ABSTRACT

Title of Document:	TASK-BASED MASS OPTIMIZATION OF RECONFIGURABLE ROBOTIC MANIPULATOR SYSTEMS
	Nathan Thomas Koelln, Master of Science, 2006
Directed By:	Dr. David L. Akin, Department of Aerospace Engineering

This work develops a method for implementing task-based mass optimization of modular, reconfigurable manipulators. Link and joint modules are selected from a library of potential parts and assembled into serial manipulator configurations. A genetic algorithm is used to search over the potential set of combinations to find mass-minimized solutions. To facilitate the automatic evaluation required by the genetic algorithm, Denavit-Hartenberg parameters are automatically generated from module combinations. Reconfigurable manipulators are shown to be lighter than fixed-topology manipulators, demonstrating the potential utility of reconfigurable robotics technology for mass reduction in space robots.

TASK-BASED MASS OPTIMIZATION OF RECONFIGURABLE ROBOT
MANIPULATOR SYSTEMS

By

Nathan Thomas Koelln

Thesis submitted to the Faculty of the Graduate School of the
University of Maryland, College Park, in partial fulfillment
of the requirements for the degree of
Master of Science
2006

Advisory Committee:
Associate Professor David Akin, Chair
Associate Professor Robert Sanner
Dr. Craig Carignan

Acknowledgements

Other than space research, nothing in life is ever done in a vacuum. Though only one pair of hands punched the keyboard to produce this thesis, many heads and hearts helped to guide those fingers. I would like to single out a few of those people that inspired me and brought this work to completion. Though this list is, as all such lists are, incomplete, nevertheless I would still like to acknowledge:

Dr. Akin for his invaluable insight and input into this body of research,

The members of my committee for their time and recommendations,

My parents Rebecca and Thomas for believing in me,

My siblings Anna, Jacob and David for forcing me to remember my place,

Shane Jacobs for not letting me quit,

Lynn Gravatt for getting me moving,

Enrico Sabelli for helping to keep the long hours lighthearted.

and last but in no way least, the great-grandfather of reconfigurable robotics, Optimus Prime, for his humble leadership and self sacrifice that we might live in a better world.

Table of Contents

List of Tables

List of Figures

List of Abbreviations

ADT – Axiomatic Design Theory
AIM – Assembly Incidence Matrix
D-H – Denavit-Hartenberg
DOF – Degree of Freedom
DP – Design Parameters
EVA – Extra-Vehicular Activity
FR – Functional Requirements
GA – Genetic Algorithm
HST – Hubble Space Telescope
ISS – International Space Station
MRSH – Multiple Restart Statistical Hill-climbing
NASA – National Aeronautics and Space Administration
PV – Process Variables
OIM – Object Incidence Matrix
SPDM – Special Purpose Dexterous Manipulator
SRMS – (Space) Shuttle Remote Manipulator System
SSL – Space Systems Laboratory
SSRMS – Space Station Remote Manipulator System

Chapter 1: Introduction

The development of on-orbit servicing technologies that can further the capabilities and boundaries of what is possible in spacecraft operations provides the backdrop for this body of work. This particular thesis is an attempt to push forward and develop the justification for further research into the use of reconfigurable manipulator systems for use in on-orbit servicing. This chapter is intended to address the inspirations and context that the research in later chapters details, and outlines what is to come in the following chapters.

1.1 On-Orbit Servicing

The plight of the Hubble Space Telescope (HST) has recently demonstrated a real and growing need for the capability to upgrade and service unique and valuable spacecraft that are in danger of failing or have already failed. In the wake of the loss of the space shuttle *Columbia*, NASA desires that each of its shuttle orbiters be capable of reaching the International Space Station (ISS) should the orbiter be damaged enough that atmospheric reentry is deemed unsafe. Due to differences in their orbital inclinations, the HST and the ISS are not reachable by a single orbiter during the course of a single mission, meaning that any further crewed missions to HST are considered to be risky. However, the loss of potential science data by a failure of the HST's remaining gyroscopes is also a loss that many are not willing to bear.

This, of course, is where robotics can step in to fill the void left by the restriction of shuttle activity. Without the astronaut extra-vehicular activity (EVA) capabilities the offered by thespace shuttle, unmanned options become the only method of servicing valuable on-orbit systems. This latter option has inspired the nexus of this thesis.

1.2 Current Robotic Servicing Strategies

The current state of robotic servicing "orthodoxy" is to use two systems to perform the tasks necessary to complete the job. One robot acts essentially as a crane to perform large-scale motions in the workspace, and the second robot serves primarily as a highly complicated end-effector of the first robot to perform the fine-scale motions necessary to complete the task. The best current example of this model is the Space-Station Remote Manipulator System (SSRMS), which is currently on orbit, picking up and moving the not-currently-on-orbit Special Purpose Dexterous Manipulator (SPDM) to and from its worksite where it can perform dexterous tasks for servicing the station (Currie, 2004). NASA's Robonaut anthropomorphic robotic system is proposed to work in much the same way, being transported to and from its worksite by large crane-like systems like the SSRMS or the Shuttle Remote Manipulator System (SRMS) (Ambrose, 2000).

While this type of set-up is conceptually and operationally simple, it is not necessarily the most efficient. As the SSRMS and SRMS class arms require forces to be reacted down a long boom, resulting in large torques at its joints, which require large and

heavy joint actuators. Additionally, two systems must be designed and maintained to do the job, making much of the crane-dexterous robot pair inactive at any given time.

1.3 Reconfigurable Modular Systems as an Alternative

As an alternative to the current orthodoxy, modular reconfigurable robotics have been suggested as a means of reducing the overall system mass and the number of total parts required to build the system. A modular reconfigurable system is one that is assembled from simple building blocks, such as individual link and joint modules, that form together to make a particular arm configuration. the arm can be broken down and reassembled into a new configuration as the situation dictates. Such a system would have the potential to act as either the crane or the dexterous robot as described in the section above, or alternatively, a reconfigurable modular robot could move hand over hand to its worksite, reconfigure itself into a dexterous arm and proceed with the task at hand.

One system that has been proposed for such work is MORPHbots, developed by the University of Maryland's Space Systems Laboratory (SSL)(Akin,2003). The system as envisioned by researchers at SSL would consist of several types of actuators, tools and nodes that could either self-reconfigure or reconfigure with the aid of another similar robot in order to achieve a desired topology for a specified task. In January 2004, a prototype two degree of freedom (2-DOF) module was built by SSL researchers, and an androgynous connector system was developed that would allow connections to be made between any two modules of the system. The MORPHbot system was never developed

3

beyond the lone 2-DOF module, though it did provide the architectural model for reconfigurable manipulators that inspired this work.

1.4 Research Goals

While reconfigurable robotics in general (and MORPHbots in particular) have been claimed to possess many possible advantages over traditional static-topology robotics, the existing literature on reconfigurable modular robotics has not explicitly shown this to be true. One of the primary claims of reconfigurable robotic systems is that by rearranging the modules of the system, the kinematics of the robot could be changed in order to provide an advantage and allow the reconfigurable system to perform tasks "better" than the their non-reconfigurable counterparts. The main goal of this research was to go beyond many of the broad, sweeping claims made about the advantages of reconfigurable robotic systems and to find a way of showing that for a specific type of work, that the advantage (or disadvantage) could actually be quantified.

There are, of course, many ways to characterize what is meant by "better" in the field of robotics. In the case of a MORPHbot style system designed for operations in outer space, one of the key measures of "better" is the overall mass of a robotic system. MORPHbots was conceived to be a system light enough to be squeezed into the mass-margins of large satellites for later servicing needs, or launched to service ailing on-orbit assets using low-cost on-demand systems such as FALCON or RASCAL. Given this mass-restrictive mission model, the overall system mass becomes of primary concern, though a lightweight system not capable of performing the tasks required of it is worse

than no manipulator at all. Therefore, a method for estimating the mass of a reconfigurable robot required for performing a task was required.

As a genetic algorithm inherently requires many evaluations of many individuals to work, a method for automatically converting a serial assemblage of modules into a kinematic representation was required. As the Denavit-Hartenberg (D-H) parameters are the standard representation for robotic manipulators, a process was developed to automatically change a serial assembly of robotic joints into its corresponding D-H parameters. The development of this process for automatically modeling the kinematics of configurations is detailed in Chapter 3. A conversion of a serial assembly of parts to a set of D-H parameters also provides access to a host of analysis tools developed for use in fixed-topology robotics, in particular the Matlab Robotics Toolbox (Corke, 2001).

In the MORPHbot paradigm, a robotic manipulator is built up from a series of modules in order to complete a given task. As will be shown in Chapter 4, even a modest number of modules can add up to billions of potential assemblies, each of which may or may not be capable of performing the task at hand. If any comparison between a reconfigurable system and a fixed-topology system is to be made, a method must be developed for searching through the many potential configurations to find the best solutions for completing the task. Brute force methods will not work over such a large search space, and traditional optimization methods such as the Newton-Raphson method fail because the relationship between a robot's configuration and its performance is difficult to define in the generic case. Therefore, a genetic algorithm was examined as a method to optimize over this very large, poorly conditioned search space, which is the topic of Chapter 4.

Of course, in order to make a fair comparison between a fixed-topology robot and a reconfigurable robotic kit, a method for determining the components of the kit is necessary. The development of a method for moving from a series of manipulator configurations to a kit of robotic parts is presented in Chapter 5. Chapter 5 is the necessary piece of the research that moves the rest of this thesis from being a method to design modular (not necessarily reconfigurable) manipulators to showing that reconfigurable robotics have the potential to save mass when compared to fixed-topology systems.

Chapter 6 gives conclusions of the research, explores potential future steps that could and should be taken to further develop the field of modular reconfigurable robotics. Though reconfigurable robots have a great number of potential benefits over fixed-topology robots, a single body of research could cover not all of them. Though branching mechanisms, closed-chain configurations, and increased reliability are all relevant topics, all of them are left for future research.

Chapter 2: Review of Literature

This chapter is intended to provide a review of the current literature relevant to the topic of this thesis, and provides justification for the course of research pursued according to the goals outlined in Chapter 1. This chapter is subdivided into three sections: on kinematic modeling of robotic systems, methodologies for synthesizing modular robotic manipulators, and random search methods. Each of these areas is critical for the process of automatically creating, modeling and evaluating reconfigurable robotic systems.

2.1 Kinematic Modeling of Robotic Systems

One of the most basic problems in the design of any robotic manipulator is expressing the position and orientation of one of its constituent components in relation to some universal reference frame and the position and orientation of the preceding components. There are two basic methods that are utilized nearly universally in robotics community: the Denevit-Hartenberg (D-H) notation (Denevit and Hartenberg, 1955) and the *product of exponential* (POE) equations which were first applied to robotics by Brockett (1984).

The D-H notation is widely used as it contains the minimum number of components to express the change in orientation and position from one joint body to the next, making the method efficient for computation. Kinematic relationships between the base frame and any other frame can be obtained by propagating the 4x4 D-H

transformation matrix up the kinematic chain formed by each of the joint and link modules. Though the D-H method represents rotary and prismatic joints in a concise manner, it does not have a method for dealing effectively with joints that combine more than one type of motion such as helical or screw joints. Fortunately, these types of joints have not appeared in many manipulator configurations, though it is not entirely clear if this limitation of the D-H notation itself has resulted in the exclusion of these particular types of joints. Nevertheless, given that most robotic mechanisms employ single degree-of-freedom actuators, D-H parameters have come to be used by many different researchers (Paredis 1995, 1997, Schneider 1994, Bi 2003, Corke 2001).

Product of exponential (POE) representation maintains the same form across different joint motion types, giving it key advantages to analyzing certain manipulators possessing certain types of joints. Though it doesn't necessarily provide a computational advantage over the D-H method (Mukundan 1987), it is useful for recursive formulations of manipulator kinematics and is used by Chen and Yang (1999) in calculating local frame transformations (but not the overall system kinematics).

Both POE and D-H notation are used for deriving the frame transformations, but it is not strictly necessary to go through either of these formalisms if some other representation can be used. In his development of a scheme for developing an automatic modeling technique, Chen (1999) simply leaves it up to the reader to determine how the frame transformations are determined, merely stating that they exist.

In this thesis, the D-H notation is used for the representation of manipulators, as it is accepted as the standard method for representing the relationships between motion frames of robotic manipulators. Being the standard, many tools have been developed for

8

use with D-H characterized manipulators, and using these tools proves to be useful in performing analyses on such manipulators. Only the kinematic components of robotic modeling were deemed important for this thesis as manipulator configurations were considered to be performing their tasks quasi-statically. The expansion into dynamics was not made in this work, though in principle, it is a step that could be made provided that the modules were modeled to a sufficient level of detail.

2.2 Modular Manipulator Design

The basic goal of modulator manipulator design is to take a pre-existing kit of modules and to assemble them into a configuration that is deemed "best" for a given task. A task could be any number of different force and motion trajectories and may or may not be constrained by position or velocity requirements. Finding the optimal design is by no means a trivial problem as the search space grows exponentially with the number of available modules in the kit. For example, Paredis (1997) showed that for a kit consisting of 23 elements to be assembled into a 7-DOF manipulator using only axially symmetric modules that have no relative orientation, there are approximately 3×10^{20} possible combinations. While a kit of 23 different modules is quite large when compared to the library size of similar studies, it does still show that that the total search space is immense even when many simplifying assumptions (7-DOF only, link-joint module connections only) are made to pare down the potential number of combinations. While it would be theoretically possible to evaluate each of the possible robot designs from a given kit of

parts and examine its performance relative to the specified task, such an exhaustive searching of the configuration space would be very impractical at best.

Despite these difficulties several rather general methods for automatic generation of modular robotic designs have been proposed (Bi 2002, Chen 1999, Paredis and Kholsa 1995 and 1997, Chocron 1997, Han 1997). Of these proposed methods, there is very little difference in the conceptual nature of them. Each starts from a pre-defined kit of joint and link modules and uses randomized search methods to find combinations of the modules that have the highest performance index (defined by the designer), but not necessarily the global optimum. Of these various studies, Paredis and Kholsa (1995) utilize a simulated annealing method to implement the random search while the rest of the studies all utilize one form or another of a genetic or evolutionary algorithm.

This thesis follows similar methods by defining a library of modules, then building up configurations of those parts, evaluating them using a mass-based optimization, and then using a genetic algorithm to evolve a solution over time. However, the above studies only specified goal positions for the end-effector to reach, while this thesis simultaneously considers goal points and the forces required at the tool-tip. Therefore, while the other studies are concerned with the design of the best manipulators to reach a series of goal points, this work is concerned with the best manipulator for exerting a given force and moment at the goal point.

2.2.1 Axiomatic Design Theory

A great deal of work has gone into the modular design of systems in other fields such as software design, organizational structures, and manufacturing. Suh (1990, 1998) has summarized many of the principles for designing the most effective design of modular systems in what he calls Axiomatic Design Theory (ADT). Using the principles of ADT, a designer is able to determine the most effective mappings from the functional requirements (FR) to the design parameters (DP) to the process variables (PV). In the case of modular robotic design, there is no process level of design in the sense of ADT, and so the most basic level of design becomes the DPs.

Unfortunately, in the field of modular robotics, a strong coupling exists between the behavior of the individual joint and link modules with the behavior of the entire system. The most comprehensive discussion of these inherent difficulties came from Bi (2002) who applied the ADT developed by Suh (1998) to the problem of modular robotic design. The independence property of ADT states that each functional requirement should be independent. As the position and orientation of a modular robotic end-effector in Cartesian space is highly coupled to the arrangement of the individual joint modules, it becomes impossible to achieve independence of the functional requirements at the joint-module level. For example, changing the initial joint module in a kinematic chain from a rotation about the global z-axis to a rotation about the global y-axis will completely change the kinematic model of the entire manipulator and of each module downstream from it. The end result of this is that existing modular design schemes developed for other fields like software or mechanism design are not applicable in the field of robotic manipulator design as the entire manipulator must be considered the "module" by the

11

principles of ADT rather than each of the individual joint and link modules as we would prefer.

The critical point to take away from this, is that the process of developing a relationship between the design inputs (modules) and the design outputs (joint-torque, actuator mass, etc.) is extremely complicated to do in the general case. For this reason, no effort was made in developing analytical relationships in this thesis between the input variables and the output variables.

2.2.2 Design Variables

In any given design, the designer has control over a certain number of input parameters that can be changed and varied in pursuit of the best design for a given set of requirements and restrictions. In launch vehicle design for example, the design variables might include the number of engines, the number of stages of a rocket, or perhaps the choice of propellant used; In the automotive industry design variables might include engine horsepower, wheelbase, and brake type.

In modular robot design, there are many possible ways to represent the different variables of the overall design, and they depend highly on the architecture of the modular system. Paredis and Kholsa (1997) used an inventory consisting of five different module topologies (pivot joint, wrist module, rotary joint, straight link, and corner link) before evaluating the fitness of the given design. In this case the design variable was the order in which the various modules were assembled, along with the relative location of the base of the manipulator with respect to the desired target. The design variables were effectively a

mixture of discrete and continuous variables, which tends to be the norm for modular robotic systems.

Chocron and Bidaud (1997) used a similar approach in their formulation of a genetic algorithm for use with 3-D modular manipulators in that the number and types of joints and the link lengths were the design variables. Each joint-link combination was encoded into an 8-bit string for use in a genetic algorithm so that the link lengths, though nominally continuous, were actually made to be discrete due to the binary encoding. In this study, the number of degrees of freedom (DOF) was not defined ahead of time, allowing the search space to expand.

In contrast, Han (1997) restricted the total number of DOFs to be the minimum needed to achieve the task at hand, creating a non-redundant configuration for the manipulator. Only two types of joints were examined, and only one type of link topology with a discrete variable length was used. Here the design variables were once again the order and type of joint modules, though the number of joints in the chain was known *a priori*.

Bi (2002) and Chen (1999, 1995) used a similar matrix formulation of the joint configuration space, which Chen termed the Assembly Incidence Matrix (AIM) and Bi called the Object Incidence Matrix (OIM). While both essentially perform the same function, Bi's formulation stored more information about the joint and link modules inside the matrix. In the example 3-DOF system shown in Figure 2.2, each element of the OIM contains a "0" if there is no connection between the link and the corresponding vector, or a series of four numbers indicating which the relationship between the joint and the link. The first number indicates that the element specified by the row is connected to

13

the element specified by the column. The second number determines whether part "A"

(1) or part "B" (0) of the joint specified by the column is connected to the corresponding

link. The third number indicates the port of the link or base that is attached to the

corresponding joint. The fourth number indicates the port on the joint which is connected

to the link or base. The basic topology given by Figure 2.2 is a Base – Joint 4 – Link 1-

Joint 10 – Link2 – Joint 4 – Gripper manipulator where "Joint 4" refers to a particular

variety of joint module stored in the library.

Figure 2.1: Object Incidence Matrix (Bi 2002)

	<Joint 4>	<Joint 10>	<Joint 4>
<Base 1>	<1,0,1,2>	0	0
<Link 1>	<1,1,1,8>	<1,0,2,2>	0
<Link 2>	0	<1,1,1,6>	<1,0,2,1>
<Gripper 1>	0	0	<1,1,1,7>

In this case, the elements of the OIM become the variables for use in the design of

the manipulator as it contains all the relevant information about the assembly

configuration. Information about each of the joint, link, and gripper modules is found by

use of a look-up table or similar data structure. Both Bi and Chen used these matrix

formulations as the starting point for generating kinematic and dynamic models.

This thesis has taken a similar approach to Bi and Chen, using a matrix describing

the number, type, orientation and connections between adjoining modules as the input

variable for the process of manipulator design and evaluation. This matrix, termed the

Assembly Matrix is presented in greater detail in Chapter 3, but all of the design variables

are essentially contained within the assembly matrix.

2.2.3 Design Evaluation

In order to determine which modular manipulator configuration is "best" for a given task, it is necessary to define the objectives for a given manipulator. Evaluation of a manipulator's performance typically follows only after it has first been shown not to violate any of the constraints imposed on it (joint angles, workspace boundaries, obstacle collision, exceeding actuator performance in force or torque, etc.). Once a modular manipulator is shown not to be in violation of any of these key requirements, the evaluation of its performance can proceed. This performance evaluation is most often dependent on the task being performed by the manipulator, and researchers exploring this problem have used a number of different performance indicators.

Manipulability, or the ability of the end-effector to move in an arbitrary direction at a given point, is a measure used often for evaluating the effectiveness of a design. A manipulator that maintains good manipulability along a given trajectory tends to be far from its singularities and is generally well-behaved in regards to controllability. Han (1997) and Chen (1995) used this as the sole measure for evaluation of the performance of a manipulator.

Chocron (1997) in contrast, rolls all of the constraints and performance criteria into one large objective function that induces a very large penalty for violation of a constraint but also includes a manipulability measure and a parameter for roughly estimating relative masses (based on arm length and number of joint actuators). Each of the terms in the objective function (reachability, linear and angular distance between

desired and actual pose, object proximity, number of modules, and manipulability) is then weighted by a scalar constant. Adjusting the ratio between these constants allows the various parameters to be given more or less weight as the situation dictates.

Paredis (1997) used the total energy expended by the manipulator in moving along its defined trajectory to be the objective function in his evaluation of fault tolerant manipulators. In this evaluation, he found that as long as several designs could be found that did not violate the constraints, that the difference between these multiple designs was 'acceptably small' in terms of the energy consumed. Bi (2002) followed this precedent and used energy expended as the objective function to be minimized, reasoning that the principle cost of operating a robot is in the amount of power consumed. This reasoning does not apply, however, across all fields of robotics. While power consumption may be the principle source of cost for a robot in an industrial setting, in on-orbit servicing, the cost of launching the robot to the worksite is the principle cost driver, making the mass of the robotic system far more important than the amount of power it consumes.

Several other studies such (Chen and Yang 1999, Paredis and Kholsa 1995) presented generic methods for employing an objective function in reconfigurable manipulator design, but no specific method for evaluating an objective function was presented.

The objective function used in this thesis, which is presented in detail in Chapter 4, is based on the estimated mass of the manipulator capable of performing each of the tasks and the manipulability of the manipulator. These two factors are linearly weighted by two constants, with the goal of achieving a 90%/10% weighting between the mass of the manipulator and its manipulability. Mass was selected as the primary weighting factor

as it is the most important parameter for many orbital systems, and plays the biggest role in determining the launch cost of a system, all other things being equal. Manipulability was maintained as part of the objective function as it is a measure of the system's performance, and helps to keep the optimizer from choosing configurations that would require the manipulator to approach a singularity in completing the specified task.

2.3 Random Search Methods

As discussed at the beginning of this section, the search space for all possible manipulator configurations is prohibitively large for evaluating all possible solutions, even if solutions that are unacceptable can be eliminated early on in the evaluation process. Additionally, the strong coupling between modular assembly and manipulator dynamics and kinematics results in a very poorly conditioned search space, and traditional optimization methods such as the Newton-Raphson Method tend to fail. Therefore, because the search space is too large to examine exhaustively and too poorly conditioned to evaluate through the use of derivatives or perturbation methods, a randomized search of the space seems to be the best option. Several algorithms have been used for performing these randomized searches, though simulated annealing and genetic algorithms seem to be predominantly used in of modular robotic manipulator design.

Paredis and Kholsa (1995) proposed using a simulated annealing procedure for introducing some randomness to the search process. Like many other algorithms, it works by randomly varying the design parameters and checking the

objective function. If the objective function is improved (usually by getting smaller), then the new variables are kept and the old ones discarded and the algorithm continues. However, sometimes a worse value of the objective function is accepted and kept by the algorithm allowing the process to occasionally move "uphill" and to avoid becoming stuck in a local minimum. Whether or not the change in variables is accepted or discarded is defined by to Equation 1.1. In a simulated annealing algorithm, the "Temperature" is gradually reduced over time, resulting in fewer random motions until eventually the solution becomes "frozen".

$$\begin{cases} \Delta F_{Obj} \leq 0 \Rightarrow accept \\ \exp(-\Delta F_{Obj}/T > random[0,1] \Rightarrow accept \end{cases} \quad (2.1)$$

Genetic algorithms (GA) seem by far the most common method for evaluating modular robots and have been applied in a number of different studies for the purposes of generating and evaluating robotic configurations. Only Paredis (1997) provided a performance comparison between a GA and multiple restart statistical hill-climbing (MRSH), a method of starting a more traditional optimizer at various points in order to avoid finding local minima (or maxima as the case may be). Just as simulated annealing takes its inspiration from the process of crystal formation in a cooling liquid, a GA looks to evolutionary biology for its intuition. The process works by creating an initial population of designs to be evaluated and then 'breeding' the fittest individuals in the population and throwing away the least fit. Design variables are encoded in binary 'chromosomes' that are subjected to recombination and mutation with each additional generation. This process helps to choose the robotic configurations of the next generation that are most likely to have superior performance while the randomized nature of the

18

mutation function helps to ensure that the algorithm will not become stuck at a local minimum which is far from the global optimum. The GA itself includes characteristics such as population size, the way the best and worst members of the population are handled, and the mutation rate, which are not typically documented in the literature.

While the fundamentals are basically the same, there are key differences in the implementation of a GA for performing the search. Paredis and Kholsa (1997) made use of an agent-based approach where the different functions of a GA were decomposed to facilitate parallel processing. Chen (1995) developed a method for encoding his AIM representation of design variables into a binary string (See Section 2.4.2). Using his similar OIM formulation of design variables (Figure 2.2), Bi (2002) explained how the OIM could be decomposed for use in the GA Toolbox developed for the Matlab computing environment by Chipperfield (1994). Functions from the GA Toolbox are used to carry out a GA optimization in this thesis, and the details of the implementation of that optimization are described in Chapter 4.

2.4 Summary

The review of literature on modular and reconfigurable manipulators has provided the starting point for the research presented in this thesis by helping to define the problem, model the manipulator kinematics, and show methods for optimizing over large, poorly conditioned search spaces. Furthermore, the review of the literature has helped to rule out several research directions, in particular, the principles of Axiomatic Design Theory. Though modular systems in other realms have been studied and codified in the

past, these principles do not apply to modular robotics, as there is no independence between the input design variables and the output design variables.

In the next chapter, the kinematic modeling procedure is outlined in detail. The automated modeling method presented is required to move from an arbitrary serial assembly of modules to a set of Denavit-Hartenberg parameters formed by the connections of those modules.

Chapter 3: Automated Kinematic Modeling

This chapter outlines a process for automatically modeling the kinematics for an open-chain modular manipulator. The process follows three basic steps: first, properly characterize the constituent modules, second, assemble these modules into an appropriate chain, and third, derive the Denavit-Hartenberg (D-H) parameters for the entire manipulator. This procedure was implemented in Matlab as the function *Assembly to Denavit-Hartenberg*, abbreviated as *Assy2DH*, and is the focus of this chapter.

The development of the capacity to automatically derive a set of D-H parameters given a chain of assembled modules is motivated by a need to examine large numbers of potential manipulator combinations, while at the same time expressing the forward kinematics of those manipulator configurations within a widely utilized convention. The module level assembly of the manipulator was chosen as the starting point for this procedure, as it is intuitively easier to build a robotic configuration and derive its D-H parameters than to begin with a desired set of D-H parameters and determine which assemblage of modules will generate them. Furthermore, using the D-H convention (as opposed to other methods) allows analysis tools that have been developed in the past, such as the Matlab Robotics Toolbox (Corke, 2001).

3.1 Module Assembly Representation

The modular reconfigurable robotic system of this thesis is inspired by the MORPHbots architecture proposed by the University of Maryland Space Systems Laboratory (SSL). The MORPHbots system, introduced is discussed in greater detail in

21

Section 2.1, has the important characteristics with regards to the generation of manipulator configurations:

- *Any two modules can be connected together.*

- *Each module composing the assembly has two ports of connection.*

- *The ports can be connected in four distinct ways by rotating the two adjacent modules 90º relative to each other around the line normal to the interface plate.*

- *Modules range from 0-DOF links up to 3-DOF.*

Although the MORPHbots architecture is generic enough to provide for robotic systems with multiple or branching structure manipulators, the configurations considered in this thesis are restricted to be single-chain serial manipulators. While this restriction limits the number of potential configurations examined, it also helps to keep the comparisons between non-reconfigurable manipulators and reconfigurable manipulators more clear. Though multi-armed and branching structures are not evaluated in this thesis, many of the techniques presented later in this thesis (kit determination and genetic algorithm optimization in particular) could be used to examine these more complicated structures.

Several constraints are imposed upon the construction of a modular manipulator. It is assumed that the first module is always connected to a base, and that the last module is considered to possess an end-effector. In between these two ends, any two modules may be connected together. That is to say, a joint module may connect either to a link module or to another joint module, and a link module may connect either to a joint module or another link module. Using this formulation of assemblies is more general than making the assumption that links may only attach to joints.

22

Four numbers are used to characterize each module in a configuration. The first number used indicates the number of degrees of freedom in the module (0,1,2, or 3), indicating what class the module falls into. The second number indicates for a given module DOF, the "part number" of the module in the library. There is no particular logic to the assignment of these part numbers; they merely serve to indicate which part should be used from the library of available modules. The third number indicates which port on the body is connected to the preceding module; 0 for the output port and 1 for the input port. Finally, the fourth number indicates the orientation of the two bodies relative to each other; 0 meaning the local x-axes of the two parts are aligned, 1 meaning there is a positive rotation of 90° about the z axis between the two parts, 2 for 180° and 3 for 270°. In principle, the method outlined developed in this chapter is general enough to handle joints of any orientation. The four variables form a column of the assembly matrix, which has the dimension 4xm where m is the number of modules in the assembly.

