
                mOmegaA = abs(minOmegaA(t,1));
            end
            if (abs(maxOmegaB(t,1)) > abs(minOmegaB(t,1)))
                mOmegaB = abs(maxOmegaB(t,1));
            else
                mOmegaB = abs(minOmegaB(t,1));
            end
            if (mOmegaA > mOmegaB)
                mOmega = mOmegaA;
            else
                mOmega = mOmegaB;
            end

            %determine greatest absolute angular acceleration (Alpha) reached in segment
            if (abs(maxAlphaA(t,1)) > abs(minAlphaA(t,1)))
                mAlphaA = abs(maxAlphaA(t,1));
            else
                mAlphaA = abs(minAlphaA(t,1));
            end
            if (abs(maxAlphaB(t,1)) > abs(minAlphaB(t,1)))
                mAlphaB = abs(maxAlphaB(t,1));
            else
                mAlphaB = abs(minAlphaB(t,1));
            end
            if (mAlphaA > mAlphaB)
                mAlpha = mAlphaA;
            else
                mAlpha = mAlphaB;
            end
…
```

Figure F.52 PathGenerator Function (Part 7/9)

```
…
            %determine greatest absolute angular jerk (Jerk) reached in segment
            if (abs(maxJerkA(t,1)) > abs(minJerkA(t,1)))
                mJerkA = abs(maxJerkA(t,1));
            else
                mJerkA = abs(minJerkA(t,1));
            end
            if (abs(maxJerkB(t,1)) > abs(minJerkB(t,1)))
                mJerkB = abs(maxJerkB(t,1));
            else
                mJerkB = abs(minJerkB(t,1));
            end
            if (mJerkA > mJerkB)
                mJerk = mJerkA;
            else
                mJerk = mJerkB;
            end
            % extract max TCP velocity for PathSegment
            mTCPVel = TCPVelocities(t,1);
            TCPVel_Max = Targets(t+1).VelocityLimit;
            %Calculate Scaling Factor for shortening time segment if need be
            shorteningFactor = ppc.InitialAcceptanceThreshold^ ...
             ((OptimisationIterations+ppc.RelativeAgeingFactor)/ppc.RelativeAgeingFactor);
            if ((mTorque>ppc.MaxTorque) || (mOmega>ppc.MaxOmega) || ...
                   (mAlpha>ppc.MaxAlpha) || (mJerk>ppc.MaxJerk) || (mTCPVel>TCPVel_Max))
                % Then need to extend path time based on either torque, omega, alpha,
                %  jerk or TCP velocity
                withinConstraints = 0; %set flag
                target1 = Targets(t);
                target2 = Targets(t+1);
                %determine which ratio to use
                if (mTorque/ppc.MaxTorque > mOmega/ppc.MaxOmega) && ...
                                    (mTorque/ppc.MaxTorque > mAlpha/ppc.MaxAlpha) && ...
                                    (mTorque/ppc.MaxTorque > mJerk/ppc.MaxJerk) && ...
                                    (mTorque/ppc.MaxTorque > mTCPVel/TCPVel_Max)
                    ratio = mTorque/ppc.MaxTorque;
                elseif (mOmega/ppc.MaxOmega > mTorque/ppc.MaxTorque) && ...
                                    (mOmega/ppc.MaxOmega > mAlpha/ppc.MaxAlpha) && ...
                                    (mOmega/ppc.MaxOmega > mJerk/ppc.MaxJerk) && ...
                                    (mOmega/ppc.MaxOmega > mTCPVel/TCPVel_Max)
                    ratio = mOmega/ppc.MaxOmega;
                elseif (mAlpha/ppc.MaxAlpha > mTorque/ppc.MaxTorque) && ...
                                    (mAlpha/ppc.MaxAlpha > mOmega/ppc.MaxOmega) && ...
                                    (mAlpha/ppc.MaxAlpha > mJerk/ppc.MaxJerk) && ...
                                    (mAlpha/ppc.MaxAlpha > mTCPVel/TCPVel_Max)
                    ratio = mAlpha/ppc.MaxAlpha;
                elseif (mJerk/ppc.MaxJerk > mTorque/ppc.MaxTorque) && ...
                                    (mJerk/ppc.MaxJerk > mOmega/ppc.MaxOmega) && ...
                                    (mJerk/ppc.MaxJerk > mAlpha/ppc.MaxAlpha) && ...
                                    (mJerk/ppc.MaxJerk > mTCPVel/TCPVel_Max)
                    ratio = mJerk/ppc.MaxJerk;
                else
                    ratio = mTCPVel/TCPVel_Max;
                end

                % Increase PathTime on next knot by a factor relative to the difference in
                % either torques or omegas (depending on which ever is greatest). Also
                % increase all following knots by the same length.
                for i=t+1:numKnots
                    target_i = Targets(i);
                    target_i.PathTime = target_i.PathTime + ...
                        (target2.PathTime-target1.PathTime) * ratio * ppc.ReactiveFactor;

                    Targets(i) = target_i; %re-insert knot back into collection of knots
                end
…
```

Figure F.53 PathGenerator Function (Part 8/9)

```
…
            elseif ((mTorque<ppc.MaxTorque*shorteningFactor) && ...
                                        (mOmega<ppc.MaxOmega*shorteningFactor) && ...
                                        (mAlpha<ppc.MaxAlpha*shorteningFactor) && ...
                                        (mJerk<ppc.MaxJerk*shorteningFactor) && ...
                                        (mTCPVel<TCPVel_Max*shorteningFactor))
                % This means that the torque, angular velocity, angular acceleration,
                % angular jerk and TCP velocity are outside a percentage of the maximum
                % for both motors. Therefore the cycle can be shortened to get more
                % performance from the mechanism.

                withinConstraints = 0; %set flag
                target1 = Targets(t);
                target2 = Targets(t+1);

                if (mTorque/ppc.MaxTorque > mOmega/ppc.MaxOmega) && ...
                                    (mTorque/ppc.MaxTorque > mAlpha/ppc.MaxAlpha) && ...
                                    (mTorque/ppc.MaxTorque > mJerk/ppc.MaxJerk) && ...
                                    (mTorque/ppc.MaxTorque > mTCPVel/TCPVel_Max)
                    ratio = 1 - mTorque/ppc.MaxTorque;
                elseif (mOmega/ppc.MaxOmega > mTorque/ppc.MaxTorque) && ...
                                    (mOmega/ppc.MaxOmega > mAlpha/ppc.MaxAlpha) && ...
                                    (mOmega/ppc.MaxOmega > mJerk/ppc.MaxJerk) && ...
                                    (mOmega/ppc.MaxOmega > mTCPVel/TCPVel_Max)
                    ratio = 1 - mOmega/ppc.MaxOmega;
                elseif (mAlpha/ppc.MaxAlpha > mTorque/ppc.MaxTorque) && ...
                                    (mAlpha/ppc.MaxAlpha > mOmega/ppc.MaxOmega) && ...
                                    (mAlpha/ppc.MaxAlpha > mJerk/ppc.MaxJerk) && ...
                                    (mAlpha/ppc.MaxAlpha > mTCPVel/TCPVel_Max)
                    ratio = 1 - mAlpha/ppc.MaxAlpha;
                elseif (mJerk/ppc.MaxJerk > mTorque/ppc.MaxTorque) && ...
                                    (mJerk/ppc.MaxJerk > mOmega/ppc.MaxOmega) && ...
                                    (mJerk/ppc.MaxJerk > mAlpha/ppc.MaxAlpha) && ...
                                    (mJerk/ppc.MaxJerk > mTCPVel/TCPVel_Max)
                    ratio = 1 - mJerk/ppc.MaxJerk;
                else
                    ratio = 1 - mTCPVel/TCPVel_Max;
                end

                % Decrease PathTime on next knot by a factor relative to the difference in
                % either torques or omegas (depending on which ever is smallest). Also
                % decrease all following knots by the same length.
                for i=t+1:numKnots
                    target_i = Targets(i);
                    target_i.PathTime = target_i.PathTime - ...
                        (target2.PathTime-target1.PathTime) * ratio * ppc.ReactiveFactor;

                    Targets(i) = target_i; %re-insert knot back into collection of knots
                end
            end
        end
    end
    % From here the process is encapsulated in a loop(from line 26) to increase
    %  Knot.PathTimes where appropriate until absolute values of max/min torques are
    %  within the constraints

    OptimisationIterations = OptimisationIterations+1;
end

pathA = PathA;
pathB = PathB;
end
```

Figure F.54 PathGenerator Function (Part 9/9)

