12-2011

A study of a novel modular variable geometry frame arranged as a robotic surface

Christopher James Salisbury
University of Nevada, Las Vegas

A STUDY OF A NOVEL MODULAR VARIABLE GEOMETRY FRAME ARRANGED AS A ROBOTIC SURFACE

by

Christopher James Salisbury

Bachelor of Science

University of Nevada, Las Vegas

2009

A thesis submitted in partial fulfillment

of the requirements for the

Master of Science in Mechanical Engineering

Department of Mechanical Engineering

Howard R. Hughes College of Engineering

Graduate College

University of Nevada, Las Vegas

December 2011

THE GRADUATE COLLEGE

We recommend the thesis prepared under our supervision by

Christopher Salisbury

entitled

A Study of A Novel Modular Variable Geometry Frame Arranged as a Robotic Surface

be accepted in partial fulfillment of the requirements for the degree of

Master of Science
Department of Mechanical Engineering

Woosoon Yim, Committee Chair

Mohamed Trabia, Committee Member

Brendan O'Toole, Committee Member

Sahjendra Singh, Graduate College Representative

Ronald Smith, Ph. D., Vice President for Research and Graduate Studies
and Dean of the Graduate College

December 2011

ABSTRACT

A Study of a Novel Modular Variable Geometry Frame

Arranged as a Robotic Surface

by

Christopher James Salisbury

Dr. Woosoon Yim, Examination Committee Chair

Professor of Mechanical Engineering

University of Nevada, Las Vegas

The novel concept of a "variable geometry frame" is introduced and explored through a three-dimensional robotic surface which is devised and implemented using triangular modules. The link design is optimized using surplus motor dimensions as firm constraints, and round numbers for further arbitrary constraints. Each module is connected by a passive six-bar mechanism that mimics the constraints of a spherical joint at each triangle intersection. A three dimensional inkjet printer is used to create a six-module prototype designed around surplus stepper motors powered by an old computer power supply as a proof-of-concept example.

The finite element method is applied to the static and dynamic loading of this device using linear three dimensional (6 degrees of freedom per node) beam

elements to calculate the cartesian displacement and force and the angular displacement and torque at each joint. In this way, the traditional methods of finding joint forces and torques are completely bypassed. An efficient algorithm is developed to linearly combine local stiffness matrices into a full structural stiffness matrix for the easy application of loads. This is then decomposed back into the local matrices to easily obtain joint variables used in the design and open-loop control of the surface.

Arbitrary equation driven surfaces are approximated ensuring that they are within the joints limits. Moving shapes are then calculated by considering the initial position of the surface, the desired position of the surface, and intermediate shapes at discrete times along the desired path.

There are no sensors on the prototype, but feedback models and state estimators are developed for future use. These models include shape sampling methods derived from existing meshing algorithms, trajectory planning using sinusoidal acceleration profiles, spline-based path approximation to allow lower curvature paths able to be traversed more quickly and/or able to be travelled with a constant velocity and optimized by iteratively calculating actuator saturation with no discontinuities, and the optimal tracking of a desired path (modeled with a time-varying ricatti equation).

ACKNOWLEDGMENTS

I would like to thank the Nevada Space Grant Consortium as well as the Department of Mechanical Engineering at the University of Nevada, Las Vegas (UNLV) for their generous financial support. I would also like to thank the Intelligent Structure Controls Laboratory (ISCL) of UNLV for the ongoing use and improvement of their facilities.

I have been fortunate to have very good instruction and recommendations for textbooks in my coursework. Dr. Woosoon Yim instructed an excellent robotics class, which introduced me to serial robotics. He also gave very good recommendations on textbooks that I might read – even loaning me some. Dr. Sahjendra Singh taught one class in digital controls and another outstanding course in optimal controls as well as recommending four excellent textbooks. Dr. Brendan O'Toole was the professor for a very influential course in energy methods, where I learned intricacies of the finite element method.

The support of the Mechanical Engineering department in general has been outstanding, and has been instrumental in my educational progress. I am very grateful to have been even a very small part of such a wonderful department.

PREFACE

This project has been incredibly stimulating, and has provided an excellent platform for learning. Because it is an extremely broad project, there are many aspects that are discussed, but none are explored fully. I have tried to balance the information presented, but in certain cases, one may find too much detail about a simple matter, or not enough detail about a complex matter. In the former case, the reader is encouraged to skim through trivialities if they are deemed to be very well understood. In the latter case, references have been included for the interested reader to begin a more in-depth analysis.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

Goals and Initial Ideas

A mechanical surface able to approximate arbitrary shapes was desired, and toward this end, many ideas were considered. Throughout this process, many possible applications were discovered and ultimately the most versatile and easily implemented design was chosen. The most interesting of these designs are discussed below.

One of the first ideas was to use rods moving up and down in a vertical fashion with the ends of the rods forming the desired surface. This would require the control of many rods in order to achieve a reasonable resolution, but is very easily scaled to large sizes as the rods are operated independently of each other. This type of device can be used as a display or a mold for casting prototypes, but cannot be used for robotic manipulation. It can also only produce shapes as a function of height. This limits the shapes to have no overhang/undercut as there can only be one height value for each rod at a given time. Using rods also allows for (and necessitates) discontinuous surfaces.

Another early idea was silicone impregnated with a ferrous metal which is then moved by changing magnetic fields. This allows for a different class of shapes to be formed, but the shape resolution is dependent on the number and location of magnetic field generators and control therefore becomes very difficult. It also requires a tremendous amount of energy.

Also among the early contenders was the idea of a variable geometry frame (VGF). The first such idea was a four-bar mechanism composed of four equal links connected by parallel revolute joints formed into rhombic modules with each module connected by revolute and prismatic joints. This is the first concept considered that had the potential for robotic manipulation and self-transportation. It is also not limited to functions of height as with the rod idea and also does not have resistive spring forces increasing the energy requirements proportionally to the distance from the neutral position as with the silicone idea. As such, a prototype model was drawn up for a more detailed analysis. Nine actuators are required for each module, and the motion is severely limited as the number of modules increases. This extremely high complexity for the mediocre results was deemed impractical, but it did show the potential of VGFs.

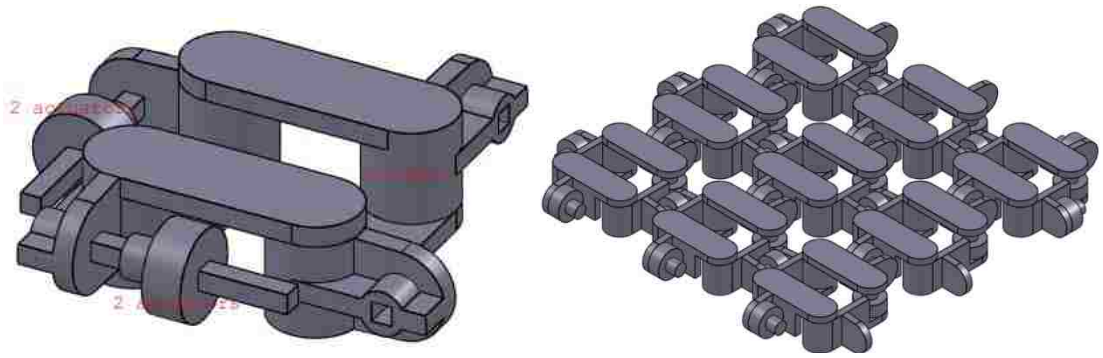

Figure 1: Left: A single rhombic module. Right: Nine modules connected with revolute and prismatic joints.

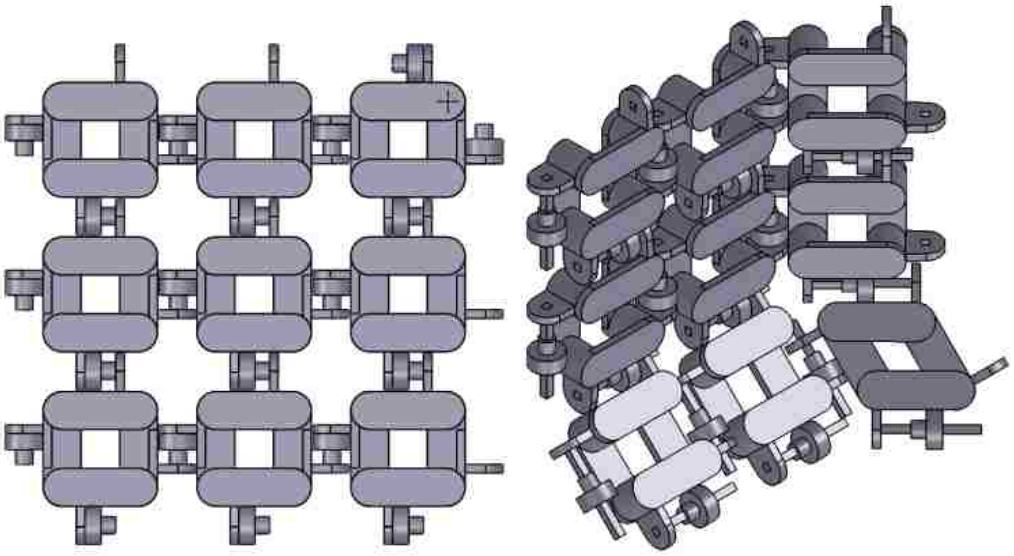

Figure 2: Example positions of a nine-module rhombic surface.

In order to reduce the complexity of the modules, triangular modules were considered. To remove the limitation of additional modules connected in parallel, variable link lengths became apparently necessary. Thus, triangular modules with variable link lengths were considered. Each module could now be connected to adjacent modules with only a revolute joint. These modules only require six actuators per module, and adding modules does not limit the motion of the existing modules in the way that the rhomboidal modules did. Initially sliding revolute joints were considered (Figure 3), but eventually these were eschewed in favor of six-bar revolute joints which reduced friction and weight. By using triangular modules of this type, we can take advantage of the wide variety of computer graphics and finite element algorithms currently available.

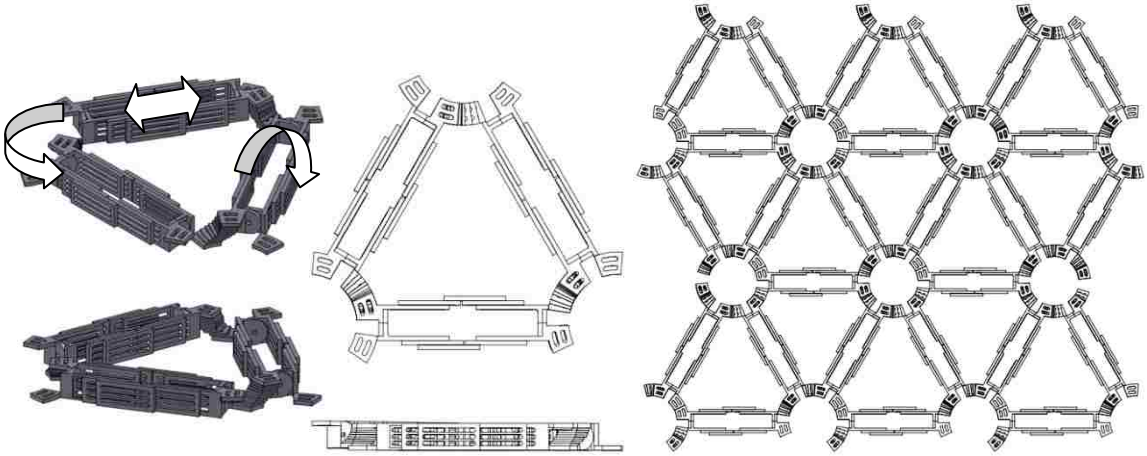

Figure 3: Left: Initial shape of one triangular module with sliding revolute joints (with the edges of adjacent modules attached to show out of plane motion). Right: Assembly of many modules. Actuators are not shown.

A robotic surface composed of triangular modules is able to perform most of the functions of a traditional robotic arm, as well as the functions of many surfaces and structures. A triangular modular approach allows us take advantage of the large number of computer graphics and finite element algorithms available to accurately approximate any arbitrary surface within the joint range as well as calculate the kinematics, dynamics, and related forces. Since the actual shape is that of beam elements arranged into triangles, there is little need to approximate the robot's shape with other discrete elements. This greatly eases the control of the surface, and allows the approximate duplication of any C^0 continuous surface. This includes surfaces other than Euclidean functions - that is to say that for a single input, there can be multiple outputs (e.g., a sphere can be approximated).

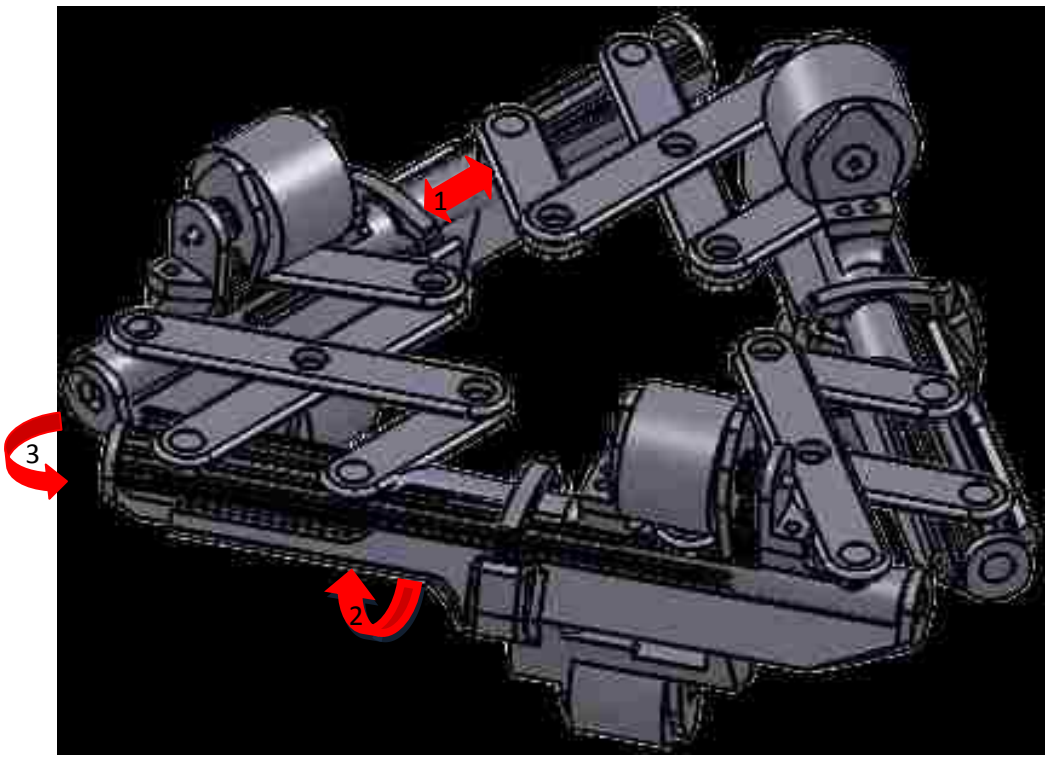

Figure 4: One module of a robotic surface – composed of three actuated prismatic joints (number 1 in the figure), three actuated rotary joints (number 2 in the figure), and three passive rotary joints (number 3 in the figure).

One module consists of three actuated revolute joints, three passive six-bar joints, and three actuated prismatic joints (shown in Figure 4). The six-bar joint is a modification of Hamlin and Sanderson's concentric multilink spherical joint [1]. It has been modified by placing the six-bar mechanism out of plane with the other rotary joints (numbered "2" in Figure 4) so that the axes of rotation intersect, avoiding the need for any offset in the individual six-bar links (Figure 5). Because the joints are in parallel rather than in series (as with a traditional robot arm), the kinematics and dynamics cannot be easily computed using traditional methods. Further, in order to produce a desired surface, the position and motion of each vertex must be controlled, rather than just an end-effector [2].

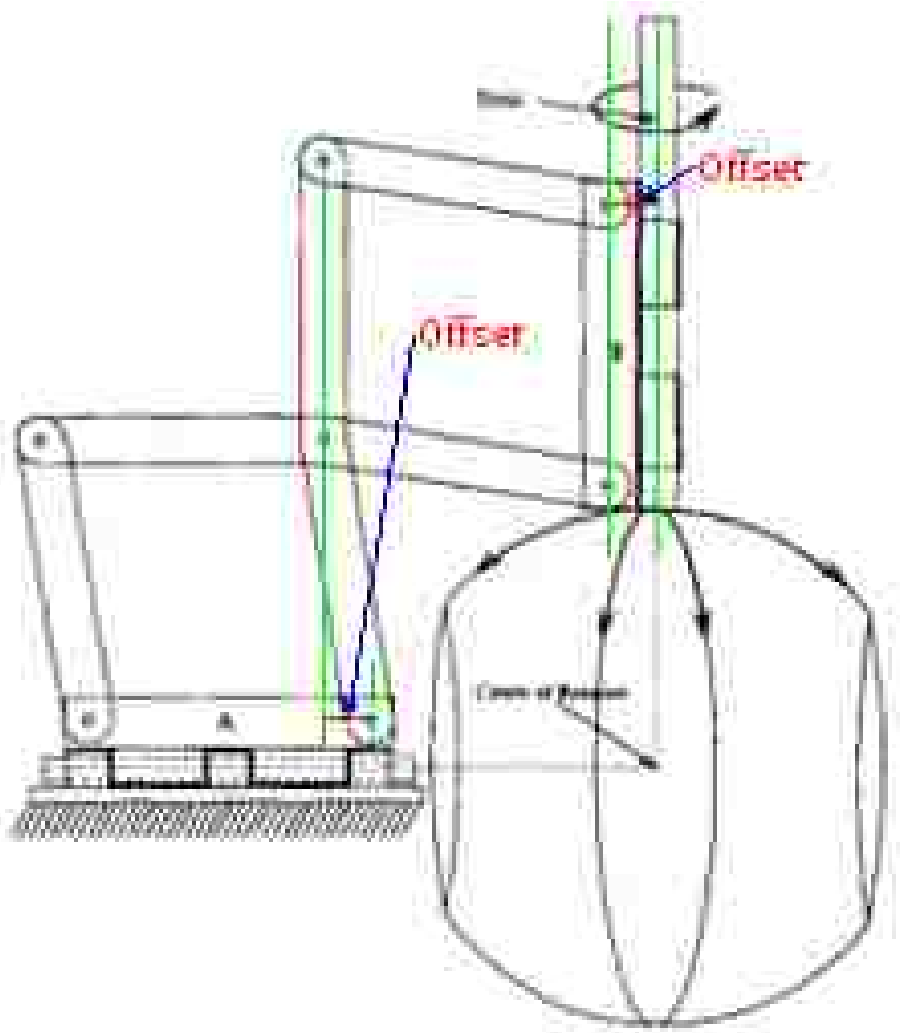

Figure 5: CMS joint with link offset. [1]

Several mechanisms similar to this have been considered in the past [2], and are usually categorized in a class known as a variable geometry truss (VGT) [3][4]. A truss is characterized by passive revolute joints causing only one-dimensional axial forces in each link (tension or compression). Assuming a three-dimensional truss (or "space truss" using spherical joints) this means that singular positions include any position in which all of the links attached to one node are co-planar. Specifically,

if in such a position there are any out-of-plane forces at the common node, the resulting forces in each of the links attached at that point will be infinite.

In order to increase the potential of such a device, a rotary actuator controlling the angle about an axis parallel to each link is included in addition to the standard linear actuator. This means that the structure is no longer able to be classified as a truss, but must instead be called a frame (or "space frame") with forces in all three local dimensions of each link. Singular and uncontrollable positions now only occur when all links attached to one node are co-linear.

Modular robotics have become very popular largely because of their versatility. Most current efforts are furthering the capabilities of modules to be connected one-dimensionally in series. The presented robotic surface implements a novel module type for the expansion to modules to be connected two-dimensionally in parallel. This opens many new possibilities and can be expanded to three dimensions without much difficulty (think of two or more parallel surfaces attached to each other). The joints of a robot directly affect one another. This can be seen positively in that they support each other and allow less powerful actuators to be used, and it can be seen negatively in that they interfere with each other and make the robot more difficult to control. In one-dimensional modules, actuators have no more than two actuators connected adjacently to them. Because distance is finite, this limits the extent to which they can aid each other. By allowing modules to interact in multiple dimensions, greater numbers of actuators can be connected adjacently. This allows greater loads to be supported at each joint for the same motion. More importantly, it allows for a different type of motion – such as surface

motion. Each joint is of course also more restricted by the joints around it, but for a surface this is also desirable as it guarantees surface continuity.

Literature Review – Variable Geometry Trusses

This summary includes a brief overview of some different types and abilities of several VGTs. Cyclic two-dimensional planar trusses and planar three-dimensional VGTs as well as non-cyclic three-dimensional non-planar VGTs are all considered.

Shape Morphing Hinged Truss Structures [3]

A planar triangular truss connected by novel spherical "hexa-pivotal" joints (Figure 6) has a rigid bi-pyramidal structure placed on each triangle (with the triangle between the two pyramids).

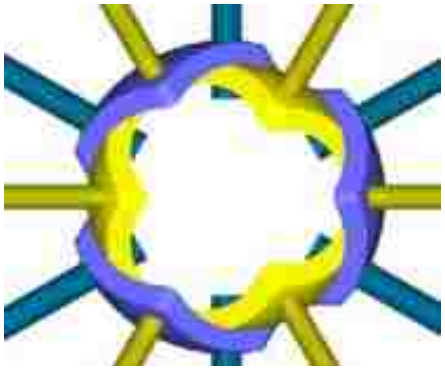

Figure 6: Spherical hexa-pivotal joint [3].

These bi-pyramidal hexahedrons provide a moment arm to be actuated with shape memory alloy wire connecting the apex of adjacent pyramids as shown in Figure 7.

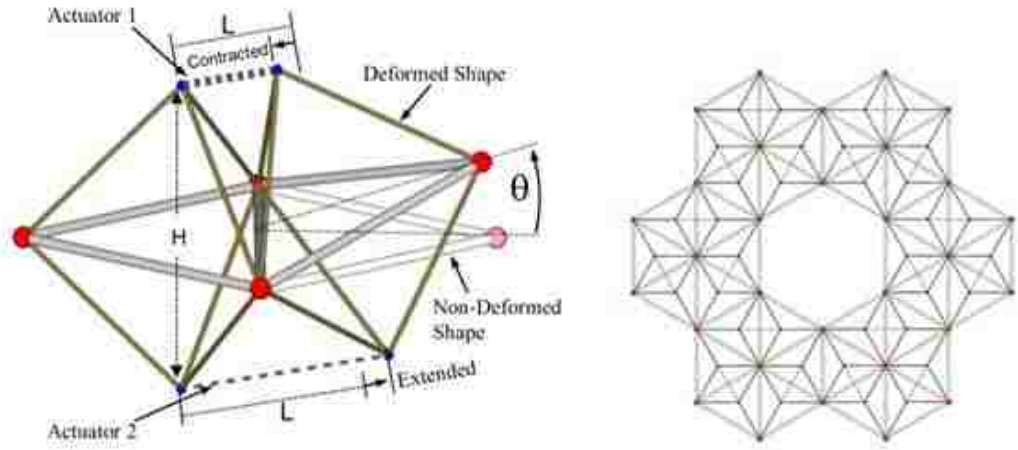

Figure 7: The apex of each adjacent pyramid is connected with Nitinol to control the angle between pyramids. A hexagonal pattern was chosen because of the increased mobility.

The shape on the right in Figure 7 was chosen because it has more degrees of freedom than if the central region were filled with the triangular shapes. An experimental model was built, and binary actuator combinations were tested resulting in simple motion. The angles were calculated with simple trigonometry, and the degrees of freedom were calculated in a traditional manner using Maxwell's stability criterion.

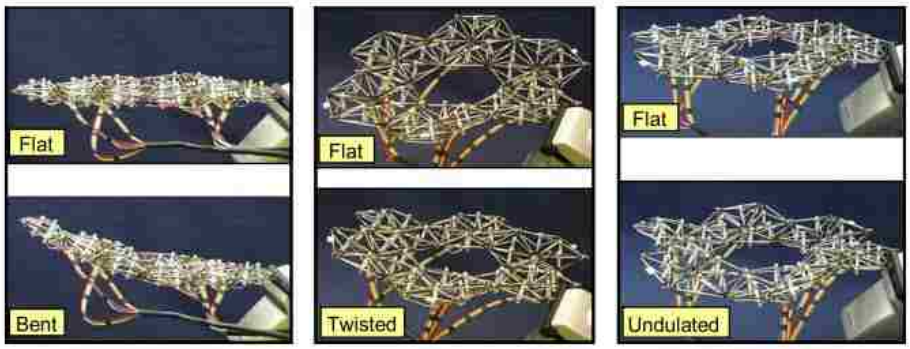

Figure 8: Multiple example positions for a hinged truss structure [3].

Trussarm — A Variable-Geometry-Truss Manipulator [5]

Two stacked octahedral truss arms are compared with a precursor to the international space station's current "Canadarm," with the results shown in Table 1.

Table 1: A comparison between Canadarm and two trussarms [5].

Parameter [Units]	Canadarm	Trussarm "D"	Trussarm "V"
Mass (incl. end effector) [kg]	411.5	413.4	411.5
Length [m]	15.53	15.17	14.47
Diameter (deployed straight) [m]	0.325	0.325	2.117
Storage Volume (approx.) [m³]	1.6	0.383	1.6
Degrees of Freedom (total)	6	99	15
Degrees of Freedom (redundant)	0	93	9
Maximal Tip Load (straight) [N]			
Axial	203,177	1,575,000	1,469,000
Normal	7,243	7,310	45,135
Modal Frequencies (first five) [Hz]		Straight	
Mode 1	0.372	0.309	2.033
Mode 2	0.387	0.309	2.049
Mode 3	3.263	2.018	12.130
Mode 4	3.580	2.019	12.306
Mode 5	9.923	5.786	25.115
Modal Frequencies (first five) [Hz]		90° Curve	
Mode 1	0.497	0.327	2.061
Mode 2	0.525	0.347	2.455
Mode 3	1.416	1.547	8.821
Mode 4	1.656	1.672	10.474
Mode 5	9.471	5.171	24.939
Computer Requirements	small	large	medium
Dexterity	small	large	medium

Note: Trussarm "D" has the same diameter as Canadarm.
Trussarm "V" has the same stowed volume as Canadarm.

The octahedral modules used for the arms are simplified 3 degree of freedom Stewart platforms. These modules were chosen from four options using a modal vibrational analysis in NASTRAN. The reasons that the chosen modules are "better"

were not well explained, but it is implied that the main criterion was the strength to mass ratio. A basic introduction to dynamics is given, but it is plain that NASTRAN was relied upon to obtain correct equations and to perform all of the calculations. A two-module experimental model was produced, but no experimental results are reported.

As a preliminary work on adaptive variable octahedral trusses, this paper provides a starting point for further work — work that has since been completed. It clearly shows that the Canadarm can be improved upon by using a VGT, and that parallel robotics have superior strength to weight ratios.

Tetrobot: A Modular Approach to Reconfigurable Parallel Robotics [4]

A novel concentric multilink spherical (CMS) joint utilizing a planar offset hinge (realized in a six-bar mechanism) is analyzed for range of motion and force propagation using classical mechanics and traditional machine dynamics methods.

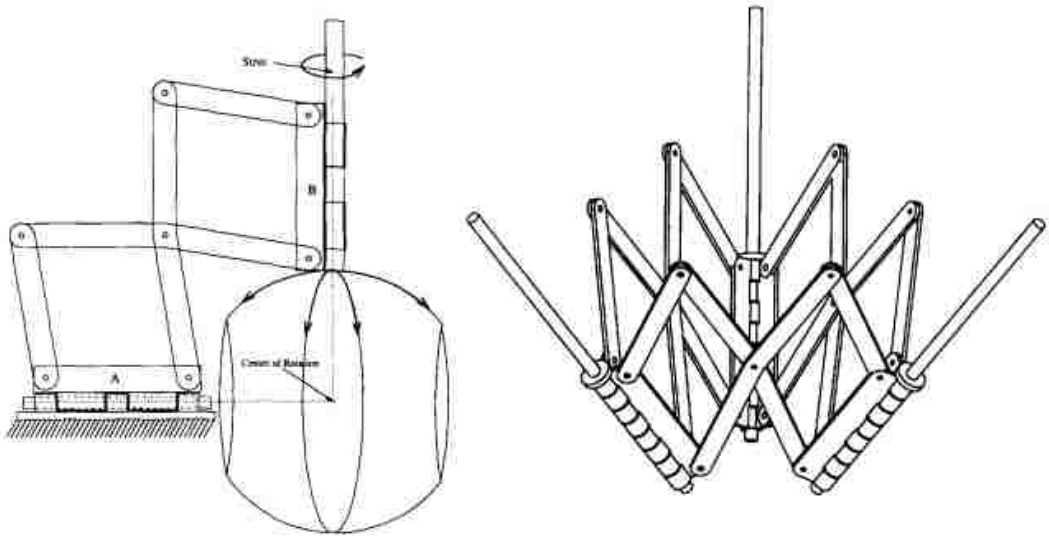

Figure 9: Basic concept of a concentric multilink spherical (CMS) joint.

The joint is used in modular parallel robotics with highly redundant degrees of freedom. Two modules are evaluated: tetrahedral and octahedral.

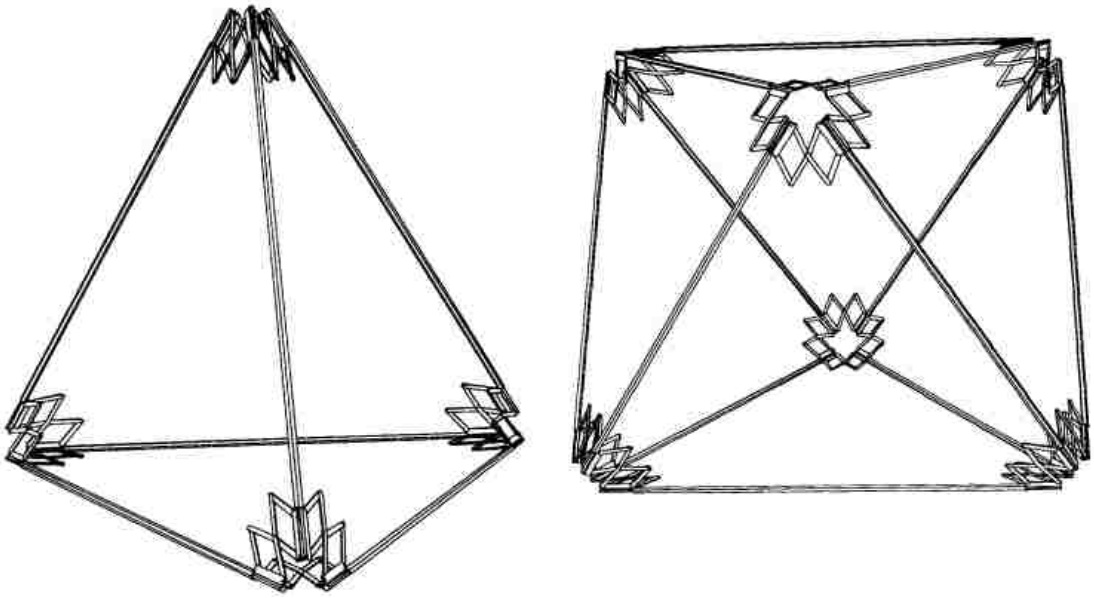

Figure 10: Tetrobot module examples.

Each link has a linear actuator and prismatic joint installed, and the modules are strictly arranged to avoid forming mechanical cycles between modules. Only the individual modules need to be solved in a parallel manner, and the propagation of motion between the modules can be treated as a traditional serial mechanism. A strong emphasis is placed on the modular nature of the robot, and many useful configurations are implemented.

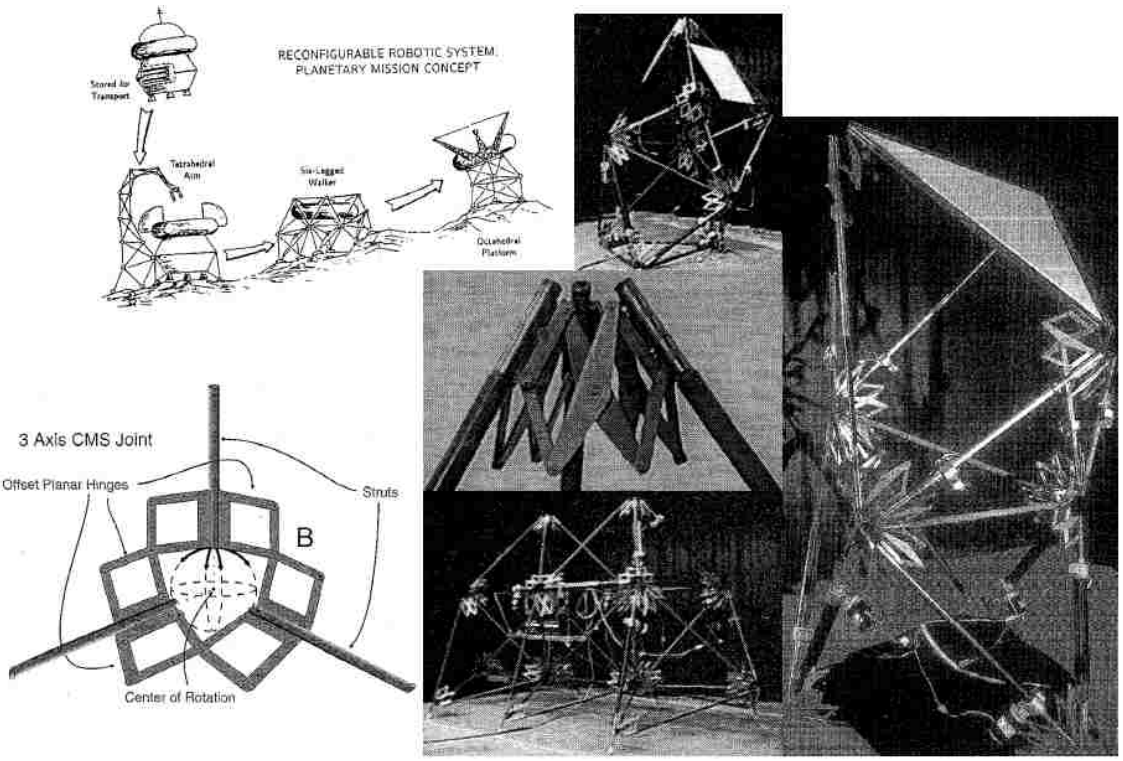

Figure 11: An overview of a Tetrobot VGT system.