For example, the following assembly matrix codes for the manipulator represented schematically in Figure 3.1.

$$
\begin{array}{r}
DOF \\
Part \\
In/Outport \\
Rotation*90^O
\end{array}
\begin{bmatrix}
1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 \\
1 & 1 & 2 & 2 & 2 & 1 & 2 & 3 \\
1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0
\end{bmatrix}
$$

*Module 2: Link Module*   *Module 7: Joint Module*
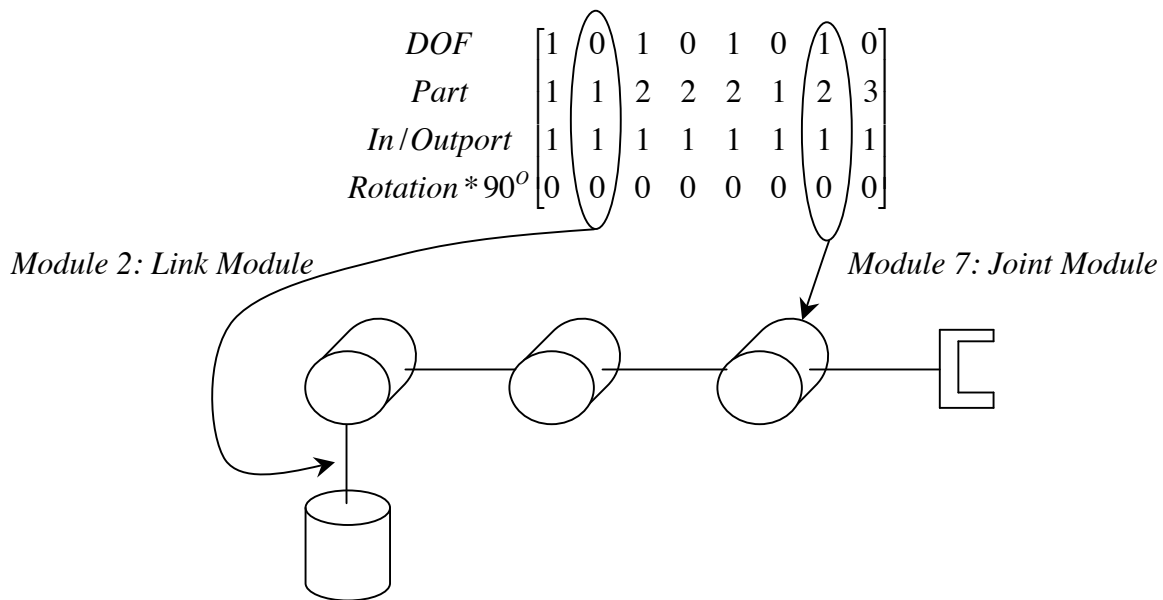
Figure 3.1: Converting an Assembly to a Manipulator

Here, the first column indicates that the first element of the manipulator is 1-DOF joint that is attached to the base using the input port (3rd row) of part number one (2nd row), and the local x-axis on the input face of the module is aligned with the base's x-axis (4th row). Note that modules of differing categories can share a part number, as the part libraries for different categories are kept separate. Thus, there is link part #1 and a 1-DOF joint part #1.

3.2 Module Characterization

Each module to be used in the building of a modular manipulator must be characterized in a standardized fashion in order for an automated procedure for developing the manipulator kinematics to work properly. Of primary importance to characterizing the modules is defining a local module coordinate system and the location and orientation of the two connection ports relative to the coordinate system of the module. After the modules are properly characterized, then for a serial chain of the modules, the location and orientation of each of these modules in the global coordinate system can be calculated. Eventually this information can be used to develop the Denavit-Hartenberg (D-H) parameters for the entire arm.

In this thesis, three different categories of modules were considered: links, grippers and joints. While the field of end-effector design is one of fundamental importance to the field of robotics the actual functioning of the end-effector is of

marginal importance. Therefore, the distal end of the final module in the chain is

considered to be the tool tip of the module chain.

3.2.1 Link Modeling

Links are treated as rigid-body connectors for the development of the robotic

kinematics, and serve to connect adjacent joints to each other. Various types of joints

exist, including straight links, corner links and twisted links, but with regards to

developing the D-H parameters what is most important is the spatial relationship between

the link's distal port and proximal ports. For links, the frame origin is always defined to

be at the center of the link's input port, and has the z-direction defined to be into the link.

While the terminology of input port and output port are used to refer to each of the

module's two ports, in practice, either one may be used as the distal or proximal

connector; the names only indicate their relationship to the coordinate system origin of

the modules. Once the input port is defined, the output port can be defined in relation to

it using a standard 4x4 transformation matrix of the following form:

$$
L_T = \begin{bmatrix} \hat{X}_{Out} \cdot \hat{X}_{In} & \hat{Y}_{Out} \cdot \hat{X}_{In} & \hat{Z}_{Out} \cdot \hat{X}_{In} & {}^{In}P_{XOut} \\ \hat{X}_{Out} \cdot \hat{Y}_{In} & \hat{Y}_{Out} \cdot \hat{Y}_{In} & \hat{Z}_{Out} \cdot \hat{Y}_{In} & {}^{In}P_{YOut} \\ \hat{X}_{Out} \cdot \hat{Z}_{In} & \hat{Y}_{Out} \cdot \hat{Z}_{In} & \hat{Z}_{Out} \cdot \hat{Z}_{In} & {}^{In}P_{ZOut} \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{3.2}
$$

where ${}^{In}P_{Out}$ is the vector describing the location of the output port in the reference frame

of the input port and the upper-left 3 x 3 of the matrix are the direction cosines (Craig,

1986). The last row of the matrix is necessary to make it square, though it contains no useful information.

At the output port, the z-direction is defined to be pointing away from the link body, and the x-axis is defined to be in the same direction as the x-axis of the input port, or in the direction of the cross-product between the z-axes of the input and output ports.
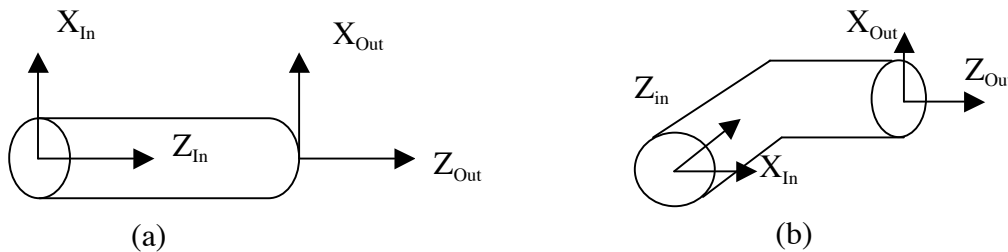

Figure 3.2: Link Module Frame Assignments

As mentioned in the introduction to this chapter, end-effectors were modeled in the same manner as the links, and were assumed to be the terminus of the arm. Two types of link topologies were included in the kit used in this thesis,:a straight link and a corner link. Various sizes of these two links were made available and the complete listing of them can be found in Appendix A.

3.2.2 Joint Modeling

In order to kinematically model a single degree of freedom rotary joint module with two possible input ports, it becomes necessary to define an origin, the axis of motion, and the relation of both ports to the origin. The origin is defined to be on the

plane separating the two parts of the joint module and on the line about which the two points pivot. The z-axis of the module is then aligned with the motion axis as shown in Figure 3.3. Affixing the frames for the input port and the output port follow the same convention as outlined for the links, where the z-axis is directed into the body for the input port and out of the body for output port.

For each of these cases, the input port and the output port are expressed as transformations from the joint module origin and are described by the same 4x4 transformation as the link frames. While each of the ports is labeled as an input port or an output port, it is in fact possible to assemble any one of these modules backwards if desired.


(a) Pivot Joint (Offset)

(b) Roll Joint

(c) Pivot Joint (No offset)
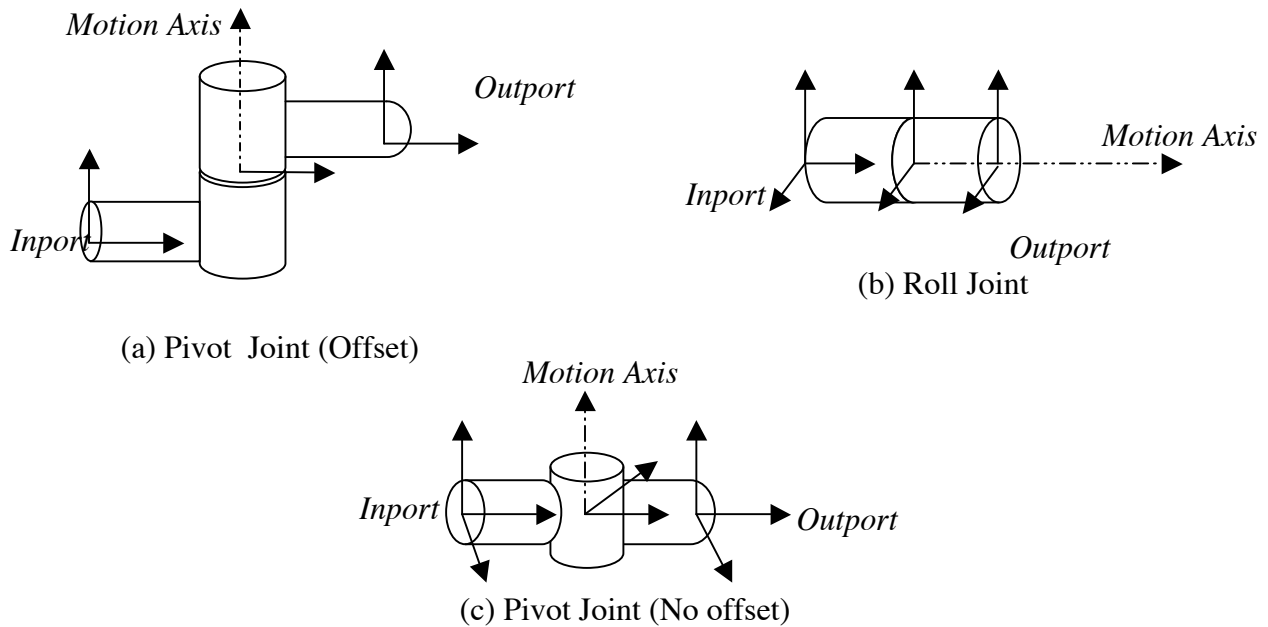
Figure 3.3 Example Joint Frames

For multiple DOF modules such as a roll-pitch module or a spherical wrist module, more frame transformations are needed to describe the position of the motion axes with respect to the frame origin. Each of these transformations is stored as a component in an element in a Matlab cell structure. The complete structure containing all available modules becomes the library listed in Appendix A.

Characterizing the various modules in this way provides a method sufficiently general for later conversion of a given manipulator assembly into an appropriate D-H parameter set, provided that the representation of the assembly is capable of accurately representing the order of the modules to be assembled and the port on each of the modules that is to be connected to the preceding module.

3.3 Calculation of Denavit-Hartenberg Parameters

Once an assembly of modules has been specified using properly characterized modules, it becomes possible to determine the Denavit-Hartenberg (D-H) parameters for the manipulator uniquely determined by the chain of modules. While it is certainly possible to represent the kinematics of the manipulator in some other form, the motivation for using the D-H convention is two-fold. First, it is the most widespread and commonly used method for representing robot kinematics thanks in part to its simplicity. Using a well-known convention makes the results of any research immediately more accessible to others. Second, using D-H parameters allows analysis tools developed by others for use with conventional robotics to be employed even for a modular robotic

system. In the case of this thesis, the Robotics Toolbox for Matlab (Corke, 2001) is used for further analysis on a given configuration of a modular robot.

The Denavit-Hartenberg method was first proposed in 1955 as a method for representing adjoining link elements by means of a 4x4 transformation (Denavit and Hartenberg, 1955). The D-H method was developed into an architecture for robotics by Paul (1981) and is referred to as the "standard" D-H convention. A "modified" convention was proposed by Craig (1986). Though both the standard and the modified convention are quite similar, it is necessary to know which method is being employed to avoid confusion, and the distinctions become crucial when developing a method to automatically generate them for given modular robotic configuration. In the case of this thesis, the standard convention is used throughout, though this choice is rather arbitrary so long as consistency is maintained.

In the language of the D-H convention, any serial manipulator consists of a series of links in a serial chain connected together by joints. Each of these joints has only one degree of freedom (DOF) and joints that have more than one degree of freedom are broken down into single DOF units. A serial manipulator that has *n* joints will have *n+1* links associated with it. The first unit of the chain is always the base link, Link 0, and links may only attach to joints and *vice versa*. The ith joint body is always the connector between link *i-1* and link *i*. It is vital to note that though the D-H convention allows only one DOF per joint body, a D-H joint body is not the same thing as a joint modules as understood in the context of this thesis.

The position and orientation of one link with respect to the next link in the chain can be represented with four parameters, two of these being properties of the link and two

29

of them properties of the joint that connects the two adjacent links. Links are defined by their length (given the name *a*), and also by the relative twist between its two axes (α), while joints are described by the distance along the axis of motion that the two links are from each other (link offset, *d*) and the angle between the two links (joint angle, θ).

In order to generate these parameters for each link, it is common to first assign coordinate axes for each joint body. Using the standard convention, the axis z_{i-1} is assigned to be along the motion axis of joint body *i* (in the modified convention z_i is assigned to be along the axis of joint *i*). The x_{i-1} axis is then assigned to be in the direction of the cross-product $z_{i-1} \times z_i$ in the case of intersecting *z* axes, or else in the direction of the normal between *z* axes. Assigning the axes automatically is the thrust of the next section of this chapter, where the details of determining the location and orientation of the D-H joint bodies from the location and orientation of the joint modules will be discussed. Although the basics are mentioned here, as the background is a necessary part of the *Assy2DH* function.

After the axes are assigned, the D-H parameters can be assigned by a recursive formulation presented in Corke (2001):

Table 3.1: DH Parameters

Link length	a_i	offset distance between z_{i-1} and z_i along the x_i axis
Link twist	α_i	angle from the z_{i-1} axis to the z_i axis about the x_i axis
Link offset	d_i	distance from the origin of frame i-1 to the x_i along the z_{i-1} axis
Joint angle	θ_i	angle between the x_{i-1} and x_i axes about the z_{i-1} axis

Either the parameter θ_i or d_i is referred to as the joint variable depending on the joint type. For revolute joints, θ_i is the variable and d_i is constant, whereas for prismatic joints, d_i is the joint variable and θ_i is constant.

Figure 3.4: Denavit-Hartenberg Parameters
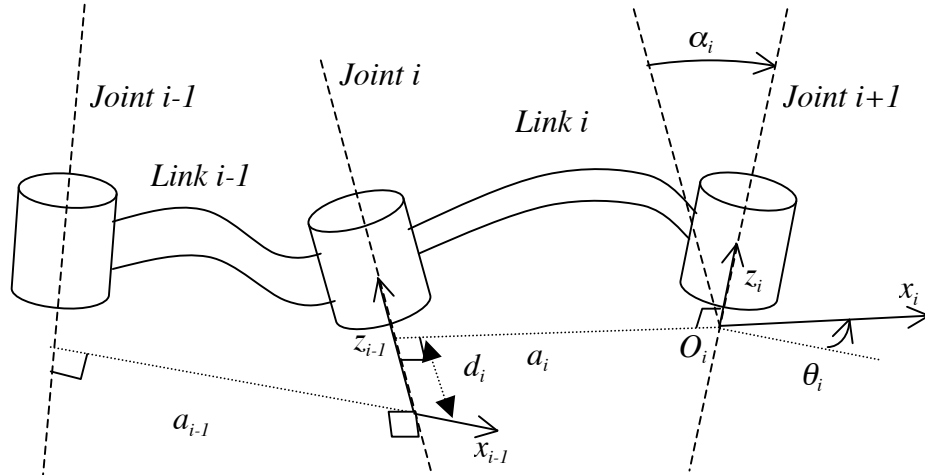
After the complete set of D-H parameters for a given manipulator is defined, it becomes possible to express the transformation between the coordinate system *i-1* and *i* as:

$$^{i-1}T_i = \begin{bmatrix} \cos\theta_i & -\sin\theta_i\cos\alpha_i & \sin\theta_i\sin\alpha_i & a_i\cos\theta_i \\ \sin\theta_i & \cos\theta_i\cos\alpha_i & -\cos\theta_i\sin\alpha_i & a_i\sin\theta_i \\ 0 & \sin\alpha_i & \cos\alpha_i & d_i \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad (3.2)$$

For a manipulator of *n* degrees of freedom, there will be *n+1* transformations required to calculate the position and orientation of the end-effector of the manipulator in the base frame. Multiplying this chain of transformations together yields:

31

$$^{Base}T_n = \left[^{Base}T_0 \right]\left[^0T_1 \right]\left[^1T_2 \right] \cdots \left[^{n-1}T_n \right] \quad (3.3)$$

Equation 3.3 is essentially what can be described as the forward kinematics of the manipulator, because given a vector of joint variables, q, where the elements of q are either joint angles or link offsets depending on the joint type, the position and orientation of the end-effector are completely determined. It should be noted that the forward kinematics of a given manipulator configuration are already known from the link and joint transformation matrices of the modules comprising the manipulator assembly. Although D-H based forward kinematics are not required to solve the forward kinematics of the manipulator arm, they are mentioned here for the sake of completeness.

It should be noted here that while the D-H parameters require a link-joint-link pattern for the kinematic chain, the MORPHbots modular robotic system does not require this, allowing the linking of two joint modules, or two link modules together. However, creating virtual link bodies can be created between any two adjacent joint modules, and combining any two adjacent link modules into one large link body easily circumvent this problem. Confusion may arise, however, with what the D-H convention terms a "link body" or a "joint body," since these do not align with what we have defined as a "link module" or a "joint module". In order to prevent this confusion, the full terms "link body" and "link module" will be used rather than simply using "link" or a "joint".

3.4 Automatic Denavit-Hartenberg Parameter Assignment

The following section outlines a procedure that was developed in Matlab for automatically identifying the Denavit-Hartenberg (D-H) parameters for a given

configuration of joint and link modules. The procedure, called "Assembly to D-H", or *Assy2DH* for short, effectively runs three loops in series. First, it loads the properties of the various joint and link modules from the inventory and labels the joint bodies. Second, it assigns the *z*-axes by identifying the orientation of the joint body directions of motion (up to three axes per joint module, depending on its degrees of freedom). Third, the procedure identifies the origins and *x*-axes of the joint body frames and calculates the D-H parameters. It should be noted that the third step is the most complicated and requires the determination of the relationship between adjacent motion (z) axes as skew, parallel or intersecting. The input to this function is the assembly matrix (see Figure 1), and the output is a matrix of D-H parameters characterizing the arm.

The development of the *Assy2DH* procedure is a critical step in allowing the analysis of large numbers of potential manipulator configurations. Without the ability to convert from the assembly matrix characterization of arbitrary module connections to a D-H characterization of the arm, none of the tools built with the D-H convention as their basis could be applied.

3.4.1 Loading Module Information

In order to assign joint axes, link frames, or D-H parameters for a given configuration, it is necessary to decode the *Assembly* matrix and read in the appropriate information regarding the modules that make up the manipulator. During this loading procedure, any joint module having more than one degree of freedom (DOF) is broken up into a series of 1-DOF joints and virtual links in order to better accommodate the alternating link-joint-link structure of the D-H convention. As the properties of each

module are read into the *Assy2DH* function, they are stored sequentially in a new combined *Link-Joint* vector whose length is equal to the number of non-joint modules plus the total DOFs. It should be noted that the original *Assembly* matrix also contains this same information, however, it was not decomposed into single DOF elements.

3.4.2 Assigning *z*-axes

Each element in the *Link-Joint* vector is now tagged as being "forward" if its input port is the proximal connector or "backwards" if its output port is the proximal connector (row 3 of the assembly matrix). The module's degree of rotation is indicated as well (row 4). At this point all the necessary information is available to begin locating and assigning each of the joint bodies a *z-axis*. This is simply a matter of starting at the base of the manipulator and continuing up the chain of bodies as defined in the *Link-Joint* vector until finding a joint module.
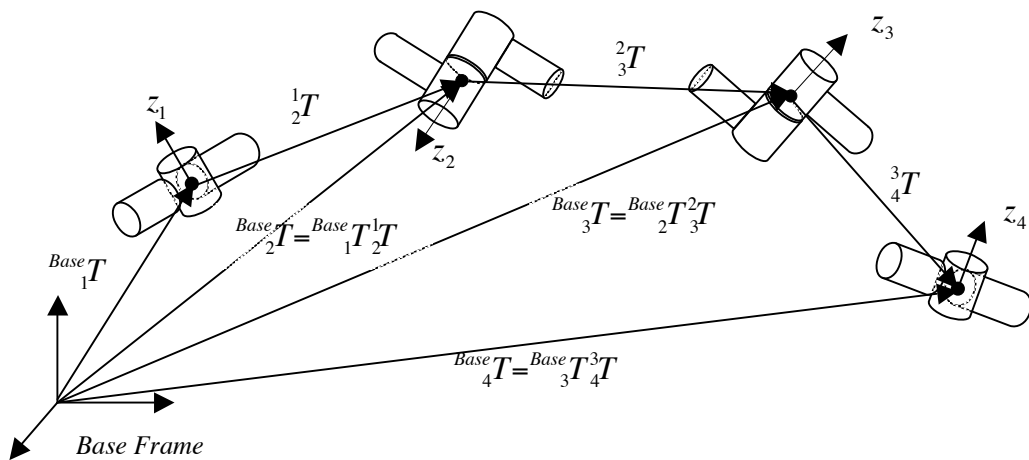

Figure 3.5: Z-axis Determination

The transforms for each of the intermediate link bodies are all rolled into a single linear transformation that describes the location and orientation of the current joint body relative to its predecessor joint body expressed in the global coordinate system. This can be summarized by

$$^{Base}_{i}T = {}^{Base}_{i-1}T \, {}^{i-1}_{i}T \quad (3.4)$$

and is essentially equivalent to solving the forward kinematics of the manipulator. Multiplying again by the transformation from the input port of the joint module into its local module coordinates yields the location of a point that its *z-axis* passes through (the joint module origin), as well as the direction of this axis (orientation of the *z-axis* with respect to the base frame origin). It is important to note, that the *x-* and *y-axis* of each joint module coordinate frame is known as well, but this information proves to be unimportant, as only the *z-axis* of the joint module will coincide with the link frames used by the D-H parameters.

3.4.3 Assigning Frame Origins, *x*-axes and D-H parameters

From prior steps, the position and orientation of each of the joint module local coordinate frames is known in the global coordinate system. Because a point (the local origin of the joint module) are defined and a unit vector in the direction of the line (the joint motion axis), the entire line of the *z-axis* is also defined in three-dimensional space, though its origin in the global frame and the direction of its *x* and *y axes* is as yet unknown. The following paragraphs discuss how the frame origin and the direction of the x_i-*axis* are assigned for each of the motion axes depending on the relationship

(a) Parallel Lines   *(b) Intersecting Lines*   *(c) Skew Lines*
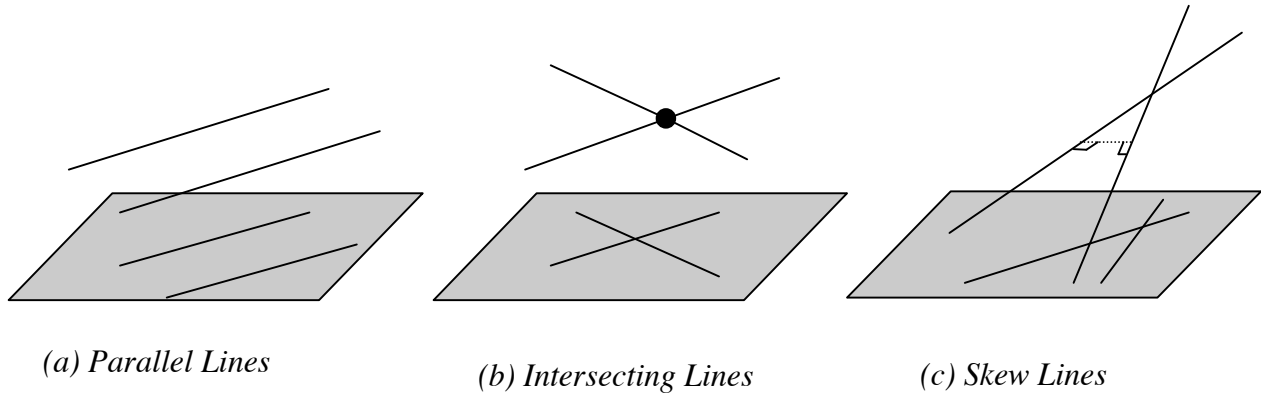
Figure 3.6: Lines in 3-Space

between the z_i and z_{i-1} axes. Once frames and axes are assigned, calculating the D-H

parameters for frame i becomes straightforward.

For two distinct lines in three-dimensional space, there are three possibilities for

the orientation of these lines with respect to each other. They may be *parallel*, in which

case the distance between any point on the first line and the point closest to it on the

second line is always constant. They may also be *intersecting*, if there is one common

point shared by both lines. Or the third possibility is that the two lines may be *skew* in

which case, they neither intersect nor share a direction. If projected onto a two-

dimensional plane, both skew lines and intersecting lines will appear to intersect. When

examining at the *z-axes* of two adjacent joint modules, each of these three cases must be

considered separately in order to properly assign the origins of the joint body frames and

to calculate the D-H parameters that describe the relationship in space between the joint

bodies. Note that D-H joint bodies are not the same thing as joint modules, as discussed

above.

Parallel Axes

If two adjacent *z-axes* are parallel to each other, then their cross product will be zero. Since the direction of the normal between these two lines (and therefore the location of the origin) is not uniquely defined, the location of the origin of *frame i* is arbitrarily set to be at the origin of the joint module through which the motion axis passes. Then, the direction of the x_i-*axis* is assigned to be in the direction of the normal from axis z_{i-1} to axis z_i.

The D-H parameter a_i is now just the distance between the z_{i-1} and z_i axes along the x_i axis as stated above in *Table 3.1*. The other D-H distance parameter, d_i, can be found by using the Pythagorean theorem as the distance between the centers of the axes is already known (represented in Figure 3.7 as c). The twist angle α_i, can be calculated by
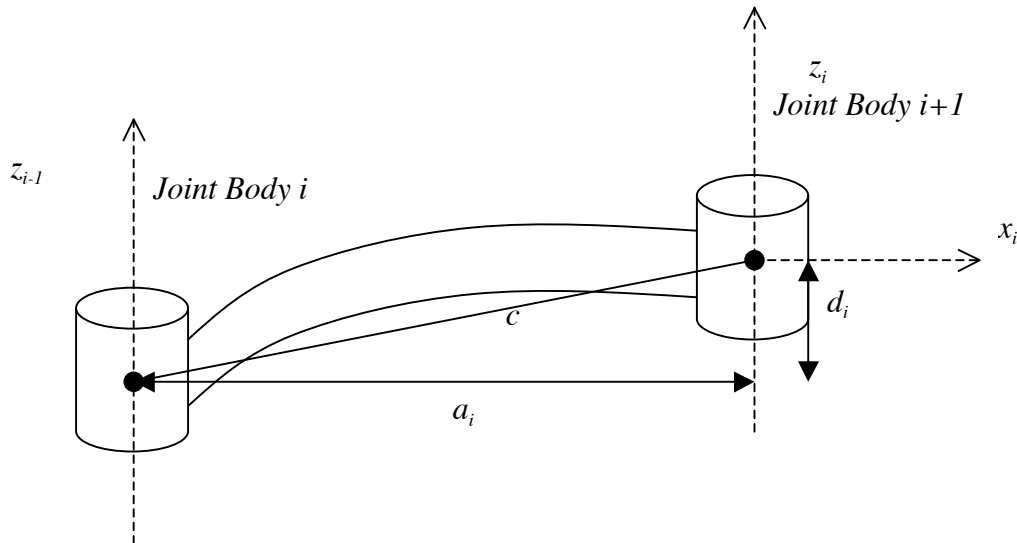

Figure 3.7: Parallel Axes

taking the inverse cosine of the vector dot product between z_{i-1} and z_i unit vectors.

Similarly, θ_i is the inverse cosine of the vector dot product between x_{i-1} and x_i unit vectors.

For rotational joints, θ is the variable of joint motion as opposed to a, d, and α which are

constant for a given configuration.

$$\theta_i = \cos^{-1}(\hat{X}_i \bullet \hat{X}_{i-1}) \qquad (3.5)$$

$$\alpha_i = \cos^{-1}(\hat{Z}_i \bullet \hat{Z}_{i-1}) \qquad (3.6)$$

Intersecting Axes

If axes z_{i-1} and z_i intersect, then the origin of *frame i* is assigned to be at the point

of intersection, with the x_i-*axis* oriented along the normal between the z_{i-1} and z_i axes.

The x_i axis is assigned to be in the direction of the cross product between z_{i-1} and z_i. In

order to determine whether the two axes intersect, the cross product between z_{i-1}
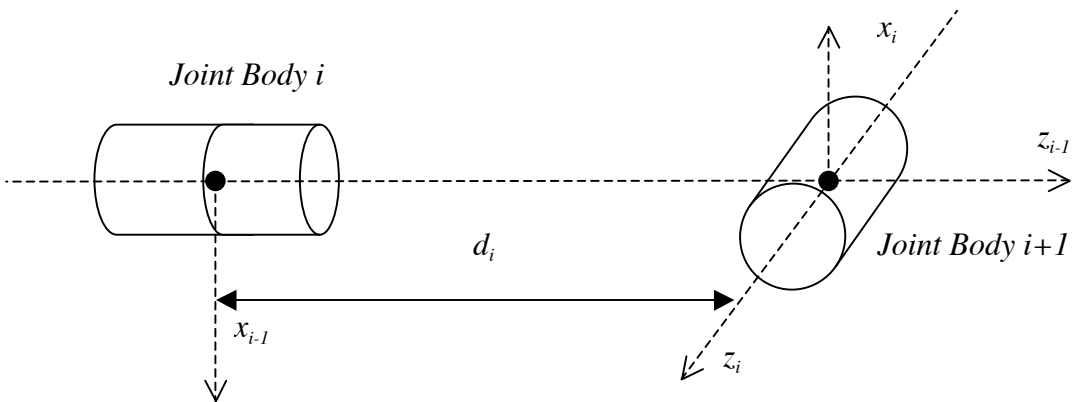

Figure 3.8: Intersecting Axes

and z_i and then the vector dot product of the result with a line, c, connecting the centers of the joint bodies $|c \cdot (z_{i-1} \times z_i)|$. If the value of this expression is zero, then the two axes *must* intersect. Since they intersect, the distance between them is zero and therefore a_i is zero. The parameter d_i is simply calculated as the distance between the origins of frame *i-1* and *i,* already already calculated as c. The two angular D-H parameters are again calculated according to (*3.5*) and (*3.6*).