```
classdef PathSegment
% Defines a segment of a path within a given time period.
% The position on the path segment is defined by a polynomial function of time.

properties
Coef            % Array of polynomial coefficients.
StartTime       % The start time for the segment.
EndTime         % The end time for the segment.
end

methods
    % Create instance of PathSegment class with variables
    function ps = PathSegment(Coef,StartTime,EndTime)
        if nargin > 0
            ps.Coef = Coef;
            ps.StartTime = StartTime;
            ps.EndTime = EndTime;
        end
    end

    % Returns the angle position at the requested time within the time segment
    function theta = getTheta(obj,time)
        % check time requested is within time defined by this PathSegment
        if ((time >= obj.StartTime)&&(time<=obj.EndTime+1e-10))
            j = 0;                  % represents the order of the polynomial coefficient
            t = time - obj.StartTime; % time since start of this segment
            thetaSum = 0;
            for i=1:length(obj.Coef)
                thetaSum = thetaSum + obj.Coef(i)*t^j;
                j=j+1;
            end
            theta = thetaSum;
        else
            disp(['Time requested (',num2str(time),...
                ') is outside this path segments definable range:',...
                num2str(obj.StartTime),'>=','time','<=',num2str(obj.EndTime)])
            error('Time requested is outside this path segments definable range')
        end
    end

    % Returns the angular velocity at the requested time within the time segment
    function omega = getOmega(obj,time)
        % check time requested is within time defined by this PathSegment
        if ((time >= obj.StartTime)&&(time<=obj.EndTime+1e-10))
            j = 0;                  % represents the order of the polynomial coefficient
            t = time - obj.StartTime; % time since start of this segment
            omegaSum = 0;
            for i=1:length(obj.Coef)
                omegaSum = omegaSum + j*obj.Coef(i)*t^(j-1);    % 1st order derivative
                j=j+1;
            end
            omega = omegaSum;

        else
            disp(['Time requested (',num2str(time),...
                ') is outside this path segments definable range:',...
                num2str(obj.StartTime),'>=','time','<=',num2str(obj.EndTime)])
            error('Time requested is outside this path segments definable range')
        end
    end
…
```

Figure F.55 PathSegment Class (Part 1/2)

```
…

% Returns the angular acceleration at the requested time within the time segment
    function alpha = getAlpha(obj,time)
        if ((time >= obj.StartTime)&&(time<=obj.EndTime+1e-10))
            j = 0;                    % represents the order of the polynomial coefficient
            t = time - obj.StartTime; % time since start of this segment
            alphaSum = 0;
            for i=1:length(obj.Coef)
                alphaSum = alphaSum + (j-1)*j*obj.Coef(i)*t^(j-2);  % 2nd order derivative
                j=j+1;
            end
            alpha = alphaSum;

        else
            disp(['Time requested (',num2str(time),...
                ') is outside this path segments definable range:',...
                num2str(obj.StartTime),'>=','time','<=',num2str(obj.EndTime)])
            error('Time requested is outside this path segments definable range')
        end
    end
end
end
```

Figure F.56 PathSegment Class (Part 2/2)

```
function [fig,h] = Plot_Knots_TCP(FigID)
% Plots the Knots and the Trajectory followed by the TCP in the SimMechanics simulation

fig = figure(FigID);    % Create new Figure
% Load Targets and Knots
load PG_Outputs\Targets_XYZ.mat Targets_XYZ
load PG_Outputs\Knots_TXY.mat Knots_TXY

%% Plot Target Points
h = scatter(Targets_XYZ(:,1),Targets_XYZ(:,2),'MarkerEdgeColor',[1 0 0],...
                                                'MarkerFaceColor',[1,0.7,0.7]);
hold on;

% Plot Centre of Target points
scatter(Targets_XYZ(:,1),Targets_XYZ(:,2),'Marker','+','MarkerEdgeColor',[1 0 0],...
                                                'SizeData',10^2,'LineWidth',2);
% Obtain the axes size (in axpos) in Points
currentunits = get(gca,'Units');
set(gca, 'Units', 'Points');
axpos = get(gca,'Position');
set(gca, 'Units', currentunits);

%% Plot Knots
%customise colours
tKnots_TXY=Knots_TXY';
numKnots = size(tKnots_TXY,2);
tf = tKnots_TXY(1,numKnots);
knotColours = zeros(numKnots,3);
for i=1:numKnots
    time = tKnots_TXY(1,i);
   knotColours(i,1) = 0.7-0.7*time/tf;
   knotColours(i,2) = 0.7-0.7*time/tf;
   knotColours(i,3) = 1;
end

scatter(Knots_TXY(:,2),Knots_TXY(:,3),'Marker','o','CData',knotColours,...
                                                'SizeData',15^2,'LineWidth',5)
%% Plot TCP Trajectory Followed

load Mdl_Outputs\TCP_XY.mat TCP_TXY     % Load TCP Path from SimMechanics
% customise colours
numPpoints = size(TCP_TXY,2);
tf = TCP_TXY(1,numPpoints);
pathColours = zeros(numPpoints,3);
for i=1:numPpoints
    time = TCP_TXY(1,i);
   pathColours(i,1) = 1;
   pathColours(i,2) = 1;
   pathColours(i,3) = 0.7-0.7*time/tf;
end

scatter(TCP_TXY(2,:),TCP_TXY(3,:),'Marker','x','CData',pathColours,'LineWidth',1.5,...
                                                'SizeData',10^2)
hold off
grid on;
set(gca,'DataAspectRatio',[1 1 1])

%% Extra Graphical Manipulation
%Scale target points to their actual size with respect to the axis
%Zones are defined in mm, so scale by 1000, but divide by 2 as it only defines the radius
scalingRatio = (1000/2);
markerWidth = (Targets_XYZ(:,3)/scalingRatio)/diff(xlim)*axpos(3); %Calculate Marker width
set(h, 'SizeData', markerWidth.^2)

end
```

Figure F.57 Plot_Knots_TCP Function

```
function Plot_Sim_Outputs(AngularUnits,SimID,config)
% Plots the output results of a Simulation
% VARIABLES:
% Angular Units - either 'Rad' or 'Deg'
% SimID - Simulation ID
% config - Instance of a Configuration class

% setup motor angle units
if (strcmp(AngularUnits,'rad'))
    scaleFactor = 1;
    unitLabel = 'Rad';
elseif (strcmp(AngularUnits,'deg'))
    scaleFactor = 180/pi;
    unitLabel = 'Deg';
else
    error(['AngularUnits must be either "rad" or "deg". "',AngularUnits,...
                                                        '" is not permissible']);
end

% Load Simulation output files
load Mdl_Outputs\M_Torque.mat M_Torque
load Mdl_Outputs\M_PVA.mat M_PVA
load Mdl_Outputs\TCP_PVA.mat TCP_PVA
load PG_Outputs\Knots_TXY.mat Knots_TXY
load PG_Outputs\Targets_XYZ.mat Targets_XYZ
load Mdl_Outputs\TorqueCalcs.mat TorqueCalcs

groupName = ['SimID: ',num2str(SimID),' - 2DOF_PPM Simulation Outputs'];

group = setfigdocked('GroupName',groupName,'GridSize',[3 3],'Maximize',1,...
                                            'GroupDocked',0,'SpanCell',[1 2 2 1]);

%% Plot Knots and TCP

% Call Plot_Knots_TCP function to create the scatter plot of the knots etc, passing back
% handles to the figure and ScatterGroup. A resize function has been added so that the
% 'zones' are resized to be in scale with the axis
[fig,scatterHandle] = Plot_Knots_TCP(SimID*100+1);
group = setfigdocked('GroupName',groupName,'Figure',gcf,'Figindex',2);
set(fig,'ResizeFcn',{@f_PlotKnotsTCP,Targets_XYZ,fig,scatterHandle});

%% Plot Motor Torques
figure(SimID*100+2)

plot(M_Torque(1,:),M_Torque(3,:),'-k',M_Torque(1,:),M_Torque(2,:),'-r',...
           TorqueCalcs(1,:),TorqueCalcs(2,:),':k',TorqueCalcs(1,:),TorqueCalcs(4,:),':r')
title('Motor Torques');
xlabel('Time (s)');
ylabel('Torque (Nm)');

grid on;
h_legend = legend('Motor A','Motor B','Motor A (est.)','Motor B (est.)',...
                                                    'Location','NorthEast');
set(h_legend,'FontSize',8);
group = setfigdocked('GroupName',groupName,'Figure',gcf,'Figindex',7);
…
```

Figure F.58 Plot_Sim_Outputs Function (Part 1/3)