The control of a robot formed from these modules is accomplished by first separating each joint into one of three categories: fixed, constrained, or unconstrained. Though three categories are used, the "fixed" and "constrained" categories can be combined with no loss of generality. Because the length of each link is variable, the position of each joint can be controlled independently. To determine the desired location of the unconstrained joints, a weighting method is used that the authors refer to as a "virtual force" method. A virtual spring and the associated spring force is attached to each link with the neutral spring position being set at the prismatic joint's zero position (this can be an arbitrary position, but a useful position is the point midway between the two joint extremes). A virtual force balance is calculated for each joint, and the resulting position is the desired

position of the unconstrained joints. In order to allow for a distributed control system, an iterative link by link sweeping method is used, but an analytical solution is also possible. This can equivalently be seen as a cost optimization method where the cost is the distance from the neutral link length position.

The work done by the authors of these articles shows the validity of VGTs, and builds a strong foundation for further work. The same principles that make static trusses more practical than solid filled volumes makes dynamic trusses more appropriate than serial robotic arms. The strength to weight ratio is much higher and as such, less material is required for the same strength — resulting in a much reduced cost. Dynamic trusses also have the added benefit of requiring much smaller actuators, also reducing the cost dramatically. Not only do they have better strength to weight ratios, but they are also highly redundant, resulting in greater dexterity and improved fault tolerance (better reliability).

The main drawback has been the ability to control a large number of actuators within extremely specific constraints (and calculating those constraints) in an efficient and logical fashion. Beginning in the late 1980s, modern computers began making this more feasible. Octahedrons have been the most complicated shapes used successfully so far, but general methods for any cyclic shape are surely not far off.

Newer actuators with higher strength to weight ratios, and ever smaller electronics make miniaturization a very exciting prospect for these types of robotic systems. Using tiny powerful actuators, physical three-dimensional displays and morphing prototypes become immediately possible. VGTs and VGFs may very well

be a precursor to the science fiction ideal of self-assembling nanobots for morphing materials. There is still an enormous amount of work before this point may be seriously considered but within a generation, crude physical three-dimensional displays and morphing prototype modelers are quite plausible.

Applications

By locking the joints, a variable geometry type system can be used as a rigid support when not being used for its robotic purposes and can form the frame of anything from a pencil holder to a vehicle chassis. When a robot is required, the surface can be powered on to perform the necessary tasks. Due to the large number of joints, there are many control points - each of which can be modified by attaching an end-effector or even a traditional robot arm, allowing for extremely versatile use.

By placing optical elements (reflectors, lenses, etc.) at each joint, this type of discrete robotic surface can replace parabolic reflectors or lenses for anything from solar concentration to maser power transmission to radio antennae – or even light collection for telescopic purposes. Because of its morphing ability, the range in which the focal area is located can be extremely broad, or it can even have multiple focal areas.

Another area of interest may be manufacturing. A surface of this nature can be used to quickly create arbitrary solid shapes for various casting, molding, and forming manufacturing methods leading to further methods for the rapid production of prototypes before investing in costly equipment for mass production.

With improved actuators a much smaller module size can be used, which allows for practical applications in personal three-dimensional displays. Stronger actuators make much higher loads possible, extending possible applications to morphing wings and fairings for improved aerodynamic performance. Using structures of this nature in the microgravity of outer space, an entire vehicle or even an entire space station would be easily reconfigured into a wide variety of desired shapes.

Active damping of vibrations has been a popular topic for a very long time. Usually, this is limited to placing a vibrating actuator on a structure in the direction of the highest vibrational amplitude or predicting the vibrational mode and changing the actuator's direction accordingly. In a variable geometry frame, the motion can be damped in any direction quickly and effectively using the actuators already built into the structure.

A surface made as a VGF can roll up to form snake-like structures to be used to clean pipes, climb trees, etc., or even as truss-arms (or "frame-arms"). It can form any number of closed shapes to be used as wheels, balloons, combustion chambers, etc. Because of its morphing ability, the applications of this type of mechanism are limited only by the creativity of the user.

CHAPTER 2

STATIC DESIGN

Static Structural Analysis (Static Joint Forces)

The finite element method (FEM) is applied to a new class of robots that can be called VGF robots. All triangle edges have actuated prismatic joints, and all internal (non-boundary) edges are rotationally actuated to control the angle between faces and avoid singularity at the in-plane positions described above. It is desired to determine the joint forces arising from static loading. Typical methods use the Jacobian matrix or Lagrangian equations of motion to find the joint forces [6][7]. Because of the highly redundant nature of this robot, these methods are not feasible. Indeed, the most complicated parallel joint configuration with a well-known closed-form solution has only six mechanical cycles comprised of six spherical joints and six prismatic joints [2][4]. In the proposed robot however, each modular increase in the length and width of the robot increases the number and size of cycles exponentially. FEA is not limited by parallel joints, and can be used very conveniently to find all of the required joint forces. It should be noted that the FEM is an approximating method that assumes link deformation causes the link to lie along a mathematical "shape" function. It also approximates the element's stiffness by only considering the stiffness of discrete lengths between nodes [8]. In the case of this study, the shape function used was a cubic polynomial. This means that the results will not be exact, but merely an approximation. As the purpose is to make sure that the actuators' maximum limits are not exceeded, this approximation should be satisfactory.

A stiffness matrix is calculated as the linear combination of the following well-known stiffness matrices: pure axial loading along the x-axis of the element's local coordinate system (x in the direction of the link), pure torsion about the local x-axis, pure bending about the local y-axis, and pure bending about the local z-axis. Each of these stiffness matrices was calculated using the cubic shape functions of an Euler-Bernoulli beam element [8].

If

$$\tilde{v} = [\Delta_1, \Delta_2]^T$$

is a displacement vector in all six degrees of freedom (translation along the x-, y-, and z-axes as well as rotation about the same) for each node of a beam element, where

$$\Delta_1 = \left[\delta_x^1, \delta_y^1, \delta_z^1, \varphi_x^1, \varphi_y^1, \varphi_z^1\right]^T$$

and

$$\Delta_2 = \left[\delta_x^2, \delta_y^2, \delta_z^2, \varphi_x^2, \varphi_y^2, \varphi_z^2\right]^T,$$

with δ representing the translational displacement and φ representing the rotational displacement, where the superscripts signify the node, and the subscripts give the direction, the following result is obtained:

Local stiffness matrix $k =$

$$\begin{bmatrix}
\frac{EA}{L} & 0 & 0 & 0 & 0 & 0 & \frac{-EA}{L} & 0 & 0 & 0 & 0 & 0 \\
0 & \frac{12EI_z}{L^3} & 0 & 0 & 0 & \frac{-6EI_z}{L^2} & 0 & \frac{-12EI_z}{L^3} & 0 & 0 & 0 & \frac{-6EI_z}{L^2} \\
0 & 0 & \frac{12EI_y}{L^3} & 0 & \frac{-6EI_y}{L^2} & 0 & 0 & 0 & \frac{-12EI_y}{L^3} & 0 & \frac{-6EI_y}{L^2} & 0 \\
0 & 0 & 0 & \frac{GI_x}{L} & 0 & 0 & 0 & 0 & 0 & \frac{-GI_x}{L} & 0 & 0 \\
0 & 0 & \frac{-6EI_y}{L^2} & 0 & \frac{4EI_y}{L} & 0 & 0 & 0 & \frac{6EI_y}{L^2} & 0 & \frac{2EI_y}{L} & 0 \\
0 & \frac{-6EI_z}{L^2} & 0 & 0 & 0 & \frac{4EI_z}{L} & 0 & \frac{6EI_z}{L^2} & 0 & 0 & 0 & \frac{2EI_z}{L} \\
\frac{-EA}{L} & 0 & 0 & 0 & 0 & 0 & \frac{EA}{L} & 0 & 0 & 0 & 0 & 0 \\
0 & \frac{-12EI_z}{L^3} & 0 & 0 & 0 & \frac{6EI_z}{L^2} & 0 & \frac{12EI_z}{L^3} & 0 & 0 & 0 & \frac{6EI_z}{L^2} \\
0 & 0 & \frac{-12EI_y}{L^3} & 0 & \frac{6EI_y}{L^2} & 0 & 0 & 0 & \frac{12EI_y}{L^3} & 0 & \frac{6EI_y}{L^2} & 0 \\
0 & 0 & 0 & \frac{-GI_x}{L} & 0 & 0 & 0 & 0 & 0 & \frac{GI_x}{L} & 0 & 0 \\
0 & 0 & \frac{-6EI_y}{L^2} & 0 & \frac{2EI_y}{L} & 0 & 0 & 0 & \frac{6EI_y}{L^2} & 0 & \frac{4EI_y}{L} & 0 \\
0 & \frac{-6EI_z}{L^2} & 0 & 0 & 0 & \frac{2EI_z}{L} & 0 & \frac{6EI_z}{L^2} & 0 & 0 & 0 & \frac{4EI_z}{L}
\end{bmatrix}$$

E is the modulus of elasticity, A is the cross-sectional area of the link, L is the length of the link, and I is the second moment of area about the axis noted in the subscript.

A transformation matrix is then computed using basic trigonometry to rotate the local coordinate system to match with the global coordinate system. There are many ways to write this type of matrix, but one popular method is the directional cosine matrix (DCM). Using the definition of the cosine function, the desired rotations can be written as shown:

Transformation matrix $T =$

$$\begin{bmatrix}
cxx & cyx & czx & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
cxy & cyy & czy & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
cxz & cyz & czz & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & cxx & cyx & czx & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & cxy & cyy & czy & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & cxz & cyz & czz & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & cxx & cyx & czx & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & cxy & cyy & czy & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & cxz & cyz & czz & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & cxx & cyx & czx \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & cxy & cyy & czy \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & cxz & cyz & czz
\end{bmatrix}$$

Where "c" stands for cosine and the next two letters indicate the argument. The first letter is the local axis and the second letter is the global axis (e.g., "cyz" indicates the cosine of the angle from the local y-axis to the global z-axis). Then

$$v = T\tilde{v}$$

and

$$\tilde{v} = T^{-1}v, \tag{1}$$

where v is a vector containing the six displacements of each node in global coordinates. The tilda denotes the same vector in local coordinates. If the forces are represented by

$$p = [p_1, p_2]^T$$

where

$$p_1 = \left[p_x^1, p_y^1, p_z^1, \tau_x^1, \tau_y^1, \tau_z^1\right]^T$$

and

$$p_2 = \left[p_x^2, p_y^2, p_z^2, \tau_x^2, \tau_y^2, \tau_z^2\right]^T,$$

and the spring constants are found in

$$k = diag[k_1, k_2]$$

where

$$k_1 = diag\left[k_x^1, k_y^1, k_z^1, k_{\varphi_x}^1, k_{\varphi_y}^1, k_{\varphi_z}^1\right]$$

and

$$k_2 = diag\left[k_x^2, k_y^2, k_z^2, k_{\varphi_x}^2, k_{\varphi_y}^2, k_{\varphi_z}^2\right],$$

then Hooke's law in local coordinates can be written as

$$\tilde{p} = \tilde{k}\tilde{v}. \tag{2}$$

In the same way as (1),

$$p = T\tilde{p}$$

and

$$\tilde{p} = T^{-1}p. \tag{3}$$

Substituting (1) into (2), gives

$$\tilde{p} = \tilde{k}T^{-1}v, \tag{4}$$

and then substituting (3) into (4) results in

$$T^{-1}p = \tilde{k}T^{-1}v. \tag{5}$$

Because T is seen to be orthogonal,

$$T^{\mathrm{T}} = T^{-1}$$

and (5) can be simplified to

$$p = T\tilde{k}T^{\mathrm{T}}v.$$

This equation can be rewritten as Hooke's law in global coordinates,

$$p = kv,$$

where

$$k = T\tilde{k}T^{\mathrm{T}}$$

is the global stiffness matrix for a single element.

Next, the element stiffness matrices must be assembled into a stiffness matrix for the entire structure. A simple and general way to do this is by indexing each node and associating those indices with their adjacent elements. Then to assemble the stiffness submatrix of a node, just add each submatrix associated with that particular node. Repeat this for each node, and the full structural matrix is complete. The twelve node VGF shown in Figure 2 is used as an example in two different positions. As this gives a 72x72 matrix (12 nodes x 6 degrees of freedom) the full position dependent stiffness matrix of the entire structure is not shown, although the relevant link forces are given in the results section. Once the model has been set up in this fashion, any position is simple to compute.

The unknown global displacements U_{unkown} are calculated from the known (applied) global forces P_{known} using the corresponding sections $K_{P_{known}}^{-1}$ of the full global stiffness matrix K by

$$U_{unknown} = K_{P_{known}}^{-1} P_{known} .$$

Once the unknown displacements have been computed, they are linearly combined with the known displacements (obtained from the boundary conditions) to give a complete vector of all displacements. If desired, this can then be used to find unknown reaction forces by

$$P = KU,$$

where P is a vector containing all global forces, K is the global stiffness matrix for the entire structure, and U is a vector containing the global displacements of all nodes.

But we are currently interested only in the joint forces. To calculate the joint forces, we first transform the displacements that are associated with the first element to the local coordinates of that element and then simply pre-multiply the local displacements by the local stiffness matrix. Taking these steps together to form a single equation,

$$\tilde{p} = \tilde{k}T^{\mathrm{T}}v,$$

where $\tilde{p}$ is again the vector of local joint forces represented in the local joint space (in this system, the local coordinate systems are defined as being coincident with the joint coordinate systems). The local coordinate systems are defined in Figures 12 and 13.

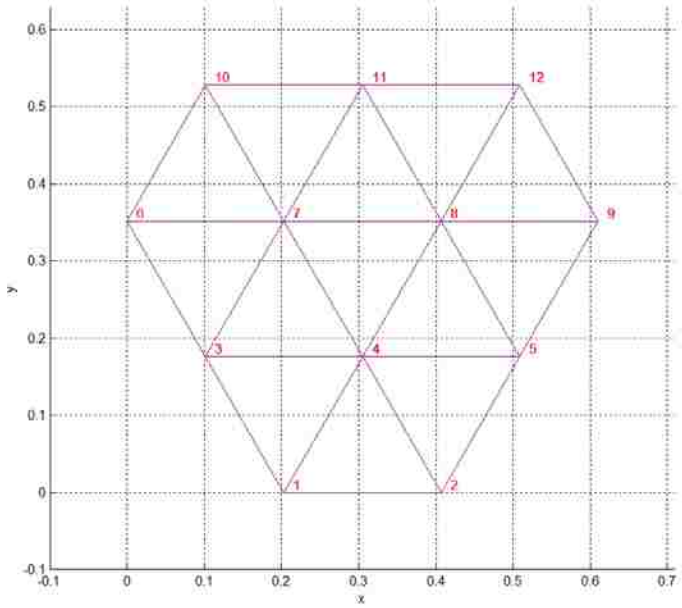

Figure 12: The node numbers of an example VGF in a planar position. The local y-axis is formed by a vector perpendicular to both the local x-axis and the global z-axis calculated by the cross product of the z unit vector and the x unit vector $\hat{\boldsymbol{z}} \times \hat{\boldsymbol{x}}$ to give a unique direction. The local z-axis is then naturally the cross product of the x and y unit vectors $\hat{\boldsymbol{x}} \times \hat{\boldsymbol{y}}$.

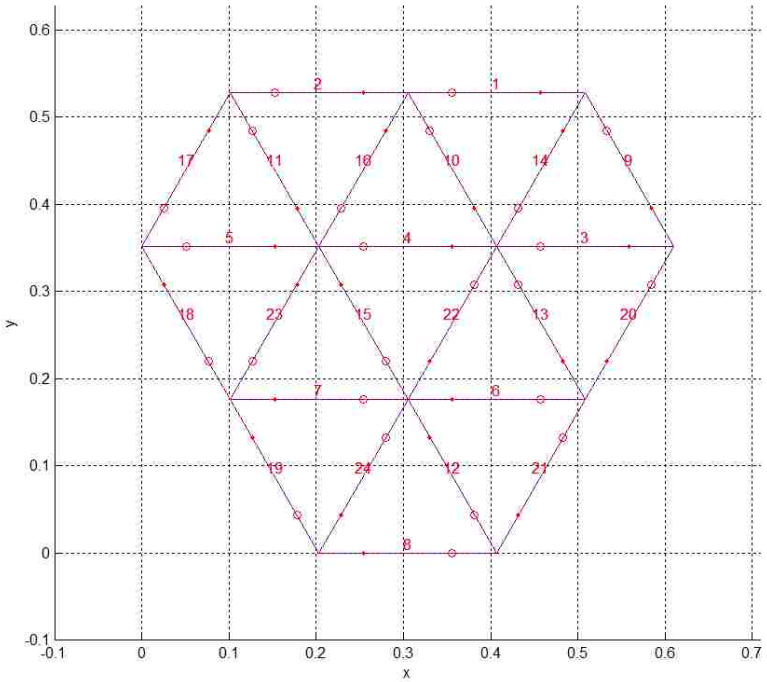

Figure 13: The element numbers of the same example VGF as in Figure 12. The local x-axis of each line starts at the node closest to the dot, and ends at the node nearest the circle.

Inverse kinematics are used to calculate the required joint positions to obtain desired control point locations. If each triangle vertex is a control point, the calculation of inverse kinematics is simple trigonometry and can be seen in the code in Appendix A. The trajectory of each control point is calculated by sampling a moving surface and ensuring that the distance between the time-varying sample points is within the acceleration and velocity limits of the actuators. This discrete motion is then uploaded to a microcontroller to control the actuated joints offline.

Two positions are shown to illustrate a range of motion and the related forces required to achieve it. The loading is the same on both shapes with the weight of the elements due to gravitational acceleration applied to each node. Node 1 is fixed in all 6 degrees of freedom as the reference node, and nodes 9 and 10 are both fixed in the z-direction, but allowed to slide in the x- and y- directions and freely rotate in all directions. In Figures 14 and 15, the blue lines represent the original shape, while the dotted red line is an exaggeration of the deformation after the load is applied. The displacements are all multiplied by 100 for all plots.

The first shape is a dome-like shape reminiscent of the geodesic domes created by Buckminster Fuller in the 1970s. Because this is a stiff position when considering only the load due to gravity, we expect relatively high axial forces and low moments.

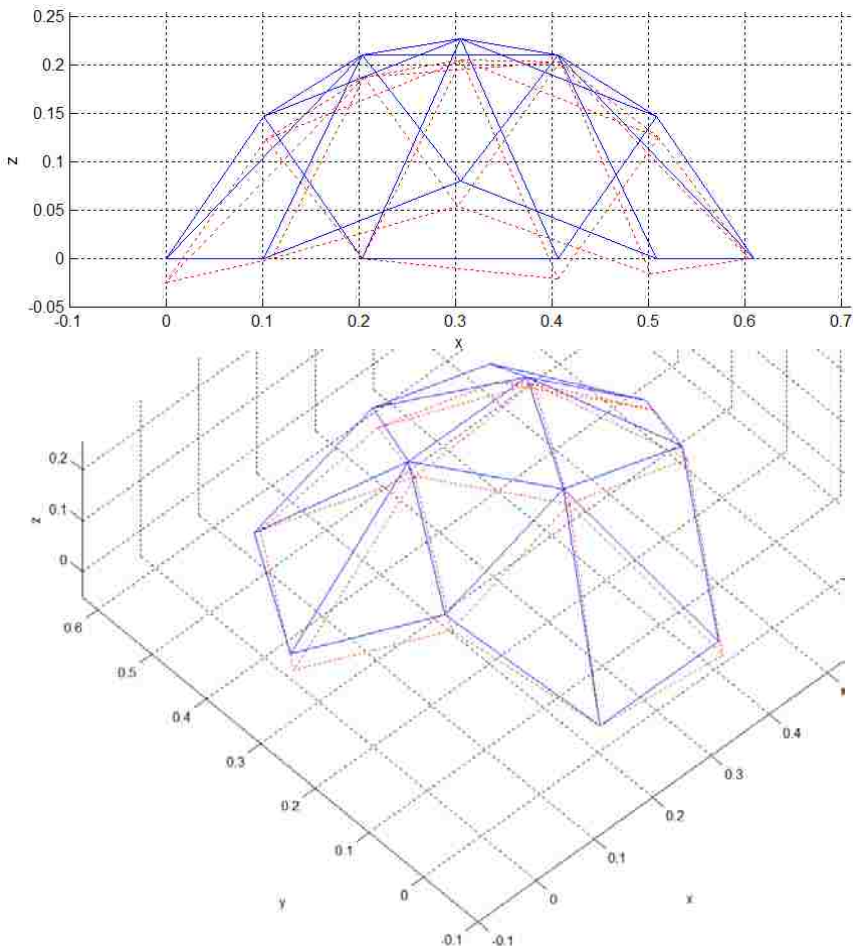

Figure 14: Very stiff convex semi-spherical position. The blue lines represent the undeformed shape, while the red dotted lines are an exaggeration of the deformation after the load is applied. Top - side view (parallel to the x-z-plane with the y-axis going into the page). Bottom - isotropic view.

The local nodal forces corresponding with vertical loading due exclusively to gravity, assuming a mass 20 grams concentrated at the ends of the links (10 grams per end) are calculated. The physical properties used for the axial stiffness are those of an aluminum ANSI #8-32 threaded rod corresponding to a potential linear actuator. The rest of the physical properties are those of a proprietary plastic that was used for the construction of the rest of the link (including the gearbox of the rotary actuator). The maximum axial force (which corresponds to the force that the

linear actuator needs to apply) for this position is 1.382 Newtons in link (element) 11 while the maximum moment about the x-axis (the rotationally actuated direction) is only 0.26 N·cm.

The second shape considered is a planar position that would be singular without the rotary actuators, and would yield infinite forces if modeled with non-bending truss elements. Because of the included actuators, this shape should have no axial force in the beams, but instead should have a shear force and a moment.

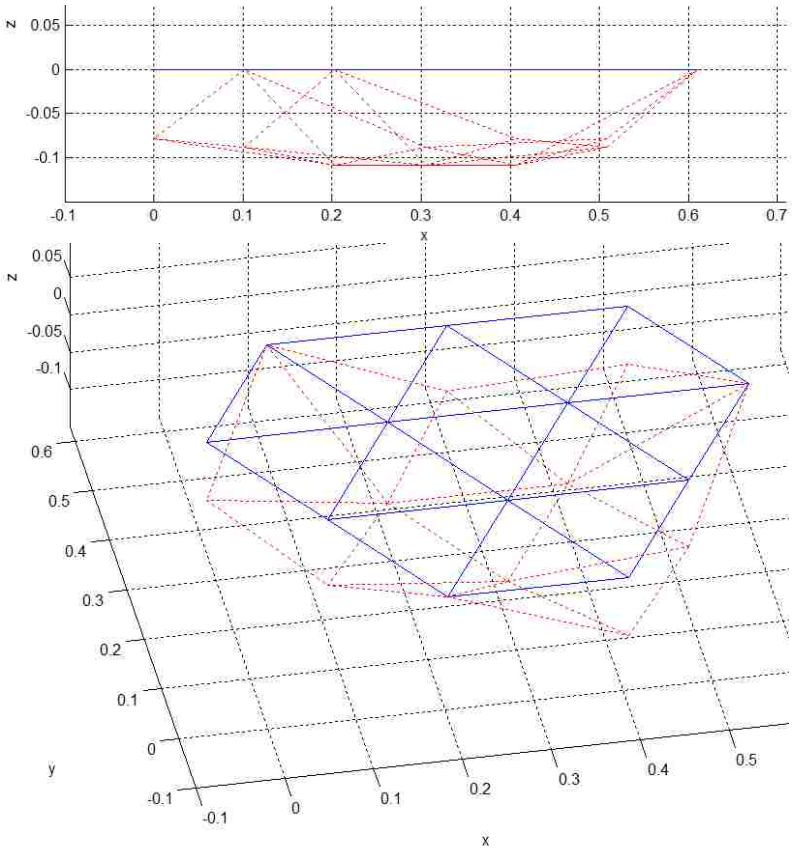

Figure 15: Zero "in-plane" position that, if not for the rotational actuators, would be singular and require infinite force in the local x-directions in order to maintain this position. With the rotational actuators, there is just a reasonable shear force and moment. The blue lines represent the undeformed shape, while the red dotted lines are an exaggeration of the deformation after the load is applied. Top - side view (parallel to the x-z-plane with the y-axis going into the page). Bottom - isotropic view.

As expected, all of the axial forces in this position are zero. The maximum moment about the x-axis (the actuated direction) in this position is 1.29 N·cm in link 2.

The other forces (in non-actuated directions) are important for the link design (and can be seen in the Exhibits), but our focus in this paper has been on finding joint forces without the aid of traditional kinematic or dynamic methods.

Mechanism Design

With the physical motion of the mechanism characterized and the critical joint forces estimated, all that was missing was a good joint for the design of this mechanism to proceed.

After a careful consideration of many spherical joints [2], a modification of the CMS joint was chosen. It was obvious that the offset links could be made straight if they were given an axis that intersected with the perpendicular "hinge" joint. This simplifies the joint, reduces weight, and increases strength and rigidity.

In a single layer surface application, all positions involving collinear adjacent links are singular to some extent. Rotary actuation is added to avoid these singularities and to allow for a strong focus on surface applications. The joint is modified to accommodate this actuation through a thorough analysis of the available linkspace, taking as much into account as seemed feasible at the time.

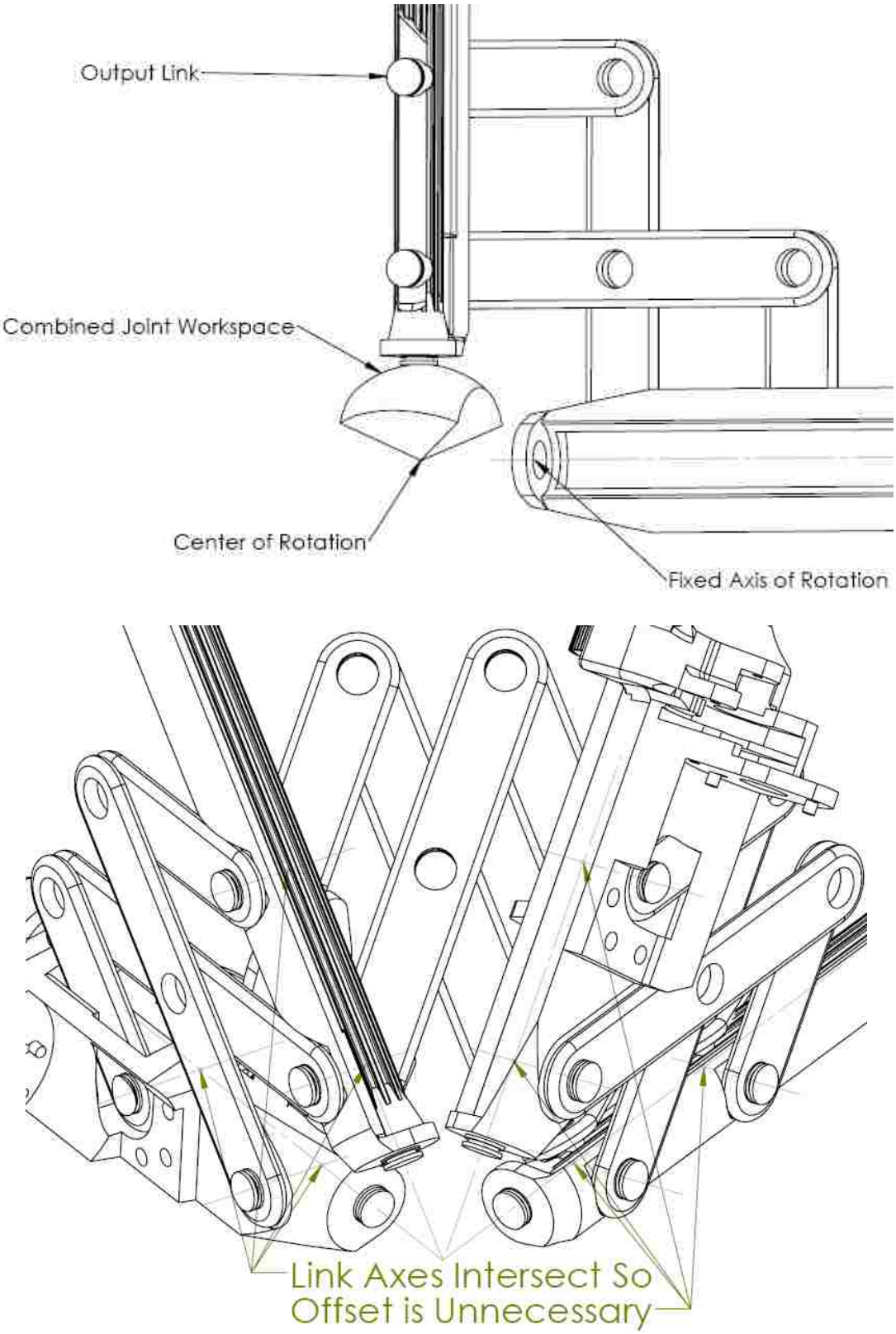

Figure 16: The CMS joint is modified to reduce complexity, increase strength and rigidity, and reduce weight.

Geometric Design Considerations

Joint range and actuator size are the main geometric constraints. When an actuator is mounted on a link, it is reasonable to consider it as part of the link. Therefore, larger an actuator is, the wider the link it is attached to becomes. The wider the links are, the less the joints can move before the links collide with one another. In the mechanism pictured in Figure 27, all of the most stringent constraints of this nature occur when the prismatic joints are at their shortest. Links must not intersect at any point throughout the joint range. This limits the links to be within a confined linkspace. The shape of this space can be defined by the joint ranges while the actuator size and placement requirements determine its scale (the actual size of the space which will have a shape that is mathematically similar to the shape determined by the joint ranges). Some of the joint ranges used for this mechanism are in turn dependent on the link size, so the two must either be solved simultaneously, or iteratively solved independently with reasonable values and then compared to find which is the more demanding constraint.

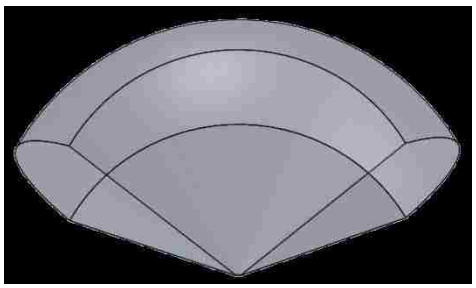

Figure 17: The overall range of the modified CMS joint is shown above using the arbitrary prismatic joint range of L to 1.5L (L+/-20%) – which corresponds with a passive joint range of ~39° to ~97° – and an arbitrary out-of-plane rotation of +/- 60°.

For the iterative approach, reasonable initial joint ranges are chosen. Beginning with the prismatic joints, a joint range of L to $1.5L$ is chosen, where L is the smallest total length of the link. This corresponds with a joint range of $\sim 39° \leq \theta_{passive} \leq \sim 97°$ for the passive revolute joints between the prismatic joints of a triangular module (calculated by comparing the extreme angles at the extreme prismatic joint positions). With no other considerations, this would result in a linkspace shape as shown below.

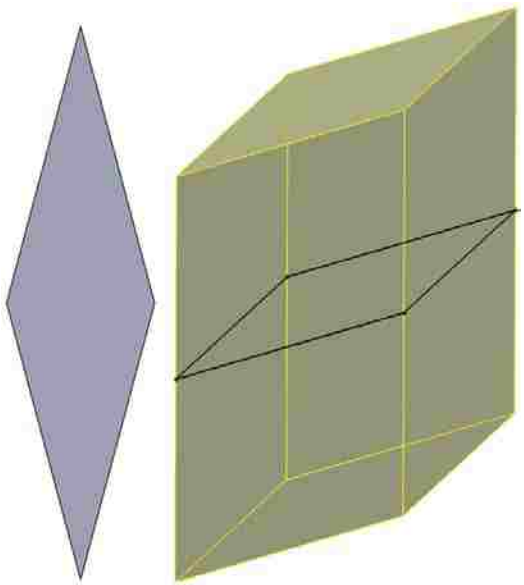

Figure 18: Joint space after considering the constraints imposed by a desired arbitrary prismatic joint range.

Next we consider the revolute joint between each module. If this joint were restricted to $-60° \leq \theta \leq 60°$, the jointspace shape appears as in Figure 18.

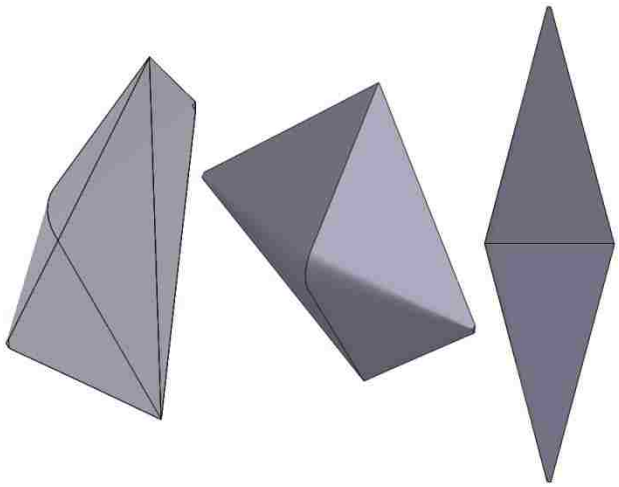

Figure 19: Three views of the linkspace after considering joint ranges in three dimensions.