Skew Axes

In the case of skew axes, assigning the direction of the *x*-axis is quite straightforward, though determining the location of the frame origin is a bit more involved. Just as in the case of intersecting axes, the *x*-axis direction is simply the cross product between z_{i-1} and z_i. In order to determine the location of the origin of frame *i,* the location of the two points of closest approach on the z_{i-1} and z_i axes must be determined. Two generic *z* axes, z_1 and z_2 are shown in Figure 3.9, with points P_1 and P_2 on z_1 and P_3 and P_4 on z_2.
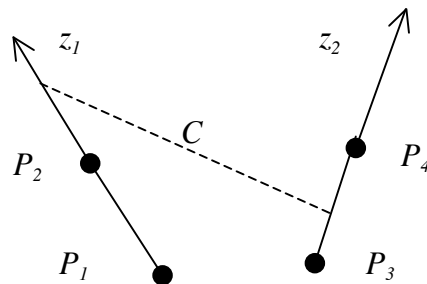

Figure 3.9: Arbitrary line between two axes

Given that the equations of the lines defining the z_{i-1} and z_i axes can be expressed as:

$$z_{i-1} = \begin{bmatrix} P_{1x} \\ P_{1y} \\ P_{1z} \end{bmatrix} + \begin{bmatrix} P_{2x} - P_{1x} \\ P_{2y} - P_{1y} \\ P_{2z} - P_{1z} \end{bmatrix} s_1 \qquad (3.7)$$

$$z_i = \begin{bmatrix} P_{3x} \\ P_{3y} \\ P_{3z} \end{bmatrix} + \begin{bmatrix} P_{4x} - P_{3x} \\ P_{4y} - P_{3y} \\ P_{4z} - P_{3z} \end{bmatrix} s_2 \qquad (3.8)$$

where s_1 and s_2 are constants. Note that the x,y,z subscripts denote the vector components in the global, base frame. From (3.7) and (3.8), a general equation for a line connecting any point on z_1 with any point on z_2 is:

$$C = z_i - z_{i-1} = \left(\begin{bmatrix} P_{1x} \\ P_{1y} \\ P_{1z} \end{bmatrix} + \begin{bmatrix} P_{2x} - P_{1x} \\ P_{2y} - P_{1y} \\ P_{2z} - P_{1z} \end{bmatrix} s_1 \right) - \left(\begin{bmatrix} P_{3x} \\ P_{3y} \\ P_{3z} \end{bmatrix} + \begin{bmatrix} P_{4x} - P_{3x} \\ P_{4y} - P_{3y} \\ P_{4z} - P_{3z} \end{bmatrix} s_2 \right) \qquad (3.9)$$

If C is set to be parallel to the direction of the x_i axis

$$\frac{C_x}{\hat{x}_{ix}} = \frac{C_y}{\hat{x}_{iy}} = \frac{C_z}{\hat{x}_{iz}} \qquad (3.10)$$

then

$$\begin{bmatrix} P_{2x} - P_{1x} & P_{3x} - P_{4x} \\ P_{2y} - P_{1y} & P_{3y} - P_{4y} \\ P_{2z} - P_{1z} & P_{3z} - P_{4z} \end{bmatrix} \begin{bmatrix} s_1 \\ s_2 \end{bmatrix} = \begin{bmatrix} \hat{x}_{ix} + P_{3x} - P_{1x} \\ \hat{x}_{iy} + P_{3y} - P_{1y} \\ \hat{x}_{iz} + P_{3z} - P_{1z} \end{bmatrix} \qquad (3.11)$$

A system of three equations and two unknowns (s_1 and s_2) thereby result, and solving the system is a straightforward matter of eliminating the redundant equation. After s_1 and s_2
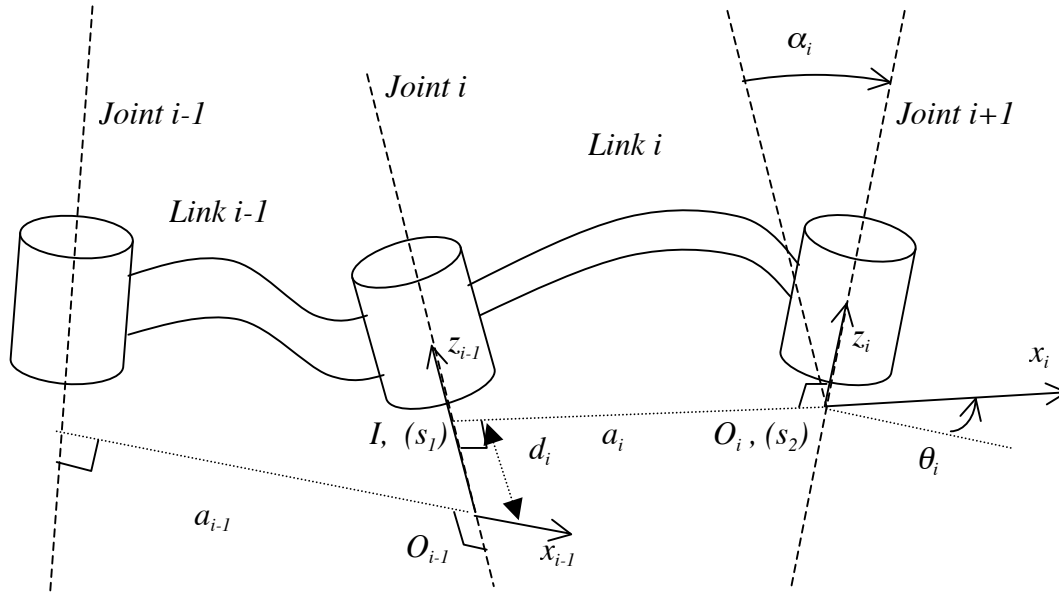

Figure 3.10: Skew Axes

have been solved, then it is a simple matter of plugging them back into (*3.7*) and (*3.8*) in order to find the points of closest approach on the z_{i-1} and z_i axes. The origin of frame *i*, O_i, is set to be at the point of closest approach on the axis of joint *i+1*.

Since the two closest points on both the axes are known, the D-H parameter a_i is simply the length of the line between these two points (labeled as *I* and O_i in Figure 3.10). The parameter d_i is obtained by finding the distance from the origin of frame *i-1* to point *I*. As always, α_i and θ_i are obtained by using (*3.5*) and (*3.6*).

3.5 Summary

In this chapter, a method for encoding the serial connections between kinematically modeled joint and link modules was developed and termed the *assembly*

41

matrix. Additionally, a method for automatically generating the D-H parameters for a manipulator from its assembly matrix was developed and codified in Matlab as the function *Assy2DH*. The choice of the D-H convention to represent the manipulator described by the assembly matrix was made for its broad acceptance in robotic manipulator applications, and the existence of many analysis tools that take D-H parameters as inputs, such as the Matlab Robotics Toolbox. The ability to automatically generate a set of D-H parameters from a matrix containing the information of how the various joint and link modules are connected to each other will prove to be an important component in automatically generating and evaluating large numbers of manipulators for performing task-based optimization using a genetic algorithm.

Chapter 4: Modular Manipulator Generation and Evaluation

This chapter addresses the potential configuration space for a manipulator generated from a library of modules, and demonstrates that the number of potential combinations is too large for performing an exhaustive search. The generation of modular manipulators from the library of modules is addressed and a method for finding task-based near-optimal solutions by of these manipulators by use of a genetic algorithm is described. The details of the objective function used to evaluate the fitness of a given manipulator configuration in the performance of a quasi-static task are highlighted.

The goal of the generation, evaluation and evolution of manipulator configuration solutions is to answer the question: "Given a task and a library of modules, what is the lightest (lowest mass) manipulator that can be constructed to achieve the task?" If this question can be answered reliably, then it will be possible to automatically evaluate any possible set of manipulator modules against any possible set of desired tasks. This flexibility will allow single-point conventional solutions to task-based designs to be broken down and compared to modular reconfigurable solutions, which is the topic of Chapter 5.

4.1 Manipulator Configuration Space

In this thesis, manipulators are generated based on a pre-defined library of potential parts that can be selected and connected in series to build up a manipulator arm. Therefore, since there are a finite number of parts in the module there is a finite number

of ways in which these parts can be combined. In theory then, it would be possible to do an exhaustive search of all possible manipulators in order to find the one configuration that is the global optimum. In practice, however, this becomes rapidly infeasible, as the number of potential configurations grows rapidly with the number of modules available in the library.

In order to place an upper bound on the solution space, no arm was allowed to contain more than fifteen modules, and the number of degrees of freedom (DOFs) was restricted to six in order to reduce the computational time required for computing the inverse kinematics. For this case, if the individual modules in the library are allowed to be one degree of freedom (1-DOF), two degree of freedom (2-DOF), three degree of freedom (3-DOF), or rigid links, there are seven possible ways to combine 1-DOF, 2-DOF and 3-DOF joints in order to sum to a total of six DOFs as shown in Table 4.1. Since there can be a maximum of fifteen total modules, the range of link modules is determined by the number of joint modules used in a particular case.

Table 4.1 Joint and Link Module Combinations

Case	1-DOF Joints	2-DOF Joints	3-DOF Joints	Link Modules	Combinations
1	6	0	0	0-9	5005
2	4	1	0	0-10	15015
3	3	0	1	0-11	5460
4	2	2	0	0-11	8190
5	1	1	1	0-12	2730
6	0	3	0	0-12	455
7	0	0	2	0-13	105

Considering each of these cases individually, it is simple to determine the number of unique combinations of module categories possible. Taking Case 1 as an example, the number of possible ways to choose six of the fifteen total modules is simply the combinational result from probability:

$$\binom{15}{6} = \frac{15!}{6!(6-15)!} = 5005 \qquad (4.1)$$

Computing the number of possible combinations is marginally more complicated when more than one type of joint module is used, as in Case 2. As Case 2 involves choosing four 1-DOF modules and one 2-DOF module, the number of possible configurations is determined by the combination of four elements from fifteen (the 1-DOF modules) multiplied by the combination of one element out of the remaining 11 modules (the 2-DOF module).

$$\binom{15}{6} * \binom{11}{1} = \frac{15!}{6!(15-6)!} * \frac{11!}{1!(10-1)!} = 15,015 \qquad (4.2)$$

Each of the other seven cases can be computed similarly, and the net result is that there are a total of 36,960 ways to combine 1-,2-, and 3-DOF modules such that they combine to be a total of 6-DOFs.

Once the number of various configurations of module categories is determined, it is a simple calculation to determine the total number of unique assembly matrix configurations. A conservative estimate is yielded by considering only the various configurations generated by joint modules and not adding in the link modules. The number of total combinations for each case when module part type, rotation and in/out connectivity are considered is calculated according to the following formula:

$$C_{Total} = (8)^{N_{joints}} (P_{1DOF})^{N_{1DOF}} (P_{2DOF})^{N_{2DOF}} (P_{3DOF})^{N_{3DOF}} C_{Case} \qquad (4.3)$$

where N_{joints} is the total number of joint modules

N_{iDOF} is the number of joint modules with $DOF = i$

P_{iDOF} is the number of parts in the library with $DOF = I$

The results of this calculation are tabulated in Table 4.2 for P_{1DOF}=3, P_{2DOF} =2, and

P_{3DOF}=1.

Even though this method is a gross under-estimate of the size of the search space

since it ignores the permutations generated by including link modules, the number of

unique assembly matrices, (and thus, the potential search space) is still on the order 10^{11}.

Some of these configurations, though they have unique assembly matrices, will yield

identical sets of D-H parameters, as some joints have isomorphic configurations when

assembled backwards or rotated.

Clearly, this is far too large a space over which to do any sort of exhaustive task-

based search, and so a genetic algorithm is employed as a method to find manipulator

configurations that approach the global optimum, even if they do not necessarily reach it.

However, before a manipulator arm can be evaluated, it must first be created. This is

detailed in the following section.

Table 4.2 Estimation of Unique Assembly Matrices

Case	DOF Combinations	Joint Module Permutations	(Combinations *Permutations)
1	5005	191102976	9.5647E+11
2	15015	2654208	3.9853E+10
3	5460	110592	6.0383E+08
4	8190	147456	1.2077E+09
5	2730	3072	8.3866E+06
6	455	13824	6.2899E+06
7	105	576	6.0480E+04
		TOTAL	9.9815E+11

4.2 Manipulator Synthesis

In order to build a modular manipulator arm, it is first necessary to have some knowledge of the modules that are available to build the arm. As mentioned earlier, the available link and joint modules are stored in a library that is segregated by the number of degrees of freedom (DOFs) for a given module. That is to say, zero-DOF modules such as links and end-effectors are stored separately from one-DOF modules, which are stored separately from two-DOF, and three-DOF modules. Within each DOF category, there are multiple choices of parts. The complete library can be found in Appendix A.

To generate the *assembly* matrix for an individual manipulator, a random process is used where the first row of the assembly matrix indicating the DOF *category* of the module is generated first with the *part type*, *in/out* value and the *rotation* values all generated subsequently. As the *part types* available for a given DOF category are different across DOF categories, these values must be randomly generated inside of the range appropriate for their category or else the assembly matrix will be invalid. For example, if there are only two types of 2-DOF joints, then a part value of "3" will be outside the bounds of the library and the assembly matrix will be meaningless. The values for in/out and rotation each have a constant valid range and so are not dependent on either the DOF category or part value. In/out is always either 0 or 1 and rotation is always 0,1,2 or 3. A population of these assembly matrices is generated as the initial population of robot manipulators.

$$[Assembly] = \begin{array}{r} DOF \\ Part \\ In/Out \\ Rotation*90° \end{array} \begin{bmatrix} [0,1,2,3] & [0,1,2,3] & [0,1,2,3] & ... \\ [1,2,3] & [1,2,3] & [1,2,3] & ... \\ [0,1] & [0,1] & [0,1] & ... \\ [0,1,2,3] & [0,1,2,3] & [0,1,2,3] & ... \end{bmatrix} \qquad (4.4)$$

47

4.3 Genetic Algorithm

As shown above, despite the fact that there are a finite number of permutations of modules to construct a manipulator, the search space is prohibitively large. Furthermore, the search space is poorly conditioned for employing traditional optimization methods such as the Newton-Raphson method as the solution space is neither smooth nor continuous. A genetic algorithm (sometimes called an evolutionary algorithm) is an optimization method well-suited to this type of problem as it handles discrete variables well, requires little knowledge of either the form of the solution or the details of the objective function, and greatly reduces the number of times that the objective function needs to be evaluated over exhaustive, brute force methods. The genetic algorithm in this thesis used many of the functions developed in the GA Toolbox for Matlab (Chipperfield et al. 1994) and the function written to execute the optimization in Matlab was called *Automatic Configuration Generation and Evaluation (ACGE)*.

A genetic algorithm (GA) takes its inspiration from the biological process of selective breeding, and the terminology of this type of optimizer tends to follow from this source of inspiration. The initial population of potential solutions is first broken down and encoded into chromosomes that contain all the information pertinent to reconstructing each individual in the population. Each of these individuals is then evaluated against an objective function and the population is put into ranked order of fitness. Individuals with higher fitness values are then crossed or bred with other high-fitness individuals, the chromosomes of their offspring are mutated, and a new population is produced. This new population is then evaluated, ranked, crossed, mutated and a third

generation is produced. In each of these generations, individuals that are more fit tend to remain a part of the population, and tend to produce offspring that have higher fitness values than the offspring of their lower-fitness counterparts. The basic structure of *ACGE* is shown in the pseudo-code below.

```
function [Population] = ACGE(PartLibrary, Tasks)
gen =0
Population = rand(Assembly) /*randomize population*/
        Population
while gen<maxgen
    gen=gen+1
    select SelectedPop(gen) from Population
    recombine pairs in SelectedPop(gen)
    mutate SelectedPop(gen)
    evaluate SelectedPop(gen)
    reinsert SelectedPop(gen) into Population
end
```

In some genetic algorithms, this process of evaluation goes on until the most-fit individual's fitness is within the tolerance limit of some pre-known solution. However, in this thesis, as the value of the global optimum is not known for any particular evaluation of the objective function, the GA proceeds until it reaches a predetermined number of generations.

4.3.1 Chromosome Encoding

The first step in getting a GA to work for a given optimization problem is to encode a potential solution in a chromosome structure. In order to do this with the assembly matrix, each column of the assembly matrix is placed sequentially into the chromosome string. Thus, the chromosome string representing an individual assembly matrix is a series of four-tuple sub-strings, with each one of the four-tuples containing the information for a particular module.

49

As not all module categories have the same number of valid parts within their libraries, it is necessary to keep the part variables (assembly matrix row 2) associated with their corresponding category value (assembly matrix row 1).

From the whole chromosome, four sub-chromosome strings are assembled: one for each row of the assembly matrix and each coding for a different type of variable (category, part, in/out and rotation). Each of these four sub-chromosomes will be recombined separately in the next step of the procedure.

$$
Assy = \begin{bmatrix} 1 & 0 & 2 & 1 & 0 & 2 \\ 2 & 2 & 1 & 2 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 & 1 & 1 \end{bmatrix}
$$

Module 1   Module 2   Module 3

$$
Chromosome = \begin{bmatrix} 1 & 2 & 0 & 1 & 0 & 2 & 0 & 1 & 2 & 1 & 0 & 1 .. \end{bmatrix}
$$

Module 1   Module 2   Module 3
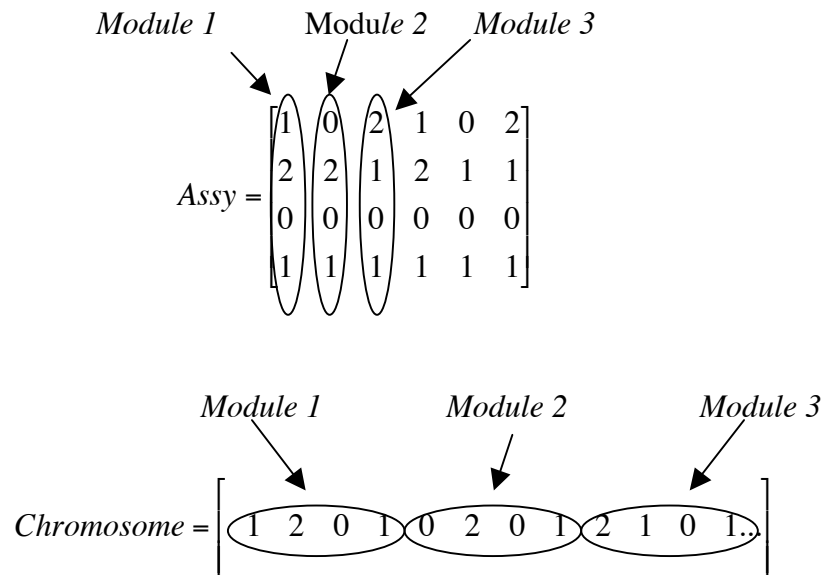
Figure 4.1: Chromosome String Structure

4.3.2 Chromosome Recombination and Mutation

As mentioned above, each of the individuals in the initial population is evaluated by the objective function and assigned a fitness value. The most-fit individuals are selected to produce the next generation of the population. Once the population of assembly matrices has been encoded into chromosomes, recombining and mutating the initial population produces this new generation of "offspring" assembly matrices.

In order to reduce the number of invalid assembly combinations during the recombination process, a two-step procedure is utilized such that the module categories are recombined and mutated first. Subsequently the Part, In/Out, and Rotation variables are placed onto sub-chromosomes and these sub-chromosomes are crossed and mutated individually. The results of this secondary recombination are checked against the corresponding category variable to verify that the results of the recombination still code for valid components within the module library.

This is best illustrated by the following example in which two assembly matrices are crossed. For the purposes of simplification, the mutation will be ignored and the two matrices have identical values of In/Out and Rotation. The two assemblies are as follows:

$$Assy1 = \begin{bmatrix} 1 & 0 & 2 & 1 & 0 & 2 \\ 2 & 2 & 1 & 2 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 & 1 & 1 \end{bmatrix} \qquad Assy2 = \begin{bmatrix} 1 & 1 & 1 & 2 & 0 & 1 \\ 2 & 1 & 1 & 2 & 3 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 & 1 & 1 \end{bmatrix}$$

Then, the first row of *Assy1* and *Assy2* are recombined. Each element has a 60% chance of crossing over and switching places with its counterpart in the other matrix.

$$Assy1Category = \begin{bmatrix} 1 & 0 & 2 & 1 & 0 & 2 \end{bmatrix}$$
$$Assy2Category = \begin{bmatrix} 1 & 1 & 1 & 2 & 0 & 1 \end{bmatrix}$$

Assembly matrices after category recombination

$$Assy1 = \begin{bmatrix} 1 & 1 & 2 & 2 & 0 & 1 \\ 2 & 2 & 1 & 2 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 & 1 & 1 \end{bmatrix} \qquad Assy2 = \begin{bmatrix} 1 & 0 & 1 & 2 & 0 & 2 \\ 2 & 1 & 1 & 2 & 3 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 & 1 & 1 \end{bmatrix}$$

Now that the assembly matrices have been reconstructed, the remaining three rows of the assembly matrix are recombined and mutated. For the part-type variables (row 2) this is done in a partitioned fashion based on the value of its corresponding category (row 1)

$$Part1 = \begin{bmatrix} 2 & 2 & 1 & 2 & 1 & 1 \end{bmatrix}$$

$$Part2 = \begin{bmatrix} 1 & 1 & 1 & 2 & 3 & 1 \end{bmatrix}$$

Assembly matrices after part recombination

$$Assy1 = \begin{bmatrix} 1 & 1 & 2 & 2 & 0 & 1 \\ 1 & 2 & 1 & 2 & 3 & 1 \\ 1 & 1 & 0 & 1 & 0 & 1 \\ 1 & 0 & 2 & 1 & 3 & 2 \end{bmatrix} \qquad Assy2 = \begin{bmatrix} 1 & 0 & 1 & 2 & 0 & 2 \\ 2 & 1 & 1 & 2 & 1 & 1 \\ 1 & 1 & 0 & 1 & 0 & 1 \\ 1 & 0 & 2 & 1 & 3 & 2 \end{bmatrix}$$

variable. Thus we have the following sub-chromosome pairs for the part- variables and the following new assembly matrices:

The result of this recombination process are two "offspring" assembly matrices which combine elements of the two "parent" assemblies. Each of these new matrices would also be subjected to a mutation process that randomly changes elements within the matrix and introduces a random element to the process which helps to maintain the solution diversity of the population.

Assemblies with Dissimilar Module Numbers

The number of columns in an assembly matrix is determined by the number of modules composing the configuration. This leads to the possibility of configurations of different lengths being crossed with each other. Certain care needs to be exercised in this case to deal with the possibility of creating configurations outside of the DOF range of the rest of the population. Additionally, a method must be developed to ensure that the lengths of the sub-chromosomes being crossed are the same.

The maximum number of modules was set to be 15 at the start of the algorithm to deal with the problem of dissimilar chromosome lengths. Each of the manipulator configurations were stored in a matrix with dimension N_{IND} by *15,* where configurations having fewer than 15 modules were appended with -1 elements. When the assembly matrices were parsed into sub-chromosomes, the -1 elements were included as well. The process of recombination leads to the -1 elements making their way into the population, and often the creation of non-sensical codings for modules. For example, if the two assembly matrices below (*Assy1* and *Assy2*) were recombined, their offspring could look something like *NewAssy1* and *NewAssy2* (for the sake of clarity, only the last column of each has been changed).

$$
Assy1 = \begin{bmatrix} 1 & 0 & 2 & 2 & 0 & 1 \\ 1 & 2 & 1 & 2 & 3 & 1 \\ 1 & 1 & 0 & 1 & 0 & 1 \\ 1 & 0 & 2 & 1 & 3 & 2 \end{bmatrix}
\qquad
Assy2 = \begin{bmatrix} 1 & 0 & 1 & 2 & 2 & -1 \\ 2 & 1 & 1 & 2 & 1 & -1 \\ 1 & 1 & 0 & 1 & 0 & -1 \\ 1 & 0 & 2 & 1 & 3 & -1 \end{bmatrix}
$$

$$
NewAssy1 = \begin{bmatrix} 1 & 0 & 2 & 2 & 0 & -1 \\ 1 & 2 & 1 & 2 & 3 & -1 \\ 1 & 1 & 0 & 1 & 0 & 1 \\ 1 & 0 & 2 & 1 & 3 & -1 \end{bmatrix}
\qquad
NewAssy2 = \begin{bmatrix} 1 & 0 & 1 & 2 & 2 & 1 \\ 2 & 1 & 1 & 2 & 1 & 1 \\ 1 & 1 & 0 & 1 & 0 & -1 \\ 1 & 0 & 2 & 1 & 3 & 2 \end{bmatrix}
$$

If the two offspring assemblies were as shown above, neither one of them would be a valid assembly matrix for evaluation. *NewAssy1* does not have the proper number of degrees of freedom, while the last module of *NewAssy2* has an invalid (1,1,-1,2) coding for its last element. During the process of mutation, it would be possible for any of the –1 values to spontaneously assume a valid coding. However, if the mutation process does not spontaneously cause a nonsense module to assume a valid coding, it must be thrown away by setting the rest of the elements in that column to be –1. When the last module of *NewAssy2* is thrown away, it would still code for a 6-DOF manipulator. However, when the last row of *NewAssy1* is thrown away, it has only 5-DOFs left. In the case where the number of DOFs of a post-recombination assembly has either one too many or one too few DOFs, it will either have a 1-DOF module spontaneously added or subtracted to return the configuration to 6-DOF.

This spontaneous addition of modules in the case where the manipulator configuration has been shortened by the process of recombination, coupled with the process of mutation helps to keep the population from becoming overly shortened by the recombination process.

Sorting and Reinsertion of Offspring

Once the new population of assembly matrices has been produced, each of the new assemblies is evaluated by the objective function, and is assigned a fitness value. Based on this fitness value, the new assembly matrices may or may not replace matrices from the parent generation. A semi-elitist reinsertion scheme is used within the ACGE function, whereby the best 25% of individuals will always remain in the population and

survive to the next generation, but most of the population will be replaced by the next generation, even if the new generation has lower fitness values. Because there are many more "bad" manipulator combinations than good ones, it is often possible that there will be no valid offspring produced for several generations, leading to the requirement that the best individuals will continue to persist in the population. Without using this semi-elitist strategy, the overall population would be at risk of filling with unworkable solutions.

Additionally, configurations that match existing configurations in the parent population are not reinserted in order to preserve the diversity of the parent population. If this step is not taken, the population can have the tendency to homogenize over time, leading to more rapid convergence, but with fewer potential configurations evaluated.

4.3.3 Objective Function

The objective function is the core of any optimization scheme, since it determines the "goodness" of any given solution. In this thesis, the overall mass of a reconfigurable modular manipulator system is the primary parameter being to be minimized. Of course, given two robotic systems which have similar masses, another criterion must be used to determine which of these systems would be better for performing a given task. Therefore, the Yoshikawa manipulability index of a manipulator contributes to the value of the objective function, and this provides a degree of guidance for selecting between manipulator configurations which are similarly massive.

As manipulator mass is the driving parameter in the objective function, it is necessary to have a way to estimate the mass of the arm. As a manipulator arm is made up of essentially two types of components (link modules and joint modules), a mass-

estimating relationship for both of these types of components must be developed. The mass of the joint is primarily determined by the amount of torque it is required to generate, while the mass of a link is primarily determined by its length. The torque required to perform a task is therefore needed to obtain an estimate for the joint mass.

Torque Calculation

A task trajectory consists of a series of desired end-effector poses that must be met by a given manipulator configuration, along with a 6x1 force vector specifying the forces and torques which the end-effector must exert at the specified position. The position and orientation required by the end-effector is given as a 4x4 transformation matrix similar to those used to compute the forward kinematics of the manipulator (see Chapter 3). Each of these trajectory points is considered separately and evaluated quasi-statically. Joint torques are found by performing the inverse kinematics to determine the joint angles required to reach the specified end-effector position and then applying the well-known equation:

$$[T] = [J]^T [F] \qquad (4.5)$$

where F is the 6x1 force/moment vector, J^T is the transpose of the Jacobian of the manipulator, and T is the vector of joint torques required to exert the desired force. Both the inverse kinematics and the calculation of the Jacobian (and its transpose) are implemented using the Robotics Toolbox (Corke, 2001).