```
…

%% Plot Motor PVAs

figure(SimID*100+3)
plot(M_PVA(1,:),M_PVA(2,:)*scaleFactor,'-m',M_PVA(1,:),M_PVA(3,:)*scaleFactor,'-c')
title('Motor Positions');
xlabel('Time (s)');
ylabel(['Position (',unitLabel,')']);
grid on;
h_legend = legend('Motor A','Motor B','Location','NorthEast');
set(h_legend,'FontSize',8);
group = setfigdocked('GroupName',groupName,'Figure',gcf,'Figindex',1);

figure(SimID*100+4)
plot(M_PVA(1,:),M_PVA(4,:)*scaleFactor,'-m',M_PVA(1,:),M_PVA(5,:)*scaleFactor,'-c')
title('Motor Angular Velocity');
xlabel('Time (s)');
ylabel(['Angular Velocity (',unitLabel,' s^-^1)']);
grid on;
h_legend = legend('Motor A','Motor B','Location','NorthEast');
set(h_legend,'FontSize',8);
group = setfigdocked('GroupName',groupName,'Figure',gcf,'Figindex',4);

figure(SimID*100+5)
plot(M_PVA(1,:),M_PVA(6,:)*scaleFactor,'-m',M_PVA(1,:),M_PVA(7,:)*scaleFactor,'-c')
title('Motor Angular Acceleration');
xlabel('Time (s)');
ylabel(['Angular Acceleration (',unitLabel,' s^-^2)']);
grid on;
h_legend = legend('Motor A','Motor B','Location','NorthEast');
set(h_legend,'FontSize',8);
group = setfigdocked('GroupName',groupName,'Figure',gcf,'Figindex',6);
…
```

Figure F.59 Plot_Sim_Outputs Function (Part 2/3)

```
…

%% Plot TCP PVA

figure(SimID*100+6)
plot(TCP_PVA(1,:),TCP_PVA(2,:),'-b',TCP_PVA(1,:),TCP_PVA(3,:),'-g')
title('TCP Position');
xlabel('Time (s)');
ylabel('Position (m)');
grid on;
h_legend = legend('X','Y','Location','NorthEast');
set(h_legend,'FontSize',8);
group = setfigdocked('GroupName',groupName,'Figure',gcf,'Figindex',3);

figure(SimID*100+7)
plot(TCP_PVA(1,:),TCP_PVA(5,:),'-b',TCP_PVA(1,:),TCP_PVA(6,:),'-g',TCP_PVA(1,:),...
                                        sqrt((TCP_PVA(5,:)).^2+(TCP_PVA(6,:)).^2),'--m')
title('TCP Velocity');
xlabel('Time (s)');
ylabel('Velocity (m s^-^1)');
grid on;
h_legend = legend('X','Y','\surd(X^2 + Y^2)','Location','NorthEast');
set(h_legend,'FontSize',8);
group = setfigdocked('GroupName',groupName,'Figure',gcf,'Figindex',5);

figure(SimID*100+8)
plot(TCP_PVA(1,:),TCP_PVA(8,:),'-b',TCP_PVA(1,:),TCP_PVA(9,:),'-g',TCP_PVA(1,:),...
                                        sqrt((TCP_PVA(8,:)).^2+(TCP_PVA(9,:)).^2),'--m')
title('TCP Acceleration');
xlabel('Time (s)');
ylabel('Acceleration (m s^-^2)');
grid on;
h_legend = legend('X','Y','\surd(X^2 + Y^2)','Location','NorthEast');
set(h_legend,'FontSize',8);

group = setfigdocked('GroupName',groupName,'Figure',gcf,'Figindex',8);

end

function XData2 = ScaleLegendLine(ch_legend1,ch_legend2)
    XData1 =  get(ch_legend1, 'XData');
    XData2 =  get(ch_legend2, 'XData');
    XScale = XData2 - XData1;
    XData2 = XData1 + XScale / 2;
end
```

Figure F.60 Plot_Sim_Outputs Function (Part 3/3)

```
PathID = 123;       % Enter PathID for search surface results

% Open Database connection
ch = mysql('open','localhost:3306','root','mysql');
db = mysql('use matlab_2dofppm');

query = ['SELECT '...
            's.MotorSeparation, '...
            's.ProxArmLength, '...
            's.DistArmLength, '...
            's.WorkspaceHeight, '...
            's.CycleTime '...
            'FROM Simulations AS s '...
            'WHERE '...
            'PathID = "',num2str(PathID),'" '...
            'ORDER BY s.CycleTime ASC '
            ];

    [
        b ...
        u ...
        l ...
        h ...
        t ...
    ] = mysql(query);

mysql('close');

tcolor = t;
hh = ceil((h-(min(h)*1.2))*400);

figure(PathID);

xlabel('Motor Separation (m)','FontWeight','bold','Color','w')
ylabel('Proximal (Upper) Arm (m)','FontWeight','bold','Color','w')
zlabel('Distal (Lower) Arm (m)','FontWeight','bold','Color','w')
set(gca, 'XColor', [0 0 0]);
set(gca, 'YColor', [0 0 0]);
set(gca, 'ZColor', [0 0 0]);
set(gca, 'Color', [1 1 1]);

% Plot search surface
s3 = scatter3(b,u,l,80,tcolor,'filled');
xlabel('Motor Separation (m)','FontWeight','bold','Color','k')
ylabel('Proximal (Upper) Arm (m)','FontWeight','bold','Color','k')
zlabel('Distal (Lower) Arm (m)','FontWeight','bold','Color','k')

set(gca, 'XColor', [0 0 0]);
set(gca, 'YColor', [0 0 0]);
set(gca, 'ZColor', [0 0 0]);
set(gca, 'Color', [1 1 1]);

map1 = jet(256);
map2 = map1(end:-1:1, :);
colormap(map2)
set(gca, 'CLim', [t(1,1), t(end,1)]);
cb = colorbar('YColor','k');
set(get(cb,'ylabel'),'String', 'Cycle-Time (s)','FontWeight','bold','Color','k');
line([-1000 1000],[u(1,1) u(1,1)],[l(1,1) l(1,1)],'Color',[1 0 1],'LineWidth',2)
line([b(1,1) b(1,1)],[-1000 1000],[l(1,1) l(1,1)],'Color',[1 0 1],'LineWidth',2)
line([b(1,1) b(1,1)],[u(1,1) u(1,1)],[-1000 1000],'Color',[1 0 1],'LineWidth',2)
XLim([min(b) max(b)])
YLim([min(u) max(u)])
ZLim([min(l) max(l)])
```

Figure F.61 Plot Search Surface Script

```
classdef PPConstraints
% Contains constraints on the path planning

properties
    MaxTorque                   % Maximum torque permissible (Nm)
    MaxOmega                    % Maximum angular velocity provided by the motors (rad)
    MaxAlpha                    % Maximum angular acceleration provided by motors(rad/s)
    MaxJerk                     % Maximum angular jerk provided by the motors (rad/s/s)
    LinearErrorFactor           % A factor to indicate number of interpolated points
                                %  during a MoveL. The number corresponds to the minimum
                                %  spacing of interpolated points in meters. Smaller the
                                %  number the more accurate it will be, but also more
                                %  computationally expensive.
    LastLinearTargetDistance    % Distance back from last target in linear move, where an
                                %  additional target is placed to ensure greater linearity
                                %  while minimising excessive targets.
    ReactiveFactor              % The proportion of the relative maximums of torque and
                                %  omega, that a path segments time is increased by.
    InitialAcceptanceThreshold  % The percentage that the torque or omega must be within
                                %  of the maximums to be considered optimal
    RelativeAgeingFactor        % A factor used in the process of discounting the
                                %  InitialAcceptanceThreshold as the number of iterations
                                %  increases.
end

end
```

Figure F.62 PPConstraints Class

```
classdef PPResults
% Contains results from path planning

properties
    PathA       % PathSegments for motor A
    PathB       % PathSegments for motor B
    Knots       % Knots(X,Y) against time
end

end
```

Figure F.63 PPResults Class

```
% Produces the reachable workspace of the 2DOFPPM with only the basic physical dimensions
% needed