Further geometric constraints require that the links be able to easily connect with one another. This will require the "trading" of linkspace with adjacent links in such a way as to maintain the similarity between links so that the modularity of the mechanism is not impinged. The locations of loading and grounding constraints must be carefully considered so as to reduce forces in unwanted directions and to maximize the rigidity as discussed in the Static Structural Design Considerations.

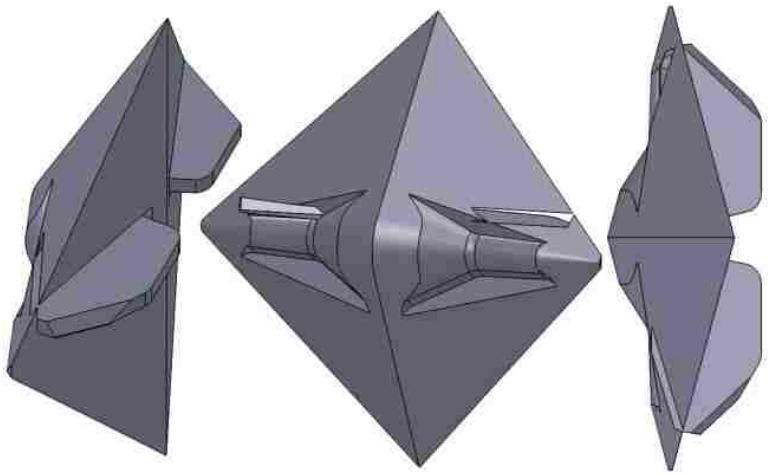

Figure 20: Three views of the linkspace after allowing for the passive joints.

Finally, the actuators must be taken into account. The location and capabilities of the actuators and the types of fasteners used are the most limiting constraints as they determine the feasibility of all of the other constraints. The motion of the actuator and any associated linkages must not be obstructed, yet must remain within the linkspace (though the space may be modified by trading with adjacent links as seen previously).

Static Structural Design Considerations

Structural constraints are mostly limited to rigidity and overall strength, though stress concentrations are considered as well. In order to obtain the optimal strength to weight ratio, the largest convex hull possible within the constraint space is desirable (or a combination of the largest – or fewest – convex hulls possible if a single convex hull violates space constraints). This convex hull is formed with the consideration of the loading and grounding sites as potential hull vertices. Because these sites are few, the different possible hulls can be quickly considered by inspection, but if they were more numerous, all of the hulls would need to have a thorough numerical inspection to find the optimal solution.

Though the strength to weight ratio is important, we must also make sure that the overall structure is strong enough not to fail under load, as well as ensuring that the structure does not deflect so much as to interfere with its operation. In the case of the mechanism being considered, the control of the mechanism is greatly simplified if the links can be treated as rigid objects. All of the most stringent constraints of this nature occur when the prismatic joints are at their longest.

Dynamic Structural Design Considerations

A desire to reduce inertia constrains the mechanism in mass and shape. As the constraints for the length of each link and the strength to weight ratio have already been added, the remaining improvements lay in reducing the thickness. This will reduce the rotational inertia about the prismatic joints' line of action, but will also reduce the flexural rigidity. The minimum thickness will be determined either by the constrained maximum deflection or by the constrained minimum strength requirements.

In case extremely high speeds are required (e.g. for a morphing flywheel), additional considerations would be required to evaluate dynamic balancing, resonant vibrational frequencies, etc. High speed applications are outside of the scope of this design, and have as a result not been evaluated.

Power Transmission

All mechanisms must transmit power from one location to another. As such, the methods of power transmission available to humanity are incredibly vast. There has been no attempt in this project to optimize the transmission. Though later changed (as discussed in Chapter 3) module 0.25 metric spur gears were initially used for all rotary power transmission (shown in Figure 20), and an ANSI #8-32 screw is fashioned into a linear screwdrive actuator for all linear power transmission. In order to transfer the rotary forces through the prismatic joints, a mechanical spline is chosen.

Although not optimized, the thickness of each gear is chosen according to the maximum amount of force each tooth will bear using a factor of safety of 5 and rounding up. This force is based on the maximum torque that the selected motors

are able to apply (~5 N·cm). Likewise, the thickness of the pins supporting each gear is chosen so that the maximum deflection of the pin will not allow the teeth of any mating gears to slip.

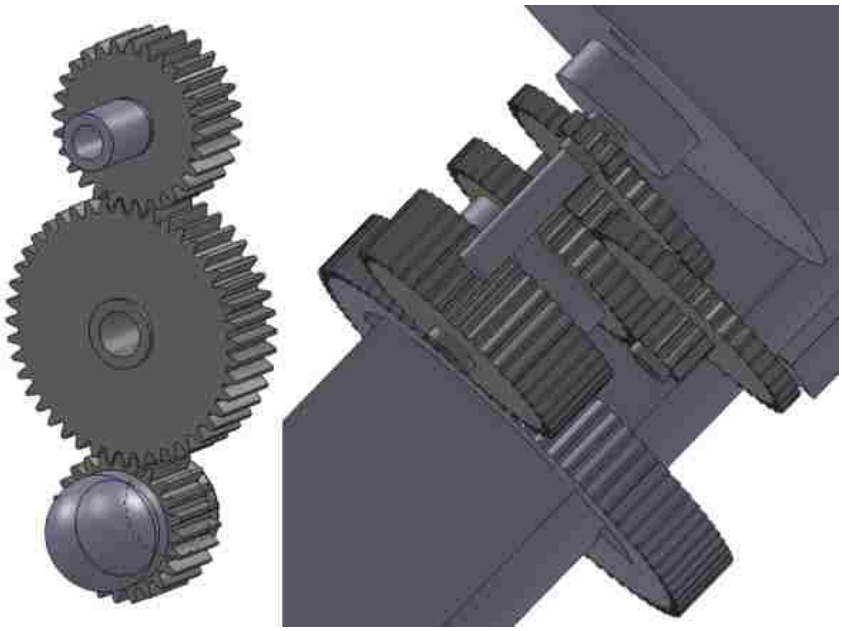

Figure 21: Initial power transmission using M0.25 spur gears. On the left is a 6:7 transmission from a motor to an ANSI #8-32 screw drive. The right is a gear reduction of ~621:1.

"Final" Link Design

Once the linkspace is finalized, the "link" as we have designed it must be separated into two links joined by a prismatic joint with enough space for a linear actuator. Because a convex hull is desirable, the center of the linkspace is chosen for the actuator. The two links are separated by maximizing the hull thickness of each link while ensuring that the desired range of motion can be covered as well as maintaining the desired rigidity. The results are shown in Figures 22-25.

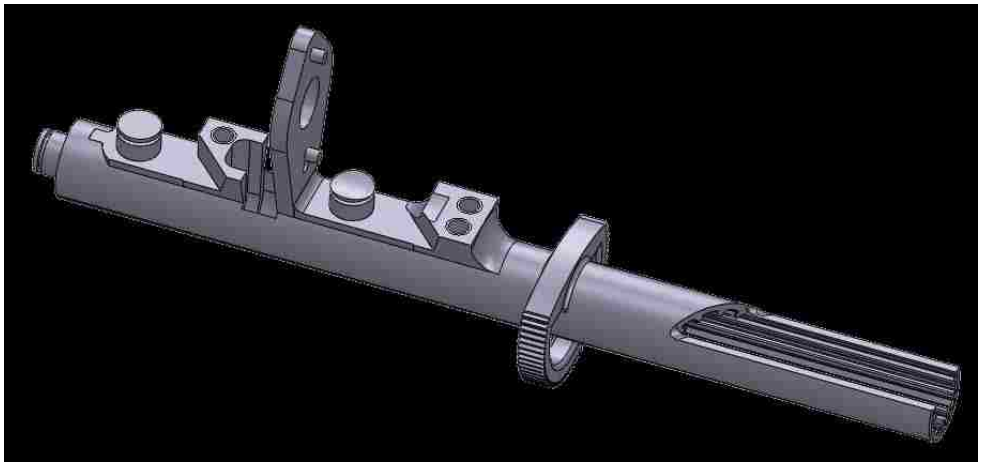

Figure 22: Linear actuator mount section of the overall link.

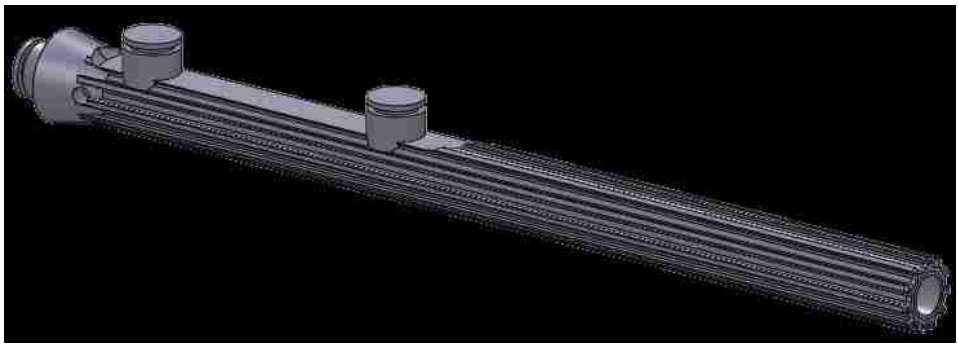

Figure 23: Linear extension section of the overall link.

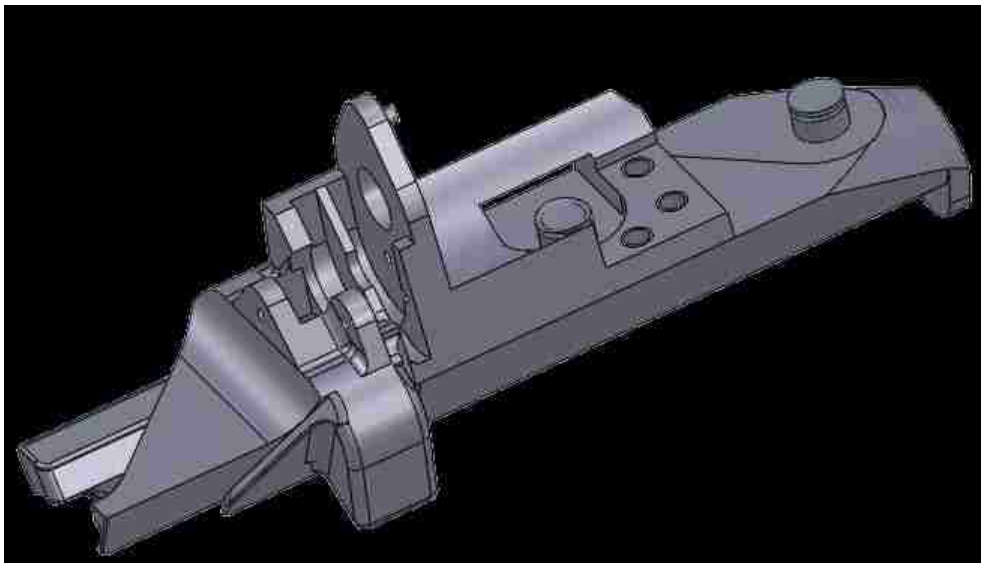

Figure 24: Rotary actuator mount section of the overall link.

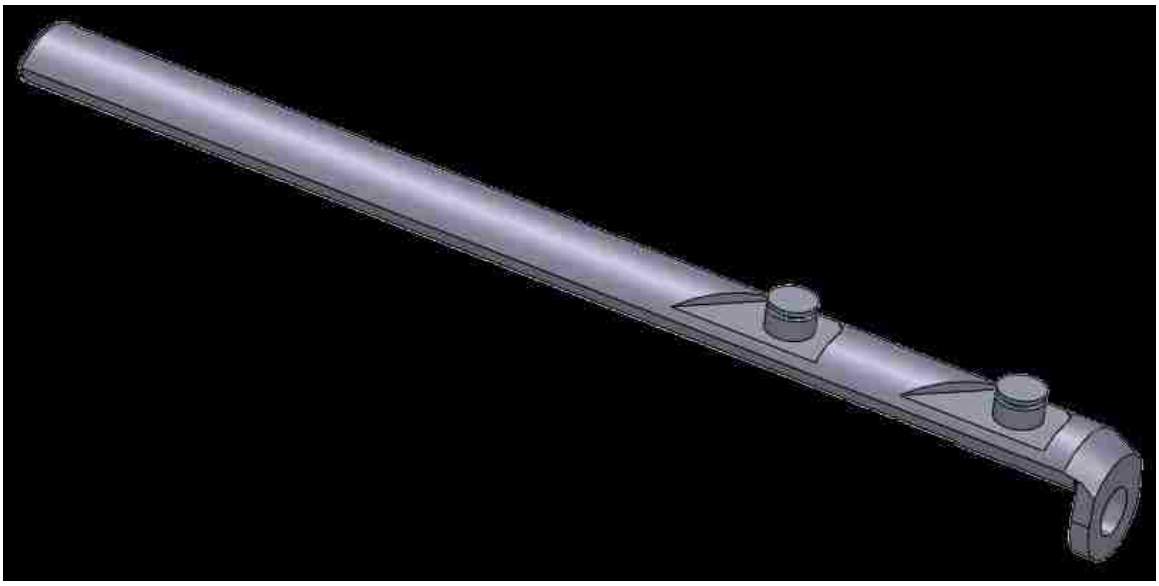

Figure 25: Rotary extension portion of the overall link.

Manufacturability Design Considerations

Because the mechanism will be printed on a 3D inkjet printer, the design for manufacturability (DFM) constraints are significantly less stringent than they might otherwise be with more traditional manufacturing methods. Nevertheless, certain constraints are necessary. The main constraint is that the support material must have an exit path to allow easy removal from the locations that require its use. This means that there will need to be holes in places that would otherwise be solid. Because the assembly will be done by hand, other constraints will require that interior parts be easily accessed by fingers or tools.

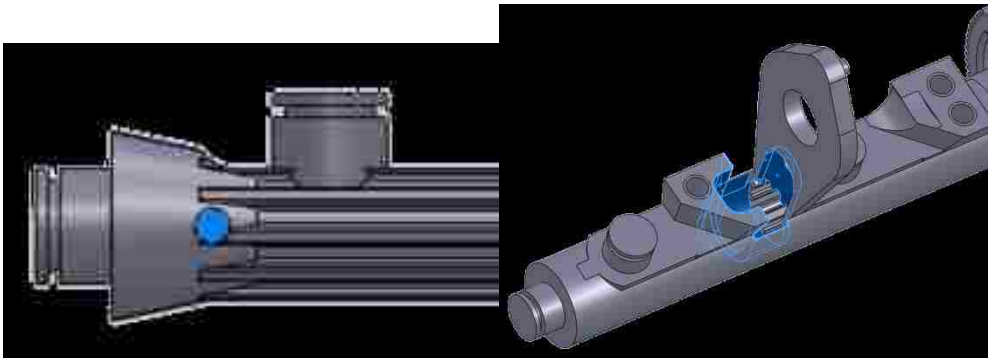

Figure 26: Extra holes and sections removed to ease manufacture and assembly are shown in blue.

The design process is circular in nature, in that it has no end and seems to repeat steps. Further manufacturing changes were made due to the limitations of the rapid prototyper. These further changes are discussed in the next chapter which explains the construction process.

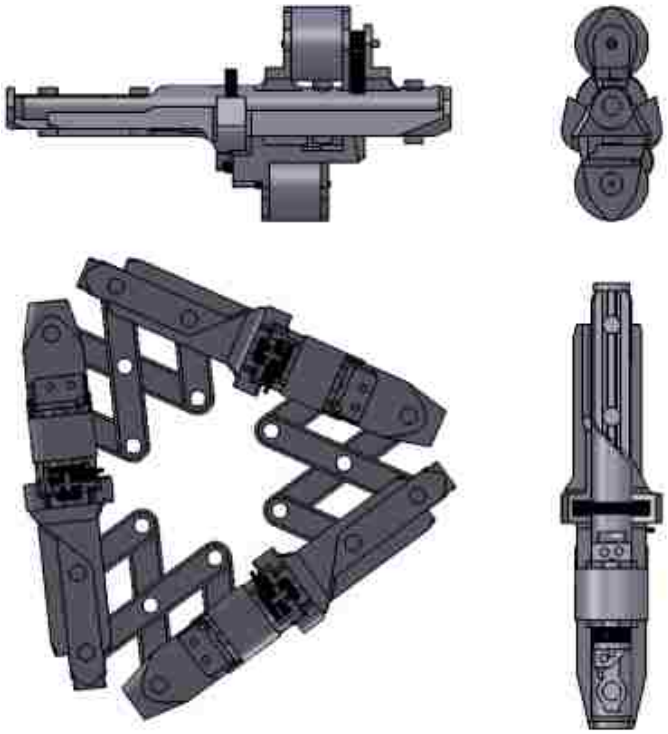

Figure 27: Single triangular module and multiple views of a single modular link.

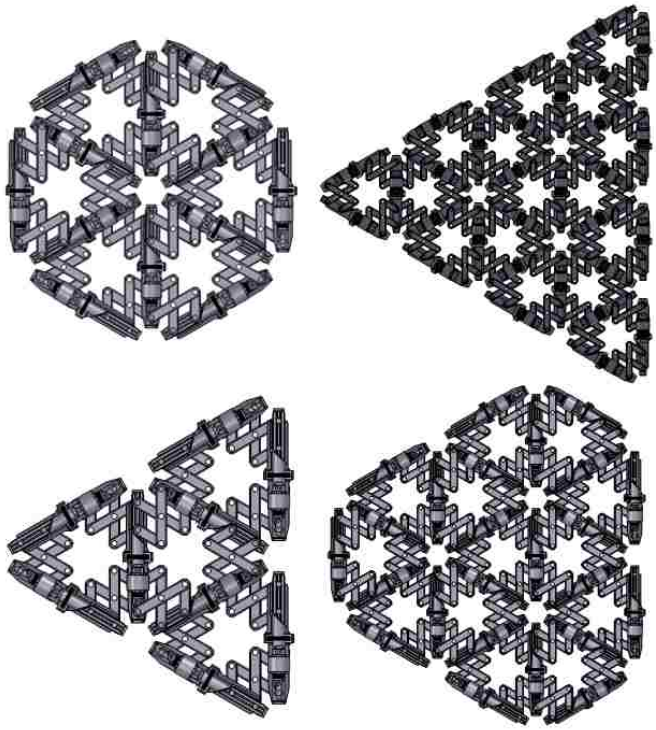

Figure 28: Example arrangements of VGF surfaces composed of triangular modules.

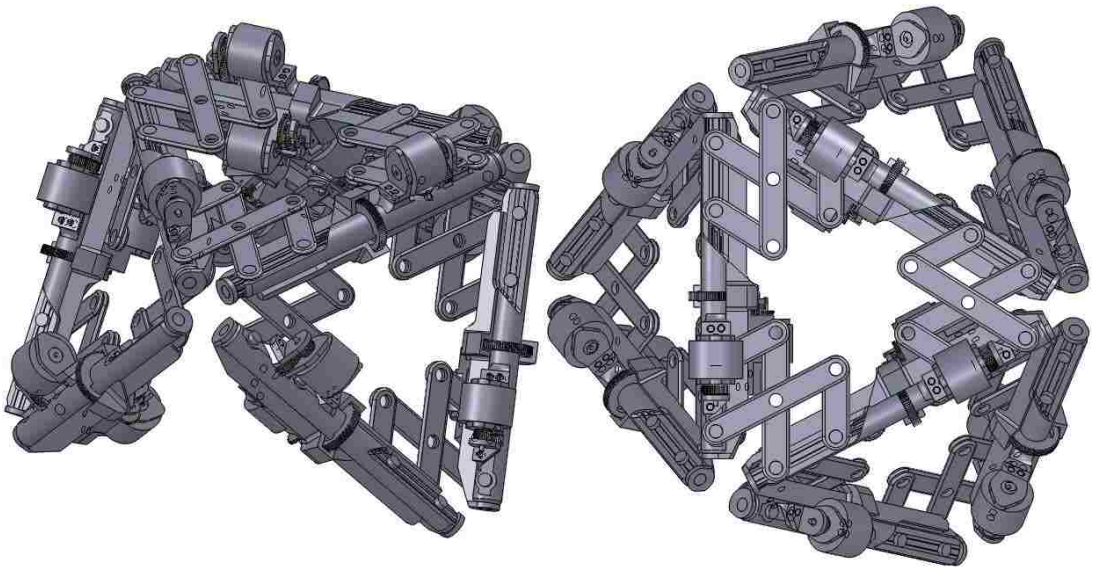

Figure 29: Example position of a four module VGF.

Kinematics

Forward Kinematics

The kinematics of parallel mechanisms are very interesting. The same actuarial principles used in serial mechanisms can be employed, but because of the closed mechanical cycles, addition considerations are required.

In a serial mechanism, the location of the first link in the chain is found with respect to the ground coordinate system. The second link in the chain is found in terms of the first joint variable and defined relative to the first link, the third link is found in terms of the second joint variable and defined relative to the second link, and so on. In this way, each link can be described by a variable vector, and the vector sum of all the links in the chain gives the location of the end point of the chain.

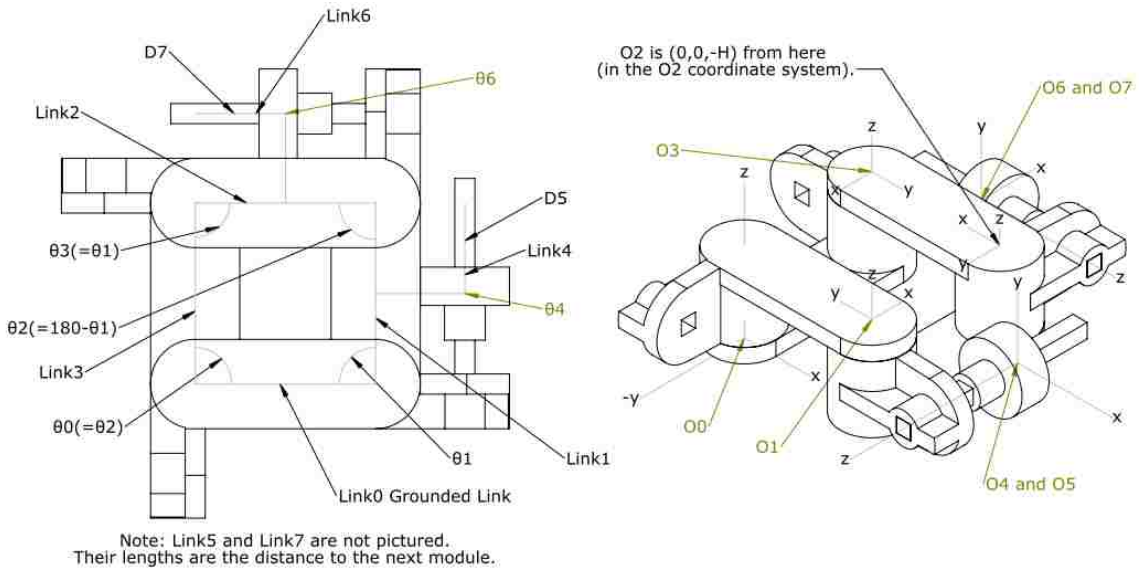

Figure 30: Coordinate definitions used in forward kinematics for a rhombic module.

Initially, this same approach can be used with parallel mechanisms. The difficulty in this step is defining the "first" link and the subsequent order of links. Any serial chains within the parallel cycles have an obvious order, but what will the order be at the branch points? This is where the extra considerations come into play. After defining each serial chain in the mechanism, each closed cycle constrains each serial chain contained within it. This "constraint" is simply that each serial chain must begin or end at the same branch points at which the other serial chains in the loop begin or end.

Constraint Equations in Forward Kinematics

This constraint can be described mathematically by equating the two vector paths (i.e. the end locations are the same). These constraint equations are often highly nonlinear, and can be very difficult to solve – usually having no known analytical solution and therefore requiring the use of numerical methods. Because of their nonlinear nature, there will also usually be multiple solutions. For three dimensional parallel mechanisms, there will likely be an infinite number of solutions.

Inverse Kinematics in Forward Kinematics

Another way to think of this is by defining one serial forward kinematic path in the usual way, and then computing the inverse kinematics for the rest of the serial paths connected at the same branch point with the target end point defined by the initially calculated path.

In highly redundant parallel mechanisms, the serial "chains" may be only one or two links long before reaching a branch point, and the branch points may have

large numbers of serial chains attached to each one. A convenient way to handle this is by defining an arbitrary serial path through the entire mechanism and calculating the positions of any links not in this initial path based on the path position (inverse kinematics).

With triangular modules arranged in a mechanism as described above, there are many possible serial paths. A convenient example might be a spiral path. This would make the inverse kinematics of all links not included in the path very simple, and would let us find a single unique solution. Each link is connected either by a prismatic and spherical joint if the links are in different modules, or a prismatic and revolute joint if they are in the same module. The spherical joints could be modeled with many different combinations of joints, but a "ZYZ" type joint is most common. All remaining links not in the initial serial path can be thought of as single links with two spherical joints with a prismatic joint in between – easily calculated from the initial path position.

Modular Definitions in Forward Kinematics

Another perhaps more appropriate method could be a modular approach. A single triangular module can be defined in terms of its joint positions, and then the interfacing joints between modules finish defining the overall shape. In the end, we want to put everything in terms of the actuated joints. This method makes this task slightly simpler than the inverse method as all of the joints used in the model are real joints in the mechanism – we simply have to calculate the passive joints in terms of the actuated joints which, for a triangle, is quite simple.

The forward kinematics were initially calculated for the rhombic surface (see Appendix B), but they were not useful enough to consider for the triangular surface.

Inverse Kinematics

The inverse kinematics of parallel mechanisms are perhaps less interesting in practice. This is because the method used for to calculate the inverse kinematics depends greatly on the desired application. Most methods consider the position of each link to be known or controlled in some way. This makes the inverse kinematics almost trivial in calculation, but produces some very interesting control schemes. The reason for this is that if only one ground position and one "end-effector" position is known, an analytical solution is not only very difficult to obtain – in fact it has been proven that there is no general solution for five or more joints connected in a loop – but also gives multiple solutions. In serial mechanisms, these multiple solutions can be a problem, but in parallel mechanisms these problems are compounded exponentially. For most parallel mechanisms, there are literally an infinite number of solutions. Rather than try to deal with these "singular spaces," it is much more sensible to control as many links as possible and use a placement rule or control law to determine the location of the uncontrolled links.

For this mechanism, a method of dynamic shape control is used, so that the motion of each link is fully defined by the controller. As such, the forward kinematics are not required. The inverse kinematics are calculated with basic trigonometry; requiring only the calculation of the distances between the endpoints of each link for the prismatic joints, and the angles between each module for the

actuated revolute joints. This fully defines the position of the mechanism and as such, all other joints are passive.

CHAPTER 3

PROTOTYPE FABRICATION

A three-dimensional inkjet-style printer that uses proprietary ultraviolet light curing plastic was used to print each part other than the stepper motors, fastening screws (steel), and the threaded rod (made from aluminum). The main links have four primary components bearing all combinations of two characteristics; either a mount for the stepper motor or an extension from the mount, and either connected to the rotary motor or to the linear motor (see Figures 31-34). Other than the main links, there are the fasteners for the motors, the passive joint links, the power transmission, and of course the actuators themselves.

An interesting feature shown in Figure 35 is the half joint formed between the linear mount and the rotary extension portions of the link using an annular space in the driven gear of the rotary drive train. This joint not only increases the range of the prismatic joint by allowing the rotary extension to extend past this gear when the prismatic joint is fully retracted, but also provides much needed support when the prismatic joint is fully extended.

The surfaces of prototypes printed with the selected printer sometimes have lower resolution than advertised (this phenomenon is described more fully later on). This led to the module size of the gears doubling from 0.25 to 0.5, causing there to no longer be enough space for the gear train that had been planned. Instead, an epicyclic gear train was required. This new transmission (shown in Figures 37-38) is more stable, more efficient, and more compact than the original transmission, so it was just as well that the change was forced.

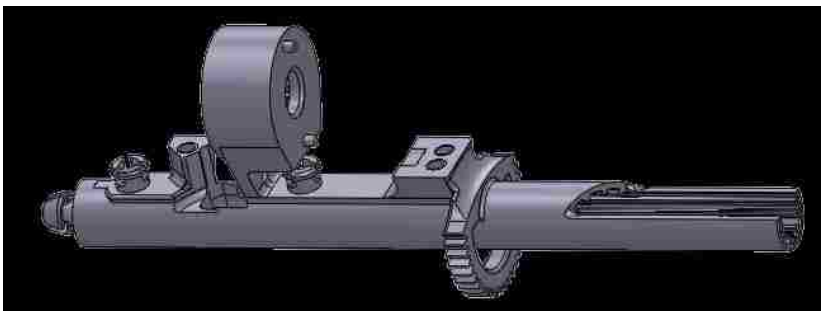

Figure 31: Linear actuator mount section shown with 0.5 M epicyclic gear housing and three-pronged snap-together clips.

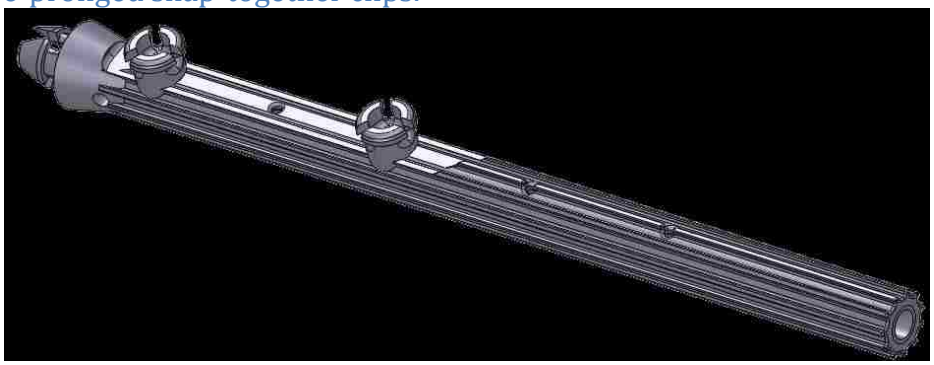

Figure 32: Linear extension section with three-pronged snap-together clips and extra holes for support material removal.

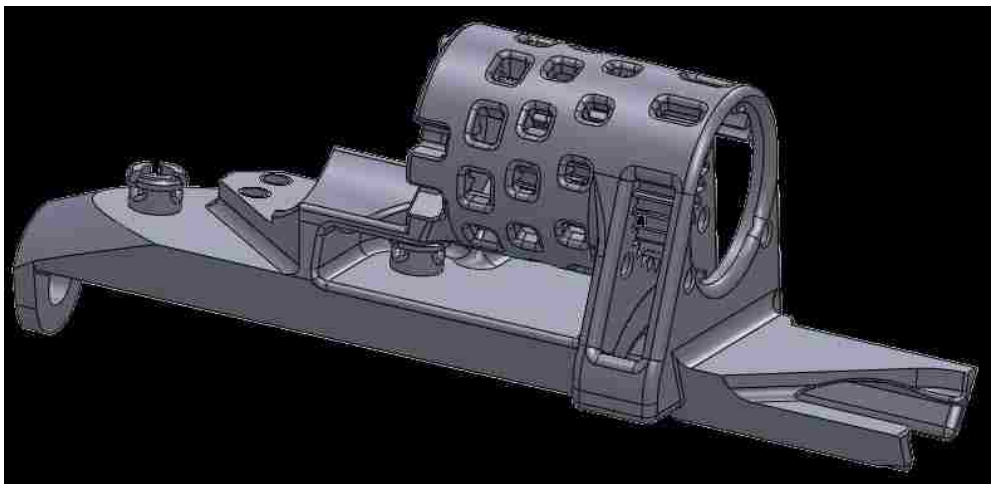

Figure 33: Rotary actuator mount section shown with 0.5 M epicyclic gear housing and three-pronged snap-together clips.

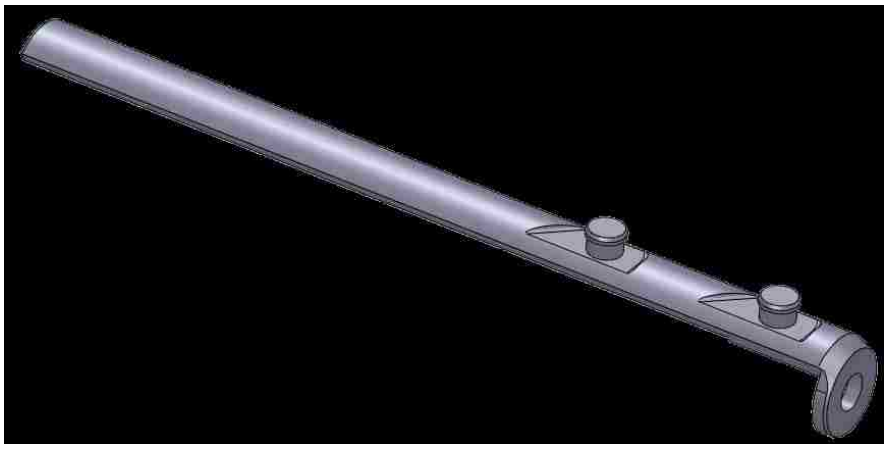

Figure 34: Rotary extension section shown with break-away reverse-tapered press fit joints.

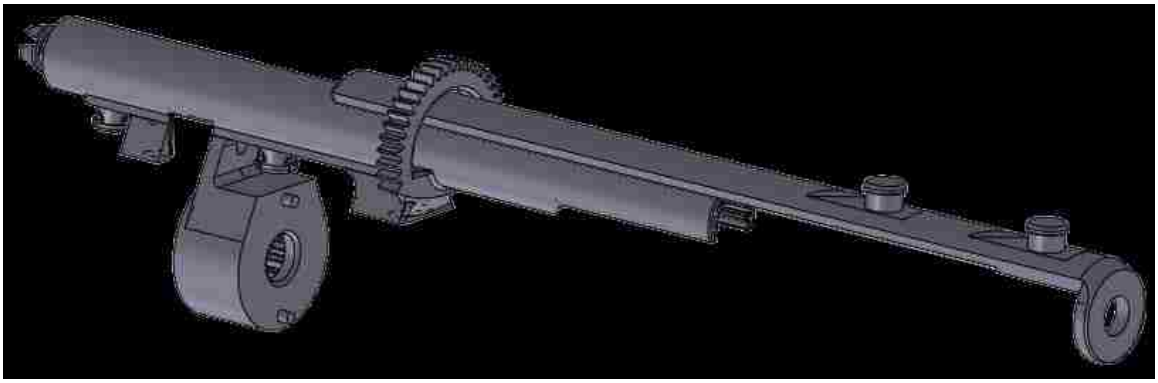

Figure 35: The prismatic extension of the rotating link component also forms a half-joint with the actuator mount side of the linear link component.