Link Mass Estimation

56

Starting with the simpler case of link modules, several assumptions are made: the modules are made of circular, tubular aluminum 7 cm in diameter, and a 1.0 m link should not deflect more than 2 cm at its tip when a point-load is applied perpendicularly to the axial direction. The maximum expected force on any one of the link elements is assumed to be 1000 N, though typical values used for the required tip force were usually closer to 100 N when *ACGE* was actually run. As beam deflection for a tubular structure is given by:

$$\delta = \frac{PL^3}{3EI} = \frac{(1000N)(1m)^3}{3(72GPa)\left(\frac{\pi}{64}\left((0.07m)^4 - (d_{inner})^4\right)\right)} \qquad (4.6)$$

Solving for d_{inner} we find that a tube with a thickness of 2 *mm* will meet the beam bending stiffness conditions specified above. Taking the cross-sectional area for a tube 7.0 *cm* in diameter and 2 *mm* in thickness, the mass per unit length of the aluminum link is calculated as:

$$\frac{Mass}{Length} = (Area)(Density) = \left(\frac{\pi}{4}\left((0.07m)^2 - (0.066m)^2\right)\right)\left(2800\frac{kg}{m^3}\right) = 1.2\frac{kg}{m} \qquad (4.7)$$

Of course, the links are not composed only of a tubular structure, but also include the two end caps that allow the link modules to connect to other link modules or joint modules. The combined weight of the two end caps is conservatively estimated to be a constant mass for each link module of 0.75 kg. Thus, the weight of any link module is estimated to be:

$$LinkMass = (1.2\frac{kg}{m})L + 0.75 \qquad (4.8)$$

where *L* is the length sum of a link module.

Since link length is a constant for a particular library link module, link masses are passed into the objective function as arguments, rather than being recalculated each time the objective function is called. This is not the case, however for the mass of the joint modules as the estimated mass of the joints varies according to the maximum torque they are expected to exert.

Joint Mass Estimation

The second component of estimating the mass for a particular manipulator configuration is the determination of joint module masses. The most common types of actuator for space-borne robotics are brushless direct current (DC) electric motors coupled with high-gear-ratio harmonic drives. In the harsh thermal and vacuum environment of outer space, these electric systems provide distinct advantages over pneumatically or hydraulically actuated manipulators. Therefore, for estimating the mass of joint modules, it was assumed that the joint actuators would be brushless DC motors with harmonic drives.

Using motor and harmonic drive data assembled from an unpublished trade study of sixteen different brushless motors paired with seven harmonic drives (Dave Hart), a regression line analysis was performed. The results of this analysis are found in Figure 4.2. Based on these results, the mass of a given actuator was estimated to be:

$$Mass[kg] = (0.007)(MaxTorque[Nm]) + 0.700 \quad (4.9)$$

From *(4.9),* the size of an actuator is sized based on the maximum torque

Actuator Mass vs. Torque

y = 0.0066x + 1.0472
y = 0.0066x + 0.6672
y = 0.0066x + 0.5762
y = 0.0074x + 0.4015
y = 0.0068x + 0.3902
y = 0.0121x + 0.0943
y = 0.0121x + 0.0743
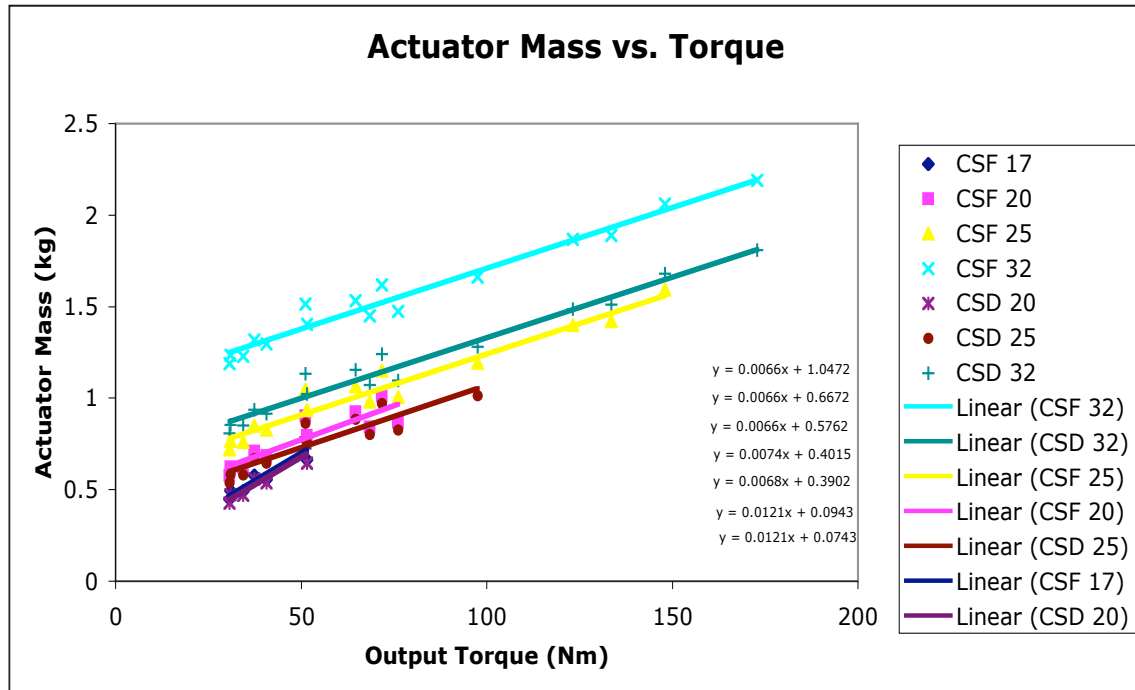
Figure 4.2: Regression Line Analysis of Harmonic Drive Actuators

expected during execution of the task.

Now that both joint masses and link masses have been determined, the total mass

of the arm can be estimated by summing the two numbers. These figures are, of course,

rather rough estimates, however, they do fulfill the basic requirements of allowing

comparisons to be made between the relative masses of different manipulator

configurations.

Dexterity Index

59

As mentioned above, it is desirable to have another performance metric for determining the relative quality of a design solution besides just its mass. In performing a task, it is generally desirable for a manipulator to have the capability to move in an arbitrary direction. A useful parameter for measuring this dexterity is the Yoshikawa manipulability index, defined as:

$$W = \sqrt{\left| J(q)J^T(q) \right|} \qquad (4.10)$$

where J is the manipulator's Jacobian for joint angles q. As the objective function seeks to minimize the manipulator mass while maximizing the manipulability (W), the manipulability is inverted such that a gain in manipulability results in a decreased objective function value. In order to avoid dividing by zero, the quotient has a 1 added to it, such that the value of this new "dexterity index" has a maximum of 1 and a minimum approaching zero.

$$DexIx = \frac{1}{(1+W)} \qquad (4.11)$$

Objective Value Calculation

The actual value of the objective function itself is given as:

$$Obj = -e^{-(M*Mass+D*DexIx)} \qquad (4.12)$$

where *Mass* is the total arm mass, *DexIx* is the "dexterity" index as given in Equation 4.10, and D and M are constant, positive weighting parameters. From *(4.12),* it is readily apparent that the objective function will always be negative and will be minimum at -1 in the case where the arm has no mass and infinite manipulability (*DexIx=0*), and will approach 0 when the mass of the manipulator becomes infinite.

As the mass of the manipulator is ultimately the parameter being minimized, it is desirable that *D* and *M* be tuned such that approximately 90% of the weight comes from

60

the manipulator mass while 10% comes from *DexIx*. Tuning these parameters to achieve

the desired 90/10 balance is rather difficult to perform *a priori* as it would require a

prediction of typical values for the mass and manipulability for a set of unknown

manipulators performing a series of tasks. The precise balance between the two

parameters is not entirely important so long as the mass tends to dominate. The

exponential form of the objective function, will prohibit the mass from becoming too

large or the manipulability going to zero.

As it turned out, there was very little difference in performance if the weighting

on manipulability was set to zero. Since joint torque and manipulability are both

functions of the Jacobian, including both parameters in the objective function is likely not

necessary, though further investigation is warranted in future studies. The inverse

kinematics solver (Robotics Toolbox function *ikine*) filtered out singular configurations

prior to the evaluation of the objective function, and the avoidance of singular

configurations is the primary reason to include manipulability in the objective function to

begin with. As a point of comparison between the two methods, over the same set of

tasks, a set without dexterity weighting was 93% of the mass of a manipulator where

dexterity weighting was included. While not enough runs were performed to do a

statistical comparison, the limited number of runs performed seemed to validate the

intuition that manipulability did not matter significantly.

GA Fitness for Infeasible Configurations

More often than not, a significant number of task points will be unreachable by a

given configuration, either because the point lies near a singular configuration of the arm

or outside the boundary of a manipulator's work space. Only configurations that can reach all of the desired task points are evaluated by the exponential objective function. However, it is desirable to distinguish between solutions that can reach most of the task points without failure, and the solutions that can't reach any points. In the language of genetic algorithms, a manipulator that can reach 80% of its goal points is better than one that can only reach 20%, and will statistically be more likely to produce "offspring" that reach all the points. A method for making this distinction is required.

In these situations, the proportion of unreachable points on the task trajectory is taken as the manipulator's "unreachability index". Thus, an arm which can reach the first nine points on a ten-point trajectory would have an unreachability index of 0.1, while an arm which can only reach the first point, but fails at the second would have an unreachability index of 0.9 (9 out of 10 points unreachable). An arm that cannot reach any points is assigned an unreachability index of 1, which is the worst-case (maximum) value (note that this is outside the 0 to -1 range of *(4.12)*). When the GA sorts the population of manipulators according to their fitness, the unreachability index of the infeasible manipulators is compared against *objective function* values of successful configurations to determine overall fitness in the population. As the GA sorts with the minimum values being considered most fit, feasible configurations are always more fit (range -1 to 0) than their infeasible counterparts (range 0 to 1), and the least fit members of the configuration population are those that cannot reach any points whatsoever. Without this distinction between total failures and near successes, the initial population can often take many generations of random flailing before it finds even a single

configuration that is capable of reaching all the task points and has a negative objective function value.

4.3.4 Generational Improvement

For a given task, the genetic algorithm is allowed to run for 100 generations of offspring, which is a total of 101 generations when the original parent generation is included. As the algorithm proceeds, the average fitness of the entire population and the fitness of the best individual created up to that point is tracked. The population average typically starts with a value near 1, as the initial population tends to have relatively few valid solutions. The population's objective function value tends to decrease nearly linearly in the first generations, as the population's elitist slots are filled (see section 4.3.2 about recombination and reinsertion). Eventually, the average begins to bounce nearly randomly as the majority of the population is replaced with each new generation, even as the elitist portion of the population remains steady. The details of the average population objective function value are not terribly important, though discussion has been included

to illustrate the nature of the genetic algorithm.
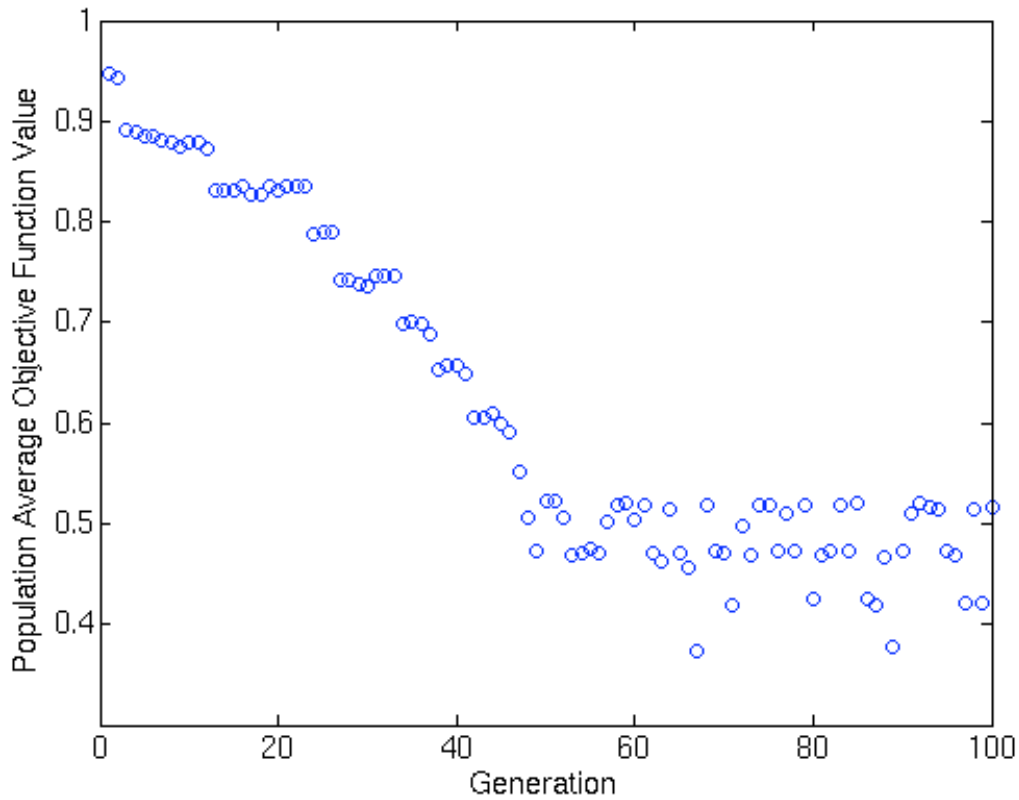
Figure 4.3: Average Population Objective Value vs. Generation

Of far more importance is the objective value of the "best" individual, which is tracked with each generation. The configuration of the "best" individual in the population is always preserved and passed to the next generation, as mentioned in Section 4.3.2, and so it is never replaced unless a new best is found. As can be seen in Figure 4.4, the objective function value for the best individual tends to exhibit a pattern of dropping and then remaining constant for several generations before dropping again. In practice, this behavior cannot be predicted, as the generation of manipulator configurations is random. Thus, it is not possible to declare with any certainty that for the value of the best

individual drops X amount in Y generations. From experience, it is typical that any major

(order of magnitude) improvement in the best individual's objective function value will

have occurred by generation 50. Therefore, the genetic algorithm is allowed to run for
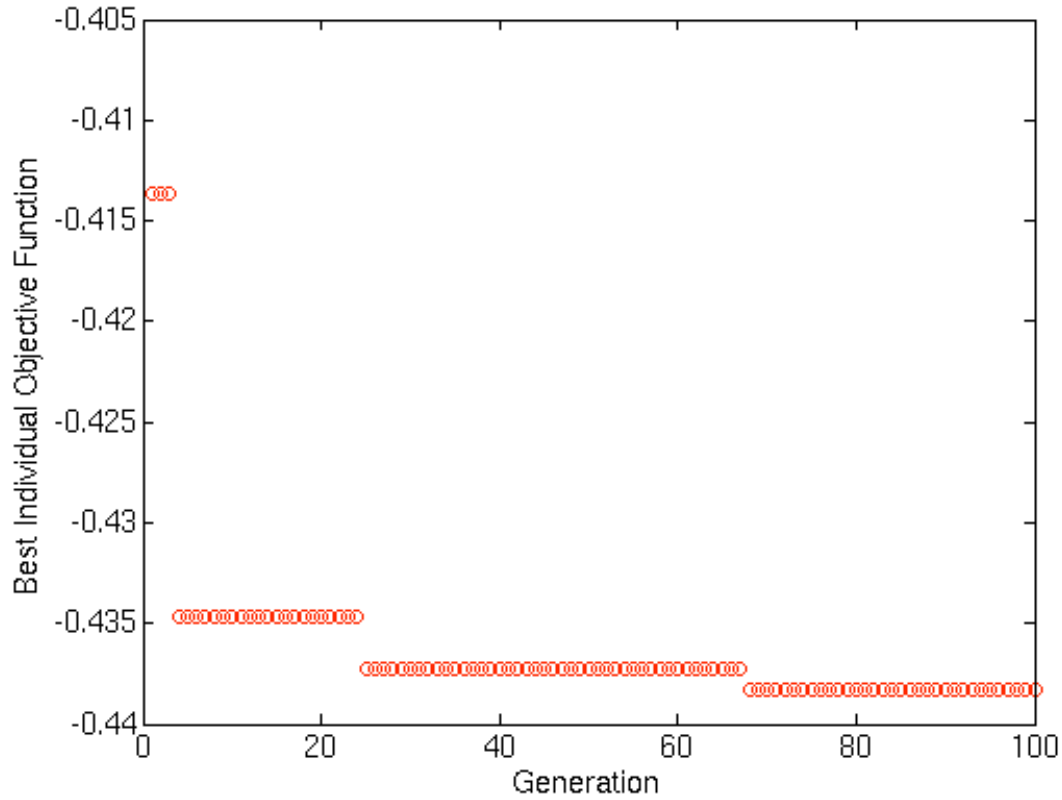
Figure 4.4: Best Individual Objective Value vs. Generation

100 generations, to provide enough of a margin to be reasonably sure that the value of the

best individual found has converged to a stable minimum.

Convergence when no First Generation Solutions Exist

65

Figure 4.5 shows a run where no acceptable solutions were found in the first generation, nor for the first 34 generations. As can be seen by comparing the two graphs, the average fitness of the entire population still decreases as it finds more and more
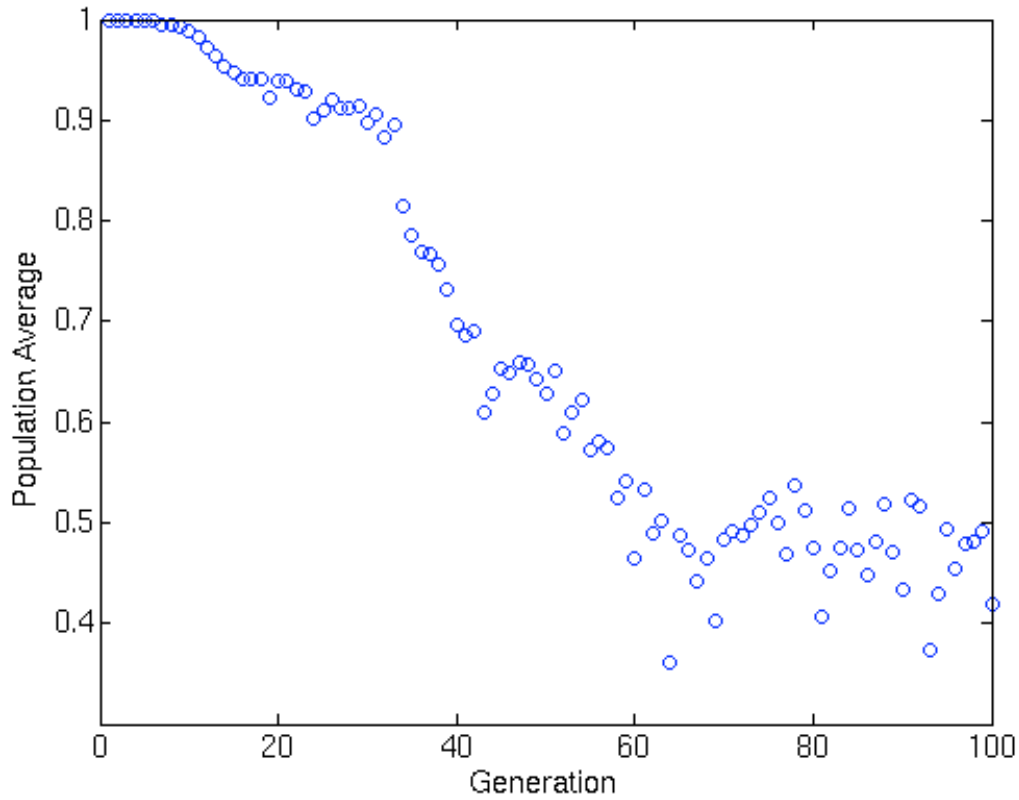
Figure 4.5: Average Population Objective Value when No First Generation Solutions Exist

partial solutions. The plot of the best individuals as shown in Figure 4.6 illustrates the utility of the method discussed in Section 4.3.3 for calculating the objective function

value of partial solutions.
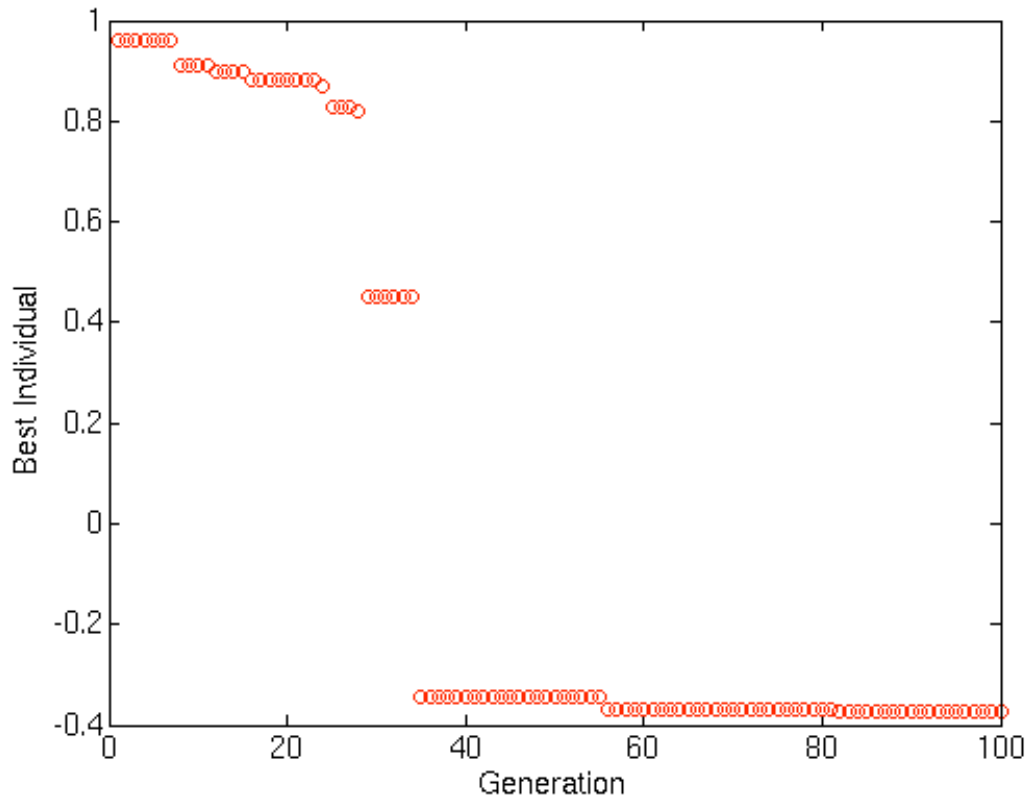
*Figure 4.6: Best Individual Objective Function Value when No First Generation
Solutions Exist*

As Figure 4.6 shows, the best solution drops from near "1", where very few points on the

trajectory are reachable, down to 0.40 where 60% of the points on the trajectory are

reachable. Eventually at generation 35, a configuration was found that could reach all the

points on the trajectory. The trend of the average population in Figure 4.5 after generation

35 is comparable to the average population trend of Figure 4.3.

4.4 Summary

As the number of unique combinations of manipulators is prohibitively large for

performing an exhaustive search, a genetic algorithm is employed for finding feasible

manipulator configurations capable of performing the designated tasks. Each manipulator combination is encoded as a chromosome string and genetic operators from the Matlab Genetic Algorithm Toolbox are used to recombine, mutate, and sort the population to search for lightweight manipulators capable of performing the task at hand, while maintaining manipulability. While the GA is not guaranteed to find the global optimum solution (minimum objective function value), it does produce results that tend to improve with each generation and as experience has shown, runs long enough to allow the algorithm to converge to a stable solution.

Chapter 5: Reconfigurable Modular Kit Determination

In Chapter 4, it was shown how a population of modular manipulators could be evolved to produce solutions to minimize manipulator mass while maintaining manipulability by using a genetic algorithm. This chapter applies the functions and scripts developed in Chapters 3 and 4 and to show the division of expected robotic tasks into subtasks can lead to a modular reconfigurable manipulator system that is lighter than a single, non-reconfigurable arm which can perform all of the tasks.

5.1 Simplified Tasks

Previous chapters specified that the tasks being considered were characterized as a series of 4x4 transformation matrices defining a position and orientation along with a 6x1 force/moment vector describing the Cartesian force and moment required to be exerted by the manipulator at its distal tip. The "task" is made up of a given number of these points strung together and reached in sequence by a manipulator. There is no check to ensure that the trajectory between points is continuous, although if the methods outlined in this thesis were ever put into practice, this would be a critical check.

A series of five different task classes were developed as test cases. The properties of each task are summarized in Table 5.1. Trajectories for each of these task archetypes can be easily generated by use of a Matlab function that allows for specification of the center of motion, the motion length, and the number of points to be generated.

Table 5.1: Simplified Task Primitives

Task	Motion	Force	EVA example
1	Translation in global X direction, constant orientation	Constant force in direction of motion	Instrument insertion/extraction
2	Translation in global Y direction, constant orientation	Constant force in direction of motion	Instrument insertion/extraction
3	Translation in global Z direction, constant orientation	Constant force in direction of motion	Instrument insertion/extraction
4	Circular translation in XY plane of radius R, between angles Theta1 and Theta2, no orientation change	Constant force in direction of motion (tangential to circle)	Generic translation
5	Circular translation in XY plane of radius R, between angles Theta1 and Theta2, orientation normal to circle	Constant force in direction of motion (tangential to circle)	Open access door
6	Circular translation in XZ plane of radius R, between angles Theta1 and Theta2, orientation normal to circle	Constant force in direction of motion (tangential to circle)	Open access door
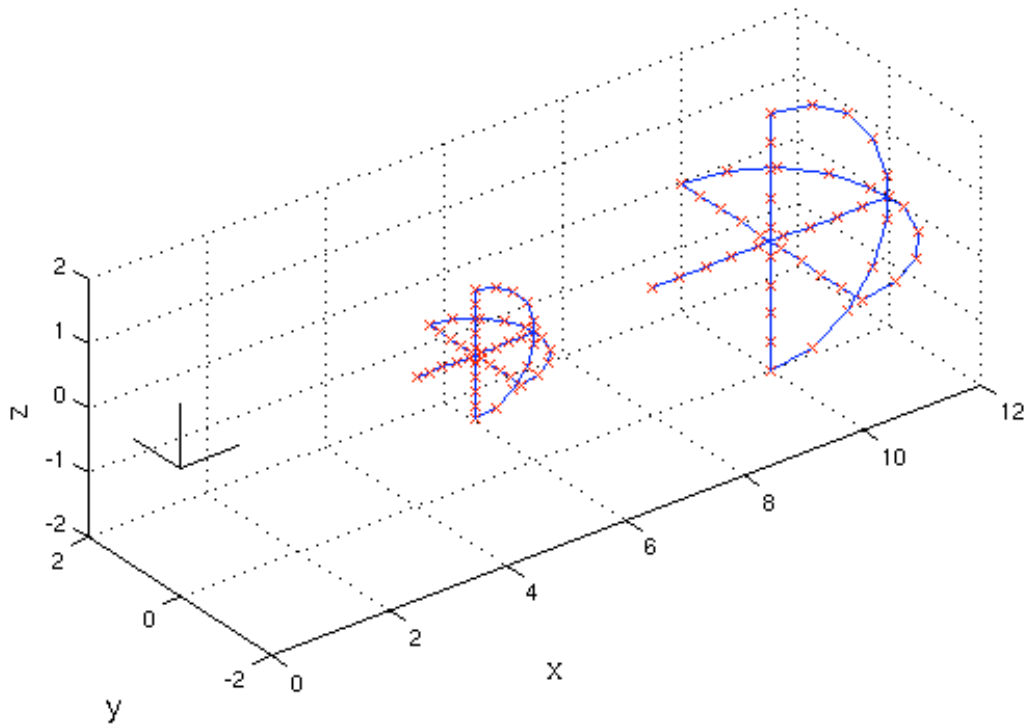
Figure 5.1: Sample Task Definitions

From these simplified tasks, a series of tasks is selected and can be used to
generate an entire range of points and forces that the manipulator is required to meet.
This range of points is termed the complete task, while each sub-element of it is
considered to be a sub-task. Figure 5.1 shows a plot of several subtasks with each
trajectory point highlighted with a red 'x'. The blue lines represent the continuous
trajectory that the arm would be following. The arm is not analyzed at each of the
intermediate points, only at the red 'x' points.

5.2 Automated Kit Determination

Once the complete task has been generated, the Automated Configuration Generation and Evaluation (ACGE) program detailed in Chapter 4 is exectuted in Matlab to find an appropriate module assembly that minimizes the mass of the manipulator (see

$$Kit.parts = \begin{cases} & Type1 \quad Type2 \quad Type3 \\ 1DOF & 0 \quad\quad 0 \quad\quad 0 \\ 2DOF & 2 \quad\quad 1 \quad\quad 0 \\ 3DOF & 0 \quad\quad 0 \quad\quad 0 \\ 0DOF(Link) & 0 \quad\quad 6 \quad\quad 0 \end{cases}$$

$$Kit.mass = \begin{cases} & Type1 \quad\quad\quad Type2 \quad\quad Type3 \\ 1DOF & 0 \quad\quad\quad\quad 0 \quad\quad\quad 0 \\ 2DOF & [m21_1, m21_2] \quad [m22_1] \quad 0 \\ 3DOF & 0 \quad\quad\quad\quad 0 \quad\quad\quad 0 \\ 0DOF(Link) & 0 \quad\quad [m02_1....m02_6] \quad 0 \end{cases}$$

$$Kit.totalmass = \sum Kit.mass$$

Figure 5.2: Example Kit Data Structure

Chapter 4). After the genetic algorithm embedded within ACGE has found a manipulator configuration, the number and masses of the required modules are recorded by the function *Assembly to Kit (Assy2Kit)*, and stored as *Kit*, a Matlab structure. This "universal" arm becomes the baseline that the various subtask arms will be evaluated against.