LengthBase = 0.3;               %Length of the base (m)
LengthUpper = 0.36;             %Length of each upper arm (m)
LengthLower = 0.88;             %Length of each lower arm (m)
minUpperArmAngle = d2r(33);     %Minimum angle allowed between upper arm and vertical
maxUpperArmAngle = d2r(175);    %Minimum angle allowed between upper arm and vertical
min1_2ArmAngle = d2r(43);       %Minimum angle allowed between upper-lower arms
max1_2ArmAngle = d2r(134);      %Maximum angle allowed between upper-lower arms
minLowerArmAngle = d2r(48);     %Minimum angle allowed between lower-lower arms
maxLowerArmAngle = d2r(71);     %Maximum angle allowed between lower-lower arms
thetaA = minUpperArmAngle;      %Angle between +Y axis and right upper arm
thetaB = maxUpperArmAngle;      %Angle between +Y axis and left upper arm
stepSize = 5;                   %Step size of motors, in degrees, for evaluating workspace
figure;
grid on;
firstValidThetaB = true;
while thetaA < maxUpperArmAngle
    % Perform direct kinematics to get TCP from motor angles
    [ tcpX, tcpY, error, errorMsg ] ...
        = Direct_2DOF_PPM( thetaA, thetaB, LengthBase,LengthUpper,LengthLower,...
            min1_2ArmAngle,max1_2ArmAngle,minLowerArmAngle,maxLowerArmAngle );

    if (error > 0)  % TCP is at the edge of the reachable workspace
        if (thetaB < minUpperArmAngle)
            % Plot point as reachable
            line([lastValidTcpX-0.01 lastValidTcpX+0.01],[lastValidTcpY lastValidTcpY],...
                'Color','b','LineWidth',3);
        end
        firstValidThetaB = true;
    else
        % Plot point as reachable
        if (firstValidThetaB == true)
            line([tcpX-0.01 tcpX+0.01],[tcpY tcpY],'Color','b','LineWidth',3);
            firstValidThetaB = false;
        end
        if (thetaB < minUpperArmAngle)
            line([tcpX-0.01 tcpX+0.01],[tcpY tcpY],'Color','b','LineWidth',3);
        end
        lastValidTcpX = tcpX;
        lastValidTcpY = tcpY;
    end

    if (thetaB < minUpperArmAngle)
        thetaA = thetaA + d2r(stepSize);    % Increment thetaA by the step size
        thetaB = maxUpperArmAngle;          % Reset thetaB for another sweep
    else
        thetaB = thetaB - d2r(stepSize);    % Decrement thetaB by the step size
    end
end
```

Figure F.64 Produce Reachable Workspace Script

```
function ProduceSearchSurface(cp,UConstraints)
% Runs Simulations evenly over the search space
% VARIABLES:
% cp - Instance of CyclePath class
% Uconstraints - Instant of UserConstraints class

% Store path and user constraint data
StorePathsUserConstraintsSQL(cp,UConstraints);

[motorID,newPPC] = SelectMotor(cp.PPC,UConstraints);    % Select Motor details from DB
cp.PPC = newPPC;

% Find limits on search area based on moves
minBase = UConstraints.MinMotorSeparation;
maxBase = 0.9*UConstraints.MaxWidth;  %allow up to 90% base + 2*5% ProxArms
minUpper = 0;
maxUpper = UConstraints.MaxWidth/2;
minLower = minBase;
maxLower = sqrt(UConstraints.MaxDepth^2 + (maxBase/2)^2);
minWSHeight = -UConstraints.MaxDepth*0.3;
maxWSHeight = UConstraints.MaxDepth*0.3;

numIntervals = 20;
params = zeros(3,0);
bb = 1;
uu = 1;
ll = 1;
hh = 1;

for b=minBase:(maxBase-minBase)/numIntervals:maxBase
    uu = 1;
    for u=minUpper:(maxUpper-minUpper)/numIntervals:maxUpper
        ll = 1;
        for l=minLower:(maxLower-minLower)/numIntervals:maxLower
            hh = 1;
            for h=minWSHeight:(maxWSHeight-minWSHeight)/10:maxWSHeight
                mvar = Configuration;
                mvar.LengthBase = b;
                mvar.LengthUpper = u;
                mvar.LengthLower = l;
                mvar.WorkspaceHeight = h;
                params(end+1,1) = b;
                params(end,2) = u;
                params(end,3) = l;
                [bb uu ll hh];
                hh=hh+1;
                try
                    mvar = SelectMVar2(cp.Moves,motorID,UConstraints,mvar);
                catch
                    continue;
                end
                try
                    [Targets_XYZ,ppr] = CompilePath(cp.Moves,mvar,cp.PPC);
                catch exception
                    continue;
                end
                StoreSimulationsSQL(mvar,cp.PPC,ppr,cp.ID,comment);
            end
            ll=ll+1;
        end
        uu=uu+1;
    end
    bb=bb+1;
end
end
```

Figure F.65 ProduceSearchSurface Function

```
% Create new instance of UserConstraints class and set variables
uc = UserConstraints;
uc.MaxMotorTorque = 300;
uc.MaxMotorVelocity = 20;
uc.MaxMotorAcceleration = 9999;
uc.MaxMotorJerk = 999999;
uc.MassGripper = 35;
uc.MinArmAng_BU = d2r(33);
uc.MinArmAng_UL = d2r(43);
uc.MinArmAng_LL = d2r(48);
uc.MaxArmAng_BU = d2r(175);
uc.MaxArmAng_UL = d2r(134);
uc.MaxArmAng_LL = d2r(71);
uc.ProxArmDensity = 2700;
uc.DistArmDensity = 2700;
uc.TorsionArmDensity = 2700;
uc.ProxArmIRadius = 0.01;
uc.DistArmIRadius = 0.01;
uc.ProxArmORadius = 0.02;
uc.DistArmORadius = 0.02;
uc.TorsionIRadius = 0.005;
uc.TorsionORadius = 0.01;
uc.MassUpperCrank = 0.2;
uc.MassLowerCrank = 0.2;
uc.UpperTorsionOffsetB_X = 0.05;
uc.UpperTorsionOffsetB_Y = 0.1;
uc.LowerTorsionOffsetTCP_X = -0.05;
uc.LowerTorsionOffsetTCP_Y = 0.1;
uc.GripperMountOffset_X = 0;
uc.GripperMountOffset_Y = -0.02;
uc.GripperLength = 0.01;
uc.MinMotorSeparation = 0.01;
uc.MaxWidth = 1.5;
uc.MaxDepth = 2;

% Specify knots for Path
k1 = Knot(-0.3,-1);
k2 = Knot(-0.3,-0.7);
k3 = Knot(0,-0.65);
k4 = Knot(0.3,-0.7);
k5 = Knot(0.3,-1);
k6 = Knot(0.3,-0.75);
k7 = Knot(0,-0.7);
k8 = Knot(-0.3,-0.75);
k9 = Knot(-0.3,-1);

% Create Move commands from Knots
m1 = MoveCMD(Target(k1),'MoveJ',10,1);
m2 = MoveCMD(Target(k2),'MoveL',10,30);
m3 = MoveCMD(Target(k3),'MoveJ',10,50);
m4 = MoveCMD(Target(k4),'MoveJ',10,30);
m5 = MoveCMD(Target(k5),'MoveL',10,1,0.2);
m6 = MoveCMD(Target(k6),'MoveL',10,20);
m7 = MoveCMD(Target(k7),'MoveJ',10,30);
m8 = MoveCMD(Target(k8),'MoveJ',10,20);
m9 = MoveCMD(Target(k9),'MoveL',10,1);

% Create new CyclePath from Moves
cp = CyclePath;
cp.ID = GetNextPathID();
cp.Moves = [m1 m2 m3 m4 m5 m6 m7 m8 m9];
cp.PPC = GetPPConstraints();

% Produce the search surface
ProduceSearchSurface(cp,uc);
```

Figure F.66 ProduceSearchSurface Start Script

```
function d = r2d(r)
%#eml
    d = r * 180/pi;     % Converts radians to degrees
end
```

Figure F.67 r2d (Radians to Degrees) Function

```
function config = RunSimulation(SimID)
% Runs a SimMechanics simulation of a previously generated configuration result.
% VARIABLES:
% SimID - Simulation ID for database