Figure 36: One full link of the prototyped mechanism with a rotary and prismatic actuator. Because all links are the same, one link might be considered to be a more fundamental module than the triangular module adopted for this project.

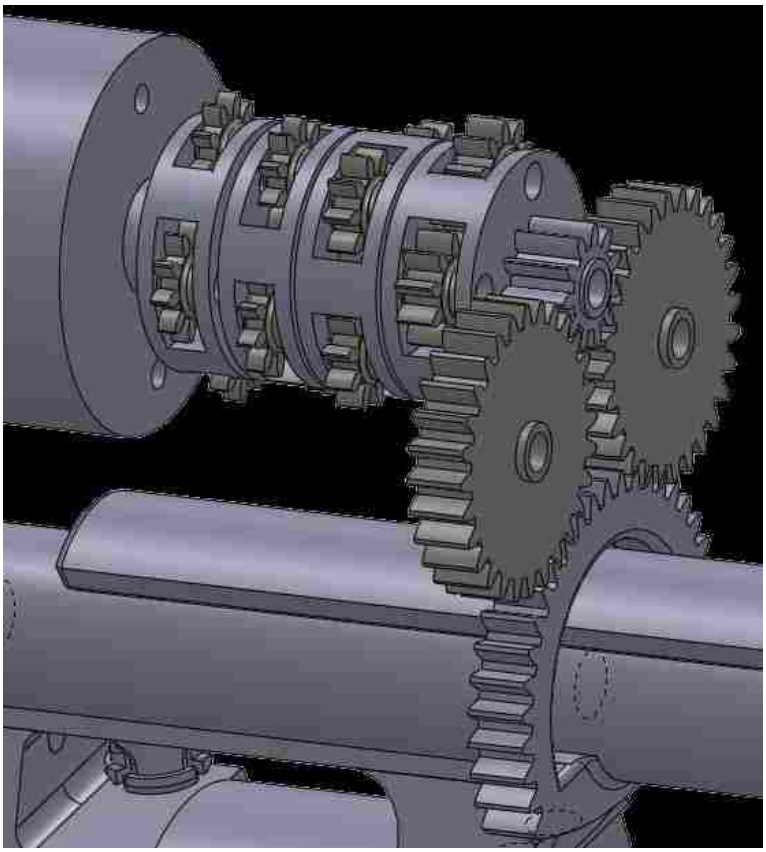

Figure 37: Geared rotary transmission using four epicyclic stages giving a total gear ratio of $\mathbf{977.\overline{45}:1}$.

In this transmission, there are four epicyclic (planetary) gear stages for each rotary actuator with each stage comprising four spur gears (each with its own pin) and one arm – with a spur gear on the arm (and all stages share a common arm) for a gear ratio of $977.\overline{45}:1$. Each linear actuator has two epicyclic gear stages that are identical to the rotary actuator's gear stages. In addition, each linear and rotary actuator both have two idler gears, the driving gear, and the driven gear for a gear ratio of 4:1 connected to the ANSI #8-32 screwdrive. With a full module having

three linear actuators and three rotary actuators, this is a total of two-hundred-twenty-two gears and one-hundred-seventy-four pins per module. Even though there are six modules in the prototype, because the outside borders of the mechanism do not need rotary actuators, this gives a grand total of only six-hundred-sixty-six gears and five-hundred-twenty-two pins in the device.

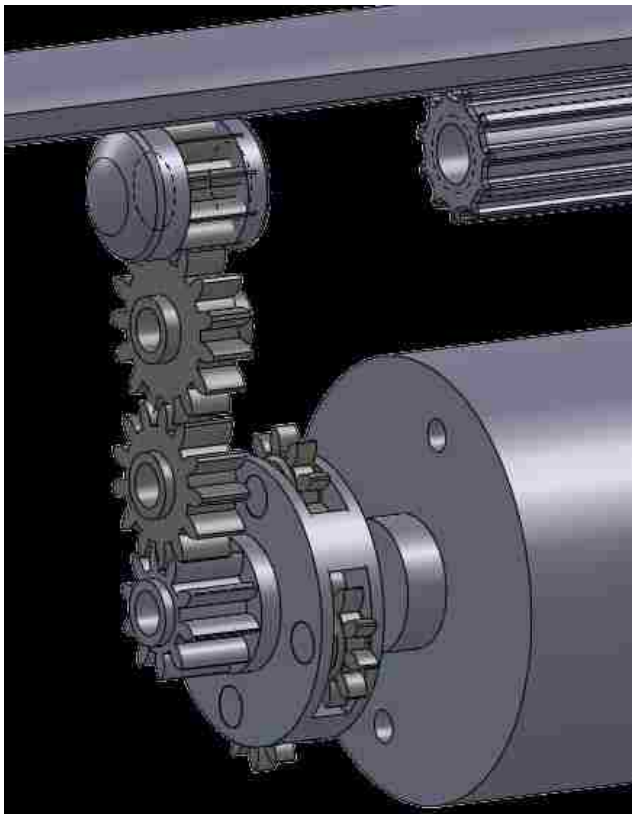

Figure 38: Geared linear transmission using one epicyclic stage for an overall gear ratio of 4:1.

Each module has six long and six short joint links making up the six-bar joints for a total of seventy-two passive joint links. There are eighteen stepper motors and twelve ~6 inch ANSI #8-32 threaded aluminum rods. There are two

types of actuator fasteners used on the mechanism; one type for the linear mounts, and one for the rotary mounts. Each linear mount fastener uses one ANSI #4-40X1/4" screw, while each rotary mount uses one of these as well as one ANSI #4-40X3/8" screw.

Figure 39: ANSI#8-32 aluminum threaded rod used in the linear actuator, fasteners for the stepper motors, and screws used to attach the fasteners.

Each stepper motor has four 36 inch color-coded power wires attached to both the actuator and to a connector to the stepper drivers. The drivers each have two power wires connecting them to the power supply, four power wires connecting them to the actuator connector, and two control wires connecting them to the microcontroller. All of the wires connected to the drivers are 6 inches in length. This is a grand total of two-hundred-sixteen wires with a total length of 96

yards (87.8 meters) of stranded aluminum wire and 17 yards (15.5 meters) of heat shrink.

Figure 40: Nest of wires and drivers.

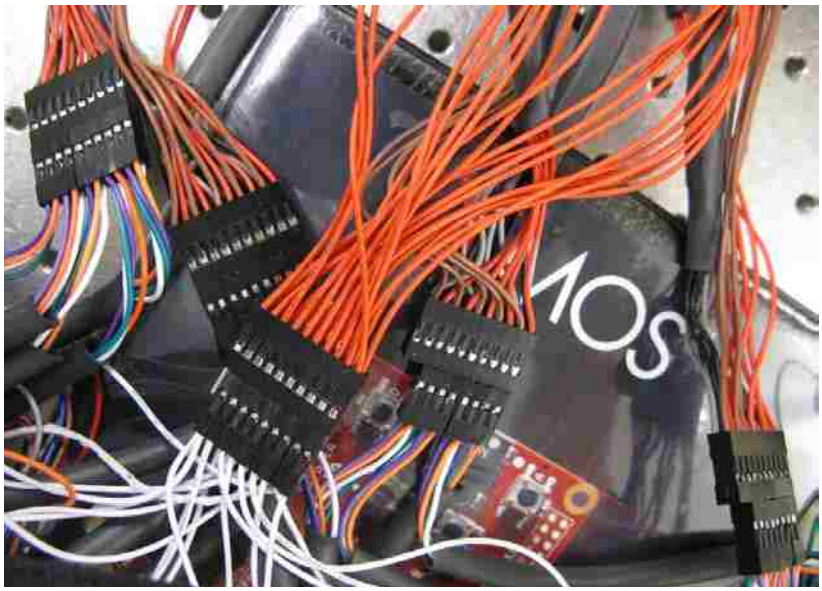

Figure 41: Power wires. Red is 5VDC and black is the corresponding ground, Brown is 12VDC and grey is the corresponding ground. All others are control wires. The white control wires are directly connected to the 5VDC wires to draw the microstepping pins high at all times.

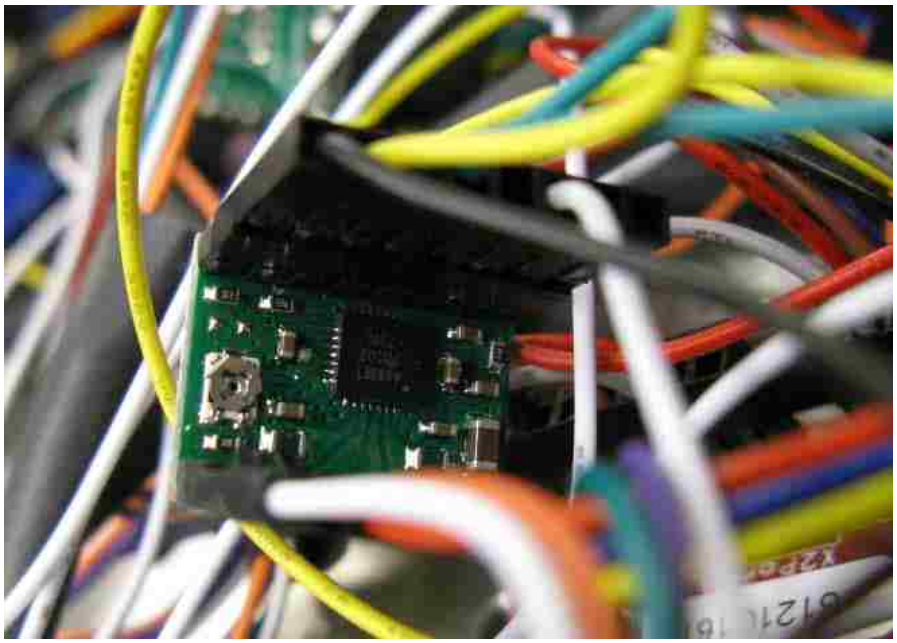

Figure 42: Stepper driver and wires. The screw on the left of the board is a potentiometer to control the current. It is left all the way open which corresponds with ~0.95A when driving with 12VDC and using long power wires.

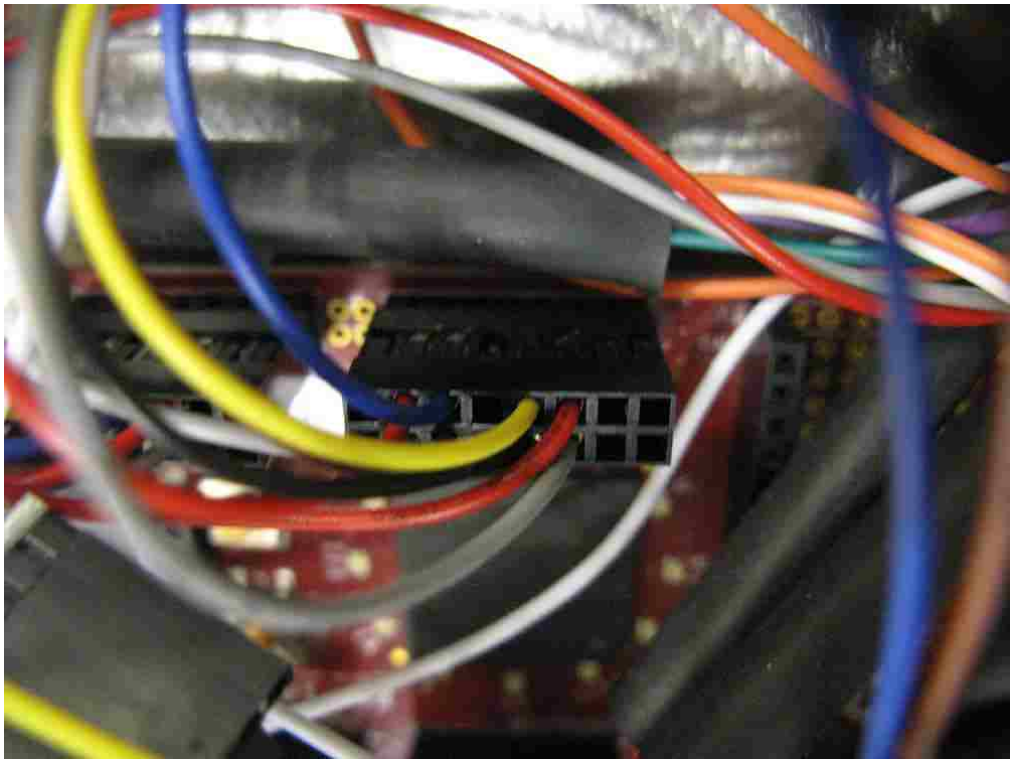

Figure 43: Control board header and control wires. For the program that I have written (which can be seen in Appendix B), the top row controls the step count, while the bottom row controls the direction.

As is typical with plastic inkjet printers, every part that was printed needed to be hand finished. The support material (a semi-water-soluble gel-like material that the printer uses to create undercut/overhang in parts) needed to be cleaned from the part (a task that was extremely arduous – especially in small enclosed spaces – and involved a wire coat hanger, chopsticks, and numerous clay sculpting tools in addition to a proprietary pressure washer), and flashing/burrs needed to be removed and smoothed.

Figure 44: Left – 3D inkjet printer. Right – pressure washer glovebox.

Figure 45: A batch of epicyclic gear arms and gear mounts with support material.

As previously mentioned, the printer that was used has a reasonably high resolution and fairly good fidelity, but creates a burr whenever there is a glossy surface joined to a matte surface. The glossy surface is a surface finish option that has significantly less friction and is significantly stronger and more rigid than the matte surface. Unfortunately, the glossy surface is only available in locations on the part that do not come into contact with support material. Because the lower friction and higher rigidity of the glossy surface was desired, all printed parts needed to have burrs removed from the glossy surface-matte surface junction. Each of the six-hundred-sixty-six gears needed to be de-burred in order to be able to spin freely;

each pin and pin-hole needed to be de-burred in order to mate properly; all fasteners and internal gears needed to be reshaped in some way; and all sliding surfaces needed to be de-burred to allow the joints to move freely.

Figure 46: This spur gear should be symmetric through a roughly horizontal line, but the burr on the bottom makes this obviously not so.

Figure 47: Notice the flat section in the upper right corner. This should be a perfect circle.

Figure 48: A side view of the same object as in Figure 47.

The last thing that had to be done in preparation of assembly was to tap each of the forty-eight threaded holes. In each of these steps, if a mistake was made, the part had to be remanufactured and processed anew. When pressure washing and otherwise removing support material, it is very easy to apply too much force and break a part. In de-burring, it is very easy to carve off a little too much and have a gear that will slip teeth. In tapping threaded holes, it is very easy to tap slightly off-axis and have a hole that applies large off-axis torques to the screw. Each of these tasks was like learning a craft. If the reader perhaps has any experience building a model, the preparation and assembly of this mechanism was very much like that.

Once the preparations were completed, the assembly began. Numerous clips pins and joining methods were attempted unsuccessfully before the current method was perfected. A reverse tapered press-fit is used for some of the passive joint links while a three-pronged snap connector is used for others. This is because the orientation that certain links had to be printed at did not allow for a strong enough prong to use the three-pronged snap connector, yet the joints that did allow for it are significantly stronger than the reverse tapered press-fit joints. The press-fit joints used a small vice to achieve the necessary force. Many prongs, pins, and clips were broken during this process, necessitating the remanufacture, re-cleaning, re-de-burring, and re-tapping of many parts.

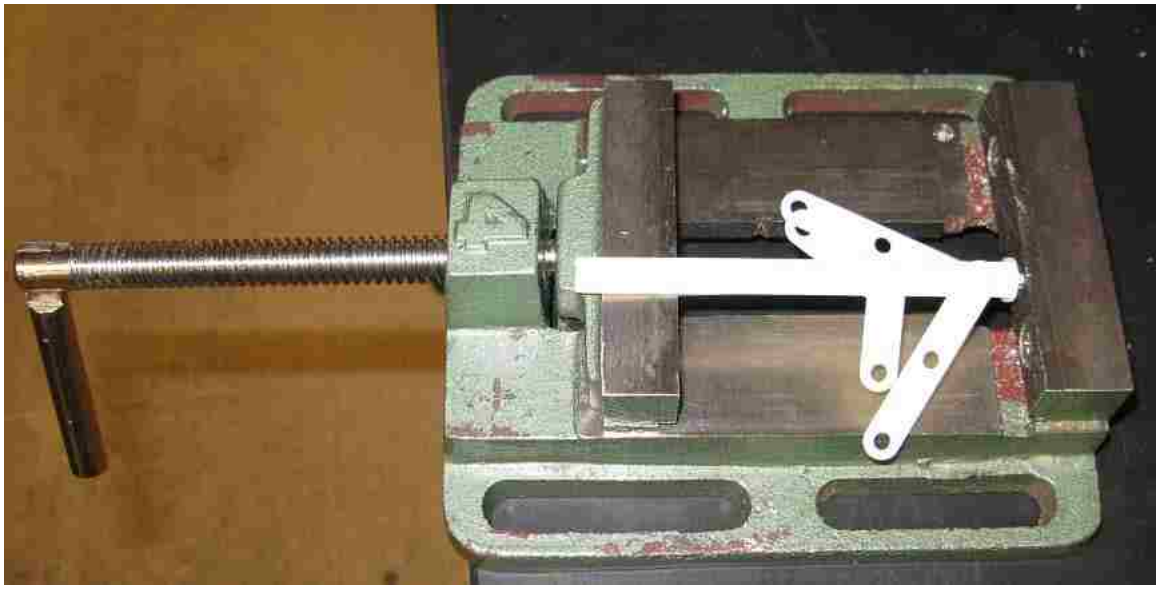

Figure 49: A clip being pressed into place with a small vise.

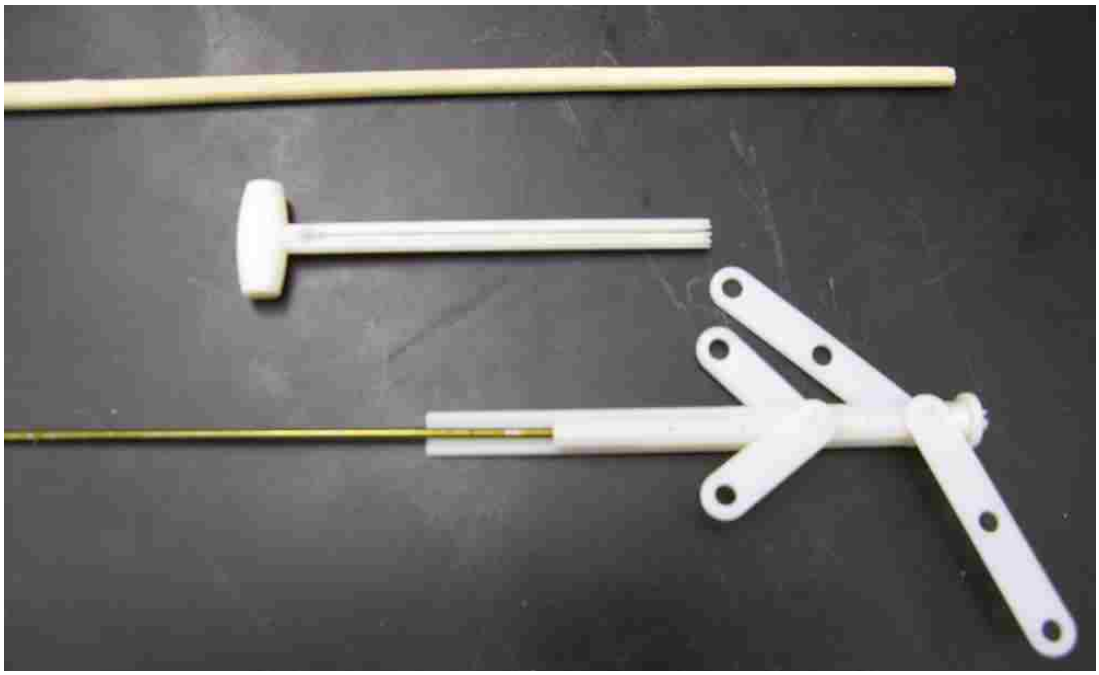

Figure 50: Tools used to remove support material include a custom made plunger (middle), a chopstick, and a wire coat hanger.

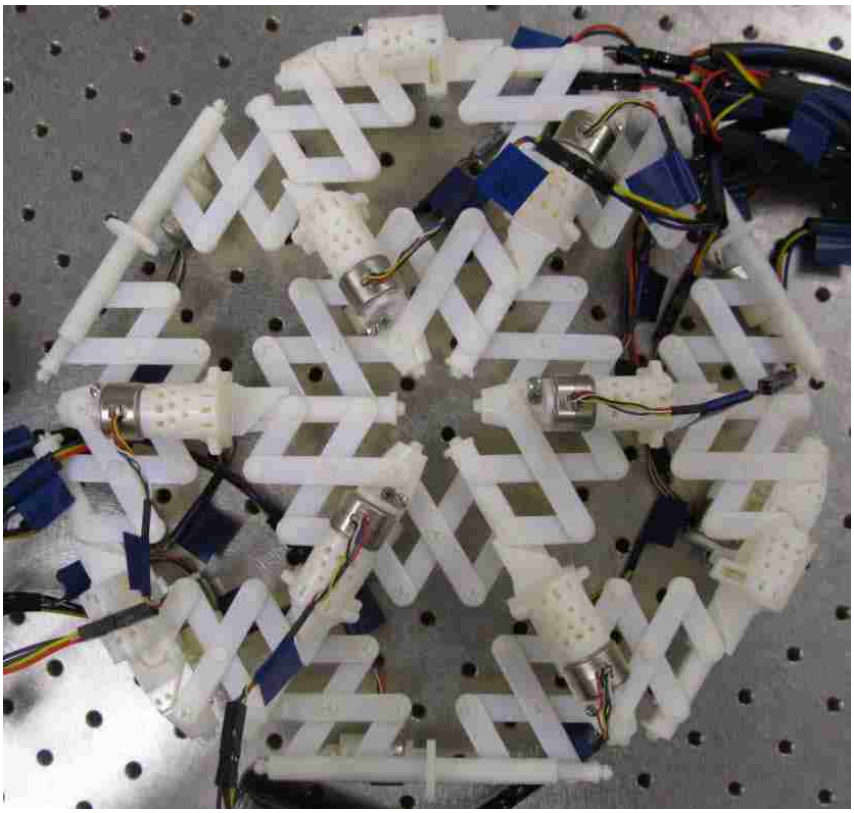

Figure 51: Stationary planar position of six module prototype.

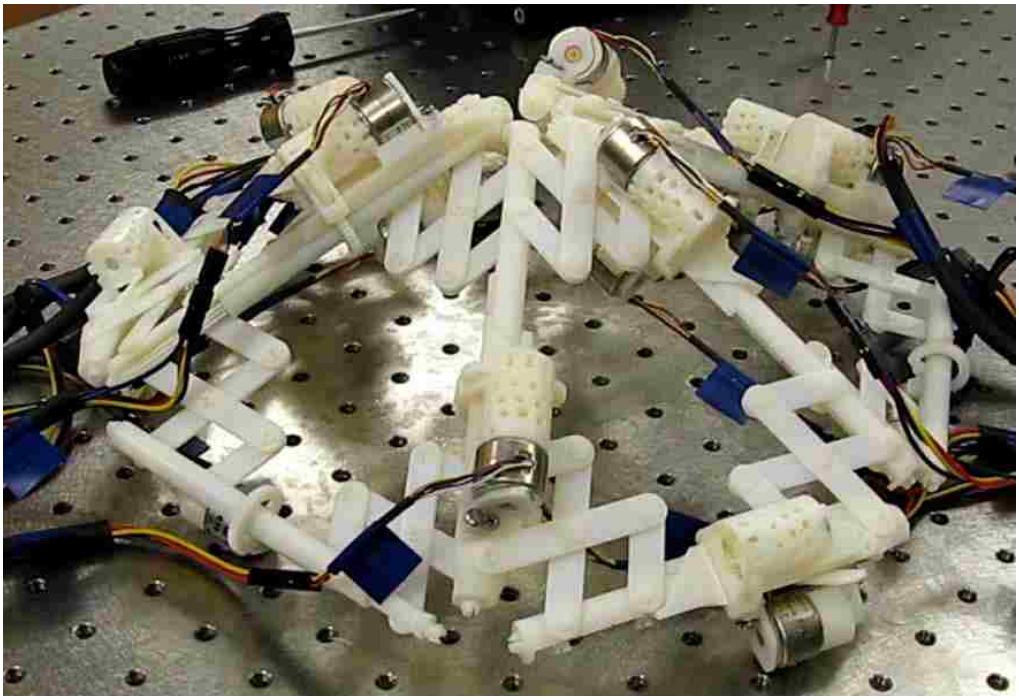

Figure 52: Moving partial sphere position of six module prototype.

CHAPTER 4

DYNAMICS AND CONTROLS

Dynamic Structural Analysis (Dynamic Joint Forces)

We now wish to find a suitable mass matrix to calculate dynamic forces. There are many methods to derive a mass matrix with various benefits and drawbacks. For our purposes, a "consistent" [8] mass matrix is chosen. Using the same methods as for the stiffness matrix, with the same definitions and shape functions, a mass matrix that has a shape consistent with the shape determined by the element's flexibility is obtained:

Local linear mass matrix $\tilde{m}_l = \frac{\rho}{420} \times$

$$\begin{bmatrix}
140 & 0 & 0 & 0 & 0 & 0 & 70 & 0 & 0 & 0 & 0 & 0 \\
0 & 156 & 0 & 0 & 0 & -22L & 0 & 54 & 0 & 0 & 0 & 13L \\
0 & 0 & 156 & 0 & -22L & 0 & 0 & 0 & 54 & 0 & 13L & 0 \\
0 & 0 & 0 & 70R^2 & 0 & 0 & 0 & 0 & 0 & 35R^2 & 0 & 0 \\
0 & 0 & -22L & 0 & 4L^2 & 0 & 0 & 0 & -13L & 0 & -3L^2 & 0 \\
0 & -22L & 0 & 0 & 0 & 4L^2 & 0 & -13L & 0 & 0 & 0 & -3L^2 \\
70 & 0 & 0 & 0 & 0 & 0 & 140 & 0 & 0 & 0 & 0 & 0 \\
0 & 54 & 0 & 0 & 0 & -13L & 0 & 156 & 0 & 0 & 0 & 22L \\
0 & 0 & 54 & 0 & -13L & 0 & 0 & 0 & 156 & 0 & 22L & 0 \\
0 & 0 & 0 & 35R^2 & 0 & 0 & 0 & 0 & 0 & 70R^2 & 0 & 0 \\
0 & 0 & 13L & 0 & -3L^2 & 0 & 0 & 0 & 22L & 0 & 4L^2 & 0 \\
0 & 13L & 0 & 0 & 0 & -3L^2 & 0 & 22L & 0 & 0 & 0 & 4L^2
\end{bmatrix}$$

where ρ is the density per unit length of each element.

Because the shape functions used do not account for the rotational inertia due to the thickness of the element, an additional rotational mass matrix is calculated in polar coordinates, with the same techniques used above, to give:

Local rotary mass matrix $\tilde{m}_r = \frac{\rho R^2}{120L} \times$

$$\begin{bmatrix}
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 36 & 0 & 0 & 0 & -3L & 0 & -36 & 0 & 0 & 0 & -3L \\
0 & 0 & 36 & 0 & -3L & 0 & 0 & 0 & -36 & 0 & -3L & 0 \\
0 & 0 & 0 & 140L^2 & 0 & 0 & 0 & 0 & 0 & 70L^2 & 0 & 0 \\
0 & 0 & -3L & 0 & 4L^2 & 0 & 0 & 0 & 3L & 0 & -L^2 & 0 \\
0 & -3L & 0 & 0 & 0 & 4L^2 & 0 & 3L & 0 & 0 & 0 & -L^2 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & -36 & 0 & 0 & 0 & 3L & 0 & 36 & 0 & 0 & 0 & 3L \\
0 & 0 & -36 & 0 & 3L & 0 & 0 & 0 & 36 & 0 & 3L & 0 \\
0 & 0 & 0 & 70L^2 & 0 & 0 & 0 & 0 & 0 & 140L^2 & 0 & 0 \\
0 & 0 & -3L & 0 & -L^2 & 0 & 0 & 0 & 3L & 0 & 4L^2 & 0 \\
0 & -3L & 0 & 0 & 0 & -L^2 & 0 & 3L & 0 & 0 & 0 & 4L^2
\end{bmatrix}$$

R is the radius of the element, assuming a circular cross section.

Then the complete local consistent mass matrix that takes into account both linear and rotary inertia is given by

$$\tilde{m} = \tilde{m}_l + \tilde{m}_r$$

Using the exact same process as with the static FEA, but with an inertial force instead of the spring force p, and an acceleration of displacement $\frac{d^2v}{dt^2}$ rather than v, the global mass matrix for a single element is derived as

$$m = T\tilde{m}T^{\mathrm{T}}.$$

This is then assembled into a full structural mass matrix in the exact same way that the full stiffness matrix was assembled.

Optimal Control

To analyze the dynamic response of the structure, a standard system of equations is used:

$$\dot{X} = AX + BU$$

and

$$z = DX.$$

The states $X = \begin{bmatrix} X1 \\ X2 \end{bmatrix}$ are chosen as the global position ($X1$), and global velocity ($X2$) of each node.

Including a viscous friction matrix

$$C \propto \|K\|$$

gives

$$A = \begin{bmatrix} 0 & I \\ -M^{-1}K & -M^{-1}C \end{bmatrix},$$

$$B = \begin{bmatrix} 0 \\ M^{-1} \end{bmatrix},$$

and

$$D = \begin{bmatrix} I & 0 \\ 0 & I \end{bmatrix},$$

where each block is a 72x72 matrix.

Another interesting feature of an actuated structure is the ability to actively damp vibrations. Using a simple optimal control algorithm [9], an input force U can

be used to stabilize the system more quickly than without using the actuators and is calculated by

$$U = -FX$$

where F is the feedback gain given by

$$F = R_2^{-1} B^T \bar{P},$$

$\bar{P}$ is the unique nonnegative-definite symmetric solution of the algebraic Riccati equation

$$0 = D^T R_3 D - PBR_2^{-1} B^T P + A^T P + PA,$$

and R_2 and R_3 are weighting matrices dictating the relative costs of the inputs and outputs respectively [10].

This gives the force at each node in global coordinates which can then be transformed to jointspace (local element coordinates) using the same transformation matrices used earlier.

There are, of course, an infinite number of loading possibilities. In order to keep the loading from becoming too complicated, yet still show an interesting response that can validate the model, a sinusoidal displacement is used. The undeformed position is a planar position that would be singular without the rotary actuators, and would yield infinite forces if modeled with non-bending truss elements. Because of the included actuators, this shape should have no axial force in the beams, but instead should have a shear force and a moment. The loading is achieved by displacing each node in the global z-direction such that the deformed position is that of a sinusoidal surface. The maximum deformation is 1.9mm. The nodes are released and the ensuing motion is analyzed. The center of mass is used

as the reference point to show free vibration. A fast Fourier transform (FFT) is performed on the output position from this analysis to determine the vibrational frequencies.

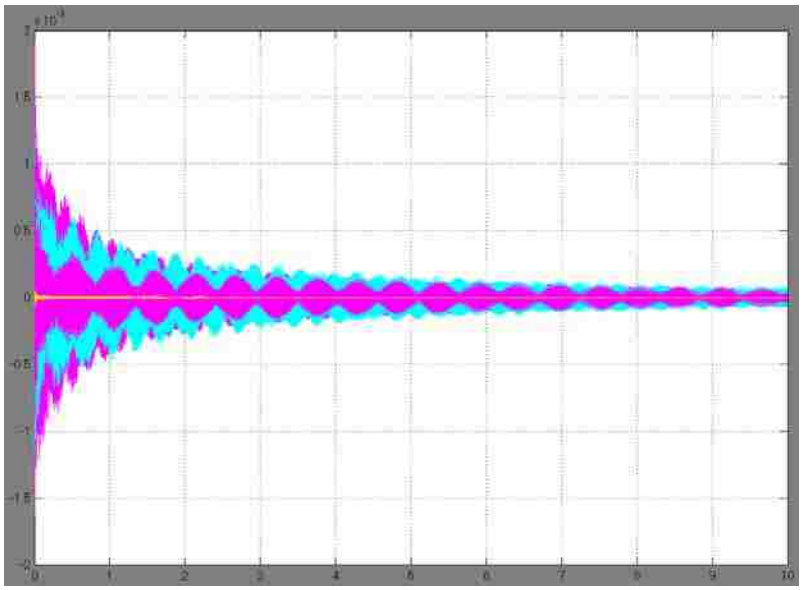

Figure 53: Qualitative graph showing the displacement in meters of each node over time in seconds without active damping.