Though the universal arm has been constructed from modules, it does not truly represent a reconfigurable solution as the same manipulator geometry is used for each and every task. To capture the advantages of a system that is both modular and

reconfigurable, the universal kit is used as the starting point for a search for manipulator

solutions to each of the subtasks rather than the complete task. Since it is complex to
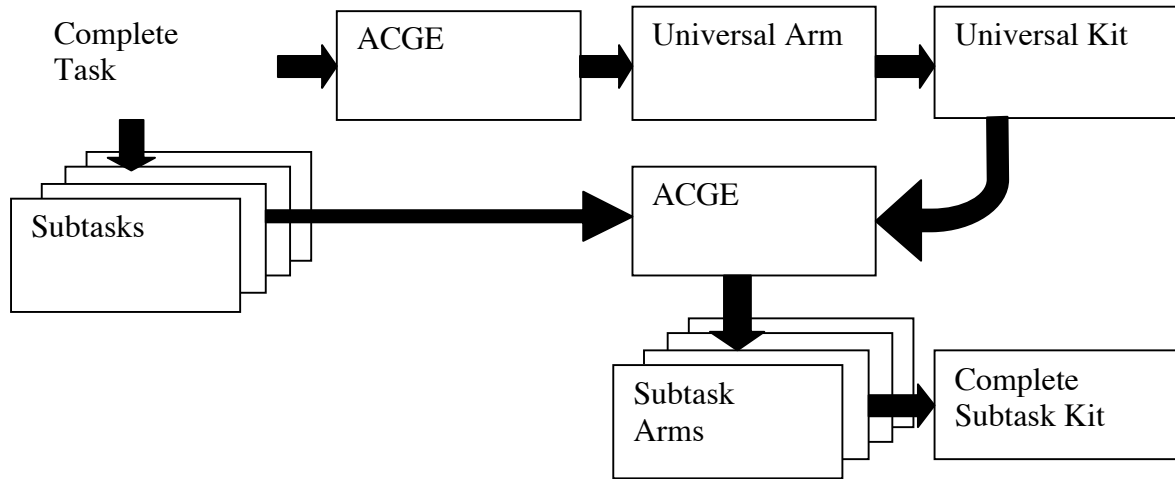
Figure 5.3: Kit Generation Flow Chart

show this using only the generic case, a concrete example will be used to illustrate this

process.

5.2.1 Universal Arm

For the task points illustrated in Figure 5.3, the universal arm output by ACGE

has the assembly matrix shown in Table 5.2 and the Denavit-Hartenberg (DH) parameters

shown in Table 5.3.

Table 5.2: Example Universal Arm Assembly Matrix

Assembly Matrix	Mod 1	Mod 2	Mod 3	Mod 4	Mod 5	Mod 6	Mod 7	Mod 8	Mod 9
Module DOF	2	0	0	2	0	0	0	0	2
Module Part	2	2	2	1	2	2	2	2	1
In/Out	1	0	1	1	1	1	0	1	1
Module Rotation	2	0	2	1	1	2	2	1	0
Module Mass (kg)	15.47	3.05	3.05	7.14	3.05	3.05	3.05	3.05	1.47

73
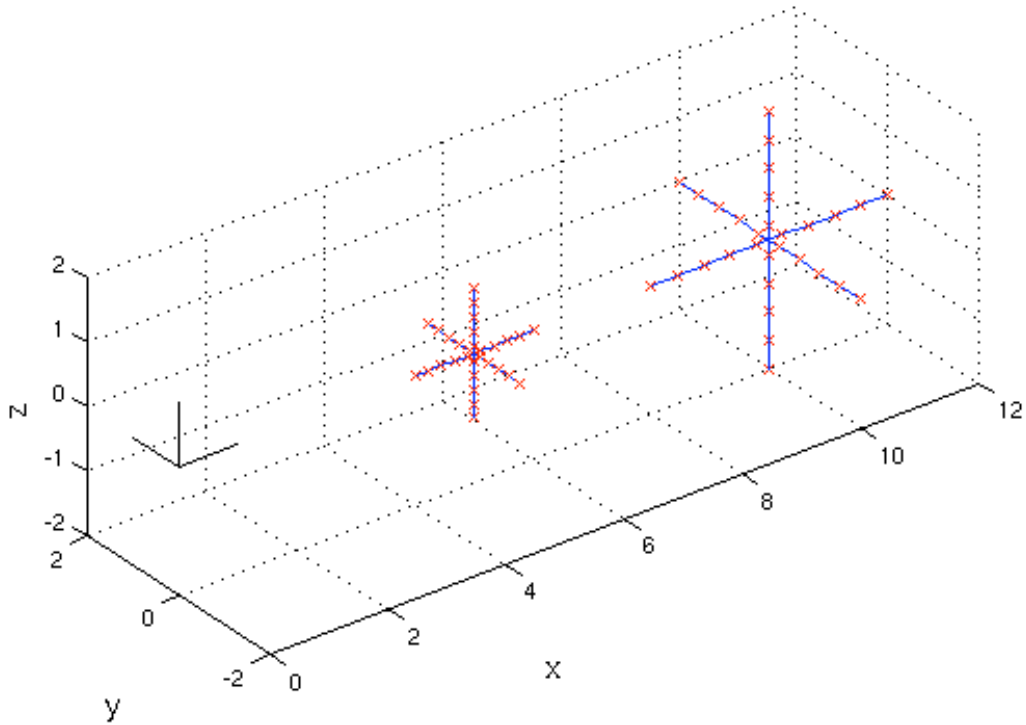
Figure 5.4: Example Complete Task

Table 5.3: Universal Arm DH Parameters

DH-Parameters		
Alpha (rad)	A (m)	D (m)
1.5708	0.1	0
0	4.2	0
1.5708	0	0
1.5708	0	8.3
1.5708	0	0
0	0	0.1

The total module kit that would be required to build the universal arm is obtained

from *Assy2Kit*, and is presented in Table 5.4. A not-to-scale diagram of the universal arm

is shown in Figure 5.4. Note that the arm has an 3-axis intersecting wrist, even though

one of the actuators is located far back in the "elbow" of the arm. While most

conventional configurations of arms have links of approximately equal length in order to

maximize the reachable workspace, workspace volume was not considered in the

evaluation of the objective function.
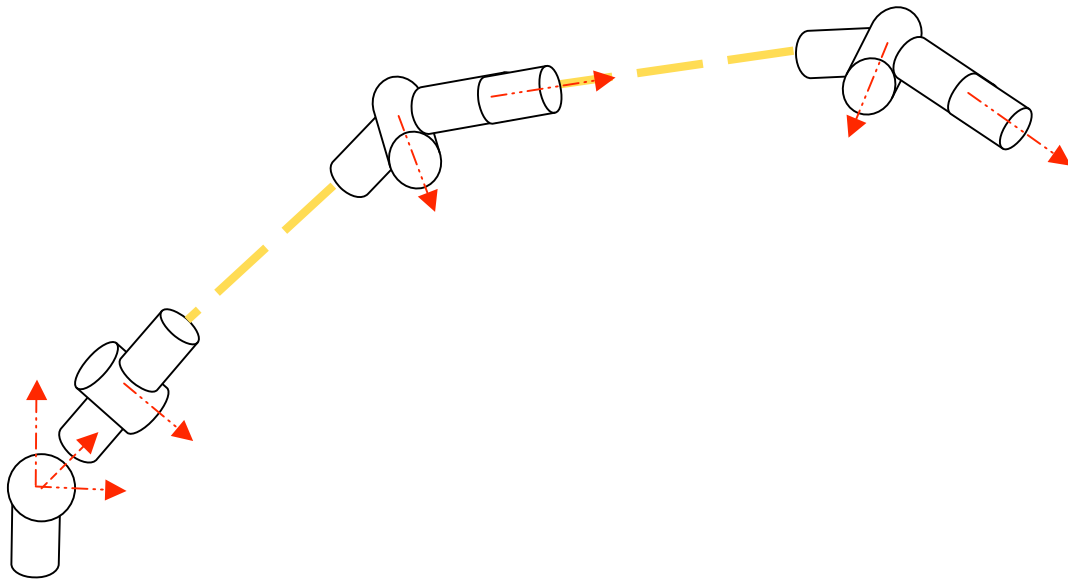
Figure 5.5: Universal Arm

Table 5.4: Universal Arm Required Kit

Required Modules [Mass]	Part 1	Part 2	Part 3
1-DOF Modules	0	0	0
2-DOF Modules	[7.14, 1.47]	[15.47]	X
3-DOF Modules	0	X	X
Link Modules	0	6, [3.05 kg each]	0
Link Mass		18.3 kg	
Joint Mass		24.1 kg	
Total Mass		42.4 kg	

5.2.2 Subtask Manipulator Kit Generation

As indicated by the flow chart in Figure 5.2, the universal kit is fed into the Automatic Configuration Generation and Evaluation (ACGE) function along with each of the individual subtasks. The universal kit constrains the number and type of modules that can be used by ACGE to construct each of the subtask arms. Each subtask is considered individually and ACGE runs independently for each of the subtasks. While the set of modules that can be selected to build each of the subtasks arms is restricted to be within the bounds of the universal kit, the joint modules are allowed to vary in mass from those of the original kit. The upshot of this is that, for a given subtask, the joint torques required are often lower than those required for the universal arm, and thus the subtask arm can be built with smaller actuators than the universal arm.

Returning to the example presented above, the complete task as shown in Figure 5.3 is composed of six subtasks, each of these a straight-line motion in the direction of one of the principle axes composed of two examples each of tasks 1, 2 and 3 from Table 5.1. Each of these six subtasks is now independently considered to be the task for which a mass optimized is to be built from the modules specified by the universal kit. For the first subtask, the assembly matrix as output by ACGE is shown as Table B.1 in Appendix B. and the kit required to build all of the subtask arms is given in Table 5.5. The columns with *Kit Limiting* equal to "1" in Table B.1 indicate the components that drive the size of the actuators to be included in the kit, and thus drive the mass of the system. Modules with *Kit Limiting* equal to "0" do not end up determining the overall mass of the kit. If the kit of parts were to contain the actuators the size of those highlighted and indicated with a Kit Limiting value of 1, then those actuators would be capable of providing sufficient torque to perform all of the subtasks.

Table 5.5: Subtask Arms Required Kit

Required Modules [Mass]	Part 1	Part 2	Part 3
1-DOF Modules	0	0	0
2-DOF Modules	[4.54, 1.87]	[8.40]	X
3-DOF Modules	0	X	X
Link Modules	0	6, [3.05 kg each]	0
Link Mass	18.3kg		
Joint Mass	14.8kg		
Total Mass	33.1kg		

The key result is that the kit required to build each of the six sub-task arms (Table 5.5, total mass 33.1 kg) is lighter than the kit that builds the universal arm (Table 5.4, total mass 42.4 kg). In addition, this process was done automatically by independently running each of the subtasks through the ACGE function and simply tallying the number of parts each solution requires. However, this particular method of assembling the kit is not effective, as optimizing individual subtask solutions does not guarantee that the merging of those solutions into a single kit will result in an overall optimization of the entire kit. In fact, for many sets of subtask arms, the total mass of the kit required to build each of the subtask arms is actually heavier than the original arm, despite each of the individual arms being far lighter than the universal arm.

An example is used to show that optimization over individual subtasks does not yield optimization for the entire kit. The SubTask5 configuration of Table 5.6 contains two of the joints that drive the total mass of the kit. However, an inspection of Table 5.5 will quickly show that there are no other instances in which a 2-DOF joint of type 1 requires such a high torque. In fact, if the SubTask5 arm is rebuilt as the SubTask5B arm as listed in Table 5.7, the overall kit mass would decrease, despite an increase in the mass of the individual manipulator.

By making the first module of the SubTask5 arm the same as the heaviest module

required across all configurations (8.4 kg for the SubTask6 arm, a Type 2 2DOF module),

the total kit mass is reduced. The 4.9 kg and 2.54 kg mass actuators required for the

SubTask5B arm (Table 5.7) are no longer the drivers of total kit mass.

Table 5.6: Alternate Subtask5 Configuration

SubTask5B Arm

Assembly Matrix	Mod 1	Mod 2	Mod 3	Mod 4	Mod 5	Mod 6
Module DOF	2	0	0	2	0	2
Module Part	2	2	2	1	2	1
In/Out	1	0	1	1	0	1
Module Rotation	1	3	3	3	1	1
Module Mass	4.9	3.05	3.05	2.54	3.05	1.47

Table 5.7: New Required Kit with SubTask5B Arm

Required Modules [Mass]	Part 1	Part 2	Part 3
1-DOF Modules	0	0	0
2-DOF Modules	[3.07, 1.55]	[8.40]	0
3-DOF Modules	0	0	0
Link Modules	0	6, [3.05 kg each]	0
Link Mass	18.3kg		
Joint Mass	13.0kg		
Total Mass	31.3kg		

5.3 Kit Refinement

Because the individual optimization of subtasks does not result in the global

optimization of the overall kit of parts (as demonstrated in Section 5.2.2), a method was

developed to ensure that a kit of parts generated from each of the subtasks would not

exceed the mass of the universal kit. By feeding the Kit of the universal arm into the

ACGE function and rejecting new combinations that have joint masses exceeding the estimated joint masses for the universal configuration, it can be guaranteed that maximum mass for the reconfigurable kit will not exceed that of the universal arm. This formulation eliminates the problem shown in Section 5.2 where the heaviest modules do not overlap across sub-task combinations, resulting in kits that would need to be heavier than the universal arm.

The results of a kit refinement process are shown in Table B.1 in Appendix B, where the task is composed of 10 subtasks. A Cartesian plot of the task locations for the ten tasks is shown in Figure 5.3. Again, an assembly matrix is presented for all of the task points (the universal arm) and each of the ten individual subtasks. Along with the assembly matrix for each of the subtasks and the mass of each of the modules, the *Kit Limiting* variable shows whether a module determines the mass of the final kit.

Note that each of the subtasks has maintained the same order of joint modules (2DOF Type2, 2DOF Type 2, 1DOF Type 2, 1DOF Type 1) when this method is used. Joints nearer to the base of a manipulator tend to have greater performance requirements as forces and torques required at the end-effector are magnified by the length of the arm, resulting in greater joint torques near the base. It is not surprising therefore, that when the allowable mass for each of the joint modules is constrained to be that of the universal arm, that the same modules tend to be found near the base while the lighter modules are to be found near the end-effector.

5.3 Kit Optimization Limitations

While basing the potential manipulator kit on the modules used by the universal arm configuration is perhaps faster, it does not mean to imply that the kit produced by the process is optimized. It is quite possible that the most ideal kit capable of building manipulators to perform the specified task shares no parts whatsoever with the universal arm for two main reasons.

The first and most important reason that a search over all possible kits was not performed is that it would take the form of a genetic algorithm search where with the input variables being the type and number of modules. Embedded within each evaluation of each possible kit would be an evaluation of the ACGE function to find the best arm configuration that could be made from the kit. As each run of the ACGE already takes around 8 hours to complete (running on an iMac with a 1.8 Ghz PowerPC G5 processor and 768 MB of RAM), the dozens of calls to ACGE required to search for the ideal kit would quickly become impractical without resorting to methods such as distributed computing.

To optimize a *particular* kit of module archetypes to find the lightest possible set of subtask arms, a concurrent optimization scheme needs to be employed. In such schemes, information transfer takes place between each of the individual subtask optimizations, helping to eliminate the problem of "shoulder" modules from one subtask configuration being used as the "wrist" modules of another. Such a scheme was not developed in this work.

The second reason that a broader search for the ideal kit was not conducted was that conducting such a search did not prove to be necessary to show that reconfigurable systems can outperform fixed-topology systems. Lower-mass kits could be found simply

80

by reconfiguring the universal arm modules and a more complete search did not need to be conducted to show this. While conducting a search for the ideal kit may be a future area of research to pursue, it was considered for this thesis.

5.4 Summary

Reconfigurable modular manipulators were shown to achieve a mass-savings over fixed topology manipulators when operating over the same range of tasks. The savings is achieved by a reduction in the required joint torques required to exert the same amount of force at the end effector due to the kinematic advantages that reconfiguring the arm provides. In order to show the difference between a fixed-topology manipulator and one capable of reconfiguring between subtasks, the complete task was broken into its constituent subtasks. The fixed-topology "universal arm" was optimized over all of the trajectory points, while the subtask arms (representing various configurations of a single kit) were optimized over only individual subtasks. It was shown that the optimization of manipulators over individual subtasks did not yield a kit of parts lighter than the universal arm.

A simplified method for finding kits which actually would be lighter than the universal arm configuration by constraining the modules comprising the kit to be lighter than those of the universal arm. While this method was not meant to yield the best possible kit of parts, an explanation of how such a kit could be found was beyond the scope of the present work.

Chapter 6: Summary, Conclusions and Future Work

6.1 Summary

A method was developed to proceed from a random serial assembly of joint and link modules to a Denavit-Hartenberg parameterization of an "optimal" manipulator formed by those constituent components. This capability of automatically generating the D-H parameters, allows large numbers of manipulators to be evaluated by an automated method of comparing task-based designs for modular reconfigurable manipulators against fixed-topology manipulators was developed. Using a genetic algorithm (GA) search of potential manipulator combinations allows for the identification of low-mass solutions.

6.2 Conclusions

This thesis has shown that reconfigurable robotic manipulators can gain a substantial advantage over fixed-topology systems in terms of system mass when performing a series of tasks. The advantage obtained by the reconfigurable manipulators comes from the capacity of the system to obtain more advantageous kinematic configurations for each task, reducing the required joint torques and actuator masses required to build the system.

While it may have been possible to reach the same conclusions by assembling series of modules and evaluating the resulting manipulators "by hand" against a fixed-topology arm, there could be no confidence that the designs analyzed were in any way optimized. To make the comparison between fixed-topology arms and a reconfigurable

system, it is necessary to compare the best examples of each. The automation of the procedure of assembling and evaluating hundreds of manipulators for a given task and evolving more effective solutions with a genetic algorithm allows this thesis to clearly show not only that a reconfigurable system can be made lighter than a universal manipulator, but that the reconfigurable system can be made lighter than the fixed-topology manipulator that has been optimized for the specified series of tasks.

The promise of lighter-weight systems should be reason enough to further pursue the technology of reconfigurable robotic manipulators, at least in the environment of on-orbit servicing where the expense of building a more complicated system can be overcome by launch-cost savings. Furthermore, while this thesis took the approach of making a one-to-one comparison of reconfigurable manipulators versus the standard fixed-topology manipulator technology, it hasn't addressed the vast possibilities of operational modes that are not available to fixed-topology technology. This thesis has shown that for a very limited set of circumstance that reconfigurable system can be better, but it is left to the work of the future to expand on the ideas presented here.

6.3 Future Work

6.3.1 Inverse Kinematics Solver

The first priority of any future work is the development of a more effective inverse kinematics solver. In this thesis the *ikine* function of the Robotics Toolbox was used to compute the required joint angles required to reach a given task point. The *ikine* function uses the pseudo-inverse of the Jacobian to compute successive numerical

83

solutions for the joint angles required to reach the goal frame, but cannot be set to make sure that the required joint angles are within the valid range for the joint modules. Given this limitation, joint angle limits were not specified for any of the joint modules within the library in any of the results presented in this thesis. Of course, in real systems the joint angle limits are critical, and this key limitation of real, physical systems is not modeled in this research.

While a joint angle optimizer using a genetic algorithm was written in Matlab that would allow for any arbitrary number of DOFs to be used and could take into account joint angles, in practice this became too computationally expensive to employ. In the future, an efficient inverse kinematics solver needs to be developed in order to allow the inclusion of the joint angle limitations. This is, of course, easier said than done.

6.3.2 Trajectory Generation

The method for generating the task trajectories would also need to be upgraded before this work could be put into practice on a real system. Currently, only a Cartesian trajectory is generated and tracked, though the joint-space trajectory of the manipulator is not. Therefore, because only the Cartesian path is tracked, the manipulator could be moving through singularities or passing through points that require it to flip between "elbow up" and "elbow down" poses. A trajectory generator should be used in the future, which can verify that for a given Cartesian path, the manipulator can move smoothly to all points on the path.

6.3.3 Expansion to Broader Solutions

The research presented in this chapter dealt only with the very limited case of single, serial manipulators, though this does not embrace the full range of capabilities that reconfigurable modular robotics have to offer. The system that inspired and motivated this research, MORPHbots, was conceived as a system capable of multi-agent and multi-armed coordinated motion with robots built up from diverse sets of modules. As mentioned in the introduction to this thesis, the real advantage of reconfigurable robotic systems for space applications may be in changing the entire orthodoxy of how servicing and assembly is performed. Moving away from the big-robot-moving-a-little-robot method of performing tasks to a mode where a single self-reconfigurable system is capable of moving to the sight, fixing itself securely to hard-points or handrails near the worksite and then performing the dexterous tasks required has the possibility of creating robots light enough to achieve the MORPHbot vision of small robot systems being launched on small and relatively cheap launchers.

In order to develop this concept, the research in this paper needs to be expanded beyond the realm of single-arm motion to involve multi-armed and multi-agent cooperation in providing the torques and forces specified by the task. The methods used in this research, especially the application of genetic algorithms (GA) to find feasible solutions, could be expanded to tackle this more complicated formulation, however the amount of computational time required to optimize manipulator configurations could be prohibitively expensive. Still, in principle, it is not difficult to see how a GA could be used to design multiple manipulators, determine the force distribution between them, and

plan the motion path. In each of these cases, a GA would work to find acceptable

solutions, and show what the expected mass savings might be.

6.3.4 Reliability Analysis

Another key promise of reconfigurable modular systems is that they are

inherently more reliable than single-topology systems as parts can be easily interchanged

with each other. While this, along with most aspects of reconfigurable robotics, is nearly

always mentioned in the literature, it is seldom (if ever) shown to be true. In the future, a

study of the trade space between carrying extra spares, the reliability of individual

modules, and the overall effect on system mass would be desirable and is an area of

research that should be pursued. Reliability estimation, however, is difficult to do

convincingly without hardware, though such an analysis may provide a target design

point for individual module reliability.

6.4.4 Reconfiguration Cost

In the current work, a reconfiguration of the robot is cost-free. Clearly, there is

some cost associated with breaking down an arm, reconfiguring its modules and

continuing on to the next subtask. A method for appropriately modeling the cost of

reconfiguration is needed. Such a method would ideally impart some cost based on the

number of disconnections/reconnections required to transform one configuration into

another.

Appendix A: Module Library

The library of available parts that are used to form manipulator combinations is made up of link modules and joint modules. There is only one classification of link modules, though the joint modules are divided into single DOF, 2 DOF and 3 DOF classifications. While the link modules simply express a transformation expressing the relationship between the two ports, the joint modules also express the location and axis of motion for each of their degrees of motion.

A.1 Link Modules

The masses of the links were estimated according to Equation 4.7, taking into account a constant mass for each of the two end-caps of the links and the length of the link module itself. While only three types of links were used in this particular thesis, in principle any number could have been used to make the kit. It was not felt that increasing the number of link modules would provide greater insight into the problem of modular reconfigurable robotic optimization or demonstrate the differences between reconfigurable and fixed-topology systems.

Table A.1 gives the properties of the link modules. When an *assembly* matrix is formed, the Part Number value is the number specified in the second row of the matrix and indicates which module to load from the library. The diameter of each of the modules was based on the diameter of link modules from the AMTEC modular robotic system,

and close in size to the diameter of the MORPHbot actuator (8.9 cm) manufactured by

the University of Maryland Space Systems Lab.

Table A.1: Link Module Library

Part Number	Diameter	Total Length	Mass	Outport Location	Description
1	0.70 m	1 m	1.90 kg	$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 \end{bmatrix}$	Straight Link
2	0.70 m	2 m	3.05 kg	$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 2 \\ 0 & 0 & 0 & 1 \end{bmatrix}$	Straight Link
3	0.7 m	1 m	2.15 kg	$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0.5 \\ 0 & -1 & 0 & 0.5 \\ 0 & 0 & 0 & 1 \end{bmatrix}$	90° Corner Link

A.2 Joint Modules

Joint modules in this thesis come in three distinct varieties: 1-DOF (degree of

freedom), 2-DOF and 3-DOF. Each joint module is characterized by assigning a local

frame origin with the z-axis along the axis of its rotation, and the x-y plane as the plane

separating the two moving pieces of the joint. Relative to this frame origin, additional

frames are assigned for the remaining joint body origins (in the case of 2 and 3 DOF

modules) and for the location of the module's input and output ports. Properties of the

modules are defined in the tables below. Three different designs are presented for 1-DOF

joints, two designs for 2 DOF joints and a single design for a 3-DOF intersecting wrist design.

Table A.2: Single Degree-of-Freedom Joint Modules
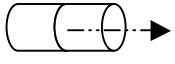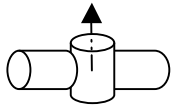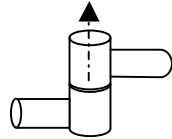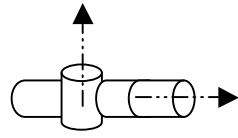
Part Number	Inport Location	Outport Location	Description	Illustration
1	$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & -0.10 \\ 0 & 0 & 0 & 1 \end{bmatrix}$	$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0.10 \\ 0 & 0 & 0 & 1 \end{bmatrix}$	Roll Joint	
2	$\begin{bmatrix} 0 & 0 & 1 & -0.10 \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$	$\begin{bmatrix} 0 & 0 & 1 & 0.10 \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$	Pitch Joint	
3	$\begin{bmatrix} 0 & 0 & 1 & -0.10 \\ 0 & -1 & 0 & 0 \\ 1 & 0 & 0 & -0.10 \\ 0 & 0 & 0 & 1 \end{bmatrix}$	$\begin{bmatrix} 0 & 0 & 1 & 0.10 \\ 0 & -1 & 0 & 0 \\ 1 & 0 & 0 & 0.10 \\ 0 & 0 & 0 & 1 \end{bmatrix}$	Yaw Joint, with offset	

Table A.3: Two Degree-of-Freedom Joint Modules

Part Number		Inport Location	Description	Illustration
1	Inport Location	$\begin{bmatrix} 0 & 0 & 1 & -0.10 \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$	Pitch-Roll Joint	
	Outport Location	$\begin{bmatrix} 0 & 0 & 1 & 0.2 \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$		
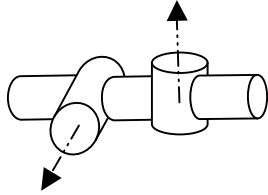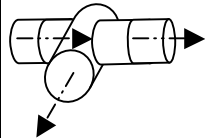
Part Number			Description	Illustration
	DOF #2 Location	$\begin{bmatrix} 0 & 0 & 1 & 0.1 \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$		
2	Inport Location	$\begin{bmatrix} 0 & 0 & 1 & -0.10 \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$	Pitch-Yaw Joint	
	Outport Location	$\begin{bmatrix} 1 & 0 & 1 & 0.10 \\ 0 & 0 & 1 & 0 \\ 0 & -1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$		
	DOF #2 Location	$\begin{bmatrix} 0 & 0 & 1 & 0.20 \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$		

Table A.4: Three Degree-of-Freedom Module

Part Number		Inport Location	Description	Illustration
	Inport Location	$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & -0.10 \\ 0 & 0 & 0 & 1 \end{bmatrix}$	Roll – Pitch – Roll Intersecting Axis Module	
	Outport Location	$\begin{bmatrix} 0 & 0 & -1 & -0.2 \\ 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0.1 \\ 0 & 0 & 0 & 1 \end{bmatrix}$		
	DOF #2 Location	$\begin{bmatrix} -1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0.1 \\ 0 & 0 & 0 & 1 \end{bmatrix}$		

DOF #3 Location	$\begin{bmatrix} 0 & 0 & -1 & -0.1 \\ 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0.1 \\ 0 & 0 & 0 & 1 \end{bmatrix}$		

Appendix B: Subtask Arm Configurations

Table B.1: Universal and Subtask Arm Assemblies

Universal Arm

Assembly Matrix	Mod 1	Mod 2	Mod 3	Mod 4	Mod 5	Mod 6	Mod 7	Mod 8	Mod 9
Module DOF	2	0	0	2	0	0	0	0	2
Module Part	2	2	2	1	2	2	2	2	1
In/Out	1	0	1	1	1	1	0	1	1
Module Rotation	2	0	2	1	1	2	2	1	0
Module Mass	15.47	3.05	3.05	7.14	3.05	3.05	3.05	3.05	1.47

SubTask1 Arm

Assembly Matrix	Mod 1	Mod 2	Mod 3	Mod 4	Mod 5	Mod 6
Module DOF	2	0	2	0	0	2
Module Part	1	2	1	2	2	2
In/Out	1	1	1	0	1	0
Module Rotation	2	1	1	2	3	0
Module Mass	1.47	3.05	3.07	3.05	3.05	1.47
Kit Limiting	0	1	0	1	1	0

SubTask2 Arm

Assembly Matrix	Mod 1	Mod 2	Mod 3	Mod 4	Mod 5	Mod 6	Mod 7	Mod 8	Mod 9
Module DOF	2	0	2	0	0	0	0	0	2
Module Part	2	2	1	2	2	2	2	2	1
In/Out	1	0	1	1	1	0	0	1	1
Module Rotation	0	2	0	1	3	3	1	1	0
Module Mass	1.4	3.05	3.07	3.05	3.05	3.05	3.05	1.4	1.47
Kit Limiting	0	1	0	1	1	0	0	0	0

SubTask3 Arm

Assembly Matrix	Mod 1	Mod 2	Mod 3	Mod 4	Mod 5	Mod 6
Module DOF	2	0	0	2	0	2
Module Part	1	2	2	2	2	1
In/Out	1	0	1	0	0	1
Module Rotation	2	0	0	3	1	3
Module Mass	2.98	3.05	3.05	2.41	3.05	1.45
Kit Limiting	0	0	0	0	0	0

SubTask4 Arm

Assembly Matrix	Mod 1	Mod 2	Mod 3	Mod 4	Mod 5	Mod 6	Mod 7	Mod 8
Module DOF	2	0	0	0	0	0	2	2
Module Part	2	2	2	2	2	2	1	1
In/Out	0	0	1	0	0	0	1	1