    % Open Database connection
    ch = mysql('open','localhost:3306','root','mysql');
    db = mysql('use matlab_2dofppm');

    query = ['SELECT '...
            's.ProxArmLength, '...
            's.DistArmLength, '...
            's.MotorSeparation, '...
            's.WorkspaceHeight, '...
            's.MotorID, '...
            's.PathID, '...
            'uc.MaxMotorTorque, '...
            'uc.MaxMotorVelocity, '...
            'uc.MassGripper, '...
            'uc.MinArmAng_BU, '...
            'uc.MinArmAng_UL, '...
            'uc.MinArmAng_LL, '...
            'uc.MaxArmAng_BU, '...
            'uc.MaxArmAng_UL, '...
            'uc.MaxArmAng_LL, '...
            'uc.ProxArmDensity, '...
            'uc.DistArmDensity, '...
            'uc.TorsionArmDensity, '...
            'uc.ProxArmIRadius, '...
            'uc.DistArmIRadius, '...
            'uc.ProxArmORadius, '...
            'uc.DistArmORadius, '...
            'uc.TorsionIRadius, '...
            'uc.TorsionORadius, '...
            'uc.MassUpperCrank, '...
            'uc.MassLowerCrank, '...
            'uc.UpperTorsionOffsetB_X, '...
            'uc.UpperTorsionOffsetB_Y, '...
            'uc.LowerTorsionOffsetTCP_X, '...
            'uc.LowerTorsionOffsetTCP_Y, '...
            'uc.GripperMountOffset_X, '...
            'uc.GripperMountOffset_Y, '...
            'uc.GripperLength, '...
            'uc.MinMotorSeparation, '...
            'uc.MaxWidth, '...
            'uc.MaxDepth '...
            'FROM Simulations AS s '...
            'JOIN paths AS p ON p.pathid = s.pathid '...
            'JOIN userconstraints AS uc ON uc.pathid = p.pathid '...
            'WHERE '...
            'SimID = "',num2str(SimID),'" '...
            ];

    uc = UserConstraints;
…
```

Figure F.68 RunSimulation Function (Part 1/3)

```
…
    [
         ProxArmLength ...
         DistArmLength ...
         MotorSeparation ...
         WorkspaceHeight ...
         MotorID ...
         PathID ...
         uc.MaxMotorTorque  ...
         uc.MaxMotorVelocity  ...
         uc.MassGripper ...
         uc.MinArmAng_BU ...
         uc.MinArmAng_UL ...
         uc.MinArmAng_LL ...
         uc.MaxArmAng_BU ...
         uc.MaxArmAng_UL ...
         uc.MaxArmAng_LL ...
         uc.ProxArmDensity ...
         uc.DistArmDensity ...
         uc.TorsionArmDensity ...
         uc.ProxArmIRadius ...
         uc.DistArmIRadius ...
         uc.ProxArmORadius ...
         uc.DistArmORadius ...
         uc.TorsionIRadius ...
         uc.TorsionORadius ...
         uc.MassUpperCrank ...
         uc.MassLowerCrank ...
         uc.UpperTorsionOffsetB_X ...
         uc.UpperTorsionOffsetB_Y ...
         uc.LowerTorsionOffsetTCP_X ...
         uc.LowerTorsionOffsetTCP_Y ...
         uc.GripperMountOffset_X ...
         uc.GripperMountOffset_Y ...
         uc.GripperLength ...
         uc.MinMotorSeparation ...
         uc.MaxWidth ...
         uc.MaxDepth ...
    ] = mysql(query);


    % Get Moves data

    query = ['SELECT '...
             'm.MSequence, '...
             'm.Knot_X, '...
             'm.Knot_Y, '...
             'm.MType, '...
             'm.MZone, '...
             'm.Speed, '...
             'm.Pause '...
             'FROM Moves AS m '...
             'WHERE '...
             'PathID = "',num2str(PathID),'" '...
             'ORDER BY MSequence ASC'...
             ];

    [
         MSequence ...
         Knot_X ...
         Knot_Y ...
         MType ...
         MZone ...
         Speed ...
         Pause  ...
    ] = mysql(query);
…
```

Figure F.69 RunSimulation Function (Part 2/3)

```
…
    Moves = repmat(MoveCMD,1,0);

    for i=1:size(MSequence,1)
        k = Knot(Knot_X(i),Knot_Y(i));
        m = MoveCMD(Target(k),MType(i),Speed(i),MZone(i),Pause(i));

        Moves(end+1) = m;
    end

    % Create matlab configuration from database results
    config = CalculateConfig(MotorSeparation,ProxArmLength,DistArmLength,...
                                                    WorkspaceHeight,Moves,MotorID,uc);
    ppc = PPConstraints;

    % get path planning constraints
    query = ['SELECT '...
             'p.LinearErrorFactor, '...
             'p.LastLinearTargetDistance, '...
             'p.ReactiveFactor, '...
             'p.InitialAcceptanceThreshold, '...
             'p.RelativeAgeingFactor '...
             'FROM Paths AS p '...
             'WHERE '...
             'PathID = "',num2str(PathID),'" '...
             ];
    [
        ppc.LinearErrorFactor ...
        ppc.LastLinearTargetDistance ...
        ppc.ReactiveFactor ...
        ppc.InitialAcceptanceThreshold ...
        ppc.RelativeAgeingFactor ...
    ] = mysql(query);

    query = ['SELECT '...
             'm.MaxTorque, '...
             'm.MaxVelocity, '...
             'm.MaxAcceleration, '...
             'm.MaxJerk '...
             'FROM Motors AS m '...
             'JOIN Simulations AS s ON s.MotorID = m.MotorID '...
             'WHERE '...
             's.SimID = "',num2str(SimID),'" '...
             ];
    [
        ppc.MaxTorque ...
        ppc.MaxOmega ...
        ppc.MaxAlpha ...
        ppc.MaxJerk ...
    ] = mysql(query);

    mysql('close');

    % Compile path
   [Targets_XYZ,ppr] = CompilePath(Moves,config,ppc);

   % Open and run SimMechanics simulation using parameters obtained from database
    open_system('TWODOF_PPM_Model');
    options = simset( 'SrcWorkspace' , 'current' );
    sss = sim('TWODOF_PPM_Model',inf,options);
    close_system('TWODOF_PPM_Model');

    Plot_Sim_Outputs('deg',SimID,config);   % Plot results
end
```

Figure F.70 RunSimulation Function (Part 3/3)

```
function [MotorID, newPPC] = SelectMotor(ppc,uc)
% Selects a random motor configuration from database for use in optimisation algorithms.
% VARIABLES:
% ppc - Path Planning Constraints
% uc - User Constraints

    % Open Database connection
    ch = mysql('open','localhost:3306','root','mysql');
    db = mysql('use matlab_2dofppm');

     query = ['SELECT MotorID, Name, Description, SpecsFolder, MaxTorque, MaxVelocity,'...
                'MaxAcceleration, MaxJerk, MomentInertia, EncoderResolution '...
                'FROM Motors '...
                'WHERE '...
                'MaxTorque <= "',num2str(uc.MaxMotorTorque),'" '...
                'AND '...
                'MaxVelocity <= "',num2str(uc.MaxMotorVelocity),'" '...
                'AND '...
                'MaxAcceleration <= "',num2str(uc.MaxMotorAcceleration),'" '...
                'AND '...
                'MaxJerk <= "',num2str(uc.MaxMotorJerk),'" '...
                'ORDER BY RAND() LIMIT 1'
        ];

    [MotorID Name Description SpecsFolder MaxTorque MaxVelocity MaxAcceleration ...
                                MaxJerk MomentInertia EncoderResolution] = mysql(query);

    newPPC = ppc;
    newPPC.MaxTorque = MaxTorque;
    newPPC.MaxOmega = MaxVelocity;
    newPPC.MaxAlpha = MaxAcceleration;
    newPPC.MaxJerk = MaxJerk;

    mysql('close');
end
```

Figure F.71 SelectMotor Function

```
function neighbouringConfig = SelectNeighbouringConfig(centralConfig,Moves,MotorID,uc,ss)
% Finds Configurations neighbouring a given config
% VARIABLES:
% centralConfig - Config around which neighbours will be found
% Moves - Instance of the MoveCMD class
% MotorID - ID linking to a motor
% uc - Instance of UserConstraints class
% ss - StepSize, how far away to look for neighbours
% RETURNS:
% neighbouringConfig - an array of neighbouring configurations

lb = centralConfig.LengthBase;
lu = centralConfig.LengthUpper;
ll = centralConfig.LengthLower;
wh = centralConfig.WorkspaceHeight;