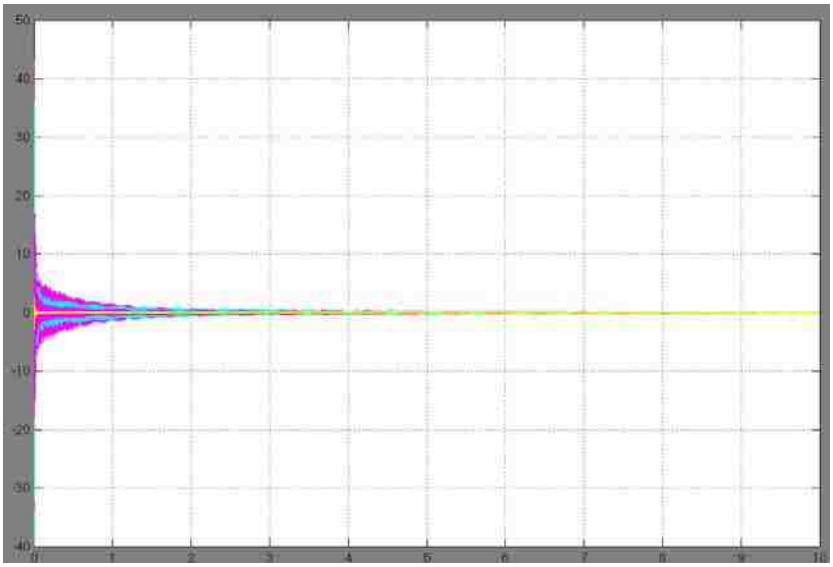

Figure 54: Velocity in meters per second without active damping.

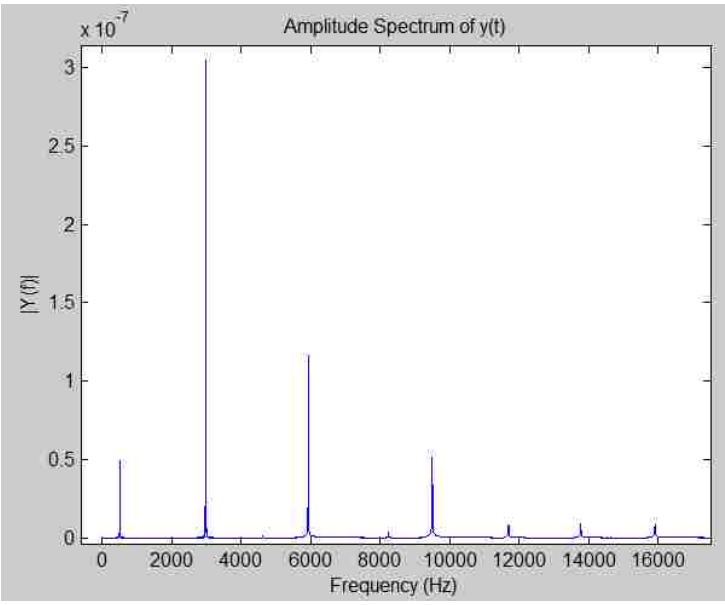

Figure 55: Spectral analysis of the vibration without active damping showing amplitude (|Y(f)|) over frequency where y(t) is the displacement of each node.

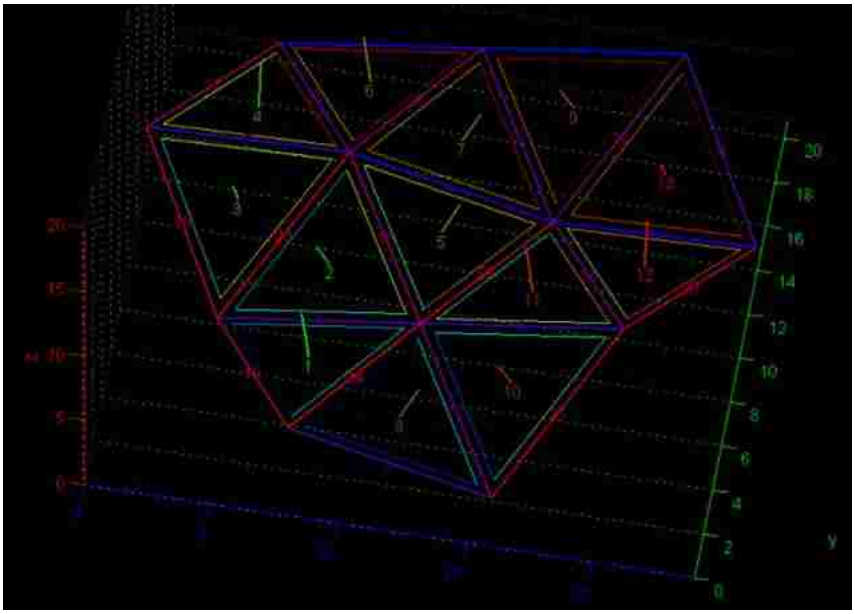

Figure 56: Sinusoidal initial position of the surface; the final position is flat on the xy plane (the vertices are only moving in the z-direction). The faces and edges are numbered, and the thickness of each link is indicated by smaller triangles. Surface normals are drawn as well to help visualize the direction that the modules are facing. The scale shown is in hundredths of inches (~0.25 mm).

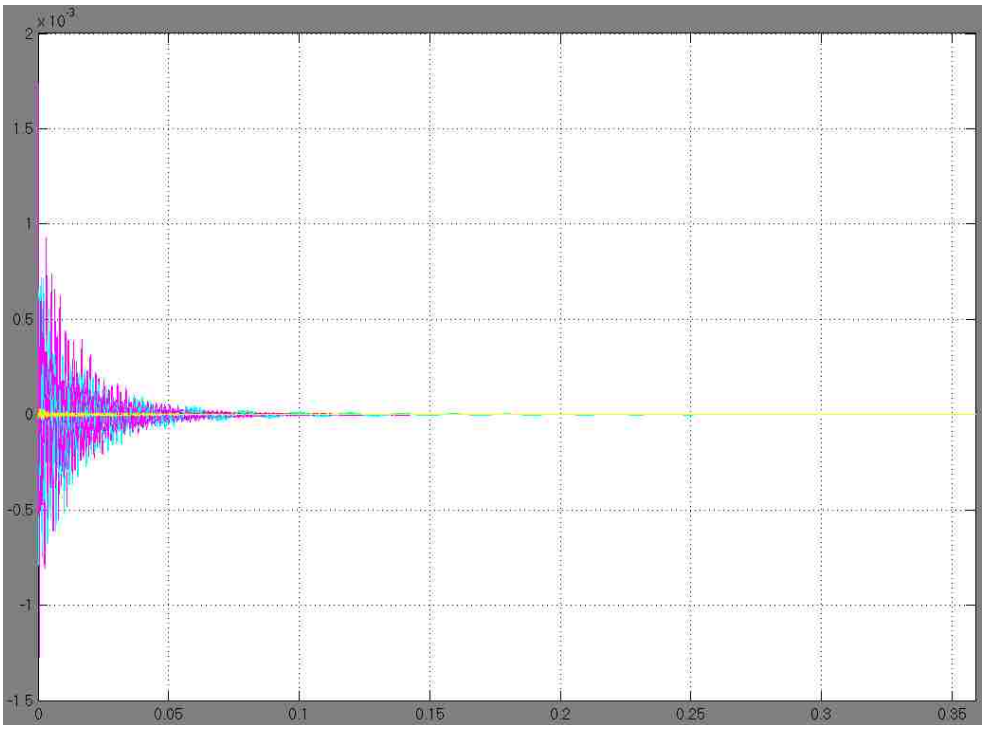

Figure 57: Qualitative graph showing the displacement in meters of each node over time with active damping. Note the drastic reduction in the time scale (shown in seconds).

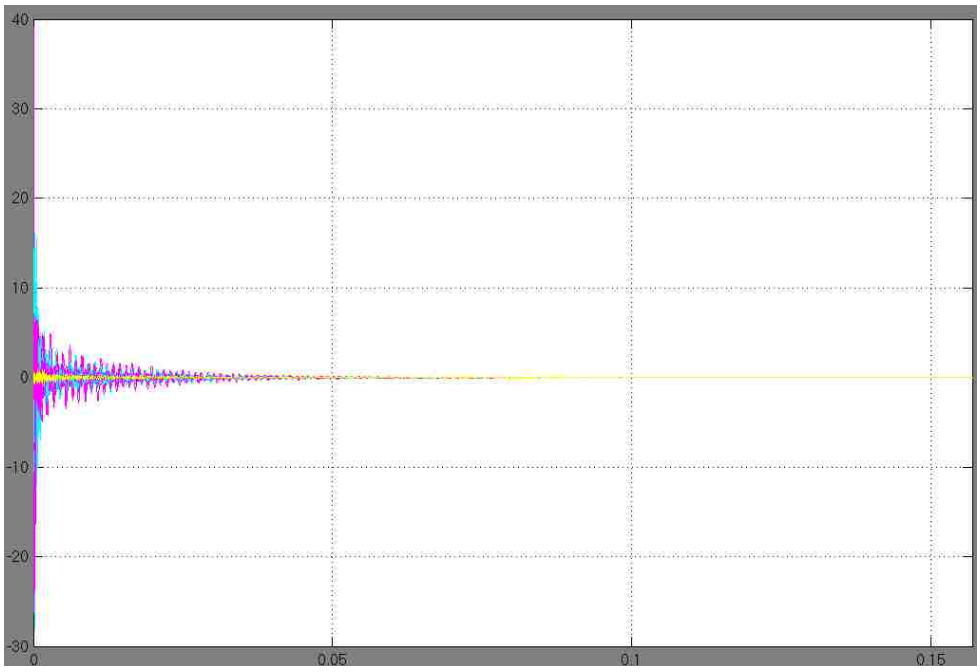

Figure 58: Velocity in meters per second of each node over time with active damping.

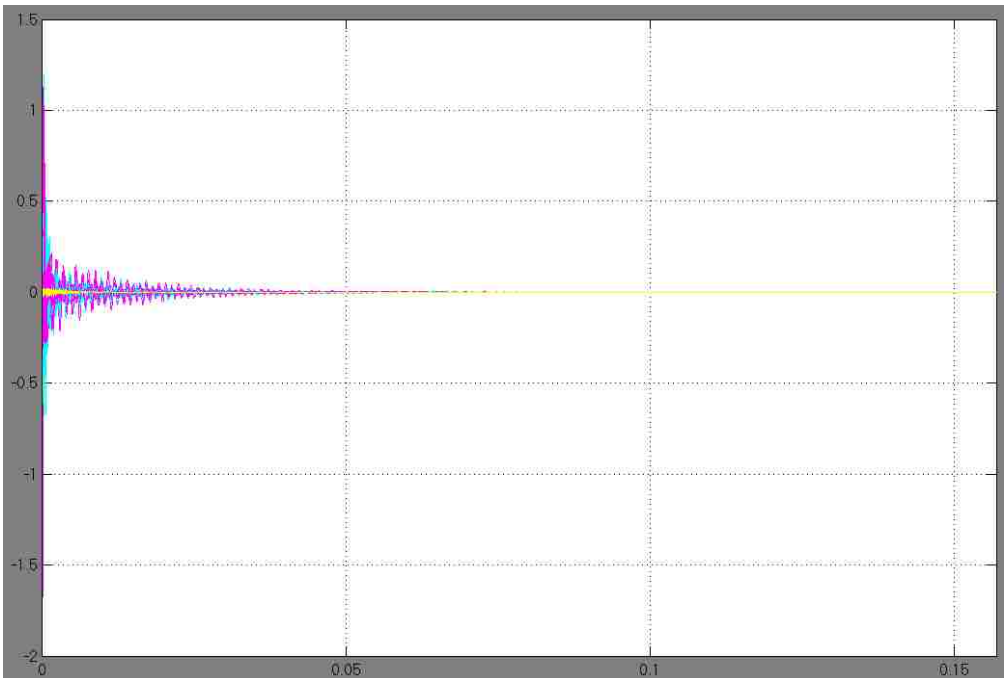

Figure 59: Active force in Newtons applied by rotary actuators.

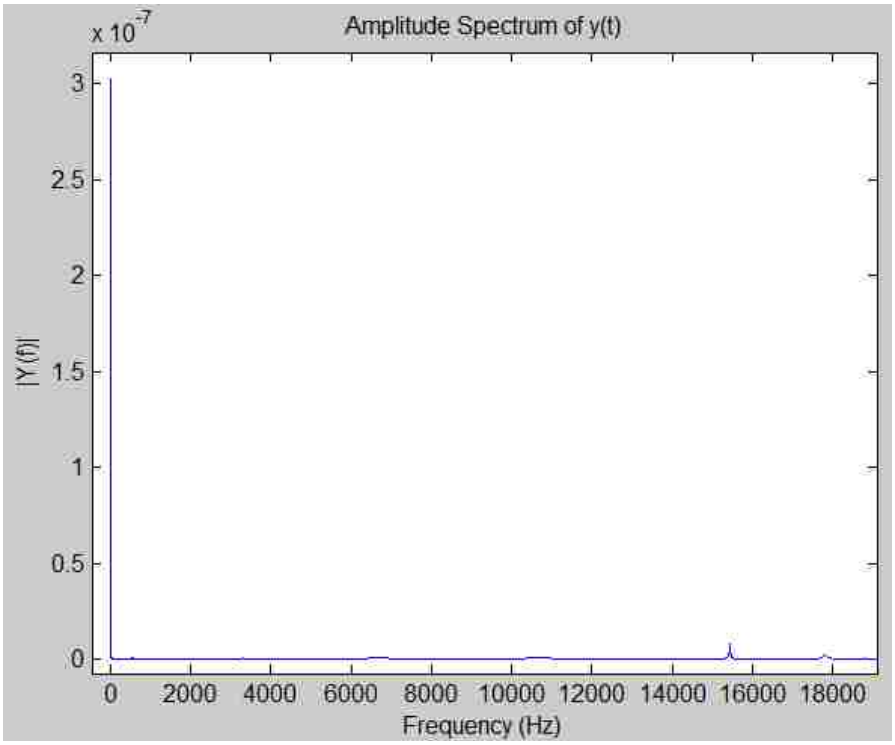

Figure 60: Spectral analysis of the vibration with active damping showing amplitude (|Y(f)|) over frequency where y(t) is the displacement of each node. The vibrations are virtually eliminated.

An interesting method of calculating an optimal force is considered for stabilization purposes. All of the states in the system used are directly a function of parameters being controlled. A continuous time algebraic Ricatti equation is solved numerically to find the forces that will minimize the time for the surface to go between two positions. In this model, each link is assumed to be rigid and friction is ignored. A standard system of equations is used:

$$\dot{X} = AX + BU$$

and

$$z = DX.$$

The states

$$X = \begin{bmatrix} X1 \\ X2 \end{bmatrix}$$

are chosen as the lengths of each link ($X1$), and the velocity of one end of a link relative to the other end ($X2$). Thirteen modules are used, giving a total of 24 links. To keep the size of the matrices low, and to reduce the tedium of printing and scrolling through these matrices, only nodal motion in the z-direction has been included. Motion in the x- and y- directions can be calculated in the exact same way.

Using the states and system of equations shown above and a lumped mass matrix gives

$$A = \begin{bmatrix} 0 & I \\ 0 & 0 \end{bmatrix},$$

$$B = \begin{bmatrix} 0 \\ \frac{1}{m} I \end{bmatrix},$$

and

$$D = \begin{bmatrix} I & 0 \\ 0 & I \end{bmatrix},$$

where each block is a 24x24 matrix and m is the mass of a link (a somewhat arbitrary value of 20 grams is used). The input force U is calculated as $-FX$ where F is the feedback gain given by $F = R_2^{-1}B^T\bar{P}$, where $\bar{P}$ is the unique nonnegative-definite symmetric solution of the algebraic Riccati equation ($0 = D^T R_3 D - PBR_2^{-1}B^T P + A^T P + PA$).

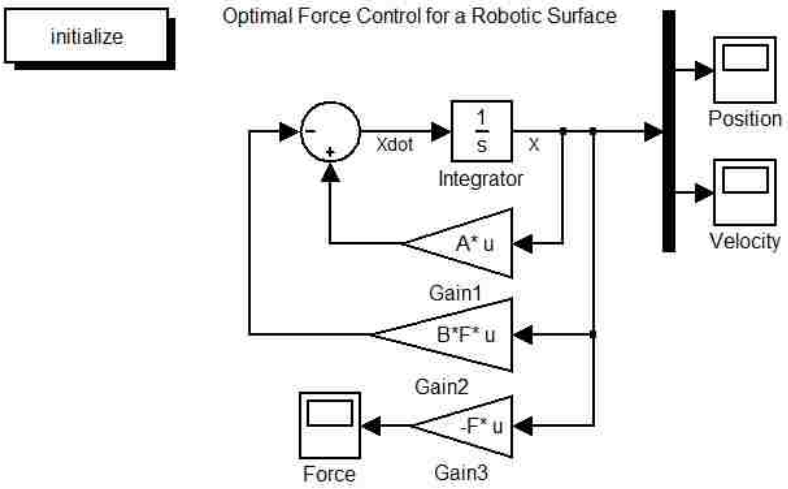

Figure 61: Block diagram from simulink model.

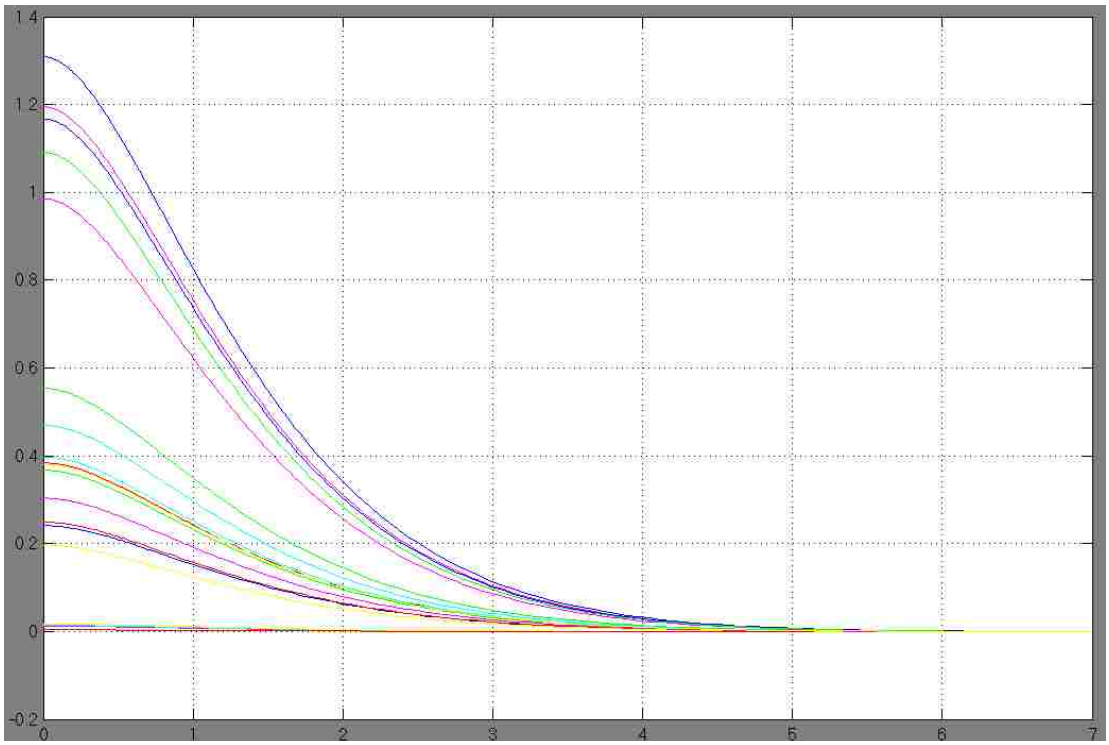

Figure 62: Vertical axis - length of links in inches, horizontal axis - time in seconds.

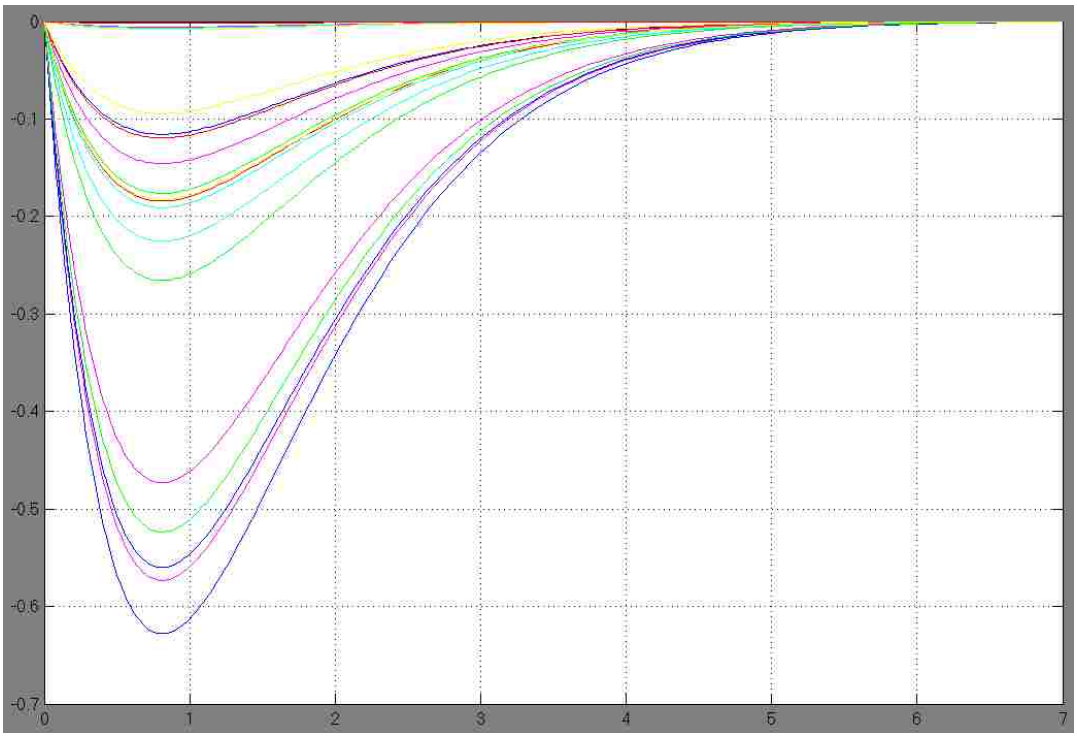

Figure 63: Vertical axis - velocity in inches per second, horizontal axis - time in seconds.

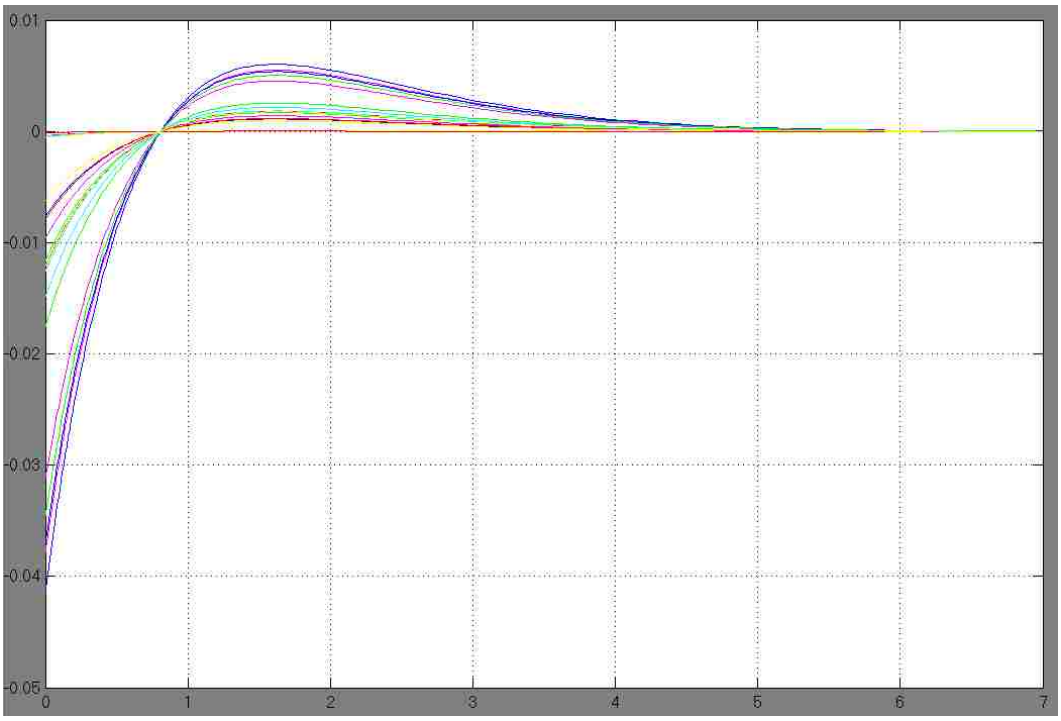

Figure 64: Vertical axis - Force in Newtons divided by 0.0254, horizontal axis - time in seconds.

The weighting matrices, R_2 and R_3 in the equations above, were decided arbitrarily as $1000 \cdot I$ and I, respectively – where I is the identity matrix. The high cost for R_2 corresponding to the force input was chosen for two main reasons. The first is that the actuators being used are quite small and since the acceleration is not directly controlled and the jerk and snap are not controlled at all the cost of higher forces is quite high. The more important second reason is that it is much easier to distinguish the different link variables on the graphs using smaller forces as it makes the relative difference between the links higher.

A state estimator was also designed by using output feedback. In this system, the output is the same as the states, but this same estimator can be used in any standard system. A stochastic disturbance is introduced and stabilized as well. The new system of equations is

$$\dot{X} = AX + BU + w_1$$

and

$$z = DX + w_2,$$

where w_1 and w_2 are white noise characterized by an intensity of V_1 and V_2 respectively. This gives the system of estimated states

$$\dot{\hat{X}} = A\hat{X} + BU + K\left(z - D\hat{X}\right)$$

where the observer gain matrix

$$K = \bar{Q}DV_2^{-1}$$

and $\bar{Q}$ is the solution to an additional continuous time algebraic Riccati equation

$$0 = V_1 - QD^TV_2^{-1}DP + AQ + QA^T.$$

As before, the same control law is used for the input U.

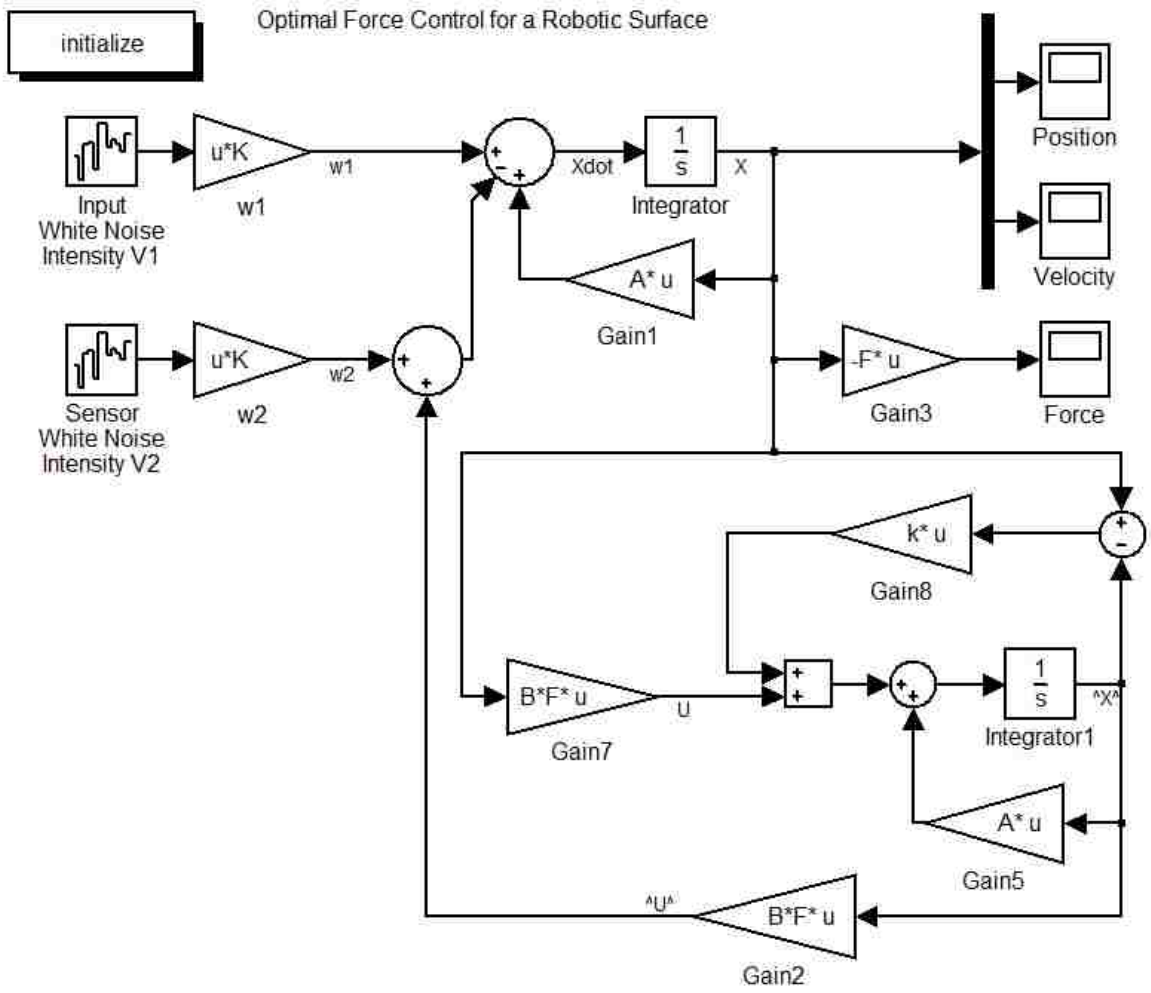

Figure 65: Block diagram of plant (top) with state estimator (bottom) and stochastic disturbance from Simulink®.

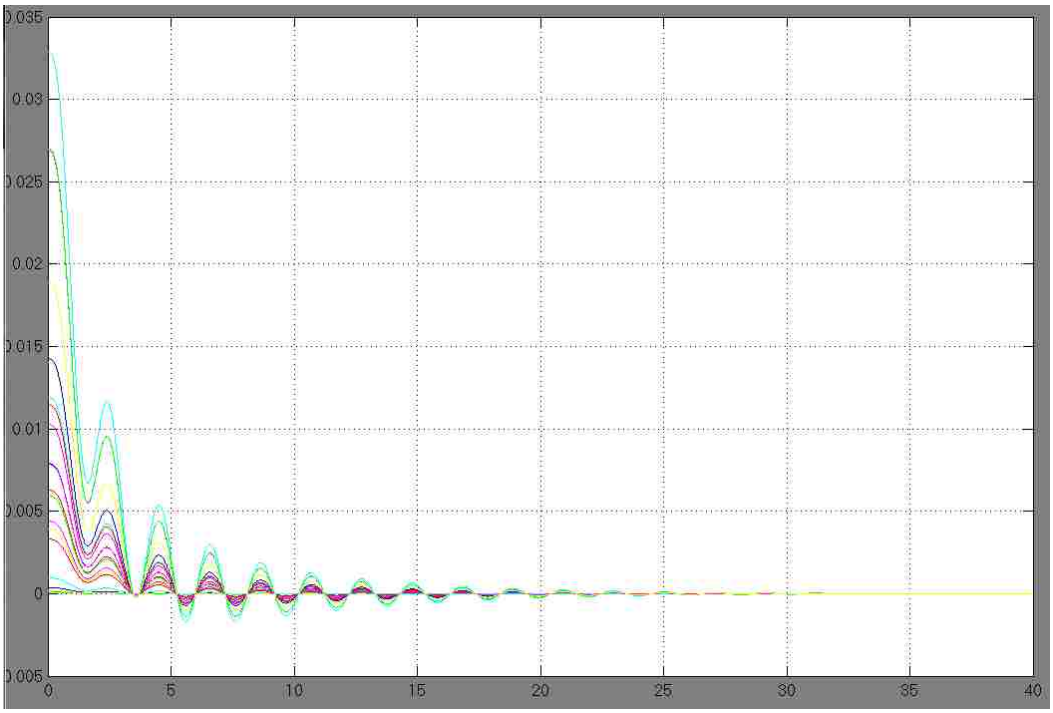

Figure 66: State estimator response with no disturbance. Vertical axis - length of links in inches, horizontal axis - time in seconds.

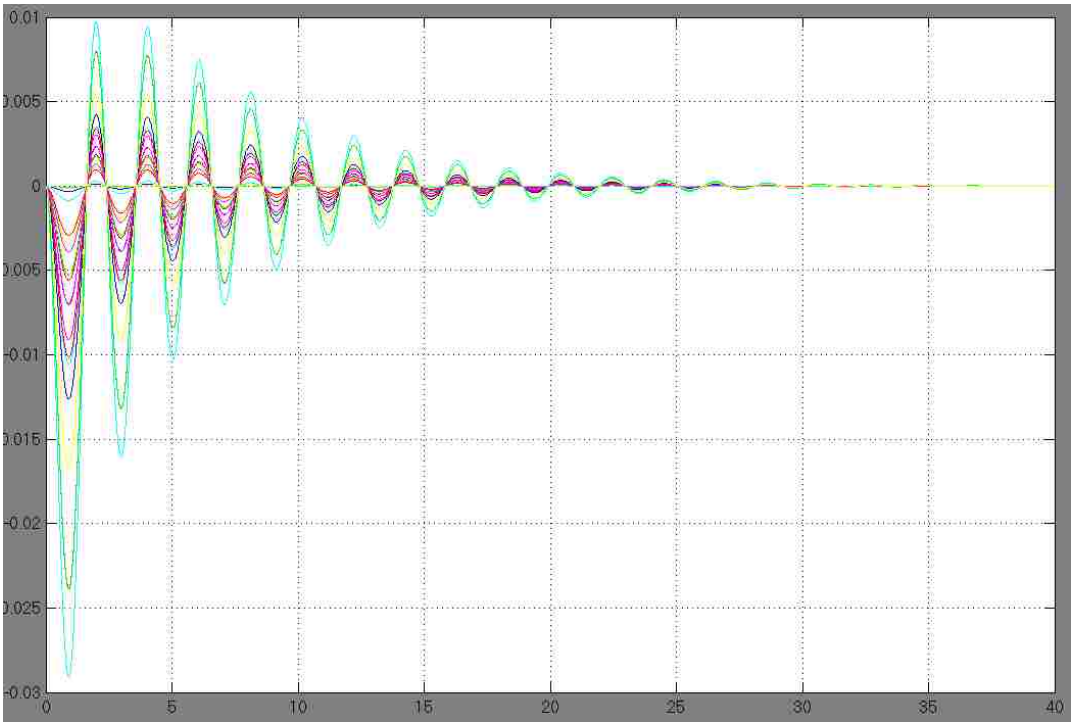

Figure 67: State estimator response with no disturbance. Vertical axis - velocity of links in in/s, horizontal axis - time in seconds.