Module Rotation	3	3	2	3	1	2	0	2
Module Mass	8.33	3.05	3.05	3.05	3.05	3.05	1.52	1.42
Kit Limiting	0	0	0	0	0	0	0	0

SubTask5 Arm

Assembly Matrix	Mod 1	Mod 2	Mod 3	Mod 4	Mod 5	Mod 6
Module DOF	2	0	0	2	0	2
Module Part	1	2	2	1	2	2
In/Out	0	0	1	1	0	0
Module Rotation	1	3	3	3	1	1
Module Mass	4.54	3.05	3.05	1.87	3.05	1.43
Kit Limiting	1	0	0	1	0	0

SubTask6 Arm

Assembly Matrix	Mod 1	Mod 2	Mod 3	Mod 4	Mod 5	Mod 6	Mod 7	Mod 8
Module DOF	2	0	0	0	0	2	0	2
Module Part	2	2	2	2	2	1	2	1
In/Out	1	0	1	1	1	1	0	1
Module Rotation	2	3	3	3	2	0	3	0
Module Mass	8.4	3.05	3.05	3.05	3.05	1.55	1.87	1.4
Kit Limiting	1	0	0	0	0	0	0	0

Table B.2: Refined Universal and Subtask Arms

Universal Arm

Assembly Matrix	Mod 1	Mod 2	Mod 3	Mod 4	Mod 5	Mod 6	Mod 7	Mod 8	Mod 9	Mod 10
Module DOF	2	0	0	0	2	0	0	0	1	1
Module Part	2	2	2	2	2	2	2	2	2	1
In/Out	1	1	1	0	1	1	0	1	0	1
Module Rotation	1	2	2	0	0	3	3	3	0	0
Module Mass (kg)	9.788	3.05	3.05	3.05	9.56	3.05	3.05	3.05	0.77	0.7

SubTask1 Arm

Assembly Matrix	Mod 1	Mod 2	Mod 3	Mod 4	Mod 5	Mod 6	Mod 7
Module DOF	2	0	0	2	0	1	1
Module Part	2	2	2	2	2	2	1
In/Out	0	0	1	0	1	0	0
Module Rotation	2	3	3	1	2	3	0
Module Mass (kg)	1.4	3.05	3.05	2.957	3.05	0.77	0.7
Kit Limiting	0	0	0	0	0	1	1

SubTask2 Arm

Assembly Matrix	Mod 1	Mod 2	Mod 3	Mod 4	Mod 5	Mod 6	Mod 7	Mod 8	Mod 9	Mod 10
Module DOF	2	0	0	0	2	0	0	0	1	1
Module Part	2	2	2	2	2	2	2	2	2	1
In/Out	1	1	1	0	1	1	0	1	0	1
Module Rotation	1	2	2	0	2	3	3	3	0	3
Module Mass (kg)	1.54	3.05	3.05	3.05	4.66	3.05	3.05	3.05	0.77	0.7
Kit Limiting	0	0	0	0	0	0	0	0	0	0

SubTask3 Arm

Assembly Matrix	Mod 1	Mod 2	Mod 3	Mod 4	Mod 5	Mod 6	Mod 7

Module DOF	2	0	0	2	0	1	1
Module Part	2	2	2	2	2	2	1
In/Out	1	0	0	0	0	0	0
Module Rotation	1	2	3	3	3	0	0
Module Mass (kg)	2.582	3.05	3.05	3.299	3.05	0.72	0.7
Kit Limiting	0	0	0	0	0	0	0

SubTask4 Arm

Assembly Matrix	Mod 1	Mod 2	Mod 3	Mod 4	Mod 5	Mod 6	Mod 7	Mod 8	Mod 9
Module DOF	2	0	0	0	2	0	0	1	1
Module Part	2	2	2	2	2	2	2	2	1
In/Out	1	0	1	0	1	1	1	0	1
Module Rotation	3	2	0	3	3	3	0	0	0
Module Mass (kg)	1.989	3.05	3.05	3.05	4.31	3.05	3.05	0.73	0.7
Kit Limiting	0	0	0	0	0	0	0	0	0

SubTask5 Arm

Assembly Matrix	Mod 1	Mod 2	Mod 3	Mod 4	Mod 5	Mod 6	Mod 7
Module DOF	2	0	0	2	0	1	1
Module Part	2	2	2	2	2	2	1
In/Out	0	0	0	0	1	1	0
Module Rotation	2	0	0	3	3	3	3
Module Mass (kg)	1.4	3.05	3.05	2.94	3.05	0.7	0.7
Kit Limiting	0	0	0	0	0	0	0

SubTask6 Arm

Assembly Matrix	Mod 1	Mod 2	Mod 3	Mod 4	Mod 5	Mod 6	Mod 7	Mod 8	Mod 9
Module DOF	2	0	0	0	2	0	0	1	1
Module Part	2	2	2	2	2	2	2	2	1
In/Out	1	1	1	0	1	0	0	0	1
Module Rotation	1	2	2	0	1	3	2	0	0
Module Mass (kg)	1.4	3.05	3.05	3.05	4.34	3.05	3.05	0.7	0.7
Kit Limiting	0	0	0	0	0	0	0	0	0

SubTask7 Arm

Assembly Matrix	Mod 1	Mod 2	Mod 3	Mod 4	Mod 5	Mod 6	Mod 7
Module DOF	2	0	0	2	0	1	1
Module Part	2	2	2	2	2	2	1
In/Out	1	1	1	0	1	0	1
Module Rotation	1	2	0	1	3	3	1
Module Mass (kg)	2.051	3.05	3.05	2.868	3.05	0.74	0.7
Kit Limiting	0	0	0	0	0	0	0

SubTask8 Arm

Assembly Matrix	Mod 1	Mod 2	Mod 3	Mod 4	Mod 5	Mod 6	Mod 7	Mod 8	Mod 9	Mod 10
Module DOF	2	0	0	0	2	0	0	0	1	1
Module Part	2	2	2	2	2	2	2	2	2	1
In/Out	1	1	1	0	1	0	0	1	1	1
Module Rotation	3	2	2	1	1	1	3	1	0	3
Module Mass (kg)	2.376	3.05	3.05	3.05	6.63	3.05	3.05	3.05	0.76	0.7
Kit Limiting	0	0	0	0	0	0	0	0	0	0

SubTask9 Arm

Assembly Matrix	Mod 1	Mod 2	Mod 3	Mod 4	Mod 5	Mod 6	Mod 7
Module DOF	2	0	0	2	0	1	1
Module Part	2	2	2	2	2	2	1
In/Out	0	1	1	0	1	1	1
Module Rotation	3	0	0	3	0	0	1
Module Mass (kg)	5.559	3.05	3.05	2.873	3.05	0.77	0.7
Kit Limiting	0	0	0	0	0	0	0

SubTask10 Arm

Assembly Matrix	Mod 1	Mod 2	Mod 3	Mod 4	Mod 5	Mod 6	Mod 7	Mod 8	Mod 9	Mod 10
Module DOF	2	0	0	0	0	0	2	0	1	1
Module Part	2	2	2	2	2	2	2	2	2	1
In/Out	1	0	0	0	1	1	1	1	0	1
Module Rotation	2	2	2	0	3	0	2	1	0	0
Module Mass (kg)	9.706	3.05	3.05	3.05	3.05	3.05	2.93	3.05	0.77	0.7
Kit Limiting	1	0	0	0	0	0	1	0	0	0

Appendix C: Matlab Functions

This section includes all of the functions used in this thesis written by the author.

Functions that were a part of the Genetics Algorithm Toolbox and the Robotics Toolbox,

as well as standard Matlab functions are not included in this appendix.

Function: Assembly to Denavit-Hartenberg (Assy2DH)

```matlab
function [dh_out, TBase, LinkMass, JointLimit ]= Assy2DH(Assy)
%
% function [DH, TBase, LinkMass, JointLims] = Assy2DH(Assy)
%
% This function is intended to convert the Assembly matrix of a given
% manipulator into that manipulator's DH parameters.
% Assembly matrix is of the form:
% Assembly = [ <ModuleType> <ModuleType> <ModuleType>... ]
%            [ <PartType>   <PartType>   <PartType> ...  ]
%            [ <In/Outport> <In/Outport> <In/Outport> ...]
%            [ <Rotation>   <Rotation>   <Rotation> ...  ]
% where Modtype indicates the type of part as follows:
% Link = 0
% Gripper = 0
% 1 DOF joint module = 1
% 2 DOF joint module = 2
% 3 DOF joint module = 3
% PartType refers to the part number of the particular module in the module
% library. At this point, grippers are stored in the same catelogue as the
% links, as gripper properties are not being modeled beyond their tooltip
% transformation
%
% This function will access the file named 'modlibrary.mat'to find the
% joint and link parameters contained within. Modlibrary should contain two
% structures: J and L that contain the fields Inport, Outport for links and
% 1-DOF modules, while 2 and 3 DOF modules will have intermediate fields
% indicating the relationship between the sub-actuators.
%
% This function is similar to the algorithm developed in Bi, pg 74.
%
% Edit History:
% 03-13-06: Added capability for handling negative values in the assembly
% matrix. Effectively, negative entries are not valid and are ignored in
% the construction of the DH parameters.
%
% 03-22-06: Added support for allowing either port of a module to be
% connected to the preceeding module (Row 3 of the Assembly Matrix)
%
% 03-29-06: Added support for changes in the rotation between modules (Row
% 4 of the Assembly Matrix)
%
% 04-03-06 Added support for 2 and 3 DOF modules
%
% 04-06-06 Added the TBase output, which is the transform from the base
% frame coordinates to the origin of the first joint frame. This
% information is lost if just the DH parameters are passed out of the
% function.
%
% 04-19-06 Added LinkMass as an output parameter.
%
% 06-08-06 Fixed a problem related to numerical round-off
%
% 06-26-06 Added joint limits as an output parameter

% Degrees of freedom = Sum of Modtypes
```

```matlab
global L DOF1 DOF2 DOF3
n=sum((Assy(1,:)>0).*Assy(1,:));
p=length(Assy);
LB=[1 0 0 0; 0 -1 0 0; 0 0 -1 0; 0 0 0 1];  %180 degree rotation about x
%Links=zeros(sum(Assy(1,:) == 0),1); %Initialize length of link and joint vectors
%Joints=zeros(n,0);
isjoint=zeros(1,n);
isforward=zeros(1,n);
rot= zeros(1,n);
ROT= zeros(4,4,4);     %Rotation matricies for forward cases
BROT = zeros(4,4,4);   %Rotation matricues for backwards cases
for i=0:3       %Precalculate the transformations for rotations in Assy(4,:)
    ROT(:,:,i+1) = round([cos(i*pi/2) -sin(i*pi/2) 0 0; sin(i*pi/2)  cos(i*pi/2) 0 0;...
                     0                0       1 0;    0                 0       0 1]);

    %Rotation matrix if part is backwards
    BROT(:,:,i+1) = round([cos(-i*pi/2) -sin(-i*pi/2) 0 0; sin(-i*pi/2)  cos(-i*pi/2) 0
0;...
                     0                0       1 0;    0                 0       0 1]);
end


i=1;, j=1; m=1; linkcount = 0;
while i <= p    % Load Link and Joint Module data for the archetypes specified in the
assembly
    if Assy(1,i) == 0  % Module is a link or gripper, assemble Link structure.
        linkcount = linkcount +1;
        LJ{j}= L{Assy(2,i),1};
        isforward(j) = Assy(3,i);
        isjoint(j)=0;
        LMass(linkcount) = L{Assy(2,i),1}.mass;
        rot(j)= Assy(4,i);
        j=j+1;
    elseif Assy(1,i)==1          % 1 DOF Joint module
        LJ{j}=DOF1{Assy(2,i),1};
        isjoint(j)=1;
        isforward(j) = Assy(3,i);
        rot(j)= Assy(4,i);
        j=j+1;
    elseif Assy(1,i) == 2        % 2 DOF Joint module
        isforward(j:j+1) = Assy(3,i);
        if isforward(j) == 1
            LJ{j}=  DOF2{Assy(2,i),1};
            LJ{j+1}=DOF2{Assy(2,i),2};
        else
            LJ{j}= DOF2{Assy(2,i),2};
            LJ{j+1}=DOF2{Assy(2,i),1};
        end
        isjoint(j:j+1)=1;
        rot(j:j+1)= [Assy(4,i), 0];
        j=j+2;
    elseif Assy(1,i) == 3          % 3 DOF Joint module
        isforward(j:j+2) = Assy(3,i);
        if isforward(j) == 1
            LJ{j}=  DOF3{Assy(2,i),1};
            LJ{j+1}=DOF3{Assy(2,i),2};
            LJ{j+2}=DOF3{Assy(2,i),3};
        else
            LJ{j}=  DOF3{Assy(2,i),3};
            LJ{j+1}=DOF3{Assy(2,i),2};
            LJ{j+2}=DOF3{Assy(2,i),1};
        end
        isjoint(j:j+2)=1;
        rot(j:j+2)= [Assy(4,i), 0, 0];
        j=j+3;
    else
        %If value of ModuleType is anything other than 0,1,2,3 ignore the module
    end

    i=i+1;
end %while i<=p
if linkcount == 0
```

```matlab
    LMass = 0;
end
% Determine locations and orientations of Z axes.
% -----------------------------------------------

%Identify joint bodies.
LT=repmat(eye(4), [1 1 n+1]);  %Initialize Link transformations
%z=zeros(4,4,n);
%zhat=zeros(3,n);   %Matrix of z unit directions expressed in base frame
jix=1;
exitloop = 0;
while jix <= n+1     % While there are still joint axes to assign

        while isjoint(m)==0  %Continue until finding a joint body

            if isforward(m)==0  %If it is assembled backwards, use the inverse
                LT(:,:,jix)=LT(:,:,jix)*LB*BROT(:,:,rot(m)+1)*inv(LJ{m}.Outport)*LB;
            else                %Otherwise use the outport location
                LT(:,:,jix)=LT(:,:,jix)*ROT(:,:,rot(m)+1)*LJ{m}.Outport;
            end
            m=m+1;
            if m > length(isjoint) % Last frame is located at end-effector
                %Only location of z is important. Orientation is assigned
                %to be the same as that of the last joint
                z(:,:,jix) = z(:,:,jix-1)*JointOut*LT(:,:,jix);
                zhat(:,jix)= zhat(:,jix-1);
                jix=jix+1;
                exitloop = 1;
                break
            end
        end
        if exitloop == 1
            break
        end
        if jix==1  %For first joint only
            if isforward(m) == 1
                z(:,:,jix)=LT(:,:,jix)*ROT(:,:,rot(m)+1)*inv(LJ{m}.Inport);
                JointOut=LJ{m}.Outport;
            else
                z(:,:,jix)=LT(:,:,jix)*LB*BROT(:,:,rot(m)+1)*inv(LJ{m}.Outport)*LB;
                JointOut = LB*LJ{m}.Inport*LB;
            end
            zhat(:,jix)=z(1:3,1:3,jix)*[0;0;1];
            JointLimit(jix,:)=LJ{m}.qlim;
            mj=m;  %Store index of joint
            m=m+1;
            jix=2;
        else
            if isforward(m) == 1
                z(:,:,jix)=z(:,:,jix-
1)*(JointOut*LT(:,:,jix)*ROT(:,:,rot(m)+1)*inv(LJ{m}.Inport));
                zhat(:,jix)=z(1:3,1:3,jix)*[0;0;1];  % Direction of z expressed in base
frame
                JointOut= LJ{m}.Outport;
                JointLimit(jix,:)=LJ{m}.qlim;
            else
                %z(:,:,jix)=z(:,:,jix-
1)*(JointOut*LT(:,:,jix)*BROT(rot(m)+1)*LB*inv(LJ{m}.Outport));
                z(:,:,jix)=z(:,:,jix-
1)*(JointOut*LT(:,:,jix)*LB*BROT(:,:,rot(m)+1)*inv(LJ{m}.Outport)*LB);
                zhat(:,jix)=z(1:3,1:3,jix)*[0;0;1];
                JointOut=LB*LJ{m}.Inport*LB;
                JointLimit(jix,:)=LJ{m}.qlim;
            end
            mj=m; %Store index of joint
            jix=jix+1;
            m=m+1;
            if m > length(isjoint) %Assign final frame and break the loop if the chain
ends in a joint module
                z(:,:,jix)= z(:,:,jix-1)*JointOut;
                zhat(:,jix)=zhat(:,jix-1);
            break
            end
```

97

```matlab
        end
end


%Direction of final z-axis (EE frame) is assigned to be in the same
%direction as the final joint's z axis and its position should be at the
%distal end of the end of the manipulator


%NOTE: z(i) is actually equal to z(i-1). That is to say, the first column
%of the matrix z is actually the direction of the z0 axis and not the z1
%axis. The matrix z looks like a transformation (a 4x4 matrix with [0 0 0
%1] as the last row) with the upper left 3x3 matrix expressing the rotation
%from the base frame to the current frame. The "position" component of each
%z transformation is the location of a point that the z-vector must pass
%through: the pivot point of a rotary joint. This point and orientation are
%used to define two points on the z-axis and are used to find the length and
%direction of the perpendicular to the z+1 axis in order to find the x axis
%as well as the D-H parameter A.

% Assign origins for each of the frames and calculate DH parameters
% -----------------------------------------------------------------

%Initialize the DH output parameters
A=zeros(n,1);
Alpha=zeros(n,1);
D=zeros(n,1);
Theta=zeros(n,1);

%Assign origin of frame 0 (has index of 1)
O(:,1)= z(1:3,4,1);    %Origin is at origin of joint module 1
if dot([0;0;1],zhat(:,1))==1  %If axis of joint 1 is global z
    xhat(:,1)=[1;0;0];        %Assign x0 to be global x
else
    xcross=cross(zhat(:,1),[0;0;1]);
    xhat(:,1)=xcross/norm(xcross);
end

for i=2:n+1  %Loop over the joints, assigning x axes and calculating DH params
    %X axes are assigned in the same manner
    p1=z(1:3,4,i-1);             %First point on z(i-1) is origin of joint mod z(i)
    p2=z(1:3,4,i-1)+zhat(:,i-1);  %Second point on z(i-1)
    p3=z(1:3,4,i);           %First point on z(i) is origin of joint mod z(i+1)
    p4=z(1:3,4,i)+zhat(:,i); %Second point on z(i)
    a=p2-p1;
    b=p4-p3;
    c=p3-p1;
    abcross=cross(a,b);

    if norm(abcross)<= 0.001  % z(i-1) and z(i) are parallel
        %disp('Running Parallel');
        %r=[ det([a(2) a(3); b(2) b(3)]) det([a(1) a(3); b(1) b(3)]) det([a(1) a(2); b(1)
b(2)])];
        %r= [det([-c(2), -c(3); zhat(2,i) zhat(3,i)]), det([-c(3), -c(1); zhat(3,i)
zhat(1,i)]),...
        %    det([-c(1), -c(2); zhat(1,i) zhat(2,i)])] %Distance between the axes
        t= dot(-c,zhat(:,i-1));
        r= p3-p1-zhat(:,i-1)*t;
        xhat(:,i)=r/norm(r);
        if sum(xhat(:,i)-zhat(:,i) <= 0.001)  %If zhat(i) and zhat(i-1) are the same line
            xhat(:,i) = xhat(:,i-1);
        end
        O(:,i)= z(1:3,4,i);  %Origin of frame (i) is set to be at the origin of joint
body i+1
        A(i-1)= abs(dot(xhat(:,i),c)); % DH paramater
        D2=sum(c.^2)-A(i-1)^2;
        if abs(D2) < 1e-8
            D2 = 0;
        end
        D(i-1)= sqrt(D2); % DH parameter D(i) by pythagorean

    elseif abs(dot(c,abcross)) <= 0.001   % Axes intersect
        %disp('Running Intersect');
        O(:,i)=p1+a* (dot(cross(c,b),abcross))/ sum(abcross.^2); %Origin of frame (i)
```

```
        xhat(:,i)=abcross;  % Unit vector x in base frame
        A(i-1)=0;
        D(i-1)=norm(O(:,i)-O(:,i-1)); %D is simply straightline distance since A is known
to be 0
    else  % z(i-1) and z(i) are skew
        %disp('Running Skew');
        A(i-1)= norm(dot(c,abcross))/(norm(abcross)); %Distance from z(i-1) to z(i)
        xhat(:,i)=abcross;
        %Calculate origin
        beta=[p1-p2 p4-p3];
        gamma= [xhat(:,i)+p1-p3];
        if det(beta(1:2,:))==0  %If eq 1 and 2 insufficient, use 2 and 3 instead
            if det(beta(2:3,:)) == 0;  %If equation 2 and 3 also insufficient, use 1 and
3
                BIx = [1 3];
            else
            BIx = [2 3];
            end
        else
            BIx = [1 2];           %Otherwise use the first two
        end
        s = inv(beta(BIx,:))*gamma(BIx);
        I = p1 +(p2-p1).*s(1);
        O(:,i)=p3+(p4-p3).*s(2);
        D(i-1)= abs(dot(O(:,i-1)-I,zhat(:,i-1)));
    end
    abdot = dot(a,b);
    if abs(1-abdot) < 0.0000001
        abdot = 1;
    elseif abs(abdot+1) < 0.0000001
        abdot = -1;
    end
    Alpha(i-1)=acos(abdot);
    xdot = dot(xhat(:,i),xhat(:,i-1));
    if abs(1-xdot) < 0.0000001
        xdot = 1;
    elseif abs(xdot+1) < 0.000001
        xdot = -1;
    end
    Theta(i-1)=acos(xdot);

end
TBase = z(:,:,1);
LinkMass = sum(LMass);
dh_out= [Alpha A Theta D];

for p=1:numrows(dh_out)
    for q=1:numcols(dh_out)
        if isreal( dh_out(p,q) )==0
            Assy
            error('DH parameters have imaginary components')
        end
    end
end
```

Function: Parametric Tasks (ParaTasks)

```
function [ForceTraj, PosTraj] = ParaTasks(TrajPoints, Task, TCent, MotionRange)
%function [ForceTraj, PosTraj] = ParaTasks(TBase, TrajPoints, Task, TCent,
%MotionRange)
%
% This function is intended to generate parametric tasks according to
% integer specified by the input Task
% The tasks are translated automatically into the base frame of the robot,
% which may be different than the global frame
%
```

```
% Table of Tasks:
% -----------------------------
% 1. Translation in the direction of global X axis
%    Motion is a straight line with centerpoint TCent, length should be
%    specified by the first element of MotionRange
%    Force applied is unit force in the direction of travel
% 2. Translation in the direction of global Y axis
%    Motion is a straight line with centerpoint TCent, length should be
%    specified by the first element of MotionRange
%    Force applied is unit force in the direction of travel
% 3. Translation in the direction of global Z axis
%    Motion is a straight line with centerpoint TCent, length should be
%    specified by the first element of MotionRange
%    Force applied is unit force in the direction of travel
% 4. Rotational motion about the point specified by TCent with the radius
%    the motion specified by MotionRange = [Radius, Angle1, Angle2] with
%    constant orientation. If Angles are omitted, a full circle is assumed.
%    If only one angle is specified, it is assumed that the motion is from
%    0 to Angle1. Angles are measured about the Z-axis of the frame
%    specified by TCent with angle 0 corresponding to the X-axis. Force
%    applied is unit force in the direction of travel.
% 5. Rotational motion about the point specified by TCent with the radius
%    of the motion specified by MotionRange = [Radius, Angle1, Angle2] with
%    the orientation maintained as being normal to the circle. If only one
%    angle is specified, it is assumed that motion is from 0 to Angle1.
%    Angles are measured about the Z-axis of the frame specified by TCent
%    with angle 0 corresponding to the X-axis. Force is directed in the
%    direction of motion. This task simulates the opening of a door about a
%    hinge.
%


switch Task
    case 1    %Translation along Base's X axis
        %Control Variables
        %Centerpoint Offsets
        L = MotionRange(1);
        StartPoint = TCent*transl(-L/2, 0, 0);
        EndPoint = TCent*transl(L/2, 0, 0);
        PosTraj = ctraj(StartPoint, EndPoint, TrajPoints);
        Force = [100 0 0 0 0 0];
        ForceTraj = repmat(Force,TrajPoints,1);

    case 2 %Translation along Base's Y axis
        L = MotionRange(1);
        StartPoint = TCent*transl( 0, L/2, 0);
        EndPoint = TCent*transl( 0, -L/2, 0);
        PosTraj = ctraj(StartPoint, EndPoint, TrajPoints);
        Force = [0 100 0 0 0 0];
        ForceTraj = repmat(Force,TrajPoints,1);


    case 3 %Translation along Base's Z axis
        L = MotionRange(1);
        StartPoint = TCent*transl(0,0,-L/2);
        EndPoint = TCent*transl(0,0,L/2);
        PosTraj = ctraj(StartPoint, EndPoint, TrajPoints);
        Force=[0 0 100 0 0 0];
        ForceTraj= repmat(Force,TrajPoints,1);

    case 4
        %Circular translation without rotation. Rotation is about the point
        %specified in TCent and the plane of the motion is the X-Y plane of
        %the rotation portion of the translation (rotation about Z axis)
        if length(MotionRange) == 1
         R = MotionRange(1);
         Theta1 = 0;
         Theta2 = 2*pi;
        elseif length(MotionRange) == 2
         R = MotionRange(1);
         Theta1 = 0;
         Theta2 = MotionRange(2);
```

```matlab
    elseif length(MotionRange) == 3
     R = MotionRange(1);
     Theta1 = MotionRange(2);
     Theta2 = MotionRange(3);
    end
     Theta = linspace(Theta1,Theta2,TrajPoints);
     %R.*cos(Theta)
     %R.*sin(Theta)
     %zeros(TrajPoints)
     %PosTraj = TCent*transl(R.*cos(Theta), R.*sin(Theta), zeros(TrajPoints));
     if Theta2 > Theta1
         for i=1:TrajPoints
             ForceTraj(i,:) = 100*[cos(Theta(i)+pi/2); sin(Theta(i)+pi/2); 0; 0; 0;
0];
             PosTraj(:,:,i) =TCent*transl(R*cos(Theta(i)),R*sin(Theta(i)),0);
         end
     elseif Theta1 > Theta2
         for i=1:TrajPoints
             ForceTraj(i,:) = 100*[-cos(Theta(i)+pi/2); sin(Theta(i)+pi/2); 0; 0; 0;
0];
             PosTraj(:,:,i) =TCent*transl(R*cos(Theta(i)),R*sin(Theta(i)),0);
         end
     end

    case 5
        %Circular translation with rotation of the end-effector. This
        %simulates the opening of a door pivoting about a hinge located at
        %TCent.
        if length(MotionRange) == 1
         R = MotionRange(1);
         Theta1 = 0;
         Theta2 = 2*pi;
        elseif length(MotionRange) == 2
         R = MotionRange(1);
         Theta1 = 0;
         Theta2 = MotionRange(2);
        elseif length(MotionRange) == 3
         R = MotionRange(1);
         Theta1 = MotionRange(2);
         Theta2 = MotionRange(3);
        end
        Theta = linspace(Theta1, Theta2, TrajPoints);
        for i=1:TrajPoints
            % Transformation from TCent frame into required EE frame
            T = [cos(pi+Theta(i)) -sin(pi+Theta(i)) 0 R*cos(Theta(i));
                 sin(pi+Theta(i))  cos(pi+Theta(i)) 0 R*sin(Theta(i));
                    0                 0          1     0;
                    0                 0          0     1];
            PosTraj(:,:,i) = TCent*T;
            if Theta2 > Theta1
                ForceTraj(i,:) = 100*[cos(Theta(i)+pi/2); sin(Theta(i)+pi/2); 0; 0; 0;
0];
            elseif Theta1 > Theta2
                ForceTraj(i,:) = 100*[-cos(Theta(i)+pi/2); sin(Theta(i)+pi/2); 0; 0; 0;
0];
            end
        end
    case 6
        %Circular translation with rotation of the end-effector. This
        %simulates the opening of a door pivoting about a hinge located at
        %TCent.
        if length(MotionRange) == 1
         R = MotionRange(1);
         Theta1 = 0;
         Theta2 = 2*pi;
        elseif length(MotionRange) == 2
         R = MotionRange(1);
         Theta1 = 0;
         Theta2 = MotionRange(2);
        elseif length(MotionRange) == 3
         R = MotionRange(1);
         Theta1 = MotionRange(2);
         Theta2 = MotionRange(3);
```

```
        end
    Theta = linspace(Theta1, Theta2, TrajPoints);
    for i=1:TrajPoints
        T = [ cos(pi+Theta(i)) 0 sin(pi+Theta(i)) R*cos(Theta(i))
              0 1 0 0
              -sin(pi+Theta(i)) 0 cos(pi+Theta(i)) R*sin(Theta(i))
              0 0 0 1];
        PosTraj(:,:,i) = TCent*T;
        if Theta2 > Theta1
            ForceTraj(i,:) = 100*[cos(Theta(i)+pi/2); 0; sin(Theta(i)+pi/2); 0; 0;
0];
        elseif Theta1 > Theta2
            ForceTraj(i,:) = 100*[-cos(Theta(i)+pi/2); 0; sin(Theta(i)+pi/2); 0; 0;
0];
        end
    end
end %Switch Task
```