neighbourLengths(1,1:4) = [lb,lu,ll,wh+ss];
neighbourLengths(2,1:4) = [lb,lu,ll+ss,wh];
neighbourLengths(3,1:4) = [lb,lu,ll+ss,wh+ss];
neighbourLengths(4,1:4) = [lb,lu+ss,ll,wh];
neighbourLengths(5,1:4) = [lb,lu+ss,ll,wh+ss];
neighbourLengths(6,1:4) = [lb,lu+ss,ll+ss,wh];
neighbourLengths(7,1:4) = [lb,lu+ss,ll+ss,wh+ss];
neighbourLengths(8,1:4) = [lb+ss,lu,ll,wh];
neighbourLengths(9,1:4) = [lb+ss,lu,ll,wh+ss];
neighbourLengths(10,1:4) = [lb+ss,lu,ll+ss,wh];
neighbourLengths(11,1:4) = [lb+ss,lu,ll+ss,wh+ss];
neighbourLengths(12,1:4) = [lb+ss,lu+ss,ll,wh];
neighbourLengths(13,1:4) = [lb+ss,lu+ss,ll,wh+ss];
neighbourLengths(14,1:4) = [lb+ss,lu+ss,ll+ss,wh];
neighbourLengths(15,1:4) = [lb+ss,lu+ss,ll+ss,wh+ss];

neighbourLengths(16,1:4) = [lb,lu,ll,wh-ss];
neighbourLengths(17,1:4) = [lb,lu,ll-ss,wh];
neighbourLengths(18,1:4) = [lb,lu,ll-ss,wh-ss];
neighbourLengths(19,1:4) = [lb,lu-ss,ll,wh];
neighbourLengths(20,1:4) = [lb,lu-ss,ll,wh-ss];
neighbourLengths(21,1:4) = [lb,lu-ss,ll-ss,wh];
neighbourLengths(22,1:4) = [lb,lu-ss,ll-ss,wh-ss];
neighbourLengths(23,1:4) = [lb-ss,lu,ll,wh];
neighbourLengths(24,1:4) = [lb-ss,lu,ll,wh-ss];
neighbourLengths(25,1:4) = [lb-ss,lu,ll-ss,wh];
neighbourLengths(26,1:4) = [lb-ss,lu,ll-ss,wh-ss];
neighbourLengths(27,1:4) = [lb-ss,lu-ss,ll,wh];
neighbourLengths(28,1:4) = [lb-ss,lu-ss,ll,wh-ss];
neighbourLengths(29,1:4) = [lb-ss,lu-ss,ll-ss,wh];
neighbourLengths(30,1:4) = [lb-ss,lu-ss,ll-ss,wh-ss];

neighbouringConfig = repmat(Configuration,1,1);

firstValidNeighbour = true;
for n=1:30
    [n1, reachable] = CalculateConfig(neighbourLengths(n,1),neighbourLengths(n,2),...
                        neighbourLengths(n,3),neighbourLengths(n,4),Moves,MotorID,uc);
    if reachable
        if (firstValidNeighbour)
            neighbouringConfig(end) = n1;
            firstValidNeighbour = false;
        else
            neighbouringConfig(end+1) = n1;
        end
    end
end
end
```

Figure F.72 SelectNeighbouringConfig Function

```
function config = SelectRandomConfig(Moves,MotorID,uc)
% Selects a random configuration based on the user constraints
% VARIABLES:
% Moves - contains Move data about path
% MotorID - motor identifier linking to motor data stored in database
% uc - UserConstraints class

config = Configuration;     % Create new configuration

reachable = false;      % Set flag
while reachable == false
%% Variable Parameters

    minBase = uc.MinMotorSeparation;
    maxBase = 0.9*uc.MaxWidth;  %allow up to 90% base + 2*5% ProxArms

    config.LengthBase = minBase + (maxBase-minBase)*rand;

    minUpper = 0;
    maxUpper = (uc.MaxWidth - config.LengthBase)/2;

    config.LengthUpper = minUpper + (maxUpper-minUpper)*rand;

    minLower = config.LengthBase;
    maxLower = sqrt((uc.MaxDepth - config.LengthUpper)^2 + (config.LengthBase/2)^2);

    config.LengthLower = minLower + (maxLower-minLower)*rand;

    minWSHeight = -uc.MaxDepth*0.1;
    maxWSHeight = uc.MaxDepth*0.1;

    config.WorkspaceHeight = minWSHeight + (maxWSHeight-minWSHeight)*rand;
…
```

Figure F.73 SelectRandomConfig Function (Part 1/4)

```
…
    %% Fixed Parameters

    config.MotorID = MotorID;

    config.MassUpper = ThickWalledTubeMass(uc.ProxArmIRadius,uc.ProxArmORadius,...
                                           config.LengthUpper,uc.ProxArmDensity);
    config.MassLower = ThickWalledTubeMass(uc.DistArmIRadius,uc.DistArmORadius,...
                                           config.LengthLower,uc.DistArmDensity);
    config.MassGripper = uc.MassGripper;
    config.MassUpperTorsion = ThickWalledTubeMass(uc.TorsionIRadius,uc.TorsionORadius,...
                                           config.LengthUpper,uc.TorsionArmDensity);
    config.MassLowerTorsion = ThickWalledTubeMass(uc.TorsionIRadius,uc.TorsionORadius,...
                                           config.LengthLower,uc.TorsionArmDensity);
    config.MassUpperCrank = uc.MassUpperCrank;
    config.MassLowerCrank = uc.MassLowerCrank;

    config.GripperMountOffset_X = uc.GripperMountOffset_X;
    config.GripperMountOffset_Y = uc.GripperMountOffset_Y;
    config.GripperLength = uc.GripperLength;

    config.UpperTorsionOffsetB_X = uc.UpperTorsionOffsetB_X;
    config.UpperTorsionOffsetB_Y = uc.UpperTorsionOffsetB_Y;
    config.LowerTorsionOffsetTCP_X = uc.LowerTorsionOffsetTCP_X;
    config.LowerTorsionOffsetTCP_Y = uc.LowerTorsionOffsetTCP_Y;

    config.InRadiusArms = uc.ProxArmIRadius;
    config.OutRadiusArms = uc.DistArmIRadius;
    config.InRadiusTorsion = uc.TorsionIRadius;
    config.OutRadiusTorsion = uc.TorsionORadius;

    config.InertiaUpper = ThickWalledTubeInertia(config.InRadiusArms,...
                              config.OutRadiusArms,config.LengthUpper,config.MassUpper);
    config.InertiaLower = ThickWalledTubeInertia(config.InRadiusArms,...
                              config.OutRadiusArms,config.LengthLower,config.MassLower);
    config.InertiaGripper = ThickWalledTubeInertia(config.InRadiusArms,...
                              config.OutRadiusArms,config.GripperLength,...
                              config.MassGripper);
    config.InertiaUpperTorsion = ThickWalledTubeInertia(config.InRadiusTorsion,...
                                  config.OutRadiusTorsion,config.LengthUpper,...
                                  config.MassUpperTorsion);
    config.InertiaLowerTorsion = ThickWalledTubeInertia(config.InRadiusTorsion,...
                                  config.OutRadiusTorsion,config.LengthLower,...
                                  config.MassLowerTorsion);
    config.InertiaUpperCrank = ThickWalledTubeInertia(config.InRadiusTorsion,...
                                  config.OutRadiusTorsion,sqrt(...
                                  config.UpperTorsionOffsetB_X^2+...
                                  config.UpperTorsionOffsetB_X^2),...
                                  config.MassUpperCrank);
    config.InertiaLowerCrank = ThickWalledTubeInertia(config.InRadiusTorsion,...
                                  config.OutRadiusTorsion,sqrt(...
                                  config.LowerTorsionOffsetTCP_X^2+...
                                  config.LowerTorsionOffsetTCP_X^2),...
                                  config.MassLowerCrank);

    config.MinUpperArmAngle = uc.MinArmAng_BU;
    config.MaxUpperArmAngle = uc.MaxArmAng_BU;
    config.Min1_2ArmAngle = uc.MinArmAng_UL;
    config.Max1_2ArmAngle = uc.MaxArmAng_UL;
    config.MinLowerArmAngle = uc.MinArmAng_LL;
    config.MaxLowerArmAngle = uc.MaxArmAng_LL;
…
```

Figure F.74 SelectRandomConfig Function (Part 2/4)

```
…

    %% Internally Computed Parameters

    config.ThetaAstart = d2r(180);
    config.ThetaBstart = d2r(180);