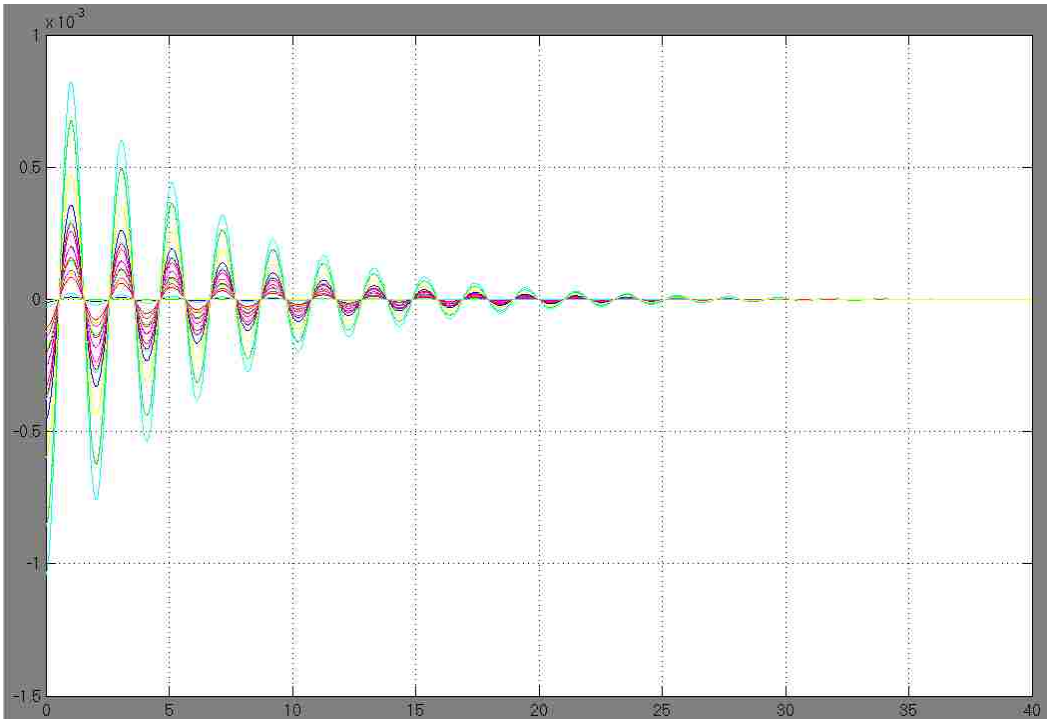

Figure 68: State estimator response with no disturbance. Vertical axis - force of links in N/0.0254, horizontal axis - time in seconds.

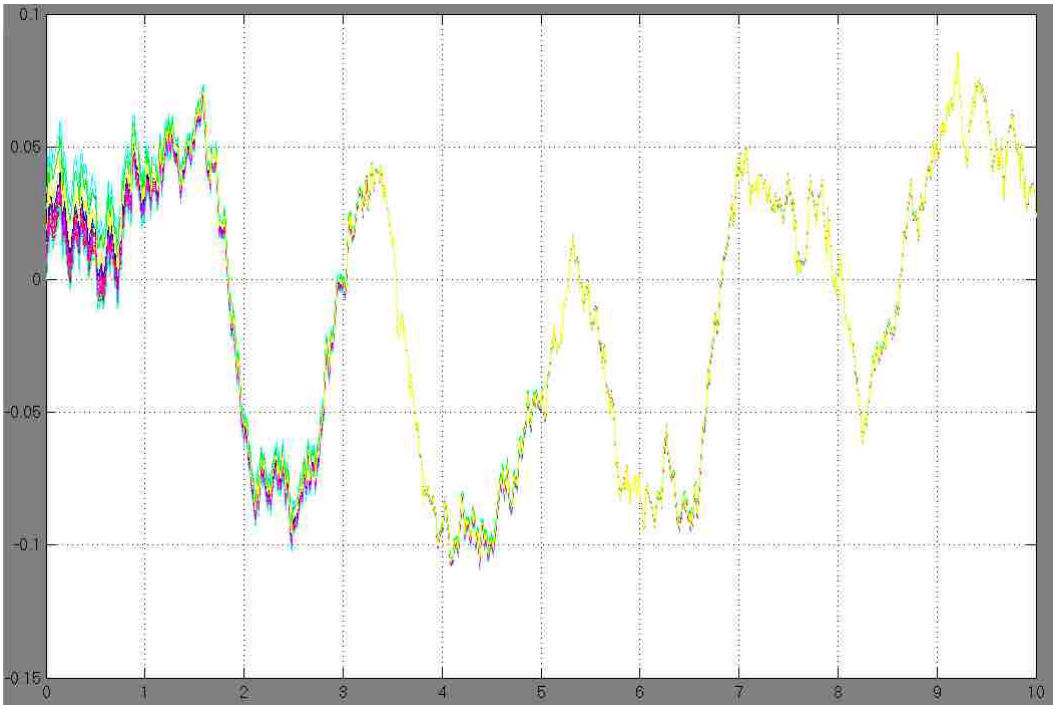

Figure 69: State estimator response with white noise disturbance acting on input and feedback sensors. Vertical axis - length of links in inches, horizontal axis - time in seconds.

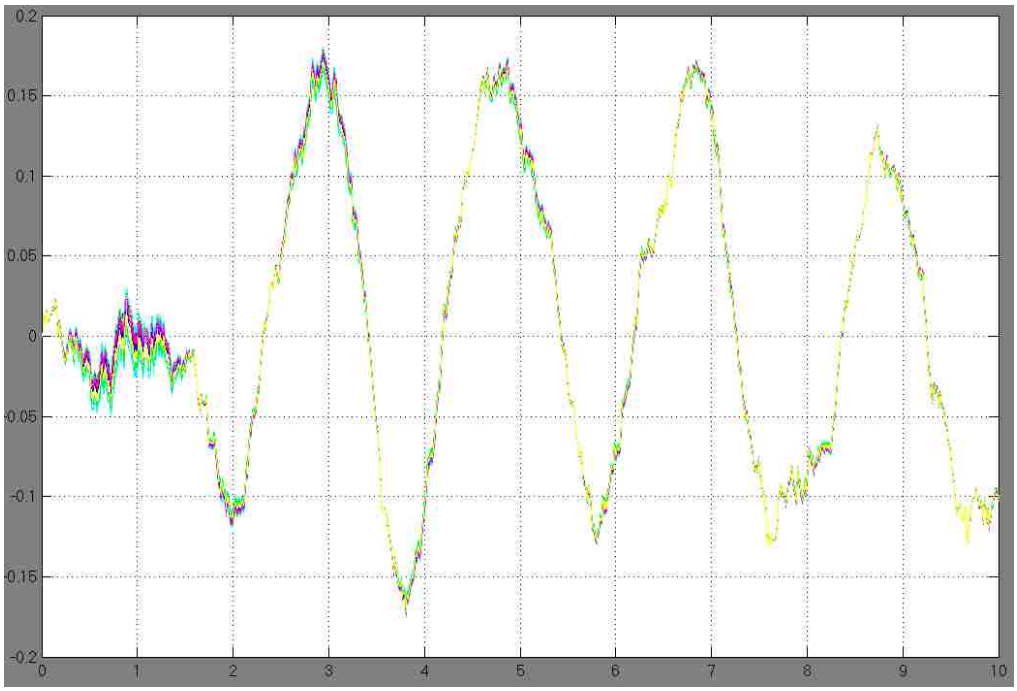

Figure 70: State estimator response with white noise disturbance acting on input and feedback sensors. Vertical axis - change of link lengths in inches per second, horizontal axis - time in seconds.

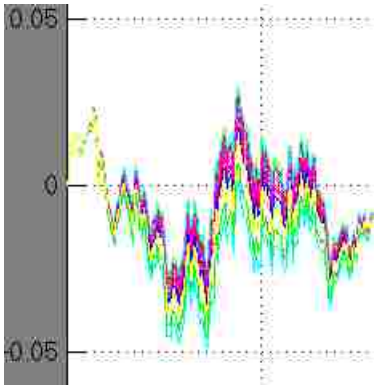

Figure 71: Close-up of the first 1.5 seconds of Figure 70.

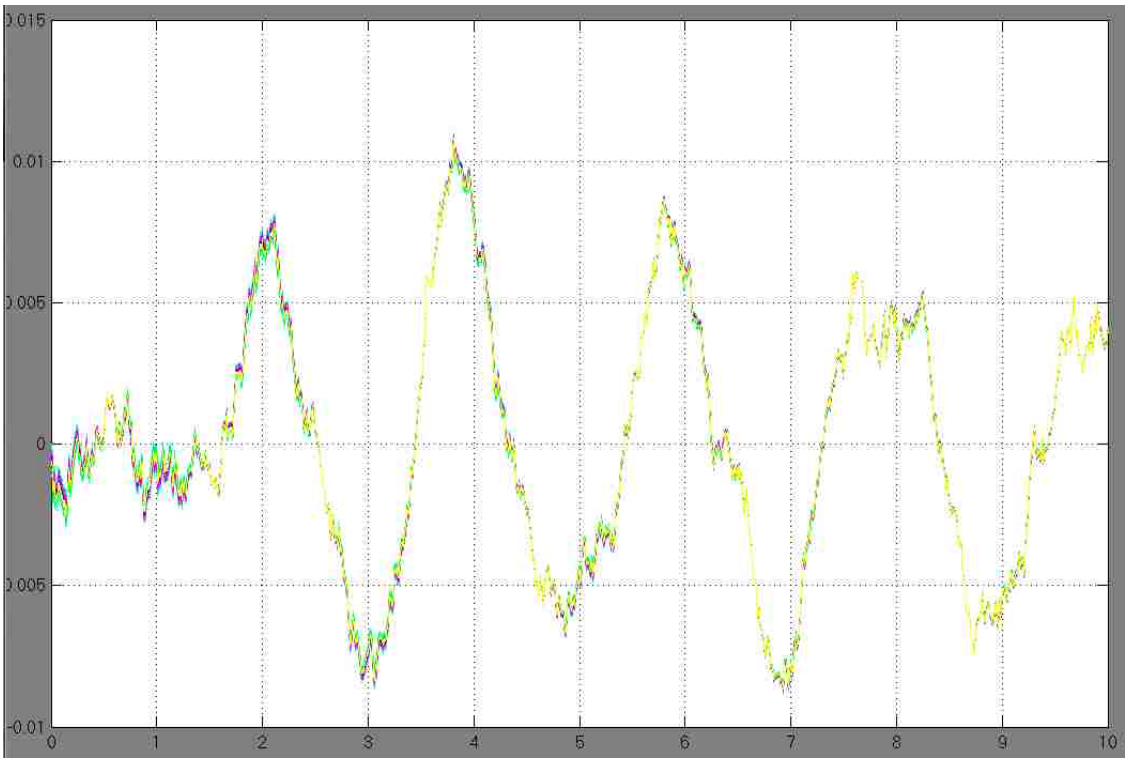

Figure 72: State estimator response with white noise disturbance acting on input and feedback sensors. Vertical axis - force of links in Newtons divided by 0.0254, horizontal axis - time in seconds.

Shape Control – Deciding on Control Point Location

Having to control each vertex is much more complicated then controlling a single end effector. Each vertex has to have a path, and that path must stay close enough to (and far enough from) adjacent vertices so that the joint range is not exceeded.

Fortunately, a tremendous amount of work [11][12] has already been done for computer graphics [13] and finite element analysis [14] that can translate almost directly to control the vertices of a robotic surface. Once a reference surface has been chosen, sampling and tessellating [15] the surface into triangles gives the

desired vertex control points for the robot, taking care to sample the surface so that the desired number of edges meets at each vertex [16]. Depending on the number of modules being used, the number of required sample points changes (the number of sample points must equal the number of vertices on the robot) and the length of each edge must be scaled to fit the surface to the robot. Using this control scheme, it is a simple matter to calculate the lengths of the edges, the angles between edges, and the angles between faces, which correspond directly with the joint variables of the robot.

Maximum forward and inverse dynamics parameters are chosen based on physical constraints, and multiple successive surfaces are chosen at specific times such that the robot does not experience motion that exceeds its actuators' capabilities, the ranges of the joints, or the yield strengths of the links. There are an infinite number of possibilities for the joint paths between two shapes, so a system of linear gain scheduling [17][18] is used (i.e., the shapes are chosen sufficiently close to each other that each vertex can follow an arbitrary desired path between the two positions in global space *and* each joint can move in a linear fashion in joint space without violating any of the constraints). This provides a linear system to be solved for each intermediate shape and eliminates the need to formulate and solve the highly implicit nonlinear differential equations that would otherwise be required to control such a system. Gain scheduling by nonlinear approximations may yield better results as there are fewer required intermediate shapes, and is a topic for future research.

The explicit control of each vertex is accomplished with equation driven motion, but in order to mimic shapes and motion with unknown equations, various methods of sampling the surface of a desired shape are explored.

Many "meshing" algorithms have been developed to represent arbitrary three-dimensional objects as triangles for computer graphics. These have been repurposed for many other applications including FEA software. The most appropriate of these to our purposes have been selected for further study.

A modified Voronoi-Delaunay shape sampling method can be used with this mechanism to approximate any arbitrary shape within joint limits. A Voronoi-Delaunay meshing scheme is named as such because it uses Delaunay Triangulation [19] on top of a Voronoi Diagram.

To make a Voronoi Diagram (sometimes referred to as Dirichlet Tessellation – Dirichlet actually came up with the idea first, though he didn't pursue the idea far enough to actually draw a diagram [20]), a defined element size first forms a 3-D grid (with spaces between points equal to the defined element size). Any point that makes up the object's point cloud is assigned to the closest point on the grid (by proximity that is). Any grid point with two or more such assignments is kept in the diagram – the rest are discarded.

In Delaunay Triangulation, triangles are formed by making edges from any two points that pass an "empty circle" test. This test calls for all circles that intersect two given test points to be drawn until one is found with no other points inside of it (the circle is "empty"). The circles typically start with a diameter equal to the distance between the two Voronoi points and vary as required in sweeping to the

left and right in search of meeting the empty circle conditions described above. Forming these triangles from the Voronoi Diagram comprises the Voronoi-Delaunay mesh.

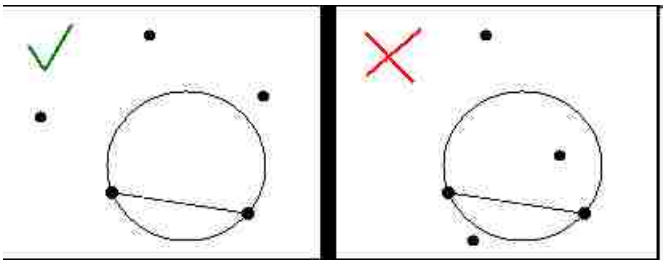

Figure 73: Graphical representation of the empty circle test. [21]

Another potential method is geodesic calculation. Given specified distances or distance ranges and the outline of the desired shape (continuously as an equation, or discretely as a point cloud), a geodesic algorithm can fill in the shape with straight-line chords with lengths equaling the specified distances or distance ranges, using any type of polygon. Restricting such an algorithm to triangles, and then even further restricting it to six chords from a single point gives appropriate control point locations for our mechanism. A combination of three start points, and/or three initial guesses can "slide" the sample surface around on the actual surface, allowing a shape to be easily mobile by using the robotic surface like tank-treads flowing around the sampled shape.

Each of the two methods mentioned so far needs to be modified to fit the number and configuration of modules in an actual robotic surface. This is not always trivial. Another more limited method is surface function mapping. A surface is "mapped" by choosing an orientation and finding the height value at specified length

and width values. Then the length is found at specific intervals of height and width, and width likewise is found and specific intervals of height and length. These point sets are compared and combined to obtain the most accurate map. If the number of points and the specified distances correspond with the robotic surface, no further calculations are required. It is simple to implement, but for complicated shapes, this is usually the least accurate of the three methods considered here.

Trajectory Planning – Deciding on a Desired Path

In the pursuit of a trajectory planner, I have implemented several control schemes. For enhanced clarity, these are all explained for a single control vertex of the mechanism travelling along a specified two-dimensional path. Though this vertex can be changed to any point on the mechanism, or any number of points on the mechanism, it may still be thought of as an end effector. The initial methods followed the reference path almost exactly, having an error tolerance of less than the lowest error tolerance of the mechanism. The first method utilized a constant acceleration profile. This of course resulted in infinite jerk and discontinuous acceleration.

Jerk is defined as the third time derivative of position. Because force equals mass times acceleration (F=ma), jerk (as the change in acceleration) is linearly proportional to the change in force. In real practical purposes, an instantaneously infinite change in force is not possible, so a very high jerk results instead. This requires a very fast change in applied force - commonly called an impulse force, as in a collision. All actuators have a limit in how quickly they can change force, but

because the proportionality constant relating the jerk and change in force is the mass being moved, the maximum allowable jerk should vary accordingly.

To avoid infinite jerk, continuous acceleration (as opposed to constant acceleration) is implemented. There are several methods for this implementation, but the one that was chosen for this project was a sinusoidal function [21]. The acceleration must go from zero to some maximum, and back to zero again with no discontinuities.

$$a = \frac{A}{2}\left[1 - \cos\left(\frac{2\pi}{T}t\right)\right]$$

Integrating once with respect to time, the velocity is given by

$$v = \frac{A}{2}\left(\frac{\mathrm{T}}{2\pi}\right)\left[\frac{2\pi}{T}t - \sin\left(\frac{2\pi}{T}t\right)\right] + v_0$$

and integrating a second time, the displacement is then given by

$$d = \frac{A}{2}\left(\frac{\mathrm{T}}{2\pi}\right)^2\left[\frac{1}{2}\left(\frac{2\pi}{T}t\right)^2 - \left(1 - \cos\left(\frac{2\pi}{T}t\right)\right)\right] + v_0 t + d_0.$$

In these equations t is time, T is the period over which acceleration occurs, and A is the maximum acceleration of which the joint is capable. Rearranging these equations for convenience gives

$$D = \frac{(v - v_0)^2}{A},$$

and

$$T = \frac{2|v - v_0|}{A},$$

where D is the total distance over which acceleration occurs. Now with a starting velocity and position, along with a desired velocity and position, the position of the vertex can be accurately calculated for any point in time. All that remains is to

coordinate the x, y and z directions so that they work together to give the desired three dimensional motion.

To allow the end effector to follow a straight line, the ratios between each parameter (position, velocity, acceleration) of each of the directions must remain geometrically similar. The linear directions x, y, and z are used as an example; rotation is analogous but not as easy to visualize. The specific ratios can be calculated from the Pythagorean Theorem ($x^2 + y^2 + z^2 = d^2$) and knowing that all of the parameters are geometrically similar. A known position (x_1, y_1, z_1) and a desired position (x_2, y_2, z_2) are given, and the displacement of each direction is given by $(x, y, z) = (x_2, y_2, z_2)$ -(x_1, y_1, z_1). d is solved for, and the ratios (that must remain mathematically similar) are given by $\frac{x}{d}, \frac{y}{d}$ and $\frac{z}{d}$, for each joint respectively. The ratio $\frac{x}{d}$ is equal to the ratio $\frac{V_x}{V}$ and $\frac{A_x}{A}$. The other two example directions have similar equivalencies. Now to keep these ratios constant between moves, $\frac{x_1}{d_1}$ must be similar to $\frac{V_{x2}}{V_2}$, but $\frac{V_{x2}}{V_2}$ still has to be similar to $\frac{x_2}{d_2}$. The only solution for this equation is $V_{x2} = 0$. This means that if a sharp corner is desired, the velocity at that corner must be zero in all directions (even if no joints change direction).

The considered control schemes first specify a new path with no sharp corners – allowing the control vertex to avoid stopping and to have a constant velocity if so desired. Quadratic b-splines take advantage of the convex hull property and minimize the number of control points required. The term "b-spline" is derived from "basis spline" or sometimes "basic spline" and is a form of spline produced by using "basis functions." These splines are useful in that every spline function of a

given degree, smoothness, and domain partition, can be represented as a linear combination of b-splines of that same degree and smoothness, and over that same partition [22]. Also, changing one knot only changes the spline over one partition, allowing for very accurate control of the spline path.

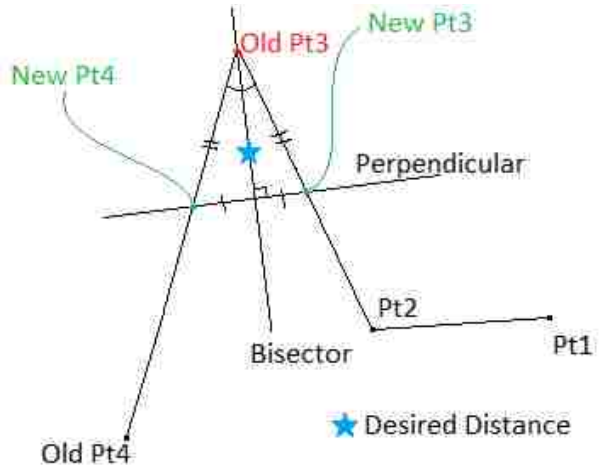

Figure 74: Example of how points are rearranged to round corners.

With a simple application of the law of cosines, a point along the acute bisector of two adjacent line segments was calculated at a predetermined distance from the intersection of these segments. Control points were then added where the bisector's perpendicular passing through this point intersected the line segments. This was accomplished numerically in MATLAB® by replacing each point other than the end points in the position vector with two points at the intersections of the bisector's perpendicular passing through the point at the desired distance from the original point and the two adjacent line segments. In this way, the corners were rounded with parabolas deviating from the original path by no more than the predetermined distance. This distance was left as a parameter to be changed by the

user, but was not allowed to be large enough to cause the bisector's perpendicular (referred to above) to intersect the original line segments past their midpoints from the direction of the point being replaced.

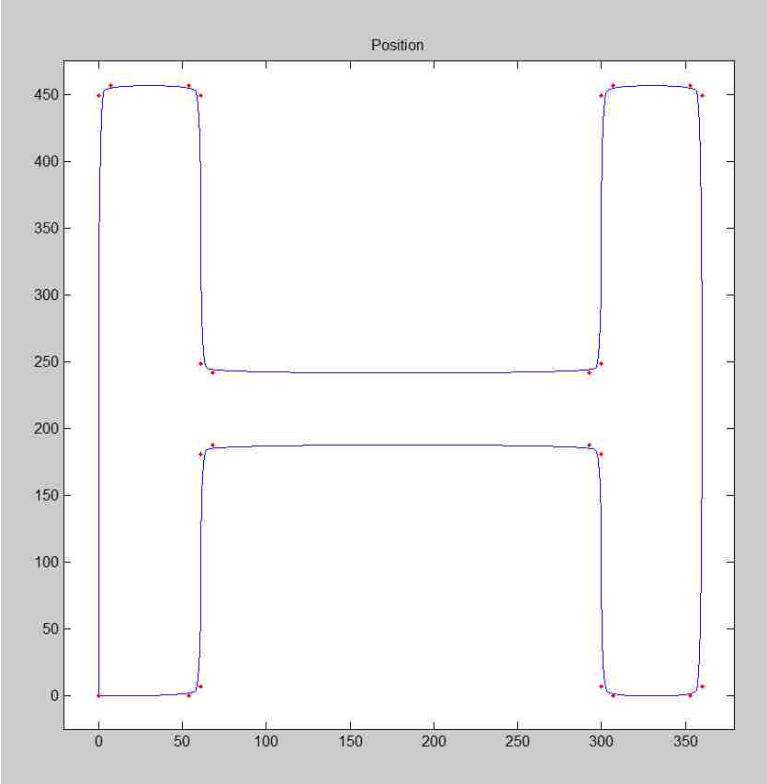

Figure 75: Example path of a quadratic b-spline in blue with the control points in red using a maximum path deviation of 5 units on the scale shown.

I next compare with interpolating cubic polynomial splines. I used two methods of adding control points since this was the simplest way to reduce path deviation. The first was very similar to the method used for cutting corners. The difference was that I multiplied the distance of the added points from the original point by three or placed them at a distance of 42% of the shortest line segment length from the original point along the respective adjacent line segments,

whichever was shorter. I also added an additional point at the intersection of the bisector and its perpendicular (since the spline now interpolates through the points). 42% was chosen because it leaves a minimum of $(2 \times (50\% - 42\%) = 16\%$ of the line length for the curvature of the parabola — preventing an entire line segment from being "blended away." The second method was simply to add control points at the midpoint of each line segment; causing the path to interpolate through the original reference path at the point of average maximum deviation. Ideally this would be based either on an error checking function or on the angles of the previous and next line segments. Maximum error calculation was considered, but was deemed to be too computationally expensive.

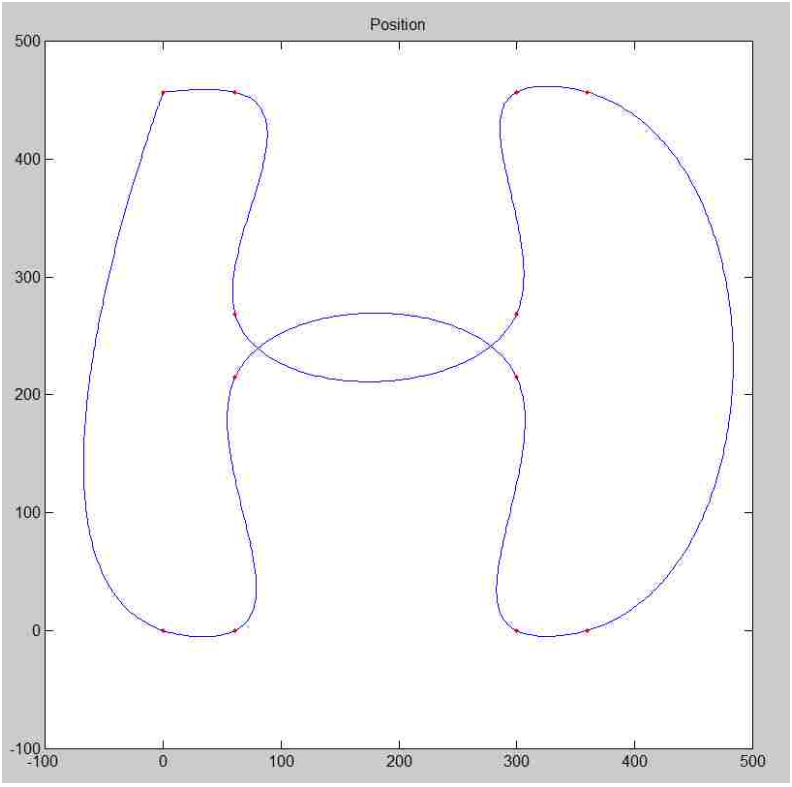

Figure 76: Cubic polynomial interpolating spline with no additional control points.

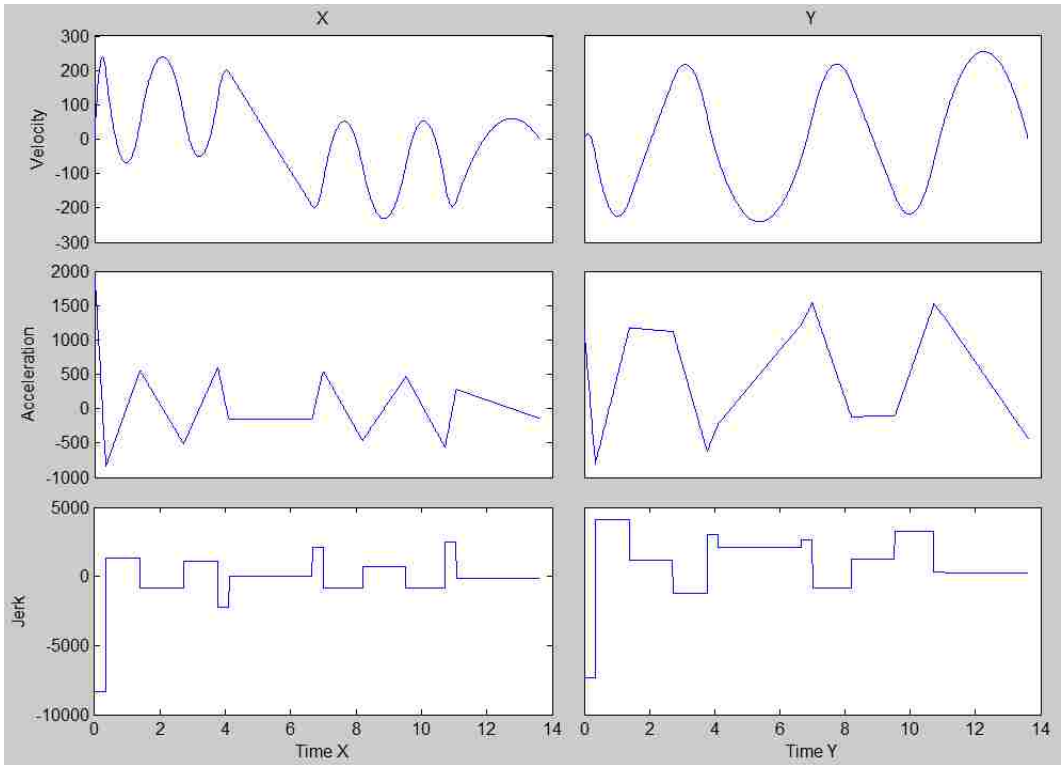

Figure 77: Velocity, acceleration, and jerk profiles for x and y directions of the example path shown in Figure 76.

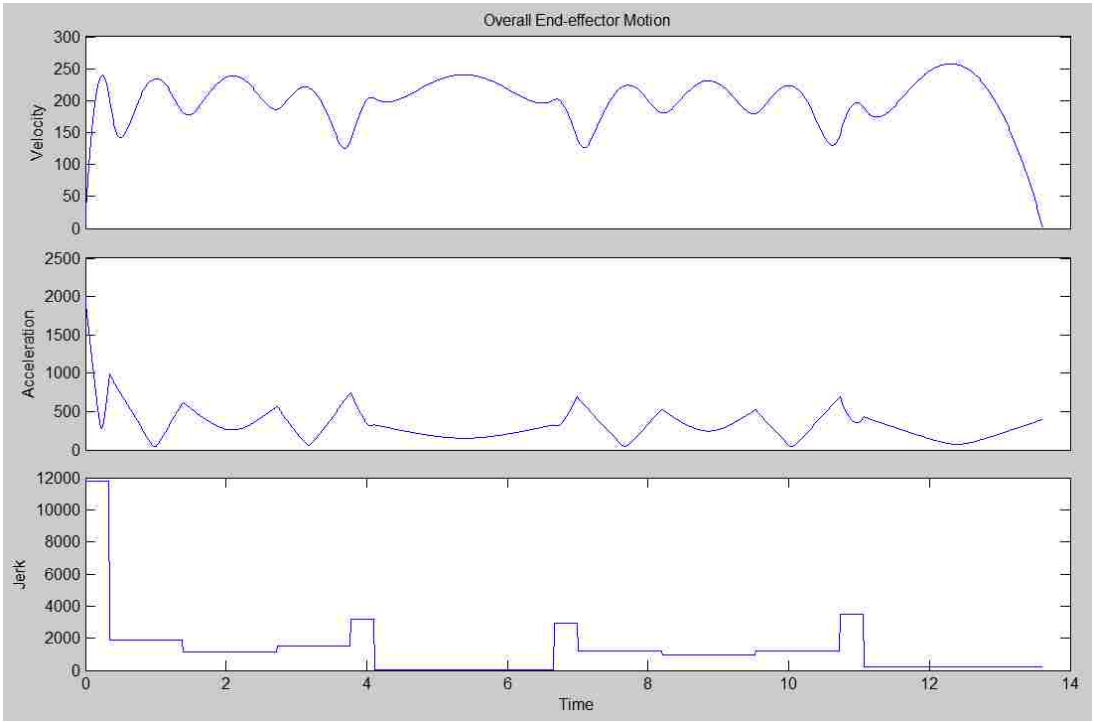

Figure 78: Overall motion profiles for the example path in Figure 76.

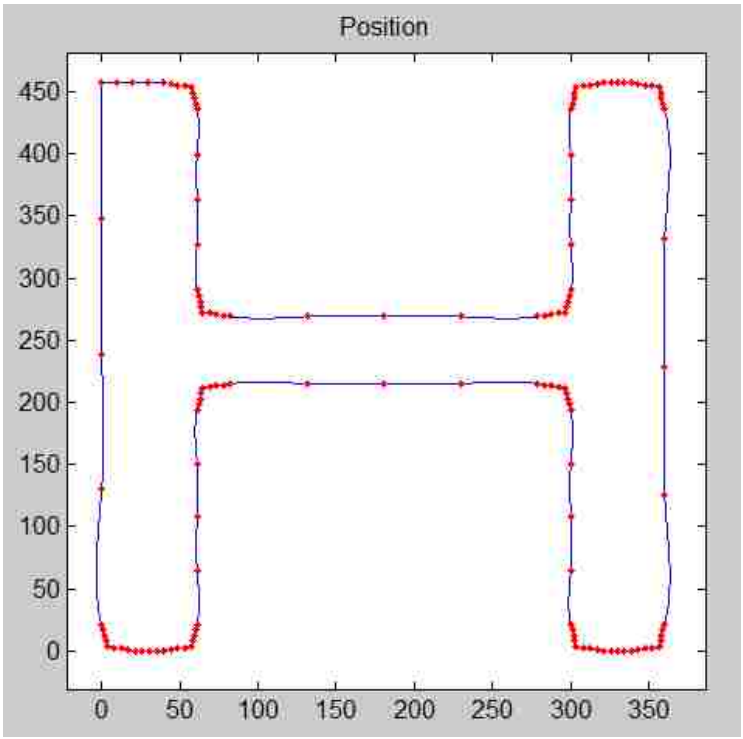

Figure 79: Example path of a cubic polynomial interpolating spline with control points added using a maximum position error of 5 near the corners and extra points added at the midpoints of each line segment two times in a row. The entire spline was then scaled to constrain the maximum velocity and maximum acceleration.