Function: ParaACGE

```
function [Best, BestAssy, PopAssy, ObjValCh, PopAvg, JointMass] =
ACGEPartLimitedOpt(ForceTraj, PosTraj, RUNS, UKit, UnivAssy)
% AUTOMATIC CONFIGURATION GENERATION AND EVALUATION
%
% This script generates configurations of modular robotic manipulators,
% and evaluates them to determine their fitness acording to an objective
% function, then uses a genetic loop to recombine and mutate the
% population to search for more appropriate solutions.
%
% The manipulators evaluated are composed of the elements specified by
% Ukit, the kit of the "universal" arm.
%
% Inputs are the force and position trajectory points of the tasks being
% performed, the variable RUNS controls how the data is saved, UKit is the
% kit of the universal arm, and UnivAssy is the assembly of the universal
% arm.
%
% Outputs are the Best individual found, the assembly matrix of the best
% individual, the final population of assemblies, the Objective values
% associated with the population elements, the Population average at each
% generation, and the mass of the joints of the best assembly.
%
% Edit History:
% 03-30-06  Incorporated recombination and mutation for module orientation
%           (Rows three and four of the assembly matrix).
% 04-04-06  Allowed the inclusion of 2 and 3 DOF modules
% 04-07-06  Introduced Recombination and Mutation Rate constants
% 05-10-06  Incorporated variable base starting positions.
% 06-08-06  Modified for parametric tasks
% 06-14-06  Fixed a major bug relating to updates of the Assy matricies for
%           each generation
% 06-22-06  Functionalized ParaACGE in order to allow it to be run multiple
%           times back to back. Non-functionalized version saved to
%           ParaACGEbkup.m
% 08-09-06  Added elements to assist in constructing optimized kits.

global L J
tic                                 %Start Timer
DOF=6;                              %Desired DOF
M = 0.025;                           %Mass Weighting
N = 1;                              %Number of Modules Weighting
D = 0;                             %Dexterity Weighting
NIND=30;                            %Number of Individuals per Generation
MAXGEN=100;                         %Number of Generations to run
MAXMOD=15;                          %Maximum number of Modules
MINMOD=5;                           %Minimum number of Modules
MAXJ1= DOF;                         %Maximum length of 1-DOF vector
MAXJ2= floor(DOF/2);                %Maximum length of 2-DOF vector
MAXJ3= floor(DOF/3);                %Maximum length of 3-DOF vector
MAXLINK= MAXMOD-2;                   %Maximum length of link vector
```

```
MAXASSY= MAXLINK+MAXJ3+MAXJ2+MAXJ1; %Maximum length of Assembly
GGAP=0.9;                           %Generation gap. # of offspring = GGAP*NIND
MUTRATE = 0.10;                     %Mutation Rate
XOVER = 0.60;                       %Cross-over rate
J1=3; J2=2; J3=1;  %Number of parts in each of the joint inventories
nL= 2;%Number of parts in the link inventory, excluding grippers
BASELOC = [1 0 0 0; 0 1 0 0; 0 0 1 0; 0 0 0 1];
KitParts = UKit.parts;


% Loop over the elements in ModType
NMods =zeros(NIND,1);   %Initialize NMods
ModType(1:NIND, 1:MAXMOD) = -1;

% Pick off module masses for UnivAssy
Jix = 1;
n_mods = 1;
while n_mods <= size(UnivAssy,2)
    if UnivAssy(1,n_mods) > 0
        UJointMass(Jix) = UnivAssy(5,n_mods);
        Jix = Jix + 1;
    end
    n_mods = n_mods + 1;
end
UJointMass


for k=1:NIND
    i=0;
    d=0;
    n_mods=0;
    clear Module


    Parts = [];
    R = [];
    count = 1;
    for f=1:numrows(KitParts);
        for g=1:numcols(KitParts);
            if KitParts(f,g) ~= 0
                Parts=[Parts repmat([f;g],1,KitParts(f,g))];
                count = count + KitParts(f,g);
            end
        end
    end
    Parts = Parts-(Parts==4).*4;
    R = rand(1,size(Parts,2));
    [R RIx]=sort(R);
    Module=Parts(1,RIx);
    PartType = Parts(2,RIx);


        NMods(k) = length(Module);
        n_mods = length(Module);
        diff = MAXMOD - n_mods;
        if diff > 0
        Module(1,n_mods+1:1:n_mods+diff)= -1;
        end
        ModType(k,:) = Module(1,:);

    for j=1:MAXMOD;
    % Read the module type and assign a part type within the bounds of the part
    % library
        if ModType(k,j)==0    %If it's a link or a gripper
            if j==n_mods  %Make last element a gripper
                % PartType(j)=3;
                  InOut(j)=1;
                  Rot(j)=floor(4*rand(1));
            else
                % PartType(j)=round((nL-1)*rand(1)+1);
                  InOut(j)=round(rand(1));
                  Rot(j)=floor(4*rand(1));
```

```matlab
                end
        elseif ModType(k,j)==1 %If it's a 1 DOF
            %PartType(j)=ceil(J1*rand(1));
            InOut(j)=round(rand(1));
            Rot(j)=floor(4*rand(1));
        elseif ModType(k,j)==2 %If it's 2 DOF
            %PartType(j)=ceil(J2*rand(1));
            InOut(j)=round(rand(1));
            Rot(j)=floor(4*rand(1));
        elseif ModType(k,j)==3 %If it's 3 DOF
            %PartType(j)=ceil(J3*rand(1));
            InOut(j)=round(rand(1));
            Rot(j)=floor(4*rand(1));
        else  %Not a valid Module, assign elements to be -1
            PartType(j) = -1;
            InOut(j) = -1;
            Rot(j) = -1;
        end

    end %for j= 1:MAXMOD


    % The base of the chromosome for PartType should be the size of the largest
    % part library used (Either Joint or Link). In other words, the base should
    % be max([L, J1, J2, J3]). The base of InOut should be 2 (binary), and the
    % base of Rot should be 4.
    TypeBase=max([nL, J1, J2, J3]);
    % Construct chromosome:
    Chrom(k,4:4:4*MAXMOD)= Rot;      % Elements 4,8,12... Indicate rotations
    Chrom(k,3:4:4*MAXMOD)= InOut;    % Elements 3,7,11... indicate In/Outport
    Chrom(k,2:4:4*MAXMOD)= PartType; % Elements 2,6,10.. are PartTypes
    Chrom(k,1:4:4*MAXMOD)= ModType(k,:); % Elements 1,5,9... are Moduletypes

    if k == 1
        Chrom(1,4:4:4*MAXMOD) = UnivAssy(4,:);
        Chrom(1,3:4:4*MAXMOD) = UnivAssy(3,:);
        Chrom(1,2:4:4*MAXMOD) = UnivAssy(2,:);
        Chrom(1,1:4:4*MAXMOD) = UnivAssy(1,:);
    end

    PopAssy(:,:,k)=[Chrom(k,1:4:4*MAXMOD); Chrom(k,2:4:4*MAXMOD);
Chrom(k,3:4:4*MAXMOD);...
        Chrom(k,4:4:4*MAXMOD)];
    if k == 1
        PopAssy(:,:,1) = UnivAssy(1:4,:);
    end

    [DH, TBase, LinkMass, JointLimits] =Assy2DH(PopAssy(:,:,k));
    TrialConfig=robot(DH);
    TrialConfig.qlim = JointLimits;
    TBase = BASELOC*TBase;

        % OBJECTIVE FUNCTION
        %------------------------------------------------------------
        [ObjValue MaxTorque JMass] = ParaEvaluateObjective(TrialConfig, TBase , LinkMass,
M, D, N,PosTraj,ForceTraj);
        % Check the joint masses against the joint masses of the universal arm.
        if ObjValue < 0 & isempty(JMass) == 0
            TrialKit = Assy2Kit(PopAssy(:,:,k),JMass);
            for a = 1:numcols(TrialKit.mass)
                for b = 1:numrows(TrialKit.mass)-1
                    if isempty(TrialKit.mass{b,a}) == 0
                        for c = 1:length(TrialKit.mass{b,a})
                            if TrialKit.mass{b,a}(c) > UKit.mass{b,a}(c)
                                ObjValue = 1;
                            end
                        end
                    end
                end
            end
        end
        ObjValCh(k,1) = ObjValue;
        %------------------------------------------------------------
```

```matlab
end %for k=1:NIND
%Arrange initial population according to the number of modules
[NMods,ModIndex] = sortrows(NMods);
for i=1:NIND
    SortChrom(i,:) = Chrom( ModIndex(i), :);
    SortObj(i,:)   = ObjValCh( ModIndex(i), :);
    SortAssy(:,:,i) = PopAssy(:,:,ModIndex(i), :);
end
Chrom = SortChrom;
ObjValCh = SortObj;
PopAssy = SortAssy;

BaseVec = crtbase( [n_mods, n_mods, n_mods, n_mods], [2,TypeBase,2,4]);
gen=0;
% Track Best individual
figure(1);
    [Best(gen+1), BestIx] = min(ObjValCh);
    BestAssy(:,:,gen+1) = PopAssy(:,:,BestIx);
    plot(Best,'ro');xlabel('generation'); ylabel('Best');
    text(0.5,0.95,['Best = ', num2str(Best(gen+1))],'Units','normalized');
    drawnow;


disp('BEGIN GENERATIONAL LOOP')

% BEGIN GENERATIONAL LOOP
% -------------------------------------------------------------------------
while gen < MAXGEN

    %Assign fitness values for the entire population
    Fitness = ranking(ObjValCh);

    %Select Individuals for breeding
    SelCh=select('sus',Chrom,Fitness,GGAP);


    NSel = size(SelCh,1);
    %Initialize the subchromosomes to be the proper size
    clear J1Ch J2Ch J3Ch LinkCh ModCh InOut1Ch InOut2Ch InOut3Ch InOutLCh...
            Rot1Ch Rot2Ch Rot3Ch RotLCh TBase
    J1Ch(1:NSel,1:MAXJ1)= -1;
    J2Ch(1:NSel,1:MAXJ2)= -1;
    J3Ch(1:NSel,1:MAXJ3)= -1;
    LinkCh(1:NSel,1:MAXLINK)= -1;
    InOut1Ch=J1Ch;
    InOut2Ch=J2Ch;
    InOut3Ch=J3Ch;
    InOutLCh=LinkCh;
    Rot1Ch=J1Ch;
    Rot2Ch=J2Ch;
    Rot3Ch=J3Ch;
    RotLCh=LinkCh;

    %Recombine and mutate just the ModuleType portion of the Assembly
    %matrix first, as this infomation is needed in order to properly
    %perform recombination on lower-leveled elements of the matrix.

    %Recombine and mutate the module types (row 1 of Assy)

    ModCh = SelCh(:, 1:4:4*MAXMOD);
    ModCh = recombin('xovsp',ModCh, XOVER);
    ModBase= crtbase(MAXMOD, 3);
    ModCh = mut(ModCh, MUTRATE, ModBase);

    %Reinsert the modified (recombined/mutated) ModuleType back into SelCh
    SelCh(:,1:4:4*MAXMOD) = ModCh;

    %Loop over each individual in the population and generate the
    %sub-chromosomes for ModType, PartType, InOut, and Rot
    for ind=1:NSel
```

```matlab
        %Check for module type
        j1=0;j2=0;j3=0;li=0; junk=0;typei=0; %Initialize indices for joints and links
        nmods =  (sum(SelCh(ind,:) >= 0))/4;
        for i=0:MAXMOD-1
            type=1+4*i;
            part= type+1;
            inout=type+2;
            rot=type+3;
            PartCh(ind,i+1)=SelCh(ind,part);
            InOutCh(ind,i+1) = SelCh(ind,inout);
            RotCh(ind,i+1) = SelCh(ind,rot);

        end %for i=0:NMods-1
    end %for ind=1:NSel


    % Recombine and mutate each of the Joint and Link subchromosomes
    % independently according to category along with their respective rotation
    % and in/out orientations

    PartCh = recombin('xovsp',PartCh, XOVER);
    InOutCh= recombin('xovsp',InOutCh,XOVER);
    RotCh  = recombin('xovsp',RotCh,  XOVER);

    PartBase = crtbase(MAXMOD, J1);
    InOutBase = crtbase(MAXMOD, 2);
    RotBase=crtbase(MAXMOD, 4);

    PartCh= modmut(PartCh, MUTRATE, PartBase);
    InOutCh = mut(InOutCh, MUTRATE, InOutBase);
    RotCh = mut(RotCh, MUTRATE, RotBase);

    %Loop over each individual in the population and each element of its
    %ModCh chromosome in order to determine whether the elements of PartCh,
    %InOutCh and RotCh are all within the bounds required by ModCh. If they
    %are not inside of these bounds, then assign them such that they are
    %within the bounds. Modules that have a -1 as the ModCh value are
    %ignored.
    for ind = 1:NSel
        for i=1:MAXMOD
            switch ModCh(ind,i)
                case 0  %Module is a link
                    if PartCh(ind,i) > nL %If outside the partlibrary bounds, assign it a
random value within the library
                        PartCh(ind,i) = ceil(nL*rand(1));
                    elseif PartCh(ind,i) == -1
                        PartCh(ind,i) = ceil(nL*rand(1));
                    end
                case 1 %Module is 1DOF
                    if PartCh(ind,i) >J1
                        PartCh(ind,i) = ceil(J1*rand(1));
                    elseif PartCh(ind,i) == -1
                        PartCh(ind,i) = ceil(J1*rand(1));
                    end
                case 2 %Module is 2DOF
                    if PartCh(ind,i) >J2
                        PartCh(ind,i) = ceil(J2*rand(1));
                    elseif PartCh(ind,i) == -1
                        PartCh(ind,i) = ceil(J2*rand(1));
                    end
                case 3 %Moduel is 3DOF
                    if PartCh(ind,i) >J3
                        PartCh(ind,i) = ceil(J3*rand(1));
                    elseif PartCh(ind,i) == -1
                        PartCh(ind,i) = ceil(J3*rand(1));
                    end
            end %Switch ModCh(ind,i)
        end %for i=1:MAXMOD
    end %for ind = 1:NSel


    % Reassemble the four link and joint sub-chromosomes into one large
```

```matlab
    % chromosome
    for ind=1:NSel %Loop over every individual in SelCh
    % Reinitialize link/joint counters
    li=0; j1=0; j2=0;j3=0;
        for i=0:MAXMOD-1
            type=1+4*i;
            part=type+1;
            inout=type+2;
            rot=type+3;
            SelCh(ind,part)= PartCh(ind,i+1);
            SelCh(ind,inout)= InOutCh(ind,i+1);
            SelCh(ind,rot)= RotCh(ind,i+1);

        end %for i =0:MAXMOD-1
    end%for ind=1:NSel

    %Evaluate Offspring
    for j=1:NSel %Loop over each of the Selected Chromosomes
        %Convert Chromosomes into the Assembly Matrix
        Assy(:,:,j)=[SelCh(j,1:4:4*MAXMOD); SelCh(j,2:4:4*MAXMOD);
SelCh(j,3:4:4*MAXMOD);...
            SelCh(j,4:4:4*MAXMOD)];
        % Now that subchromosomes are back into the Assy matrix, we
        % desire to eliminate the elements that are simply placeholders
        % prior to the objective function being called. Additionally,
        % assembly matricies that code for robots outside the bound of the
        % desired DOF range are discarded.

        junk=0; %Initialize junk index counter
        clear Garbage junkmodule
        junkmodule = [];
        for col=1:MAXMOD  %loop across the colums of the Assy matrix
            if any(Assy(:,col,j) == -1 )  %If any element of the column is -1, make the
whole column -1
                Assy(:,col,j) = [-1;-1;-1;-1];
                junk = junk+1;
                junkmodule(junk) = col;
            end
        end
        % Now, dump the junk modules to the end of the Assembly matrix.
        % This will help to keep the structure of the chromosomes neat
        % later on and keeps the placeholder sections of the matrix
        % together at the end of the matrix.
        if isempty(junkmodule) ~= 1
            TempAssy = Assy(:,:,j);
            Garbage = TempAssy(:,junkmodule);
            TempAssy(:,junkmodule)= [];
            TempAssy=[TempAssy(:,:) Garbage];
            Assy(:,:,j) = TempAssy;
        end

        %Compute the DOFs of the assembly and if it is within one of the
        %desired assembly, add or subtract a single DOF in order to meet
        %the desired DOF for the entire assembly. This tends to ensure that
        %the objective function will be called a reasonable number of times
        %in any given generation. Without this step, the number of
        %unacceptable configurations is too great.
        dof = sum((Assy(1,:,j)>0).*Assy(1,:,j));
        if dof == DOF +1 %Drop final DOF if there are one too many
            %Replace the last single DOF joint with the -1 column vector
            for g = MAXMOD:-1:1
                if Assy(1,g,j) == 1
                    Assy(:,g,j) = [-1;-1;-1;-1];
                    break
                end
            end
        elseif dof == DOF -1
            %Replace the first junk module with a single DOF joint of
            %random part type and untwisted orientation
            for g= 1:MAXMOD
                if  Assy(1,g,j) == -1
                    Assy(1,g,j) = 1;
                    Assy(2,g,j) = ceil((J1)*rand(1));
```

107

```matlab
                Assy(3,g,j) = 1;
                Assy(4,g,j) = 0;
                break
            end
        end
end %if dof == DOF

%Check that the Assembly matrix is not using more parts than
%allowed by the kit restraints.
AssyKit = Assy2Kit(Assy(:,:,j));
KitDiff = KitParts-AssyKit.parts;
DispAssy = Assy(:,:,j);
for rows = 1:numrows(KitDiff)
    for cols = 1:numcols(KitDiff)
        %If the value of KitDiff is negative, search through the
        %Assembly matrix and remove the modules that put the
        %Assembly over-budget.
        if KitDiff(rows,cols) < 0
            CutMods=KitDiff(rows,cols);
            for cut=1:abs(KitDiff(rows,cols));
                modcount = 1;
                while CutMods < 0 & modcount<=MAXMOD
                    if rows == 4
                        rows = 0;
                    end
                    if Assy(1,modcount,j) == rows & Assy(2,modcount,j) == cols
                        TempAssy = Assy(:,:,j);
                        TempAssy(:,modcount)= [];
                        TempAssy(:,MAXMOD) = [-1;-1;-1;-1];
                        Assy(:,:,j) = TempAssy;
                        CutMods = CutMods+1;
                    else
                        modcount=modcount+1;
                    end
                end %while CutMods
                if rows == 0
                    rows = 4;
                end
            end
        end %for cut=1:KitDiff
    end %if KitDiff(rows,cols) <0
end %for cols = 1:numcols(KitDiff)
end %for rows = 1:numrows(KitDiff)
DispAssy = Assy(:,:,j);



% Check the DOFs as the adjustment may have failed if there were no
% junk modules or single DOF modules to change.
dof = sum ((Assy(1,:,j) > 0).*Assy(1,:,j));

if dof == DOF  %Run objective function if DOF matches desired DOF
    [DH, TBase, LinkMass] =Assy2DH(Assy(:,:,j));
    TrialConfig=robot(DH);
    TBase = BASELOC*TBase;

        %OBJECTIVE FUNCTION
        %------------------------------------------------------
        [ObjVal MaxTorque JMass] = ParaEvaluateObjective(TrialConfig, TBase,
LinkMass, M, D, N, PosTraj, ForceTraj);
        % Check the joint masses against the joint masses of the universal arm.
        if ObjVal < 0 & isempty(JMass) == 0
            TrialKit = Assy2Kit(Assy(:,:,j),JMass);
            for a = 1:numcols(TrialKit.mass)
                for b = 1:numrows(TrialKit.mass)-1
                    if isempty(TrialKit.mass{b,a}) == 0
                        for c = 1:length(TrialKit.mass{b,a})
                            if TrialKit.mass{b,a}(c) > UKit.mass{b,a}(c)
                                ObjVal = 1;
                            end
                        end
                    end
                end
            end
```

```matlab
            end
            ObjValSel(j,1) = ObjVal;
            %--------------------------------------------------------

        else %Otherwise, assign objective output to be high and do not run
            ObjValSel(j,1) = 1;
        end %if dof == DOF

        % Because the Assembly matrix has changed due to conditioning, we need to
        % update its corresponding chromosomes so that they match.
        SelCh(j,1:4:4*MAXMOD) = Assy(1,:,j); %Module Types
        SelCh(j,2:4:4*MAXMOD) = Assy(2,:,j); %Part Types
        SelCh(j,3:4:4*MAXMOD) = Assy(3,:,j); %In/Out
        SelCh(j,4:4:4*MAXMOD) = Assy(4,:,j); %Rotation

        %Check if the elements of SelCh match any existing Chromosome. If
        %SelCh matches an existing member of Chrom, then set its objective
        %function to be high in order to maintain population diversity.
        %Without this step, the population tends to homogenize over time.
        for ChkDup=1:NIND
            if sum( SelCh(j,:) == Chrom(ChkDup,:)) == 4*MAXMOD
                ObjValSel(j,1) = 1;
            end
        end
    end %End the loop over each of the selected chromosomes for j=1:NSel

    % Reinsert Offspring into population
    %disp('Before reinsertion')

    %Calls the reinsertion function, and bases reinsertion on fitness
    %[ChromReins ObjValReins]= reins(Chrom, SelCh, 1, [1 5/(GGAP*NIND)], ObjValCh,
ObjValSel);
    [ChromReins ObjValReins]= reins(Chrom, SelCh, 1, [1 0.75], ObjValCh, ObjValSel);


    Chrom = ChromReins;
    ObjValCh = ObjValReins;
    for j=1:NIND
    PopAssy(:,:,j)=[Chrom(j,1:4:4*MAXMOD); Chrom(j,2:4:4*MAXMOD);
Chrom(j,3:4:4*MAXMOD);...
            Chrom(j,4:4:4*MAXMOD)];
    end
    %disp('After reinsertion')

    %Increment Counter
    gen = gen +1
    %Update display and record current best individual
    figure(1);
    [Best(gen+1), BestIx] = min(ObjValCh);
    BestAssy(:,:,gen+1) = PopAssy(:,:,BestIx);

    PopAvg(gen+1) = sum(ObjValCh)/length(ObjValCh);
    Top25Avg(gen) = sum(ObjValCh(1:5))/5;
    plot(Best,'ro'); xlabel('generation'); ylabel('Best');
    text(0.5,0.95,['Best = ', num2str(Best(gen+1))],'Units','normalized');
    drawnow;
    figure(2);
    plot(PopAvg, 'bo');
    figure(3);
    plot(PopAvg, 'go');
    drawnow
end %END GENERATIONAL LOOP
        [DH, TBase, LinkMass] =Assy2DH(BestAssy(:,:,101));
        TrialConfig=robot(DH);
        TBase = BASELOC*TBase;
        [ObjVal MaxTorque JointMass DexIxMean] = ParaEvaluateObjective(TrialConfig,
TBase, LinkMass, M, D, N, PosTraj, ForceTraj);

switch RUNS
    case 1
        save Data0825Kit1
    case 2
        save Data0825Kit2
```

109

```
    case 3
        save Data0825Kit3
    case 4
        save Data0825Kit4
    case 5
        save Data0825Kit5
    case 6
        save Data0825Kit6
    case 7
        save Data0825Kit7
    case 8
        save Data0825Kit8
    case 9
        save Data0825Kit9
    case 10
        save Data0825Kit10
end %End Switch
toc
%-----------------------------------------------------------------------
```

Function: ACGEPartLimitedOpt

```
function [Best, BestAssy, PopAssy, ObjValCh, PopAvg, JointMass] =
ACGEPartLimitedOpt(ForceTraj, PosTraj, RUNS, UKit, UnivAssy)
% AUTOMATIC CONFIGURATION GENERATION AND EVALUATION
%
% This script generates configurations of modular robotic manipulators,
% and evaluates them to determine their fitness acording to an objective
% function, then usues a genetic loop to recombine and mutate the
% population to search for more appropriate solutions.
%
% The manipulators evaluated are composed of the elements specified by
% Ukit, the kit of the "universal" arm.
%
% Inputs are the force and position trajectory points of the tasks being
% performed, the variable RUNS controls how the data is saved, UKit is the
% kit of the universal arm, and UnivAssy is the assembly of the universal
% arm.
%
% Outputs are the Best individual found, the assembly matrix of the best
% individual, the final population of assemblies, the Objective values
% associated with the population elements, the Population average at each
% generation, and the mass of the joints of the best assembly.
%
% Edit History:
% 03-30-06  Incorporated recombination and mutation for module orientation
%           (Rows three and four of the assembly matrix).
% 04-04-06  Allowed the inclusion of 2 and 3 DOF modules
% 04-07-06  Introduced Recombination and Mutation Rate constants
% 05-10-06  Incorporated variable base starting positions.
% 06-08-06  Modified for parametric tasks
% 06-14-06  Fixed a major bug relating to updates of the Assy matricies for
%           each generation
% 06-22-06  Functionalized ParaACGE in order to allow it to be run multiple
%           times back to back. Non-functionalized version saved to
%           ParaACGEbkup.m
% 08-09-06  Added elements to assist in constructing optimized kits.

global L J
tic                                  %Start Timer
DOF=6;                               %Desired DOF
M = 0.025;                            %Mass Weighting
N = 1;                               %Number of Modules Weighting
D = 0;                             %Dexterity Weighting
NIND=30;                             %Number of Individuals per Generation
MAXGEN=100;                          %Number of Generations to run
MAXMOD=15;                           %Maximum number of Modules
MINMOD=5;                            %Minimum number of Modules
MAXJ1= DOF;                          %Maximum length of 1-DOF vector
```

110

```
MAXJ2= floor(DOF/2);                    %Maximum length of 2-DOF vector
MAXJ3= floor(DOF/3);                    %Maximum length of 3-DOF vector
MAXLINK= MAXMOD-2;                       %Maximum length of link vector
MAXASSY= MAXLINK+MAXJ3+MAXJ2+MAXJ1; %Maximum length of Assembly
GGAP=0.9;                               %Generation gap. # of offspring = GGAP*NIND
MUTRATE = 0.10;                         %Mutation Rate
XOVER = 0.60;                           %Cross-over rate
J1=3; J2=2; J3=1;   %Number of parts in each of the joint inventories
nL= 2;%Number of parts in the link inventory, excluding grippers
BASELOC = [1 0 0 0; 0 1 0 0; 0 0 1 0; 0 0 0 1];
KitParts = UKit.parts;



% Loop over the elements in ModType
NMods =zeros(NIND,1);   %Initialize NMods
ModType(1:NIND, 1:MAXMOD) = -1;

% Pick off module masses for UnivAssy
Jix = 1;
n_mods = 1;
while n_mods <= size(UnivAssy,2)
    if UnivAssy(1,n_mods) > 0
        UJointMass(Jix) = UnivAssy(5,n_mods);
        Jix = Jix + 1;
    end
    n_mods = n_mods + 1;
end
UJointMass


for k=1:NIND
    i=0;
    d=0;
    n_mods=0;
    clear Module

    Parts = [];
    R = [];
    count = 1;
    for f=1:numrows(KitParts);
        for g=1:numcols(KitParts);
            if KitParts(f,g) ~= 0
                Parts=[Parts repmat([f;g],1,KitParts(f,g))];
                count = count + KitParts(f,g);
            end
        end
    end
    Parts = Parts-(Parts==4).*4;
    R = rand(1,size(Parts,2));
    [R RIx]=sort(R);
    Module=Parts(1,RIx);
    PartType = Parts(2,RIx);


        NMods(k) = length(Module);
        n_mods = length(Module);
        diff = MAXMOD - n_mods;
        if diff > 0
        Module(1,n_mods+1:1:n_mods+diff)= -1;
        end
        ModType(k,:) = Module(1,:);

    for j=1:MAXMOD;
    % Read the module type and assign a part type within the bounds of the part
    % library
        if ModType(k,j)==0    %If it's a link or a gripper
            if j==n_mods  %Make last element a gripper
                % PartType(j)=3;
                InOut(j)=1;
                Rot(j)=floor(4*rand(1));
            else
```