    [tcpX, tcpY, error, errorMsg ] = Direct_2DOF_PPM(config.ThetaAstart,...
                config.ThetaBstart,config.LengthBase,config.LengthUpper,...
                config.LengthLower,config.Min1_2ArmAngle,config.Max1_2ArmAngle,...
                config.MinLowerArmAngle,config.MaxLowerArmAngle);

    if error > 1
        reachable = false;
        errorMsg
        continue;
    end

    config.CS1_UpperA = [0, 0, 0];
    config.CS1_LowerA = [0, 0, 0];
    config.CS1_UpperB = [0, 0, 0];
    config.CS1_LowerB = [0, 0, 0];
    config.CS1_Gripper = [0, 0, 0];
    config.CS1_UpperTorsion = [0, 0, 0];
    config.CS1_UpperCrank = [0, 0, 0];
    config.CS1_LowerTorsion = [0, 0, 0];
    config.CS1_LowerCrank = [0, 0, 0];

    config.CS2_UpperA = [ajX-aBaseX,ajY-aBaseY,0];
    config.CS2_LowerA = [tcpX-ajX,tcpY-ajY,0];
    config.CS2_UpperB = [bjX-bBaseX,bjY-bBaseY,0];
    config.CS2_LowerB = [tcpX-bjX,tcpY-bjY,0];
    config.CS2_Gripper = [0,-config.GripperLength,0];
    config.CS2_UpperTorsion = [bjX-bBaseX,bjY-bBaseY,0];
    config.CS2_UpperCrank =[config.UpperTorsionOffsetB_X,config.UpperTorsionOffsetB_Y, 0];
    config.CS2_LowerTorsion = [tcpX-bjX,tcpY-bjY,0];
    config.CS2_LowerCrank = [config.LowerTorsionOffsetTCP_X,...
                                config.LowerTorsionOffsetTCP_Y,0];

    config.CS3_LowerB = [0 0 0];
    config.CS3_UpperB = [0 0 0];
    config.CS3_Gripper = [-0.05,0,0];
    config.CS3_UpperCrank = [config.LowerTorsionOffsetTCP_X,...
                                config.LowerTorsionOffsetTCP_Y,0];
    config.CS3_LowerCrank = [config.GripperMountOffset_X,config.GripperMountOffset_Y,0];

    config.CS4_Gripper = [0.05,0,0];
…
```

Figure F.75 SelectRandomConfig Function (Part 3/4)

```
…
    config.CG_UpperA = [(ajX-aBaseX)/2,(ajY-aBaseY)/2,0];
    config.CG_LowerA = [(tcpX-ajX)/2,(tcpY-ajY)/2,0];
    config.CG_UpperB = [(bjX-bBaseX)/2,(bjY-bBaseY)/2,0];
    config.CG_LowerB = [(tcpX-bjX)/2,(tcpY-bjY)/2,0];
    config.CG_Gripper = [0,-config.GripperLength/2,0];
    config.CG_UpperTorsion = [(bjX-bBaseX)/2,(bjY-bBaseY)/2,0];
    config.CG_UpperCrank = [(config.UpperTorsionOffsetB_X+...
                             config.LowerTorsionOffsetTCP_X)/2,...
                             config.UpperTorsionOffsetB_Y/2,0];
    config.CG_LowerTorsion = [(tcpX-bjX)/2,(tcpY-bjY)/2,0];
    config.CG_LowerCrank = [config.LowerTorsionOffsetTCP_X/2,0,0];

    config.OrientCG_UpperA = [0,0,0];
    config.OrientCG_LowerA = [0,0,0];
    config.OrientCG_UpperB = [0,0,0];
    config.OrientCG_LowerB = [0,0,0];
    config.OrientCG_Gripper = [0,0,0];
    config.OrientCG_UpperTorsion = [0,0,0];
    config.OrientCG_UpperCrank = [0,0,0];
    config.OrientCG_LowerTorsion = [0,0,0];
    config.OrientCG_LowerCrank = [0,0,0];

    config.Gpoint_1 = [aBaseX, aBaseY, 0];
    config.Gpoint_2 = [bBaseX, bBaseY, 0];
    config.Gpoint_3 = [bBaseX+config.UpperTorsionOffsetB_X, ...
                       bBaseY+config.UpperTorsionOffsetB_Y,0];

    [ thetaA, thetaB, error, errorMsg ] = Inverse_2DOF_PPM(tcpX,tcpY,...
                             config.LengthBase,config.LengthUpper,config.LengthLower);
    if error ~=0
        reachable = false;
        errorMsg
        continue;
    end
    config.ThetaA_IC = mod(thetaA + config.ThetaAstart,pi);
    config.ThetaB_IC = mod(thetaB + config.ThetaBstart,pi);

    %% Check reachability
    reachable = CheckReachability(Moves,config);
end

end
```

Figure F.76 SelectRandomConfig Function (Part 4/4)

```
function StorePathsUserConstraintsSQL(cp,uc)
% Stores the path Move data along with UserConstraints
% VARIABLES:
% cp - Instance of CyclePath class
% uc - Instance of UserConstraints

    % Open database connection
    ch = mysql('open','localhost:3306','root','mysql');
    db = mysql('use matlab_2dofppm');

    % Store Paths
    query = ['INSERT INTO paths ('...
        'PathID,'...
        'LinearErrorFactor,'...
        'LastLinearTargetDistance,'...
        'ReactiveFactor,'...
        'InitialAcceptanceThreshold,'...
        'RelativeAgeingFactor)'...
        'VALUES ('...
        '"',num2str(cp.ID),'",'...
        '"',num2str(cp.PPC.LinearErrorFactor),'",'...
        '"',num2str(cp.PPC.LastLinearTargetDistance),'",'...
        '"',num2str(cp.PPC.ReactiveFactor),'",'...
        '"',num2str(cp.PPC.InitialAcceptanceThreshold),'",'...
        '"',num2str(cp.PPC.RelativeAgeingFactor),'"'...
        ')'...
        ];
    t = mysql(query);

    % Store Moves
    numMoves = size(cp.Moves,2);
    for m=1:numMoves
        query = ['INSERT INTO moves ('...
            'PathID,'...
            'MSequence,'...
            'Knot_X,'...
            'Knot_Y,'...
            'MType,'...
            'MZone,'...
            'Speed,'...
            'Pause)'...
            'VALUES ('...
            '"',num2str(cp.ID),'",'...
            '"',num2str(m),'",'...
            '"',num2str(cp.Moves(m).Target.Knot.X),'",'...
            '"',num2str(cp.Moves(m).Target.Knot.Y),'",'...
            '"',cp.Moves(m).MoveType,'",'...
            '"',num2str(cp.Moves(m).Zone),'",'...
            '"',num2str(cp.Moves(m).Velocity),'",'...
            '"',num2str(cp.Moves(m).Pause),'"'...
            ')'...
            ];
        t = mysql(query);
    end


    % Store UserConstraints
    query = ['INSERT INTO userconstraints ('...
        'PathID,'...
        'MaxMotorTorque,'...
        'MaxMotorVelocity,'...
        'MaxMotorAcceleration,'...
        'MaxMotorJerk,'...
        'MassGripper,'...
        'MinArmAng_BU,'...
…
```

Figure F.77 StorePathsUserConstraintsSQL Function (Part 1/2)

```
…
        'MinArmAng_UL,'...
        'MinArmAng_LL,'...
        'MaxArmAng_BU,'...
        'MaxArmAng_UL,'...
        'MaxArmAng_LL,'...
        'ProxArmDensity,'...
        'DistArmDensity,'...
        'TorsionArmDensity,'...
        'ProxArmIRadius,'...
        'DistArmIRadius,'...
        'ProxArmORadius,'...
        'DistArmORadius,'...
        'TorsionIRadius,'...
        'TorsionORadius,'...
        'MassUpperCrank,'...
        'MassLowerCrank,'...
        'UpperTorsionOffsetB_X,'...
        'UpperTorsionOffsetB_Y,'...
        'LowerTorsionOffsetTCP_X,'...
        'LowerTorsionOffsetTCP_Y,'...
        'GripperMountOffset_X,'...
        'GripperMountOffset_Y,'...
        'GripperLength,'...
        'MinMotorSeparation,'...
        'MaxWidth,'...
        'MaxDepth)'...
        'VALUES ('...
        '"',num2str(cp.ID),'",'...
        '"',num2str(uc.MaxMotorTorque),'",'...
        '"',num2str(uc.MaxMotorVelocity),'",'...
        '"',num2str(uc.MaxMotorAcceleration),'",'...
        '"',num2str(uc.MaxMotorJerk),'",'...
        '"',num2str(uc.MassGripper),'",'...
        '"',num2str(uc.MinArmAng_BU),'",'...
        '"',num2str(uc.MinArmAng_UL),'",'...
        '"',num2str(uc.MinArmAng_LL),'",'...
        '"',num2str(uc.MaxArmAng_BU),'",'...
        '"',num2str(uc.MaxArmAng_UL),'",'...
        '"',num2str(uc.MaxArmAng_LL),'",'...
        '"',num2str(uc.ProxArmDensity),'",'...
        '"',num2str(uc.DistArmDensity),'",'...
        '"',num2str(uc.TorsionArmDensity),'",'...
        '"',num2str(uc.ProxArmIRadius),'",'...
        '"',num2str(uc.DistArmIRadius),'",'...
        '"',num2str(uc.ProxArmORadius),'",'...
        '"',num2str(uc.DistArmORadius),'",'...
        '"',num2str(uc.TorsionIRadius),'",'...
        '"',num2str(uc.TorsionORadius),'",'...
        '"',num2str(uc.MassUpperCrank),'",'...
        '"',num2str(uc.MassLowerCrank),'",'...
        '"',num2str(uc.UpperTorsionOffsetB_X),'",'...
        '"',num2str(uc.UpperTorsionOffsetB_Y),'",'...
        '"',num2str(uc.LowerTorsionOffsetTCP_X),'",'...
        '"',num2str(uc.LowerTorsionOffsetTCP_Y),'",'...
        '"',num2str(uc.GripperMountOffset_X),'",'...
        '"',num2str(uc.GripperMountOffset_Y),'",'...
        '"',num2str(uc.GripperLength),'",'...
        '"',num2str(uc.MinMotorSeparation),'",'...
        '"',num2str(uc.MaxWidth),'",'...
        '"',num2str(uc.MaxDepth),'"'...
        ')'...
        ];
    t = mysql(query);