Initially a uniform parametric spline was used, but soon after a non-uniform spline was calculated using the distances between points as the weighting factor of the parameter. The entire spline was then scaled by dividing the parameter function by the smaller of the quotient of the maximum desired velocity divided by the actual maximum velocity and the square root of the quotient of the maximum desired acceleration divided by the actual acceleration. The purpose of this scaling was to cause the actuators to stay within the constraints of a maximum velocity and acceleration.

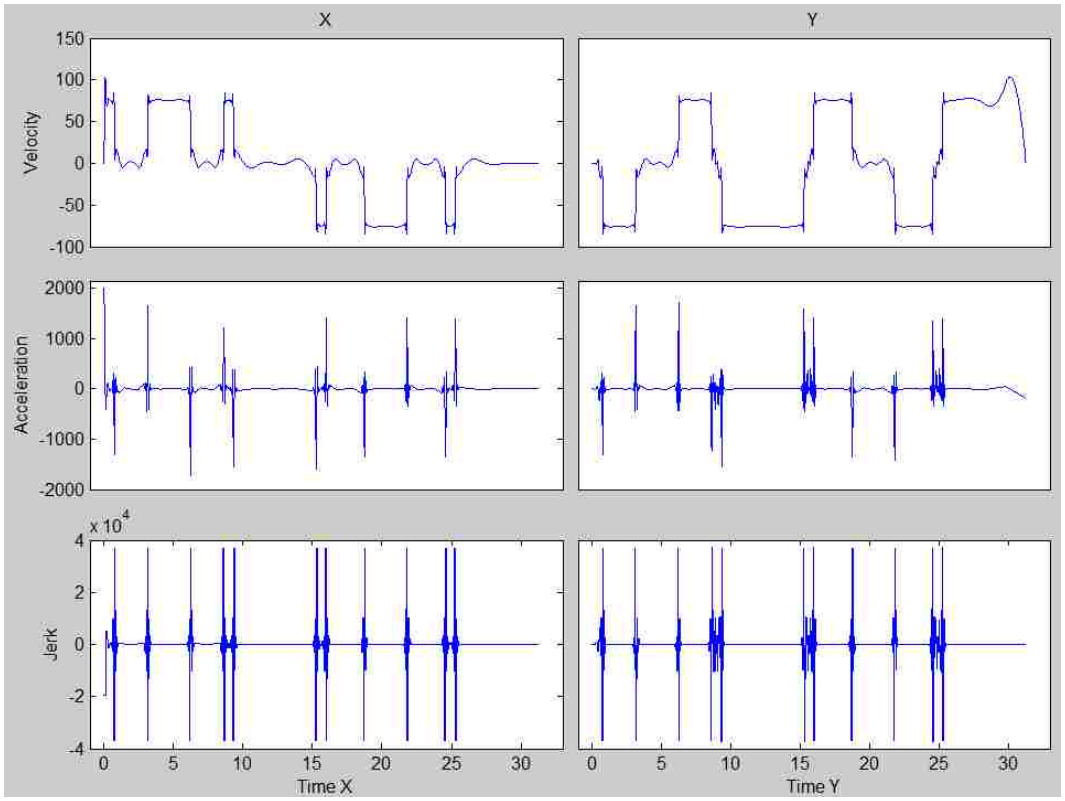

Figure 80: Velocity, acceleration, and jerk profiles for x and y directions of the example path shown in Figure 79.

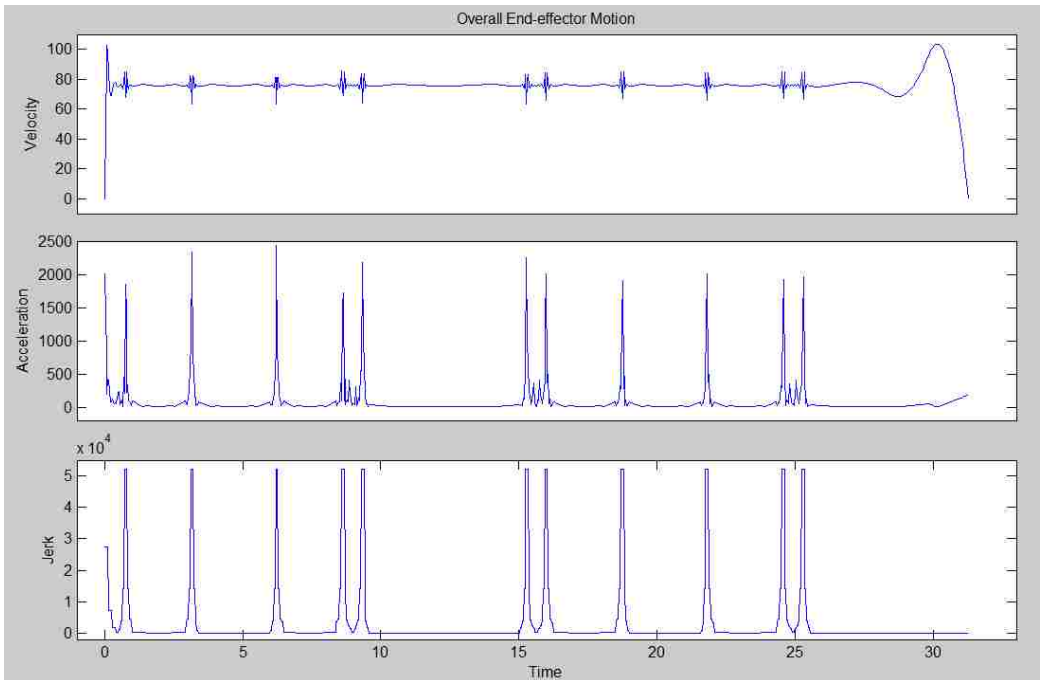

Figure 81: Overall motion profiles for the example path in Figure 79.

Finally, each line segment within the spline was rescaled by the same method as above to ensure that either the maximum velocity or the maximum acceleration was reached during each segment. This caused discontinuities, so the process was iterated until the values converged [22]. Except for a few special cases, the more line segments there are, the more benefit there will be from this method. The overall time taken to follow the path is reduced significantly for large datasets, and a straighter path is followed.

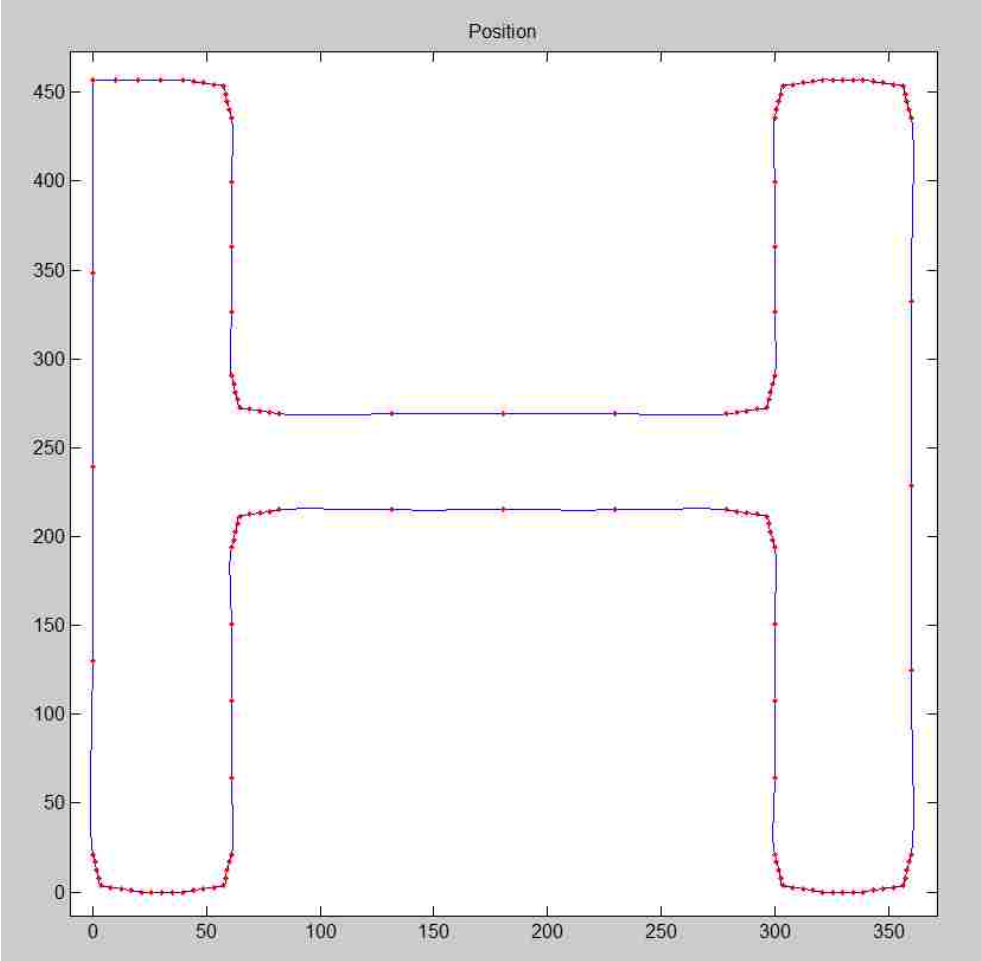

Figure 82: Same dataset as in Figure 79, but with the spline scaled at each partition.

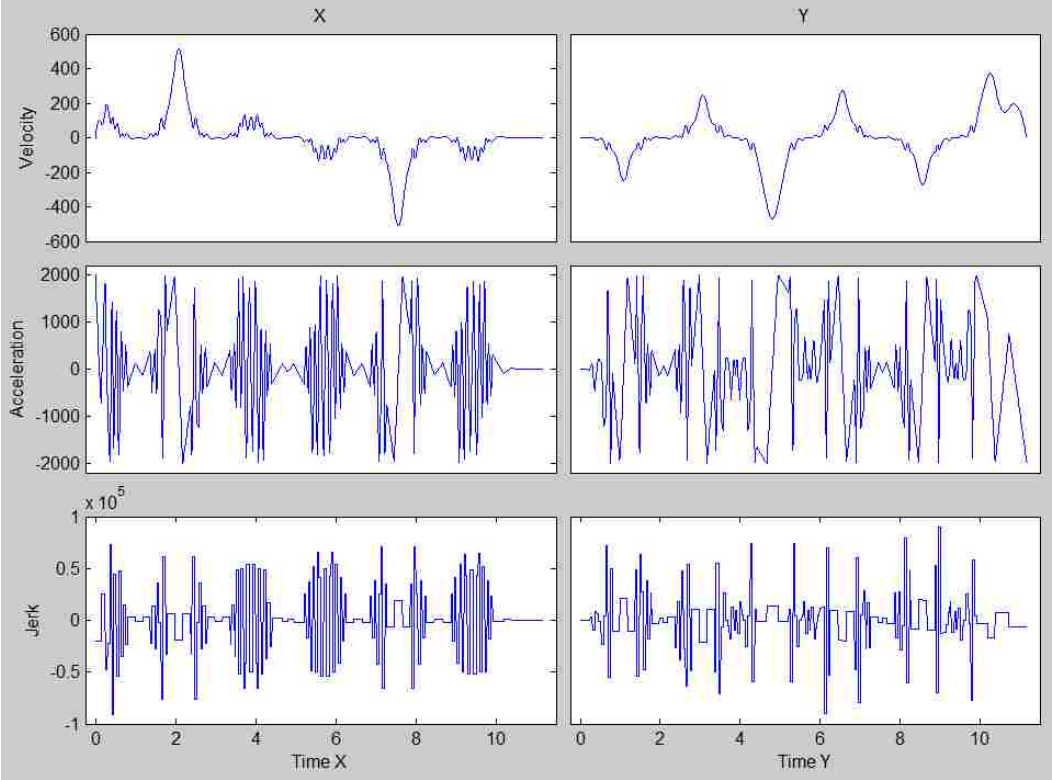

Figure 83: Velocity, acceleration, and jerk profiles for x and y directions of the example path shown in Figure 82.

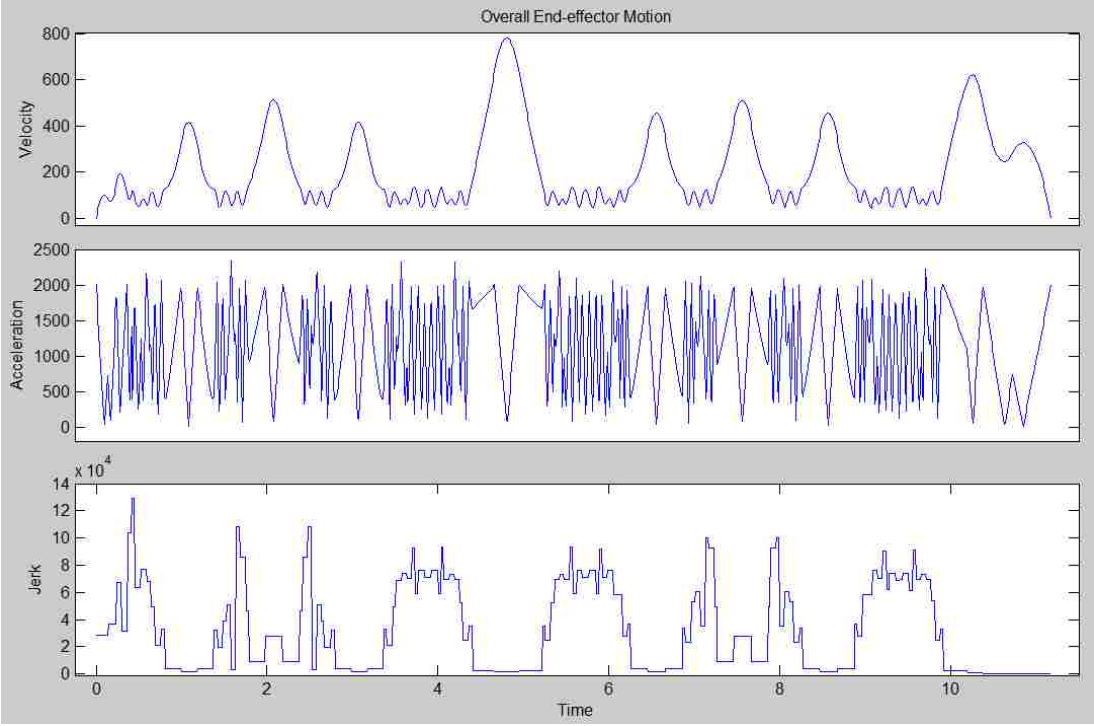

Figure 84: Overall motion profiles for the example path in Figure 82.

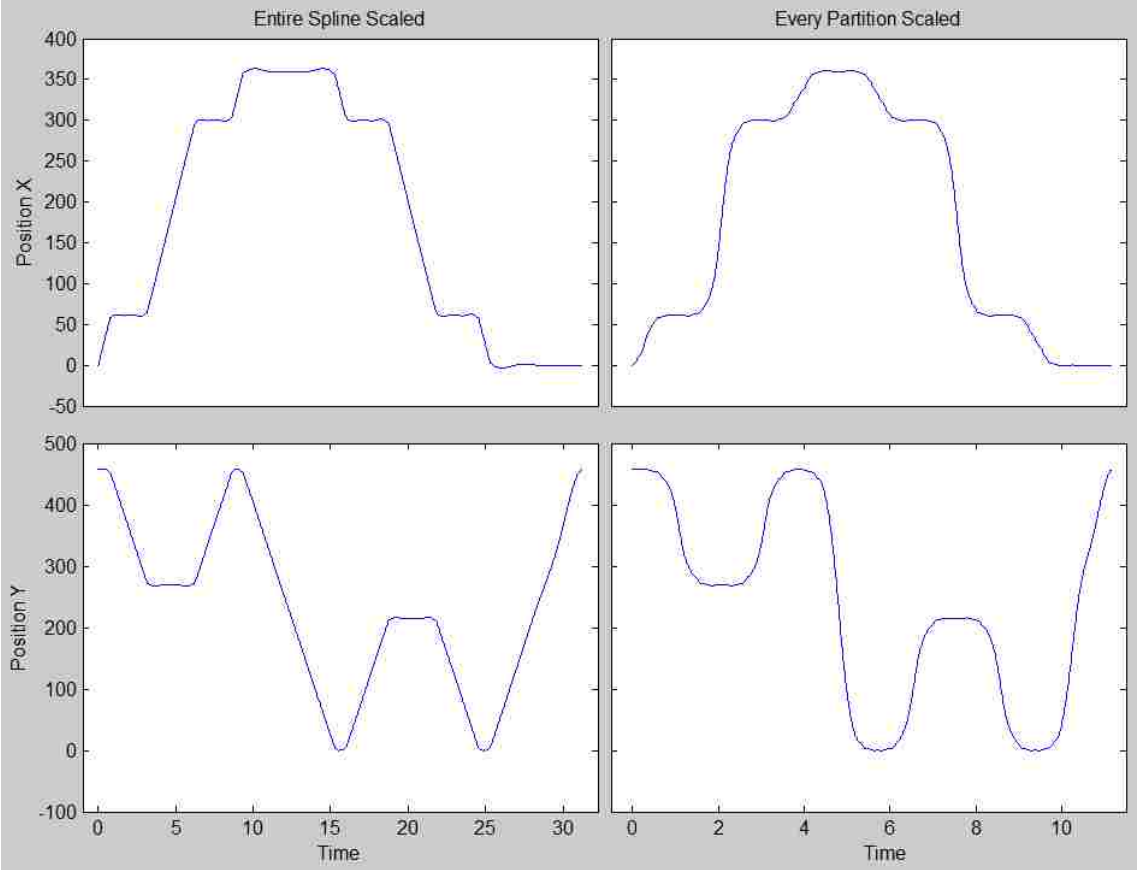

Figure 85: Comparison of position over time between the scaling of the entire spline versus scaling each partition.

Further benefit can be achieved by causing the velocity or acceleration to remain at the maximum for as long as possible. The minimum possible required time to pass through any set of points is of course with constant maximum acceleration or deceleration at all times. Because this is not technically possible, a maximum jerk can be accounted for; giving a mathematically continuous acceleration profile. This is also not possible, as it causes infinite snap, but for most lightweight machines, the snap should stay within the machine's capabilities without additional control. Unfortunately, these measures cannot be accomplished with a simple scaling law as they require more complicated parameter functions.

Adding many extra control points and recalculating by iteration add significantly to the computational expense. Though the cost of computing is decreasing exponentially, it is still a reasonable factor to take into consideration. Additionally, if straight lines are desired, all of the spline methods described previously are often inappropriate. Even if the path stays within a desired error amount, a straight line will not be produced. A more perfect compromise would use straight line segments as well as splines or even arcs for maximum path fidelity deviating only at corners. This would take a small amount of extra time, but would match the given reference path much more accurately and would be substantially less computationally intense so it may be implemented in the future.

Optimal Trajectory Tracking – Following the Desired Path

A time-varying version of the model used in the active damping section above is used for the optimal tracking of a planned trajectory.

$$\dot{X} = AX + BU$$

and

$$z = DX$$

The states

$$X = \begin{bmatrix} X1 \\ X2 \end{bmatrix}$$

are chosen as the global position ($X1$), and global velocity ($X2$) of each node. Repeated for convenience,

$$A = \begin{bmatrix} 0 & I \\ -M^{-1}K & -M^{-1}C \end{bmatrix},$$

$$B = \begin{bmatrix} 0 \\ M^{-1} \end{bmatrix},$$

and

$$D = \begin{bmatrix} I & 0 \\ 0 & I \end{bmatrix},$$

where each block is a 72x72 matrix. The stiffness matrix K, the friction matrix C, and the mass matrix M are no longer constant, but are functions of time so that $A = A(t)$ and $B = B(t)$.

Using the system of equations described above, an optimally tracking control can be planned to follow the trajectory between an initial state and a final state. The error between the desired position and the actual position is minimized by minimizing a quadratic cost function in which the costs of the error and the input over time as well as the cost of the final error are weighted.

$$J = \frac{1}{2}\left[\int_{t_0}^{T} [e(t)R_1(t)e(t) + u(t)R_2(t)u(t)]dt + e(T)R_3e(T)\right]$$

$R_1,\ R_2,$ and R_3 are weighting matrices representing the relative cost of each term in the cost function. The "1/2" can be omitted, but its inclusion makes the results available in a more convenient form [9].

The Hamiltonian is given by

$$H = \frac{1}{2}[z - Dx]R_1[z - Dx] + \frac{1}{2}uR_2u + Axp + Bup$$

where p is the co-state from the Hamiltonian equations of motion:

$$\dot{p} = -\frac{\partial H}{\partial x}$$

and

$$\dot{x} = \frac{\partial H}{\partial p}$$

Along the optimal tracking trajectory,

$$\frac{\partial H}{\partial u(t)} = 0$$

so

$$\frac{\partial H}{\partial u(t)} = R_2 u + B^T p = 0$$

which means that

$$u = -{R_2}^{-1} B^T p. \tag{1}$$

We now have a system of equations that can be solved:

$$\dot{p} = -\frac{\partial H}{\partial x} = -D^T R_1 D x - A^T p + D^T R_1 z$$

$$\dot{x} = Ax - B{R_2}^{-1} B^T p$$

or in standard reduced canonical form

$$\begin{bmatrix} \dot{x} \\ \dot{p} \end{bmatrix} = \begin{bmatrix} A & -B{R_2}^{-1} B^T \\ -D^T R_1 D & -A^T \end{bmatrix} \begin{bmatrix} x \\ p \end{bmatrix} + \begin{bmatrix} 0 \\ D^T R_1 z \end{bmatrix}.$$

Using the final time T to obtain boundary conditions, we find that the co-state p is linearly proportional to the state vector x.

$$p(T) = D^T(T) R_3 D(T) x(T) - D^T(T) R_3 z(T)$$

By defining

$$\mathrm{G}(T) = D^T(T) R_3 D(T) \tag{2}$$

and

$$g(T) = D^T(T) R_3 z(T) \tag{3}$$

we can substitute into (1) for our control law

$$u = -R_2{}^{-1}B^T(Gx - g)$$

and recognizing the structure of the equations, we can solve for G using the Riccati equation form

$$\dot{G} = -GA - A^TG + GBR_2{}^{-1}B^TG - D^TR_1D$$

with the boundary condition given by (2).

g can then be solved with

$$\dot{g} = \left[BR_2{}^{-1}B^TG - A\right]^Tg - D^TR_1z$$

using (3) as the boundary condition. Finally, the optimal tracking trajectory for the state variables is obtained from

$$\dot{x} = Ax + Bu$$

or

$$\dot{x} = \left[A - BR_2{}^{-1}B^TG\right]x + BR_2{}^{-1}B^Tg$$

Because this is a time varying system that is dependent on a time varying final value problem, the solution is computationally expensive when compared to the other options that have been discussed. Tracking achieved by stabilizing an initial position to a final position using the stabilization algorithm discussed at the beginning of this chapter using rigid links is only very slightly sub-optimal if very small increments are chosen, as the system will change very little. Because the actuators are discrete, the increments chosen can be made as small as the actuator steps to give exactly optimal results within the system's constraints using many fewer calculations and not requiring the storage of the solution to a final value problem.

Microcontroller

Microcontrollers have been advancing at the same rate as other electronics, and as such are now extremely powerful. For the implementation of a six module surface requiring eighteen actuators, a single XMOS microcontroller is used. The controller that was chosen has a multi-threaded quad-core processor, allowing the entire control program to be written without a single interrupt. Each core has eight threads, meaning that thirty-two separate functions can be run simultaneously. The processor communicates with the forty-eight input/output pins via a system of buffered ports. This means that up to twelve pins can be conveniently controlled with a single thread. As each actuator requires two pins, this means that six actuators can be easily controlled with one thread without using any interrupts or worrying about cycling between actuators within a single function. The program that was written to use this microcontroller was written in the XC language, and can be found in Appendix B.

Stepper Motors/Drivers

In an attempt to keep the prototype as simple as possible, there are no sensors or feedback of any kind. As a strictly open loop mechanism, stepper motors were chosen to simplify the model and to allow for minor disturbances. To maximize the performance of the motors, driver chips were used. The Allegro A4983 was selected for its high performance/cost ratio. The motors are current driven with the current held very near one amp. To allow for higher efficiency and improved torque at higher speeds, the motors are driven with two poles and micro-

stepping is used at sixteen increments per step [24]. To avoid slipping or loss of power and efficiency, all changes in velocity occur only at actual step locations (not at the incremental interstices).

CHAPTER 5

FUTURE WORK

Better Representation of the Mass

The mass (and the inertia in general) of this system is estimated and approximated using standard density and volume approaches. This is compared favorably with the weight as measured on a scale, but the exact location of the mass and its effect on the rotational inertia is not perfect. The mass and flexibility of the six-bar joint mechanism is built into the mass and flexibility of the links, with less than ideal results. A possible addition to future research can be an improvement on the representation of the mass and geometry of the mechanism.

Improved Actuators and Smaller Module Design

As is the case with all parallel mechanisms, using improved actuators with a better power to weight ratio can allow larger numbers of modules to be used in parallel and therefore show even more interesting uses for this type of device. Using piezoelectric or otherwise ultrasonic actuators can easily allow for significantly smaller module designs, which will open up many new applications and can be an interesting area for further research. The power transmission in this device was not carefully optimized and as such, significant improvements can almost certainly be made. Power transmission is a very interesting topic that will be essential in the future of this type of mechanism.

Sensitivity

Another aspect that has not yet been considered is the sensitivity of this system to variations in the tolerances of the joints. If there is a large force in a non-actuated direction, it is assumed in this model that the force does not at all affect the joint force. In reality, however, this may not be entirely true. If there is significant play in a joint, there will be a component of this other force that will add to (or subtract from) the joint force. Depending on the magnitude of the force and on the tolerance of the joint, this could significantly and adversely affect the usability of this system. Future research consequently includes a sensitivity and robustness analysis.

Feedback Control

Though several optimal control schemes have been modeled, without sensors on the prototype, these models cannot be physically tested. Future work can include sensors on a physical prototype to implement and rigorously test the optimal models that have been developed, as well as to develop new more sophisticated models.

Online Control

Because of time and software constraints, the current prototype has only been controlled offline with pre-calculated trajectories. Because the storage space available on the microcontroller is limited, the number of frames that can be shown in a single demonstration is restricted. By using the vastly larger storage capacity of

a computer or by performing on the fly calculations to determine future positions would remove this restriction, and allow for familiar "remote control" type abilities where a user can interact with the device using the computer's keyboard, mouse, or joystick.

Communication Using C++/FTDI

An interface to communicate online with the prototype was developed using a Future Technology Devices International (FTDI) chip and drivers to convert the universal serial bus (USB) protocol signal from the computer to the universal asynchronous receive/transmit (UART) protocol that is used by the XMOS processor. This code can be seen in Appendix D. Further development of this software to integrate it into the rest of the control software is essential in realizing the online controls described above.

Uncontrolled Points

In all of the methods evaluated thus far all control points are controlled all the time. Though it is probably always preferable to control each vertex of the mechanism, there are many situations in which the motion of only a few vertices matter and the motion of the rest of the vertices must simply support the motion of the few truly desired motions. In these cases, it may be useful to have a method of placing these points automatically. Three possibilities are listed here for further research: maximum stiffness, maximum compliance, and maximum mobility.

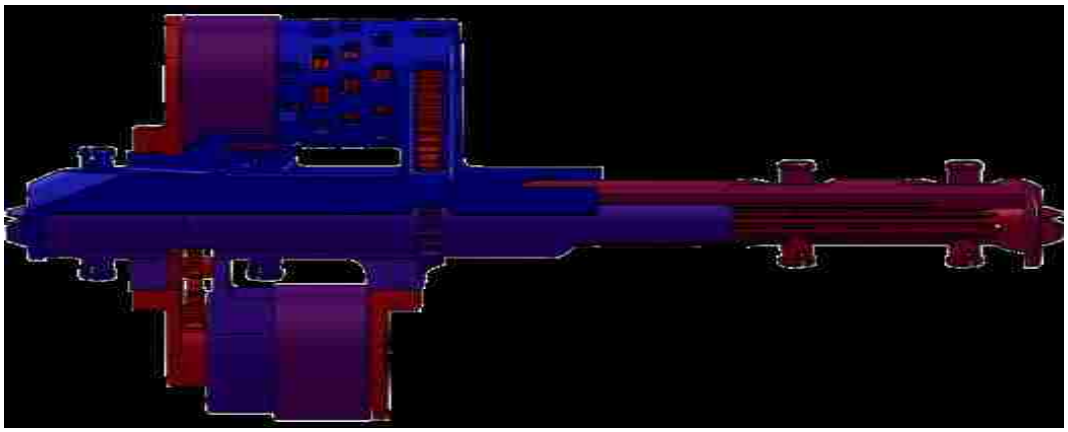

Figure 86: One full link with all parts colored according to mass with blue being the most massive and red the least massive.

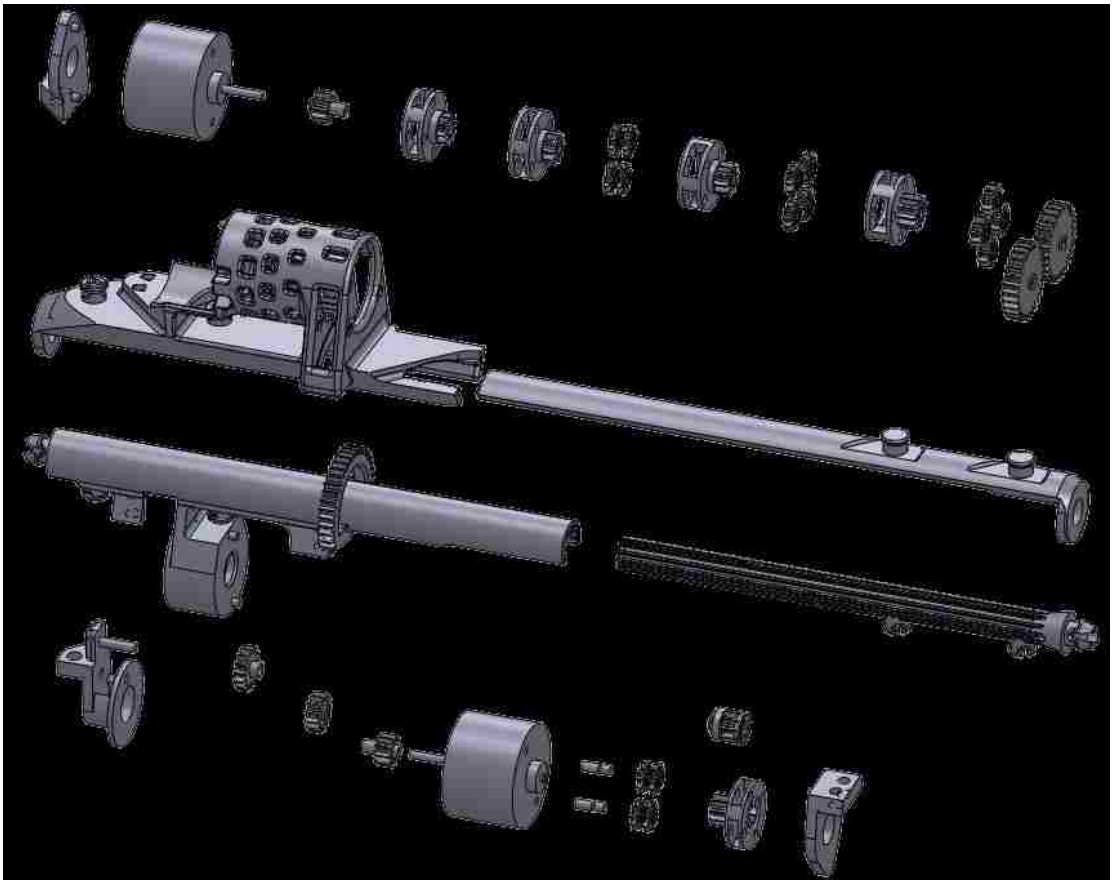

Figure 87: Exploded view of one link. For improved visualization, pins are shown for only the linear epicyclic stage, no screws are shown, and the first epicyclic stage of the rotary actuator is shown without planet gears.

Table 2: Nodal forces for the position in Figure 14.