111

```matlab
%        PartType(j)=round((nL-1)*rand(1)+1);
         InOut(j)=round(rand(1));
         Rot(j)=floor(4*rand(1));
      end
   elseif ModType(k,j)==1 %If it's a 1 DOF
      %PartType(j)=ceil(J1*rand(1));
      InOut(j)=round(rand(1));
      Rot(j)=floor(4*rand(1));
   elseif ModType(k,j)==2 %If it's 2 DOF
      %PartType(j)=ceil(J2*rand(1));
      InOut(j)=round(rand(1));
      Rot(j)=floor(4*rand(1));
   elseif ModType(k,j)==3 %If it's 3 DOF
      %PartType(j)=ceil(J3*rand(1));
      InOut(j)=round(rand(1));
      Rot(j)=floor(4*rand(1));
   else  %Not a valid Module, assign elements to be -1
      PartType(j) = -1;
      InOut(j) = -1;
      Rot(j) = -1;
   end

end %for j= 1:MAXMOD


% The base of the chromosome for PartType should be the size of the largest
% part library used (Either Joint or Link). In other words, the base should
% be max([L, J1, J2, J3]). The base of InOut should be 2 (binary), and the
% base of Rot should be 4.
TypeBase=max([nL, J1, J2, J3]);
% Construct chromosome:
Chrom(k,4:4:4*MAXMOD)= Rot;      % Elements 4,8,12... Indicate rotations
Chrom(k,3:4:4*MAXMOD)= InOut;    % Elements 3,7,11... indicate In/Outport
Chrom(k,2:4:4*MAXMOD)= PartType; % Elements 2,6,10.. are PartTypes
Chrom(k,1:4:4*MAXMOD)= ModType(k,:); % Elements 1,5,9... are Moduletypes

if k == 1
   Chrom(1,4:4:4*MAXMOD) = UnivAssy(4,:);
   Chrom(1,3:4:4*MAXMOD) = UnivAssy(3,:);
   Chrom(1,2:4:4*MAXMOD) = UnivAssy(2,:);
   Chrom(1,1:4:4*MAXMOD) = UnivAssy(1,:);
end

PopAssy(:,:,k)=[Chrom(k,1:4:4*MAXMOD); Chrom(k,2:4:4*MAXMOD);
Chrom(k,3:4:4*MAXMOD);...
   Chrom(k,4:4:4*MAXMOD)];
if k == 1
   PopAssy(:,:,1) = UnivAssy(1:4,:);
end

[DH, TBase, LinkMass, JointLimits] =Assy2DH(PopAssy(:,:,k));
TrialConfig=robot(DH);
TrialConfig.qlim = JointLimits;
TBase = BASELOC*TBase;

   % OBJECTIVE FUNCTION
   %----------------------------------------------------------
   [ObjValue MaxTorque JMass] = ParaEvaluateObjective(TrialConfig, TBase , LinkMass,
M, D, N,PosTraj,ForceTraj);
   % Check the joint masses against the joint masses of the universal arm.
   if ObjValue < 0 & isempty(JMass) == 0
      TrialKit = Assy2Kit(PopAssy(:,:,k),JMass);
      for a = 1:numcols(TrialKit.mass)
         for b = 1:numrows(TrialKit.mass)-1
            if isempty(TrialKit.mass{b,a}) == 0
               for c = 1:length(TrialKit.mass{b,a})
                  if TrialKit.mass{b,a}(c) > UKit.mass{b,a}(c)
                     ObjValue = 1;
                  end
               end
            end
         end
      end
   end
```

```matlab
        end
        ObjValCh(k,1) = ObjValue;
        %-------------------------------------------------------------


end %for k=1:NIND
%Arrange initial population according to the number of modules
[NMods,ModIndex] = sortrows(NMods);
for i=1:NIND
    SortChrom(i,:) = Chrom( ModIndex(i), :);
    SortObj(i,:)   = ObjValCh( ModIndex(i), :);
    SortAssy(:,:,i) = PopAssy(:,:,ModIndex(i), :);
end
Chrom = SortChrom;
ObjValCh = SortObj;
PopAssy = SortAssy;

BaseVec = crtbase( [n_mods, n_mods, n_mods, n_mods], [2,TypeBase,2,4]);
gen=0;
% Track Best individual
figure(1);
    [Best(gen+1), BestIx] = min(ObjValCh);
    BestAssy(:,:,gen+1) = PopAssy(:,:,BestIx);
    plot(Best,'ro');xlabel('generation'); ylabel('Best');
    text(0.5,0.95,['Best = ', num2str(Best(gen+1))],'Units','normalized');
    drawnow;


disp('BEGIN GENERATIONAL LOOP')

% BEGIN GENERATIONAL LOOP
% ----------------------------------------------------------------------
while gen < MAXGEN

    %Assign fitness values for the entire population
    Fitness = ranking(ObjValCh);

    %Select Individuals for breeding
    SelCh=select('sus',Chrom,Fitness,GGAP);


    NSel = size(SelCh,1);
    %Initialize the subchromosomes to be the proper size
    clear J1Ch J2Ch J3Ch LinkCh ModCh InOut1Ch InOut2Ch InOut3Ch InOutLCh...
          Rot1Ch Rot2Ch Rot3Ch RotLCh TBase
    J1Ch(1:NSel,1:MAXJ1)= -1;
    J2Ch(1:NSel,1:MAXJ2)= -1;
    J3Ch(1:NSel,1:MAXJ3)= -1;
    LinkCh(1:NSel,1:MAXLINK)= -1;
    InOut1Ch=J1Ch;
    InOut2Ch=J2Ch;
    InOut3Ch=J3Ch;
    InOutLCh=LinkCh;
    Rot1Ch=J1Ch;
    Rot2Ch=J2Ch;
    Rot3Ch=J3Ch;
    RotLCh=LinkCh;

    %Recombine and mutate just the ModuleType portion of the Assembly
    %matrix first, as this infomation is needed in order to properly
    %perform recombination on lower-leveled elements of the matrix.

    %Recombine and mutate the module types (row 1 of Assy)

    ModCh = SelCh(:, 1:4:4*MAXMOD);
    ModCh = recombin('xovsp',ModCh, XOVER);
    ModBase= crtbase(MAXMOD, 3);
    ModCh = mut(ModCh, MUTRATE, ModBase);

    %Reinsert the modified (recombined/mutated) ModuleType back into SelCh
    SelCh(:,1:4:4*MAXMOD) = ModCh;
```

113

```matlab
%Loop over each individual in the population and generate the
%sub-chromosomes for ModType, PartType, InOut, and Rot
for ind=1:NSel
    %Check for module type
    j1=0;j2=0;j3=0;li=0; junk=0;typei=0; %Initialize indices for joints and links
    nmods =  (sum(SelCh(ind,:) >= 0))/4;
    for i=0:MAXMOD-1
        type=1+4*i;
        part= type+1;
        inout=type+2;
        rot=type+3;
        PartCh(ind,i+1)=SelCh(ind,part);
        InOutCh(ind,i+1) = SelCh(ind,inout);
        RotCh(ind,i+1) = SelCh(ind,rot);

    end %for i=0:NMods-1
end %for ind=1:NSel


% Recombine and mutate each of the Joint and Link subchromosomes
% independently according to category along with their respective rotation
% and in/out orientations

PartCh = recombin('xovsp',PartCh, XOVER);
InOutCh= recombin('xovsp',InOutCh,XOVER);
RotCh  = recombin('xovsp',RotCh,  XOVER);

PartBase = crtbase(MAXMOD, J1);
InOutBase = crtbase(MAXMOD, 2);
RotBase=crtbase(MAXMOD, 4);

PartCh= modmut(PartCh, MUTRATE, PartBase);
InOutCh = mut(InOutCh, MUTRATE, InOutBase);
RotCh = mut(RotCh, MUTRATE, RotBase);

%Loop over each individual in the population and each element of its
%ModCh chromosome in order to determine whether the elements of PartCh,
%InOutCh and RotCh are all within the bounds required by ModCh. If they
%are not inside of these bounds, then assign them such that they are
%within the bounds. Modules that have a -1 as the ModCh value are
%ignored.
for ind = 1:NSel
    for i=1:MAXMOD
        switch ModCh(ind,i)
            case 0  %Module is a link
                if PartCh(ind,i) > nL %If outside the partlibrary bounds, assign it a
random value within the library
                    PartCh(ind,i) = ceil(nL*rand(1));
                elseif PartCh(ind,i) == -1
                    PartCh(ind,i) = ceil(nL*rand(1));
                end
            case 1 %Module is 1DOF
                if PartCh(ind,i) >J1
                    PartCh(ind,i) = ceil(J1*rand(1));
                elseif PartCh(ind,i) == -1
                    PartCh(ind,i) = ceil(J1*rand(1));
                end
            case 2 %Module is 2DOF
                if PartCh(ind,i) >J2
                    PartCh(ind,i) = ceil(J2*rand(1));
                elseif PartCh(ind,i) == -1
                    PartCh(ind,i) = ceil(J2*rand(1));
                end
            case 3 %Moduel is 3DOF
                if PartCh(ind,i) >J3
                    PartCh(ind,i) = ceil(J3*rand(1));
                elseif PartCh(ind,i) == -1
                    PartCh(ind,i) = ceil(J3*rand(1));
                end
        end %Switch ModCh(ind,i)
    end %for i=1:MAXMOD
end %for ind = 1:NSel
```

```matlab
        % Reassemble the four link and joint sub-chromosomes into one large
        % chromosome
        for ind=1:NSel %Loop over every individual in SelCh
        % Reinitialize link/joint counters
        li=0; j1=0; j2=0;j3=0;
            for i=0:MAXMOD-1
                type=1+4*i;
                part=type+1;
                inout=type+2;
                rot=type+3;
                SelCh(ind,part)= PartCh(ind,i+1);
                SelCh(ind,inout)= InOutCh(ind,i+1);
                SelCh(ind,rot)= RotCh(ind,i+1);

            end %for i =0:MAXMOD-1
        end%for ind=1:NSel

        %Evaluate Offspring
        for j=1:NSel %Loop over each of the Selected Chromosomes
            %Convert Chromosomes into the Assembly Matrix
            Assy(:,:,j)=[SelCh(j,1:4:4*MAXMOD); SelCh(j,2:4:4*MAXMOD);
SelCh(j,3:4:4*MAXMOD);...
                SelCh(j,4:4:4*MAXMOD)];
            % Now that subchromosomes are back into the Assy matrix, we
            % desire to eliminate the elements that are simply placeholders
            % prior to the objective function being called. Additionally,
            % assembly matricies that code for robots outside the bound of the
            % desired DOF range are discarded.

            junk=0; %Initialize junk index counter
            clear Garbage junkmodule
            junkmodule = [];
            for col=1:MAXMOD  %loop across the colums of the Assy matrix
                if any(Assy(:,col,j) == -1 )  %If any element of the column is -1, make the
whole column -1
                    Assy(:,col,j) = [-1;-1;-1;-1];
                    junk = junk+1;
                    junkmodule(junk) = col;
                end
            end
            % Now, dump the junk modules to the end of the Assembly matrix.
            % This will help to keep the structure of the chromosomes neat
            % later on and keeps the placeholder sections of the matrix
            % together at the end of the matrix.
            if isempty(junkmodule) ~= 1
                TempAssy = Assy(:,:,j);
                Garbage = TempAssy(:,junkmodule);
                TempAssy(:,junkmodule)= [];
                TempAssy=[TempAssy(:,:) Garbage];
                Assy(:,:,j) = TempAssy;
            end

            %Compute the DOFs of the assembly and if it is within one of the
            %desired assembly, add or subtract a single DOF in order to meet
            %the desired DOF for the entire assembly. This tends to ensure that
            %the objective function will be called a reasonable number of times
            %in any given generation. Without this step, the number of
            %unacceptable configurations is too great.
            dof = sum((Assy(1,:,j)>0).*Assy(1,:,j));
            if dof == DOF +1 %Drop final DOF if there are one too many
                %Replace the last single DOF joint with the -1 column vector
                for g = MAXMOD:-1:1
                    if Assy(1,g,j) == 1
                        Assy(:,g,j) = [-1;-1;-1;-1];
                        break
                    end
                end
            elseif dof == DOF -1
                %Replace the first junk module with a single DOF joint of
                %random part type and untwisted orientation
                for g= 1:MAXMOD
```

```matlab
                if  Assy(1,g,j) == -1
                    Assy(1,g,j) = 1;
                    Assy(2,g,j) = ceil((J1)*rand(1));
                    Assy(3,g,j) = 1;
                    Assy(4,g,j) = 0;
                    break
                end
            end
        end %if dof == DOF

        %Check that the Assembly matrix is not using more parts than
        %allowed by the kit restraints.
        AssyKit = Assy2Kit(Assy(:,:,j));
        KitDiff = KitParts-AssyKit.parts;
        DispAssy = Assy(:,:,j);
        for rows = 1:numrows(KitDiff)
            for cols = 1:numcols(KitDiff)
                %If the value of KitDiff is negative, search through the
                %Assembly matrix and remove the modules that put the
                %Assembly over-budget.
                if KitDiff(rows,cols) < 0
                    CutMods=KitDiff(rows,cols);
                    for cut=1:abs(KitDiff(rows,cols));
                        modcount = 1;
                        while CutMods < 0 & modcount<=MAXMOD
                            if rows == 4
                                rows = 0;
                            end
                            if Assy(1,modcount,j) == rows & Assy(2,modcount,j) == cols
                                TempAssy = Assy(:,:,j);
                                TempAssy(:,modcount)= [];
                                TempAssy(:,MAXMOD) = [-1;-1;-1;-1];
                                Assy(:,:,j) = TempAssy;
                                CutMods = CutMods+1;
                            else
                                modcount=modcount+1;
                            end
                        end %while CutMods
                        if rows == 0
                            rows = 4;
                        end
                    end %for cut=1:KitDiff
                end %if KitDiff(rows,cols) <0
            end %for cols = 1:numcols(KitDiff)
        end %for rows = 1:numrows(KitDiff)
        DispAssy = Assy(:,:,j);


        % Check the DOFs as the adjustment may have failed if there were no
        % junk modules or single DOF modules to change.
        dof = sum ((Assy(1,:,j) > 0).*Assy(1,:,j));

        if dof == DOF  %Run objective function if DOF matches desired DOF
            [DH, TBase, LinkMass] =Assy2DH(Assy(:,:,j));
            TrialConfig=robot(DH);
            TBase = BASELOC*TBase;

                %OBJECTIVE FUNCTION
                %--------------------------------------------------------
                [ObjVal MaxTorque JMass] = ParaEvaluateObjective(TrialConfig, TBase,
LinkMass, M, D, N, PosTraj, ForceTraj);
                % Check the joint masses against the joint masses of the universal arm.
                if ObjVal < 0 & isempty(JMass) == 0
                    TrialKit = Assy2Kit(Assy(:,:,j),JMass);
                    for a = 1:numcols(TrialKit.mass)
                        for b = 1:numrows(TrialKit.mass)-1
                            if isempty(TrialKit.mass{b,a}) == 0
                                for c = 1:length(TrialKit.mass{b,a})
                                    if TrialKit.mass{b,a}(c) > UKit.mass{b,a}(c)
                                        ObjVal = 1;
                                    end
                                end
                            end
```

116

```matlab
                    end
                end
            end
        end
        ObjValSel(j,1) = ObjVal;
        %--------------------------------------------------------

    else %Otherwise, assign objective output to be high and do not run
        ObjValSel(j,1) = 1;
    end %if dof == DOF

    % Because the Assembly matrix has changed due to conditioning, we need to
    % update its corresponding chromosomes so that they match.
    SelCh(j,1:4:4*MAXMOD) = Assy(1,:,j); %Module Types
    SelCh(j,2:4:4*MAXMOD) = Assy(2,:,j); %Part Types
    SelCh(j,3:4:4*MAXMOD) = Assy(3,:,j); %In/Out
    SelCh(j,4:4:4*MAXMOD) = Assy(4,:,j); %Rotation

    %Check if the elements of SelCh match any existing Chromosome. If
    %SelCh matches an existing member of Chrom, then set its objective
    %function to be high in order to maintain population diversity.
    %Without this step, the population tends to homogenize over time.
    for ChkDup=1:NIND
        if sum( SelCh(j,:) == Chrom(ChkDup,:)) == 4*MAXMOD
            ObjValSel(j,1) = 1;
        end
    end
end %End the loop over each of the selected chromosomes for j=1:NSel

% Reinsert Offspring into population
%disp('Before reinsertion')

%Calls the reinsertion function, and bases reinsertion on fitness
%[ChromReins ObjValReins]= reins(Chrom, SelCh, 1, [1 5/(GGAP*NIND)], ObjValCh,
ObjValSel);
[ChromReins ObjValReins]= reins(Chrom, SelCh, 1, [1 0.75], ObjValCh, ObjValSel);


Chrom = ChromReins;
ObjValCh = ObjValReins;
for j=1:NIND
PopAssy(:,:,j)=[Chrom(j,1:4:4*MAXMOD); Chrom(j,2:4:4*MAXMOD);
Chrom(j,3:4:4*MAXMOD);...
        Chrom(j,4:4:4*MAXMOD)];
end
%disp('After reinsertion')

%Increment Counter
gen = gen +1
%Update display and record current best individual
figure(1);
[Best(gen+1), BestIx] = min(ObjValCh);
BestAssy(:,:,gen+1) = PopAssy(:,:,BestIx);

PopAvg(gen+1) = sum(ObjValCh)/length(ObjValCh);
Top25Avg(gen) = sum(ObjValCh(1:5))/5;
plot(Best,'ro'); xlabel('generation'); ylabel('Best');
text(0.5,0.95,['Best = ', num2str(Best(gen+1))],'Units','normalized');
drawnow;
figure(2);
plot(PopAvg, 'bo');
figure(3);
plot(PopAvg, 'go');
drawnow
end %END GENERATIONAL LOOP
    [DH, TBase, LinkMass] =Assy2DH(BestAssy(:,:,101));
    TrialConfig=robot(DH);
    TBase = BASELOC*TBase;
    [ObjVal MaxTorque JointMass DexIxMean] = ParaEvaluateObjective(TrialConfig,
TBase, LinkMass, M, D, N, PosTraj, ForceTraj);

switch RUNS
    case 1
```

117

```
            save Data0825Kit1
        case 2
            save Data0825Kit2
        case 3
            save Data0825Kit3
        case 4
            save Data0825Kit4
        case 5
            save Data0825Kit5
        case 6
            save Data0825Kit6
        case 7
            save Data0825Kit7
        case 8
            save Data0825Kit8
        case 9
            save Data0825Kit9
        case 10
            save Data0825Kit10
end %End Switch
toc
%-----------------------------------------------------------------------
```

Function: ParaEvaluate Objective

```
% function [ObjVal MaxTorque DexIxMean JointMass] = ParaEvaluateObjective(TrialRobot,
TBase, LinkMass, M, D, N, Traj, Force)
%
% This is the objective function used to evaluate the fitness of each
% reconfigurable robotic manipulator. The various terms of the function are
% as follows:
%
% Mass Estimator:
% -----------------------------------------------------------------------
% The mass of the function is evaluated based on the number of link and
% joint modules as well as the torque required of each of the actuators to
% complete the specified tasks. The mass estimating relationships used in
% these equations were developed for the links by performing a regression
% line analysis on the links of the AMTEC robotic system. Mass estimations
% for the joint modules was developed by looking at pairing Kollmorgen
% brushless DC motors with Harmonic Drive LLC harmonic gears. The
% relationship developed is a heuristic, which assumes an 80% actuator
% efficiency.
% Mass estimates for the links were done using the AMTEC reconfigurable
% system and are stored in the link library. LinkMass is a parameter that
% must be passed into EvaluateObjective. The total link mass of a
% manipulator is calculated by summing up the mass of all links used in a
% given configuration.
%
% Dexterity:
% -----------------------------------------------------------------------
% To differentiate systems that are similarly massive, a dexterity measure
% is also included. This is essentially a measurement of the ability of the
% arm to move in any arbitrary direction at a given moment in time.
%
% Number of Modules:
% -----------------------------------------------------------------------
% Manipulators are weighted based on the total number of modules used. The
% theory is that the use of fewer modules should be seen as a benefit in
% terms of maintaining structural simplicity. This parameter ideally should
% be given a lower weighting than either the Mass or Dexterity parameters
% as it has very little impact on the functionality of the manipulator.
%
% Weighting Parameters
% -----------------------------------------------------------------------
% M is the relative weighting assigned to the mass of the manipulator
% D is the relative weighting assigned to the dexterity parameter
%
% The objective function itself is of the form:
% ObjVal = exp(- (M*Mass + D*Dexterity + N*NumMods)
```

118

```
%
% If inverse kinematic solutions cannot be found for all points,
% ObjVal = (1-PointOfFailure)/(TotalNumberOfPoints)
%
% Note: The task specification should be input at some point here as well,
% though at present, the sample task is hard-coded into the beginning of
% the function.

function [ObjVal MaxTorque JointMass DexIxMean] = ParaEvaluateObjective(TrialRobot,
TBase, LinkMass, M, D, N, Traj, Force)
JointMass = [];
Mass = [];
DexIxMean = [];
MaxTorque = [];
disp('Running ParaEvaluateObjective')
% Get the Task points form the TaskDefinitions function
%[ForceTraj, PosTraj] = TaskDefinitions(TBase, TRAJPOINTS, TASK);
if size(Traj,3) ~= size(Force,1)
    error('Length of position trajectory and force trajectory must match')
end

Tinv = inv(TBase);
Tforce = zeros(6,6);
Tforce(1:3,1:3) = Tinv(1:3,1:3);
Tforce(4:6,4:6) = Tinv(1:3,1:3);

for page = 1:size(Traj,3)
    PosTraj(1:4,1:4, page) = Tinv*Traj(:,:,page);
    ForceTraj(:,1,page) = Tforce*Force(page,:)'; %Rotate the force, torque vector
end

isreachable = ones(1,size(PosTraj,3));
for i=1:size(PosTraj,3)
    if TrialRobot.n ==6 %If manipulator is 6 DOF
        Q = ikinemod(TrialRobot, PosTraj(:,:,i));
    else %If manipulator has more than 6 DOF, use GA optimizer.
        Q = GAPose(PosTraj(:,:,i), ForceTraj(:,:,i), TrialRobot);
    end
        if isnan(Q)
            FailPoint = i;
            ObjVal = 1-(FailPoint-1)/size(PosTraj,3);
            return
            %isreachable(1,i) = 0;
        else
            Jacobian(:,:,i) = jacob0(TrialRobot, Q);
            if rank(Jacobian(:,:,i)) < 6
                ObjVal= 1;
                disp('Jacobian of insufficient rank')
                FailPoint = i;
                return
            end
        end

    Torque(:,:,i) = abs(Jacobian(:,:,i)'*ForceTraj(:,:,i));
    W(i) = sqrt( det(Jacobian(:,:,i)*Jacobian(:,:,i)')); %Yoshikawa manipulability index
    DexIx(i) = 1/(1+W(i));
end
MaxTorque = max(Torque,[],3);
%Calculate actuator masses based on the maximum torque each joint is
%expected to see at any point in the task.
JointMass = 0.007.*MaxTorque + 0.70;
%Calculate link masses based on their volume. A diameter of seven
%centimeters (to go along with actuator size) is assumed for the purposes
%of calculating volume.
Mass = sum(JointMass) + LinkMass;
DexIxMean = mean(DexIx);
% Set the Objective value to be negative, as the GA functions will try to
% minimize the value of the objective.
ObjVal = -exp(- (M*Mass  + D*DexIxMean));
```

References

Ambrose, Robert O., Hal Aldridge, R. Scott Askew, Robert R. Burridge, William Bluethmann, Myron Diftler, Chris Lovchik, Darby Magruder, and Fredrik Rehnmark. "Robonaut: NASA's Space Humanoid", IEEE Intelligent Systems, July/August 2000, pp. 56-63.

Akin, David L. "Lightweight Dexterous Modular Manipulators for Space Applications, Phase I Progress Report" University of Maryland Space Systems Lab Document Number: DT19-10005. January 30, 2004

Beer, Ferdinand P. and E. Russell Johnston, Jr. *Vector Mechanics for Engineers, Sixth Edition*. Boston, Massachusetts: WCB McGraw-Hill, 1996.

Bi, Zhuming. "On Adaptive Robot Systems for Manufacturing Applications", PhD. Thesis, University of Saskatchewan, Saskatoon, Saskatchewan, Fall 2002.

Bi, Z.M., W.A. Gruver, W.J Zhang. "Adaptability of Reconfigurable Robotic Systems", Proceedings of the IEEE International Conference on Robotics and Automation, September 14-19 2003, pp 2317-2322

Brener, Nicolas, Faiz Ben-Amar, and Philippe Bidaud. "Analysis of Self-Reconfigurable Modular Systems: A Design Proposal for Multi-Modes Locomotion", Proceedings of the IEEE International Conference on Robotics and Automation, New Orleans, LA, April 2004, pp. 996-1001.

Chen, I-Ming and Joel W. Burdick. "Determining Task Optimal Modular Robot Assembly Configurations", Proceedings of the IEEE International Conference on Robotics and Automation, Japan, 1995, pp. 132-137.

Chen, I-Ming, Song Huat Yeo, Guang Chen, Guilin Yang. "Kernel for Modular Robot Applications: Automatic Modeling Techniques", International Journal of Robotics Research, Vol. 18, No. 2, February 1999, pp. 225-242

Chipperfield, A., Fleming, P. J., Pohlheim, H. and Fonseca, C. M. "Genetic Algorithm Toolbox for use with Matlab**."** Technical Report No. 512, Department of Automatic Control and Systems Engineering, University of Sheffield, 1994.

Chocron, O. and P. Bidaud. "Genetic Design of 3D Modular Manipulators", Proceedings of the IEEE International Conference on Robotics and Automation, Albuquerque, USA, April 20-25, 1997.

Corke, Peter I., "A Robotics Toolbox for MATLAB", IEEE Robotics and Automation Magazine, Volume 3(1), March1996, pp. 24-32.

Craig, John J. *Introduction to Robotics:Mechanics and Control, Second Edition.* Reading, MA: Addison-Wesley, 1986.

Curried, Nancy J., Jennifer Rochlis. "Command and Telemetry Latency Effects on Operator Performance during International Space Station Robotic Operations", Human Factors and Ergonomics Society 48th Annual Meeting. New Orleans, LA, September 20-24, 2004.

Devore, Jay L. *Probability and Statistics for Engineering and the Sciences, Sixth Edition*. Belmont, CA: Brooks/Cole, 2004.

Ellis, Robert and Denny Gullick. *Calculus with Analytic Geometry Fifth Edition*. Fort Worth: Saunders College Publishing, Harcourt Brace College Publishers, 1994.

Farritor, Shane, Steven Dubowsky, Nathan Rutman and Jeffrey Cole. "A Systems-Level Modular Design Approach to Field Robotics", Proceedings of the IEEE International Conference on Robotics and Automation, Minneapolis, MN, April 1996, pp. 2890-2895.

Gere, James, M. *Mechanics of Materials, Fifth Edition*. Pacific Grove, CA: Brooks/Cole, 2001.

Han, Jeongheon, W.K. Chunk, Y. Youm, and S.H. Kim. "Task Based Design of Modular Robot Manipulator using Efficient Genetic Algorithm", Proceedings of the IEEE International Conference on Robotics and Automation, Albuqurque, NM, 1997, pp. 507-512.

Hayashi, Ryoichi, Saburo Matunaga, and Yoshiaki Ohkami. "Capability Evaluation of Reconfigurable Brachiating Space Robot", Industrial Electronics Society, 2000. IECON 2000. 26th Annual Conference of the IEEE, pp. 2461-2466

Howe, Scott A. "The Ultimate Construction Toy: Applying Kit-of-Parts Theory to Habitat and Vehicle Design", AIAA Space Architecture Symposium, October 10-11, 2002, Houston TX. pp. 1-36

Kim, Jin-Oh and Pradeep K. Kholsa. "Design of Space Shuttle Tile Servicing Robot: An Application of Task Based Kinematic Design", Proceedings of the IEEE International Conference on Robotics and Automation, Atlanta, GA, May 1993. vol. 3, pp. 867-874.

Kang, Seong-Ho, Daniel J. Cox, Delbert Tesar. "Standard Modular Actuator Test and Characterization", IECON 2001: The 27th Annual Conference of the IEEE Industrial Electronics Society, 2001, pp. 462-467.

Luo, Xiaoping, and Wei Wei. "A New Immune Genetic Algorithm and Its Application in Redundant Manipulator Path Planning", Journal of Robotic Systems, Vol. 21, No.

3, 2004, pp. 141-151.

Mukundan, Rangaswamy and A.K. Pradeep. "Comments on 'On the Computational Aspect of the Matrix Exponentials and Their Use in Robot Kinematics", IEEE Transactions on Automatic Control, Vol. AC-32, No. 11, November 1997, pg. 1021.

Nokleby, Scott B. and Ron P. Podhorodeski. "Pose Optimization of Serial Manipulators Using Knowledge of Their Velocity-Degenerate (Singular) Configurations", Journal of Robotic Systems, Vol 20, No. 5, 2003, pp. 239-249.

Nokleby, Scott B. and Ron P. Podhorodeski, "Velocity Degeneracy Determination for the Kinematically Redundant CSA/ISE STEAR Testbed Manipulator", Journal of Robotic Systems, Vol 17, No. 11, 2000, pp. 633-642.

Paredis, Christiaan J.J. and Pradeep K. Kholsa. "Agent-Based Design of Fault Tolerant Manipulators for Satellite Docking", Proceedings of the IEEE International Conference on Robotics and Automation, Albuquerque, NM, April 1997, pp. 3473-3480.

Paredis, Christiaan J.J. and Pradeep K. Kholsa. "Design of Modular Fault Tolerant Manipulators", Proceedings of the Workshop on Algorithmic Foundations of Robotics, San Fransisco, CA, 1995, pp. 371-383.

Park, Frank C. "Computational Aspects of the Product-of-Exponentials Formula for Robot Kinematics", IEEE Transactions on Automatic Control, Vol. 39, No. 3, March 1994, pp. 643-647.

Salam, Fathi M. A. and Choong S. Yoon. "On the Computational Aspect of the Matrix Exponentials and Their Use in Robot Kinematics", IEEE Transactions on Automatic Control, Vol AC-31, No. 4, April 1986, pp 376-378.

Schneider, Dean L., Delbert Tesar, J. Wesley Barnes. "Development and Testing of a Reliability Performance Index for Modular Robotic Systems", Reliability and Maintainability Symposium, 1994, pp. 263-271.

Sciavicco, L., and B. Siciliano. *Modelling and Control of Robot Manipulators*. London: Springer, 1996.

Skaff, Sarjoun, Peter K. Staritz, and William Whittaker. "Skyworker: Robotics for Space Assembly, Inspection and Maintenance", Space Studies Institute Conference, 2001.

Staritz, Peter J., Sarjoun Skaff, Chris Urmson, and William Whittaker. "Skyworker: A Robot for Assembly, Inspection and Maintenance of Large Scale Orbital Facilities", IEEE International Conference on Robotics and Automation, Seoul, Korea, May 21-26, 2001, pp. 4180-4185

Suh, Nam P. "Axiomatic Design Theory for Systems", Research in Engineering Design, Vol. 10, No. 4, December, 1998, pp 189-209.

Thomas, Ulrike, Bernd Finkemeyer, Torsten Kroger and Friedrich M. Wahl. "Error-Tolerant Execution of Complex Robot Tasks Based on Skill Primitives", Proceedings of the IEEE International Conference on Robotics and Automation, Taipei, Taiwan, September 14-19, 2003. pp. 3069-3075.

Yim, Mark. Kimon Roufas, David Duff, Ying Zhang, and Sam Homans. "Modular Reconfigurable Robots in Space Applications", Autonomous Robots, Vol 4, Nos. 2-3, March 2003, pp. 225-237