    mysql('close');
end
```

Figure F.78 StorePathsUserConstraintsSQL Function (Part 2/2)

```
function StoreSimulationsSQL(config,ppc,ppr,pathID,comment,comment2,iteration)
% Stores results of simulation run
% VARIABLES:
% config - Instance of Configuration class
% ppc - Instance of PathPlanningConstraints class
% ppr - Instance of PathPlanningResults class
% pathID - unique path identifier
% comment - ability to store text associated with simulation
% comment2 - another ability to store text associated with simulation
% iteration - ability to store what optimisation iteration simulation occured on

    % Open database connection
    ch = mysql('open','localhost:3306','root','mysql');
    db = mysql('use matlab_2dofppm');

    % Retrieve next available SimID from database
    simID = mysql('SELECT IFNULL(MAX(SimID)+1,1) FROM simulations');

    % Store simulation data
    query = ['INSERT INTO simulations ('...
        'SimID,'...
        'ProxArmLength,'...
        'DistArmLength,'...
        'MotorSeparation,'...
        'WorkspaceHeight,'...
        'MotorID,'...
        'CycleTime,'...
        'ExecutionDT,'...
        'PathID,'...
        'Comment,'...
        'Comment2,'...
        'Iteration)'...
        'VALUES ('...
        '"',num2str(simID),'",'...
        '"',num2str(config.LengthUpper),'",'...
        '"',num2str(config.LengthLower),'",'...
        '"',num2str(config.LengthBase),'",'...
        '"',num2str(config.WorkspaceHeight),'",'...
        '"',num2str(config.MotorID),'",'...
        '"',num2str(ppr.PathA(size(ppr.PathA,1)).EndTime),'",'...
        'NOW(),'...
        '"',num2str(pathID),'",'...
        '"',comment,'",'...
        '"',comment2,'",'...
        '"',num2str(iteration),'"'...
        ')'...
        ];

    t = mysql(query);

    mysql('close');     % Close database connection

end
```

Figure F.79 StoreSimulationsSQL Function

```
classdef Target
% Defines a target (i.e. Knot position at a given PathTime).

properties
    Knot        % Position and Orientation of Target
    PathTime    % Time along path at which knot is reached
    VelocityLimit   % Maximum TCP velocity permitted during travel to knot
end

methods
    % Create instance of Target class with variables
    function t = Target(Knot,PathTime,VelocityLimit)
        if nargin == 1
            t.Knot = Knot;
            t.PathTime = 0; % Set to zero if initialised only with Knot data.
                            % PathTime will be updated later.
        elseif nargin == 2
            t.Knot = Knot;
            t.PathTime = PathTime;
        elseif nargin ==3
            t.Knot = Knot;
            t.PathTime = PathTime;
            t.VelocityLimit = VelocityLimit;
        end
    end
end
end
```

Figure F.80 Target Class

```
classdef TerminationCondition
% Contains conditions for termination of the optimisation process

properties
    CycleTime   % Path cycle time. Optimised value must be less than this to be
                %  considered optimised.
    Iterations  % Number of optimisation iterations.
end

end
```

Figure F.81 TerminationCondition Class

```
function inertia = ThickWalledTubeInertia(r1,r2,h,m)
% Calculates the inertia of a thick-walled cylindrical tube with open ends
% VARIABLES:
% r1 - Inner Radius (m)
% r2 - Outer Radius (m)
% h - Length (m)
% m - Mass (kg)

inertia = [((1/12)*m*(3*(r1^2+r2^2)+h^2)), 0, 0;
            0,((1/12)*m*(3*(r1^2+r2^2)+h^2)),0;
            0,0,((1/2)*m*(r1^2+r2^2));];

end
```

Figure F.82 ThickWalledTubeInertia Function

```
function mass = ThickWalledTubeMass(r1,r2,length,density)
% Calculates the mass of a thick-walled cylindrical tube with open ends
% VARIABLES:
% r1 - Inner Radius (m)
% r2 - Outer Radius (m)
% length - Length of cylinder (m)
% density - density of tube material (kg/m^3)

    volume = (pi*r2^2-pi*r1^2)*length;

    mass = density*volume;

end
```

Figure F.83 ThickWalledTubeMass Function

```
classdef UserConstraints
% Contains constraints on manipulator specified by the user

properties
    MaxMotorTorque          % Maximum torque available from motor (Nm)
    MaxMotorVelocity        % Maximum angular velocity available from motor (rad/s)
    MaxMotorAcceleration    % Maximum angular acceleration available from motor (rad/s2)
    MaxMotorJerk            % Maximum angular jerk available from motor (rad/s3)
    MassGripper             % Mass of the gripper and any load (kg)
    MinArmAng_BU            %Minimum angle allowed between base and upper/proximal arm(rad)
    MinArmAng_UL            % Minimum angle allowed between upper/proximal arm and
                            %  lower/distal arm (rad)
    MinArmAng_LL            % Minimum angle allowed between the two lower/distal arms (rad)
    MaxArmAng_BU            %Maximum angle allowed between base and upper/proximal arm(rad)
    MaxArmAng_UL            % Maximum angle allowed between upper/proximal arm and
                            %  lower/distal arm (rad)
    MaxArmAng_LL            % Maximum angle allowed between the two lower/distal arms (rad)
    ProxArmDensity          % Density of upper/proximal arm (kg/m3)
    DistArmDensity          % Density of lower/distal arm (kg/m3)
    TorsionArmDensity       % Density of stabiliser arm (kg/m3)
    ProxArmIRadius          % Inner Radius of proximal arm (m)
    DistArmIRadius          % Inner Radius of distal arm (m)
    ProxArmORadius          % Outer Radius of proximal arm (m)
    DistArmORadius          % Outer Radius of distal arm (m)
    TorsionIRadius          % Inner Radius of stabiliser arm (m)
    TorsionORadius          % Outer Radius of stabiliser arm (m)
    MassUpperCrank          % Mass of the upper crank (kg)
    MassLowerCrank          % Mass of the lower crank (kg)
    UpperTorsionOffsetB_X   % Offset from center of motor B for base point of stabiliser
                            %  arm(X)
    UpperTorsionOffsetB_Y   % Offset from center of motor B for base point of stabiliser
                            %  arm(Y)
    LowerTorsionOffsetTCP_X % Offset from center of 'TCP' for lower torsion bar (X)
    LowerTorsionOffsetTCP_Y % Offset from center of 'TCP' for lower torsion bar (Y)
    GripperMountOffset_X    % Offset from bottom revolute joint where the gripper mounts(X)
    GripperMountOffset_Y    % Offset from bottom revolute joint where the gripper mounts(Y)
    GripperLength           % Length of the gripper (m)
    MinMotorSeparation      % Minimum separation distance between centers of motors (m)
    MaxWidth                % Maximum width of the manipulator as defined as
                            %  base length + 2x upper arm length
    MaxDepth                % Maximum depth of the manipulator
end

end
```

Figure F.84 UserConstraints Class