		Link 1	Link 2	Link 3	Link 4	Link 5	Link 6
Node 1	Force X	0.2125	0.2278	-1.0837	-0.5561	0.0897	0.0841
	Force Y	-0.0419	-0.3884	0.1166	-0.0399	-0.1907	0.1613
	Force Z	0.0645	-0.1996	0.0681	-0.0448	0.0104	0.0171
	Moment About X	-0.0013	0.0026	-0.001	0	0.0002	-0.0013
	Moment About Y	-0.0043	0.0254	-0.0068	0.006	-0.0011	-0.0044
	Moment About Z	0.0132	0.0341	-0.0161	0.0049	0.0251	-0.0119
Node 2	Force X	-0.2125	-0.2278	1.0837	0.5561	-0.0897	-0.0841
	Force Y	0.0419	0.3884	-0.1166	0.0399	0.1907	-0.1613
	Force Z	-0.0645	0.1996	-0.0681	0.0448	-0.0104	-0.0171
	Moment About X	0.0013	-0.0026	0.001	0	-0.0002	0.0013
	Moment About Y	-0.0098	0.0182	-0.0131	0.0032	-0.0019	0.0006
	Moment About Z	-0.004	0.0507	-0.018	0.0032	0.0306	-0.0234

		Link 7	Link 8	Link 9	Link 10	Link 11	Link 12
Node 1	Force X	0.509	0.1672	0.2403	0.1403	-1.382	0.2321
	Force Y	0.0228	0.2834	0.3116	0.1608	0.0786	-0.1854
	Force Z	0.0114	0.2074	0.2111	-0.0001	-0.0375	-0.016
	Moment About X	0.0007	0.0008	-0.0021	-0.0006	-0.0002	-0.0001
	Moment About Y	-0.0062	-0.023	-0.0238	-0.0024	0.0093	0.0026
	Moment About Z	-0.0043	-0.0289	-0.0315	-0.0121	-0.009	0.0247
Node 2	Force X	-0.509	-0.1672	-0.2403	-0.1403	1.382	-0.2321
	Force Y	-0.0228	-0.2834	-0.3116	-0.1608	-0.0786	0.1854
	Force Z	-0.0114	-0.2074	-0.2111	0.0001	0.0375	0.016
	Moment About X	-0.0007	-0.0008	0.0021	0.0006	0.0002	0.0001
	Moment About Y	0.0037	-0.0192	-0.0191	0.0025	0.0017	0.0022
	Moment About Z	-0.0007	-0.0287	-0.0318	-0.0267	-0.014	0.0318

		Link 13	Link 14	Link 15	Link 16	Link 17	Link 18
Node 1	Force X	0.3303	0.0995	-0.3356	0.4185	0.3413	0.2435
	Force Y	-0.0035	-0.1512	-0.0035	0.0455	0.3288	-0.0834
	Force Z	-0.0105	0.0086	0.0462	-0.0076	0.1708	0.1116
	Moment About X	0.001	-0.0002	0.0003	0.0009	-0.0016	-0.0015
	Moment About Y	-0.0032	-0.0025	-0.0033	-0.0026	-0.0176	-0.0102
	Moment About Z	-0.0022	0.0273	0.0006	-0.007	-0.0366	0.01
Node 2	Force X	-0.3303	-0.0995	0.3356	-0.4185	-0.3413	-0.2435
	Force Y	0.0035	0.1512	0.0035	-0.0455	-0.3288	0.0834
	Force Z	0.0105	-0.0086	-0.0462	0.0076	-0.1708	-0.1116
	Moment About X	-0.001	0.0002	-0.0003	-0.0009	0.0016	0.0015
	Moment About Y	0.0054	0	-0.0062	0.0044	-0.0171	-0.0178
	Moment About Z	0.0029	0.0169	0.0001	-0.004	-0.0302	0.0109

		Link 19	Link 20	Link 21	Link 22	Link 23	Link 24
Node 1	Force X	-0.0788	-0.0119	0.0249	-0.1051	-0.1881	-1.0415
	Force Y	-0.2513	-0.3048	-0.0547	-0.03	0.1682	0.0509
	Force Z	-0.263	-0.1904	0.111	-0.0238	0.0666	0.1378
	Moment About X	0.0005	0.0011	-0.0017	0.0001	-0.0015	-0.0004
	Moment About Y	0.0314	0.0268	-0.0105	0.0045	-0.0095	-0.0218
	Moment About Z	0.0225	0.0239	0.0079	0.002	-0.0121	-0.0059
Node 2	Force X	0.0788	0.0119	-0.0249	0.1051	0.1881	1.0415
	Force Y	0.2513	0.3048	0.0547	0.03	-0.1682	-0.0509
	Force Z	0.263	0.1904	-0.111	0.0238	-0.0666	-0.1378
	Moment About X	-0.0005	-0.0011	0.0017	-0.0001	0.0015	0.0004
	Moment About Y	0.0345	0.0209	-0.0173	0.0003	-0.0046	-0.0202
	Moment About Z	0.0405	0.0525	0.0058	0.0041	-0.0237	-0.0096

Table 3: Local nodal forces for the position shown in Figure 15.

		Link 1	Link 2	Link 3	Link 4	Link 5	Link 6
Node 1	Force X	0	0	0	0	0	0
	Force Y	0	0	0	0	0	0
	Force Z	-0.0292	-0.532	0.2012	-0.1524	-0.1419	0.0494
	Moment About X	0.0091	0.0129	0.0003	0.0042	0.0058	-0.0073
	Moment About Y	0.0421	0.0681	0.0162	0.0428	0.006	0.0177
	Moment About Z	0	0	0	0	0	0
Node 2	Force X	0	0	0	0	0	0
	Force Y	0	0	0	0	0	0
	Force Z	0.0292	0.532	-0.2012	0.1524	0.1419	-0.0494
	Moment About X	-0.0091	-0.0129	-0.0003	-0.0042	-0.0058	0.0073
	Moment About Y	-0.0361	0.04	-0.0571	-0.0118	0.0228	-0.0278
	Moment About Z	0	0	0	0	0	0

		Link 7	Link 8	Link 9	Link 10	Link 11	Link 12
Node 1	Force X	0	0	0	0	0	0
	Force Y	0	0	0	0	0	0
	Force Z	0.074	0.3694	0.4101	0.0211	-0.1965	-0.0377
	Moment About X	0.0012	0.0041	-0.013	-0.0094	-0.0006	0
	Moment About Y	-0.0115	-0.0504	-0.0381	0.0225	0.0574	-0.0096
	Moment About Z	0	0	0	0	0	0
Node 2	Force X	0	0	0	0	0	0
	Force Y	0	0	0	0	0	0
	Force Z	-0.074	-0.3694	-0.4101	-0.0211	0.1965	0.0377
	Moment About X	-0.0012	-0.0041	0.013	0.0094	0.0006	0
	Moment About Y	-0.0036	-0.0246	-0.0452	-0.0267	-0.0175	0.0172
	Moment About Z	0	0	0	0	0	0

		Link 13	Link 14	Link 15	Link 16	Link 17	Link 18
Node 1	Force X	0	0	0	0	0	0
	Force Y	0	0	0	0	0	0
	Force Z	0.1668	0.145	-0.2164	0.1315	0.3893	-0.0469
	Moment About X	-0.0027	0.0047	0.0016	-0.0017	-0.0127	0.0054
	Moment About Y	-0.0206	-0.0245	0.0464	-0.0179	-0.0394	0.0373
	Moment About Z	0	0	0	0	0	0
Node 2	Force X	0	0	0	0	0	0
	Force Y	0	0	0	0	0	0
	Force Z	-0.1668	-0.145	0.2164	-0.1315	-0.3893	0.0469
	Moment About X	0.0027	-0.0047	-0.0016	0.0017	0.0127	-0.0054
	Moment About Y	-0.0133	-0.005	-0.0025	-0.0088	-0.0397	-0.0278
	Moment About Z	0	0	0	0	0	0

		Link 19	Link 20	Link 21	Link 22	Link 23	Link 24
Node 1	Force X	0	0	0	0	0	0
	Force Y	0	0	0	0	0	0
	Force Z	-0.568	-0.4724	0.0374	-0.0558	-0.0546	0.6869
	Moment About X	-0.0044	0.0123	0.0084	0.0038	-0.0081	0.0022
	Moment About Y	0.0423	0.0578	0.0174	0.0256	0.0167	-0.0921
	Moment About Z	0	0	0	0	0	0
Node 2	Force X	0	0	0	0	0	0
	Force Y	0	0	0	0	0	0
	Force Z	0.568	0.4724	-0.0374	0.0558	0.0546	-0.6869
	Moment About X	0.0044	-0.0123	-0.0084	-0.0038	0.0081	-0.0022
	Moment About Y	0.0731	0.0382	-0.025	-0.0143	-0.0055	-0.0475
	Moment About Z	0	0	0	0	0	0

Appendix A: Numerical Model and Simulation

Many different shapes were tested, and many minor changes were made to produce the various plots included in this paper – only the last version is included in this appendix. There are many lines for alternate results or diagnostic purposes included as comments that are not executed. I have not received any programming training, so the format and comments of these programs are likely far from standard. This code is included only as a reference.

```
%Triangular Surface
clear all
close all
clc
 L=8*0.0254 ; % shortest link length
Dia=0.75*0.0254; % link thickness limited by actuator diameter
X=[(L:L:2*L)',zeros(2,1),zeros(2,1);
   (L/2:L:5/2*L)',sind(60)*L*ones(3,1),zeros(3,1);
   (0:L:3*L)',sind(60)*2*L*ones(4,1),zeros(4,1);
   (L/2:L:5/2*L)',sind(60)*3*L*ones(3,1),zeros(3,1)];
 Tri = DelaunayTri(X(:,1:2));
Face=Tri.Triangulation;
Vertex=Tri.X;
%% Temporary Shape to Find Unique Edges
% find edge indices
% assuming x and y distribution is not inversely quadratic, this gives
```

```
% linearly independent edge lengths - other functions can easily be chosen
Vertex(:,3)=(1:length(Vertex)).^0.5;
for i=1:length(Face)
   for j=1:3
      P(j,:)=Vertex(Face(i,j),:);
   end
   tempVertex(:,:)=[P(1,:);P(2,:);P(3,:);P(1,:)];
   x=diff(tempVertex(:,1));% link lengths in x direction
   y=diff(tempVertex(:,2));% link lengths in y direction
   z=diff(tempVertex(:,3));% link lengths in z direction
   d(i,:)=hypot(hypot(x,y),z);% overall link lengths
end
[Unused M ~]=unique(d,'first');% M is an index vector for unique edges
[Unused1 M1 ~]=unique(d);%M1 is for the same edges attached to adjacent faces
%% Calculate Edge Indices
for i=1:length(M)
   if M(i)>(2*length(d))
      N(i,1)=M(i)-2*length(d);
   else
      N(i,1)=M(i)+length(d);
   end
end
Edge=[Face(M) Face(N)];% index for endpoints of unique edges
```

```
%% Animate Surface Shape and Plot
% 2D offset sinusoid
%Vertex(:,3)=2*sin(Vertex(:,1)/0.0254)+2*cos(Vertex(:,2)/0.0254);
%
% circle with center (H,K)
H=(min(Vertex(:,1))+max(Vertex(:,1)))/2;
K=(min(Vertex(:,2))+max(Vertex(:,2)))/2;
R=0.28;% radius in meters
Vertex(:,3)=(R^2-(Vertex(:,1)-H).^2-(Vertex(:,2)-K).^2).^0.5; % sphere
clear H K R
%}
 %Vertex(:,3)=0;
 % set nodes 1,9, and 3 as the low points
Vertex([1 9 10],3)=min(Vertex(:,3))*ones(3,1);
% Change all points not on function to zero
for i=1:length(Vertex)
   if imag(Vertex(i,3))~=0
      Vertex(i,3)=0;
   end
end
% set lowest point(s) to 0
Vertex(:,3)=Vertex(:,3)-min(Vertex(:,3));
% make sure links do not exceed max length - x and y are fixed
```

```
while max(Vertex(:,3))>(((1.5*L)^2-L^2)^0.5)
   Vertex(:,3)=Vertex(:,3)*0.999;
end
[Normal newVertex newFace d]=NormAndOffset(Face,Vertex,Dia);
Angle=CalculateAngle(M,M1,d,Normal);
GeometryData={Face,Edge,Vertex,newVertex,newFace,Normal,Angle};
GraphObjectData=plotvariables(GeometryData,cell(1,4),'b','-');
% Full Static Loading (including angular displacement and moment)
E=2e11; % Pa - Steel
A=pi*(127/8e4)^2; % m^2 - #8-32 threaded rod
G=7.72e10; % Pa - Steel
% modeled as a solid rectangular prism; needs work
% *
b=127/4e4; % base of cross-sectional area of rectangular prism
h=127/4e4; % height of cross-sectional area of rectangular prism
I=1/12*[b*h^3 b^3*h b*h*(b^2+h^2)]; % moment of inertia for rectangle
% *
% assemble the full global stiffness matrix
K=zeros(72,72); % initialize matrix
for i=1:24
   [k Tloc kloc]=GlobalBeamElementGeneralStiffness(Vertex(Edge(i,:),:),E,A,G,I);
   kLocal(:,:,i)=kloc;
   T(:,:,i)=Tloc;
```

```
    kaa=k(1:6,1:6);  % sub-matrix associated with node a only
    kab=k(1:6,7:12); % sub-matrix associated with node a and b
    kba=k(7:12,1:6); % kab'
    kbb=k(7:12,7:12);% sub-matrix associated with node b only
    a=Edge(i,1); % global index of node a
    b=Edge(i,2); % global index of node b
    K(6*a-5:6*a,6*a-5:6*a)=K(6*a-5:6*a,6*a-5:6*a)+kaa;
    K(6*a-5:6*a,6*b-5:6*b)=K(6*a-5:6*a,6*b-5:6*b)+kab;
    K(6*b-5:6*b,6*a-5:6*a)=K(6*b-5:6*b,6*a-5:6*a)+kba;
    K(6*b-5:6*b,6*b-5:6*b)=K(6*b-5:6*b,6*b-5:6*b)+kbb;
end
% mass of half of a link (assumed to be concentrated at one node)
Mass=0.01;
Pg=-9.81*Mass*[0;0;1;0;0;0]; % weight of half of a link
%{
%P(1)=[0;0;unknown]; % forces at node 1
P(2)=3*Pg; % forces at node 2
P(3)=4*Pg; % forces at node 3
P(4)=6*Pg; % forces at node 4
P(5)=4*Pg; % forces at node 5
P(6)=3*Pg; % forces at node 6
P(7)=6*Pg; % forces at node 7
P(8)=6*Pg; % forces at node 8
```

```
%P(9)=[0;0;unknown]; % forces at node 9
%P(10)=[0;0;unknown]; % forces at node 10
P(11)=4*Pg; % forces at node 11
P(12)=3*Pg; % forces at node 12
%}
% point 1 fixed; points 9 and 10 z-coordinate fixed:
P=[    3*Pg;4*Pg;6*Pg;4*Pg;3*Pg;6*Pg;6*Pg;0;0; 0;0;0;0;0; 0;0;0;4*Pg;3*Pg];
U=K([7:50 52:56 58:72],[7:50 52:56 58:72])\P;
U=[zeros(6,1);U(1:44);0;U(45:49);0;U(50:64)];
P=K*U;
Displacement=zeros(12,3);
ExaggerationFactor=100;
for i=1:12
   Displacement(i,1:3)=ExaggerationFactor*U(6*i-5:6*i-3)';
end
% exaggerated by ExaggerationFactor
DisplacedVertex=Vertex+Displacement;
[Normal newVertex newFace d]=NormAndOffset(Face,DisplacedVertex,Dia);
Angle=CalculateAngle(M,M1,d,Normal);
GeometryData={Face,Edge,DisplacedVertex,newVertex,newFace,Normal,Angle};
DisplacedObjectData=plotvariables(GeometryData,cell(1,4),'r',':');
% solve for forces in each rod
for i=1:24
```

```
   LinkForce(:,i)=kLocal(:,:,i)*T(:,:,i)'*...
                  [U(6*Edge(i,1)-5:6*Edge(i,1));
                   U(6*Edge(i,2)-5:6*Edge(i,2))];
end
LinkForce % N forces in each link in local coordinates
% Other Loading Conditions Tested
%{
% points 1,2,6,9,10,12 fixed:
P=[       4*Pg;6*Pg;4*Pg;  6*Pg;6*Pg;    4*Pg   ];
%    1:2 3:5  6 7:8  9:10  11   12 <--Node Indices
U=K([   13:30  37:48     61:66   ],[13:30 37:48 61:66])\P;
U=[zeros(12,1);U(1:18);zeros(6,1);U(19:30);zeros(12,1);U(31:36);zeros(6,1)];
P=K*U;
%+[3*Pg;3*Pg;zeros(18,1);3*Pg;zeros(12,1);3*Pg;3*Pg;zeros(6,1);3*Pg]
%-[3*Pg;3*Pg;zeros(18,1);3*Pg;zeros(12,1);3*Pg;3*Pg;zeros(6,1);3*Pg]
%}
%{
% points 1,9,10 fixed:
P=[    3*Pg;4*Pg;6*Pg;4*Pg;3*Pg;6*Pg;6*Pg;    4*Pg;3*Pg];
%    1 2:8 9:10 11:12 <--Node Indices
U=K([  7:48     61:72],[7:48 61:72])\P;
U=[zeros(6,1);U(1:42);zeros(12,1);U(43:54)];
P=K*U;
```

```
%}
%{
% point 1 x,y, and z fixed; points 9 and 10 z-coordinate fixed:
P=[   0;0;0;3*Pg;4*Pg;6*Pg;4*Pg;3*Pg;6*Pg;6*Pg;0;0; 0;0;0;0;0; 0;0;0;4*Pg;3*Pg];
U=K([4:50 52:56 58:72],[4:50 52:56 58:72])\P;
U=[zeros(3,1);U(1:47);0;U(48:52);0;U(53:67)];
P=K*U;
%}
function [Normal newVertex newFace d]=NormAndOffset(Face,Vertex,Dia)
% Show Link Thickness by Offsetting Vertices
% redefine faces to accommodate extra vertices
newFace=zeros(length(Face),3);
for i=1:length(Face)
   for j=1:3
      x(j,:)=Vertex(Face(i,j),:);
      newFace(i,j)=3*i+j-3;
   end
   tempVertex(:,:,i)=[x(1,:);x(2,:);x(3,:);x(1,:)];
end
% offset vertices to show link thickness and calculate face normals
for k=1:length(Face)
  x=diff(tempVertex(:,1,k));% link lengths in x direction
  y=diff(tempVertex(:,2,k));% link lengths in y direction
```

```
  z=diff(tempVertex(:,3,k));% link lengths in z direction
  d(:,k)=(x.^2+y.^2+z.^2).^0.5;% overall link lengths
  A(1,1)=acosd((-d(2,k)^2+d(1,k)^2+d(3,k)^2)/(2*d(1,k)*d(3,k)));%angle from L1
to L3
  A(2,1)=acosd((-d(3,k)^2+d(2,k)^2+d(1,k)^2)/(2*d(2,k)*d(1,k)));%angle from L1
to L2
  A(3,1)=acosd((-d(1,k)^2+d(2,k)^2+d(3,k)^2)/(2*d(2,k)*d(3,k)));%angle from L2
to L3
  d1=Dia./2./sind(A);% offset distance in direction of links
  sx=x./d(:,k);% scale factors in x direction
  sy=y./d(:,k);% scale factors in y direction
  sz=z./d(:,k);% scale factors in z direction
  dx=(sx-[sx(3);sx(1);sx(2)]).*d1;% offset distance of vertices in x
  dy=(sy-[sy(3);sy(1);sy(2)]).*d1;% offset distance of vertices in y
  dz=(sz-[sz(3);sz(1);sz(2)]).*d1;% offset distance of vertices in z
  newVertex(3*k-2:3*k,:)=tempVertex(1:3,:,k)+[dx dy dz];% new vertices
  temp=cross([x(1),y(1),z(1)],[x(2),y(2),z(2)]);% compute vectors normal to faces
  Normal(k,:)=temp/(hypot(hypot(temp(1),temp(2)),temp(3)));% use unit normal
vectors
end
d=d';
end
function [Angle]=CalculateAngle(M,M1,d,Normal)
```

```
% calculate angles between faces
   Angle=zeros(length(M),1);
   for i=1:length(M)
      if M(i)~=M1(i)
         temp=M(i);
         temp1=M1(i);
         while temp>length(d)
            temp=temp-length(d);
         end
         while temp1>length(d)
            temp1=temp1-length(d);
         end
         Angle(i)=acosd(dot(Normal(temp,:),Normal(temp1,:)));
      end
   end
end
function [GraphObjectData]=plotvariables(GeometryData,GraphObjectData,
Color,Linestyle)
% plot edges, face numbers, face normals, and edge angles or edge numbers
   Face=GeometryData{1};
   Edge=GeometryData{2};
   Vertex=GeometryData{3};
   newVertex=GeometryData{4};
```

```
    newFace=GeometryData{5};
    Normal=GeometryData{6};
    link=GraphObjectData{1};
    linknum=GraphObjectData{2};
    NormalLine=GraphObjectData{3};
    facenum=GraphObjectData{4};
    linewidth=1.1;
    h=findobj('Type','figure','Name','Triangular Surface');
    if isempty(h)
       h=figure('Name','Triangular
Surface','NumberTitle','Off','BackingStore','Off','Color','w');
       hold on;
       set(gca,'DrawMode','Fast');
       set(gca,'color','w','xcolor','k','ycolor','k','zcolor','k');
       axis equal vis3d;
       axis([-0.1 max(Vertex(:,1))+0.1 -0.1 max(Vertex(:,2))+0.1 -0.15 10*0.0254]);
       grid on;
       xlabel('x');ylabel('y');zlabel('z');
       camorbit(10,-30);
       rotate3d on;
    end
    %% Initialize Plot Variables
    if isempty(GraphObjectData{1})
```

```
      %Edges
      for i=1:length(Edge)
         P=Vertex(Edge(i,:),:);
link(i)=line(P(:,1),P(:,2),P(:,3),'Color',Color,'LineStyle',Linestyle,'LineWidth',linewidt
h);
         %link(2*i)=plot3(sum([P(1,1) sum(P(:,1))/2])/2,sum([P(1,2)
sum(P(:,2))/2])/2,sum([P(1,3)
sum(P(:,3))/2])/2,'Color',Color,'LineStyle','.','LineWidth',linewidth);
         %link(3*i)=plot3(sum([P(2,1) sum(P(:,1))/2])/2,sum([P(2,2)
sum(P(:,2))/2])/2,sum([P(2,3)
sum(P(:,3))/2])/2,'Color',Color,'LineStyle','o','LineWidth',linewidth);
%linknum(i)=text(sum(P(:,1))/2,sum(P(:,2))/2,sum(P(:,3))/2,num2str(i),'Horizont
alAlignment','center','VerticalAlignment','bottom','Color',Color,'LineStyle','-
','LineWidth',linewidth);% Edge numbers and angles
         %linknum(i)=text(P(1,1),P(1,2),P(1,3),['   '
num2str(Edge(i,1))],'HorizontalAlignment','left','VerticalAlignment','bottom','Color',
Color,'LineStyle','-','LineWidth',linewidth);
      end
   end
   %% Set Graph Object Data
   %Edges
   for i=1:length(Edge)
      P=Vertex(Edge(i,:),:);
```

```
      set(link(i),'Xdata',P(:,1),'Ydata',P(:,2),'Zdata',P(:,3));
    %
set(linknum(i),'Position',[sum(P(:,1))/2,sum(P(:,2))/2,sum(P(:,3))/2],'String',num2
str(i));% Edge numbers
   end
   figure(h);
   drawnow
   GraphObjectData={link,linknum,NormalLine,facenum};
end
function [kGlobal,T,k]=GlobalBeamElementGeneralStiffness(X,E,A,G,I)
% Calculates an element stiffness matrix in global coordinates for any
% general truss element given the location of the two endpoints, the
% modulus of elasticity, and the cross-sectional area.
Ix=I(1);
Iy=I(2);
Iz=I(3);
%X1=X(1:3);% x, y, and z coordinates of node 1
%X2=X(4:6);% x, y, and z coordinates of node 2
% choose a local z-axis on a plane formed by the local x- and global z-axes

X=(X(1,:)-X(2,:))';
Y=cross([0;0;1],X);
Y=Y/((sum(Y.^2))^0.5);
```

```
Z=cross(X,Y);
Z=Z/((sum(Z.^2))^0.5);
L=(sum(X.^2))^0.5; % total length of link
cxx=X(1)/L;% cosine of angle between link x- and global x-axes
cxy=X(2)/L;% cosine of angle between link x- and global y-axes
cxz=X(3)/L;% cosine of angle between link x- and global z-axes
cyx=Y(1);% cosine of angle between link y- and global x-axes
cyy=Y(2);% cosine of angle between link y- and global y-axes
cyz=Y(3);% cosine of angle between link y- and global z-axes
czx=Z(1);% cosine of angle between link z- and global x-axes
czy=Z(2);% cosine of angle between link z- and global y-axes
czz=Z(3);% cosine of angle between link z- and global z-axes
k=[E*A/L    0     0      0     0     0   -E*A/L   0      0     0     0     0    ;
   0  12*E*Iz/L^3 0      0     0 -6*E*Iz/L^2 0 -12*E*Iz/L^3  0     0     0 -6*E*Iz/L^2;
   0      0 12*E*Iy/L^3  0 -6*E*Iy/L^2 0     0      0 -12*E*Iy/L^3 0 -6*E*Iy/L^2 0    ;
   0      0     0   G*Ix/L   0     0     0     0      0   -G*Ix/L  0     0    ;
   0      0 -6*E*Iy/L^2  0   4*E*Iy/L  0     0      0  6*E*Iy/L^2  0    2*E*Iy/L 0    ;
   0  -6*E*Iz/L^2 0      0     0   4*E*Iz/L  0   6*E*Iz/L^2  0     0     0   2*E*Iz/L ;
  -E*A/L    0     0      0     0     0    E*A/L   0      0     0     0     0    ;
   0 -12*E*Iz/L^3 0      0     0  6*E*Iz/L^2 0  12*E*Iz/L^3  0     0     0  6*E*Iz/L^2;
   0      0 -12*E*Iy/L^3 0  6*E*Iy/L^2 0     0      0  12*E*Iy/L^3 0  6*E*Iy/L^2 0    ;
   0      0     0   -G*Ix/L  0     0     0     0      0    G*Ix/L  0     0    ;
   0      0  -6*E*Iy/L^2 0  2*E*Iy/L  0     0      0  6*E*Iy/L^2  0   4*E*Iy/L  0    ;
```

```
   0 -6*E*Iz/L^2 0     0     0  2*E*Iz/L  0  6*E*Iz/L^2  0     0     0  4*E*Iz/L ];
% u1  v1  w1  k1  e1  z1  u2  v2  w2  k2  e2  z2
T=[ cxx cyx czx  0   0   0   0   0   0   0   0   0  ; % u1
   cxy cyy czy  0   0   0   0   0   0   0   0   0  ; % v1
   cxz cyz czz  0   0   0   0   0   0   0   0   0  ; % w1
    0   0   0  cxx cyx czx  0   0   0   0   0   0  ; % k1
    0   0   0  cxy cyy czy  0   0   0   0   0   0  ; % e1
    0   0   0  cxz cyz czz  0   0   0   0   0   0  ; % z1
    0   0   0   0   0   0  cxx cyx czx  0   0   0  ; % u2
    0   0   0   0   0   0  cxy cyy czy  0   0   0  ; % v2
    0   0   0   0   0   0  cxz cyz czz  0   0   0  ; % w2
    0   0   0   0   0   0   0   0   0  cxx cyx czx ; % k2
    0   0   0   0   0   0   0   0   0  cxy cyy czy ; % e2
    0   0   0   0   0   0   0   0   0  cxz cyz czz ];% z2
kGlobal=T*k*T';
end
```

Motion Simulator for the rhombic surface:

```
function nestedRSS
% Robotic Rhombic Surface
%
clear all;close all;clc;
%}
%% RSS
```

```
   Dia=1.75*2.54; H=(0.75+0.68)*2.54;
% alpha(i-1) , A(i-1),   D    , theta(degrees)
    DH=[  0 , 3*Dia ,   H    ,  90 ;  % link 1
          0 , 3*Dia ,  -H    ,  90 ;  % link 2
          0 , 3*Dia ,   H    ,  90 ;  % link 3
         90 ,  Dia  ,-1.5*Dia,   0 ;  % link 4
          0 ,   0   ,   0    ,   0 ;  % link 5
         90 ,  Dia  ,-1.5*Dia,   0 ;  % link 6
          0 ,   0   ,   0    ,   0 ;  % link 7
        -90 ,  Dia  ,  -H/2  ,   0 ;  % link 8 (between modules right)
        -90 ,  Dia  ,  -H/2  , -90 ]; % link 9 (between modules top)
% joint range
   % range1=30;range2=150; %theta1
   % range3=-45;range4=45; %theta4 and theta6
   % range5=-10;range6=10; %D5 and D7
   border=0; % joints on the borders aren't attached
% desired joint variables
   %  theta1 theta4 theta6  D5     D7
   Q =[ 90  ,  0  ,  0  , 0   ,  0  ;  % module 1
        90  ,  0  ,  0  , 0   ,  0  ;  % module 2
        90  ,border,  0  ,border,  0  ;  % module 3
        90  ,  0  ,  0  , 0   ,  0  ;  % module 4
        90  ,  0  ,  0  , 0   ,  0  ;  % module 5
```

```
    90  ,border,   0  ,border,   0  ;  % module 6
    90  ,   0  ,border,  0   ,border;  % module 7
    90  ,   0  ,border,  0   ,border;  % module 8
    90  ,border,border,border,border]; % module 9
  [mT12 mT14 p1]=position(DH,Q,1);
  [mT23 mT25 p2]=position(DH,Q,2);
  [mT3border mT36 p3]=position(DH,Q,3);
  [mT45 mT47 p4]=position(DH,Q,4);
  [mT56 mT58 p5]=position(DH,Q,5);
  [mT6border mT69 p6]=position(DH,Q,6);
  [mT78 mT7border p7]=position(DH,Q,7);
  [mT89 mT7border p8]=position(DH,Q,8);
  [mT9border mT9border p9]=position(DH,Q,9);
  p2=mT12*p2;
  p3=mT12*mT23*p3;
  p4=mT14*p4;
  p5=mT14*mT45*p5;
  p6=mT14*mT45*mT56*p6;
  p7=mT14*mT47*p7;
  p8=mT14*mT47*mT78*p8;
  p9=mT14*mT47*mT78*mT89*p9;
% plots initial position and sets variables for animation
  azimuth=-45;
```

```
   link1=plotRSS(p1,0,azimuth,1);
   link2=plotRSS(p2,0,azimuth,1);
   link3=plotRSS(p3,0,azimuth,1);
   link4=plotRSS(p4,0,azimuth,1);
   link5=plotRSS(p5,0,azimuth,1);
   link6=plotRSS(p6,0,azimuth,1);
   link7=plotRSS(p7,0,azimuth,1);
   link8=plotRSS(p8,0,azimuth,1);
   link9=plotRSS(p9,0,azimuth,1);
%}
%
% animation
for theta=-45:45
   ACS=cosd(theta);
   theta4=atand(sind(theta));%theta4 and 6 for corner modules...
   theta1=asind(cosd(theta)/cosd(atand(sind(theta))));%for corners
   D5=-(cosd(theta4)*(DH(6,2) + DH(7,2)*cosd(theta4) + DH(9,2)*cosd(theta4) -
cosd(theta4)*(DH(2,2)/2 + DH(4,3)) - DH(9,3)*sind(theta4) + DH(4,2)*(1 -
ACS^2/cosd(theta4)^2)^(1/2) - sind(theta4)*(DH(5,2)/2 + DH(8,2)/2 + DH(1,3) -
DH(3,3)/2 + (3^(1/2)*DH(8,3))/2) + DH(5,2)*cosd(theta4)*(1 -
ACS^2/cosd(theta4)^2)^(1/2) + DH(8,2)*cosd(theta4)*(1 -
ACS^2/cosd(theta4)^2)^(1/2) - DH(8,3)*sind(theta4)*(1 -
ACS^2/cosd(theta4)^2)^(1/2) - (ACS*(DH(2,2) + DH(6,2) - DH(9,3)/2 +
```

```
(3^(1/2)*DH(7,2))/2 + (3^(1/2)*DH(9,2))/2))/cosd(theta4) + cosd(theta4)*(1 -
ACS^2/cosd(theta4)^2)^(1/2)*(DH(1,2) - DH(3,2)/2 + DH(6,3)) - sind(theta4)*(1 -
ACS^2/cosd(theta4)^2)^(1/2)*(DH(7,2)/2 + DH(9,2)/2 + DH(1,3) + DH(2,3)/2 +
(3^(1/2)*DH(9,3))/2) + (3*ACS*DH(2,2))/(2*cosd(theta4)) +
(ACS*DH(4,3))/cosd(theta4)))/ACS;
   D7=-D5;
   %    theta1 theta4 theta6   D5     D7
   Q =[ theta1,theta4,theta4,  D5  ,  D7  ;  % module 1
      90    ,theta4,theta , -D5  ,   0  ;  % module 2
    180-theta1,border,theta4,border, -D7  ;  % module 3
      90    ,theta ,theta4,  0   , -D7  ;  % module 4
      90    ,theta ,theta ,  0   ,   0  ;  % module 5
      90    ,border,theta4,border,  D7  ;  % module 6
    180-theta1,theta4,border, -D5  ,border;  % module 7
      90    ,theta4,border,  D5  ,border;  % module 8
      theta1,border,border,border,border]; % module 9
   [mT12 mT14 p1]=position(DH,Q,1);
   [mT23 mT25 p2]=position(DH,Q,2);
   [mT3border mT36 p3]=position(DH,Q,3);
   [mT45 mT47 p4]=position(DH,Q,4);
   [mT56 mT58 p5]=position(DH,Q,5);
   [mT6border mT69 p6]=position(DH,Q,6);
   [mT78 mT7border p7]=position(DH,Q,7);
```

```
    [mT89 mT7border p8]=position(DH,Q,8);
    [mT9border mT9border p9]=position(DH,Q,9);
    p2=mT12*p2;
    p3=mT12*mT23*p3;
    p4=mT14*p4;
    p5=mT14*mT45*p5;
    p6=mT14*mT45*mT56*p6;
    p7=mT14*mT47*p7;
    p8=mT14*mT47*mT78*p8;
    p9=mT14*mT47*mT78*mT89*p9;
    azimuth=theta;
    link1=plotRSS(p1,link1,azimuth,0);
    link2=plotRSS(p2,link2,azimuth,0);
    link3=plotRSS(p3,link3,azimuth,0);
    link4=plotRSS(p4,link4,azimuth,0);
    link5=plotRSS(p5,link5,azimuth,0);
    link6=plotRSS(p6,link6,azimuth,0);
    link7=plotRSS(p7,link7,azimuth,0);
    link8=plotRSS(p8,link8,azimuth,0);
    link9=plotRSS(p9,link9,azimuth,0);
end
%}
function [mT0right mT0top p]=position(DH,Q,R)
```

```
% forward kinematics using transformation matrices for RSS
    T01=[cosd(Q(R,1))          , -sind(Q(R,1))          ,     0     , DH(1,2)          ;
        sind(Q(R,1))*cosd(DH(1,1)), cosd(Q(R,1))*cosd(DH(1,1)), -sind(DH(1,1)), -
sind(DH(1,1))*DH(1,3) ;
        sind(Q(R,1))*sind(DH(1,1)), cosd(Q(R,1))*sind(DH(1,1)), cosd(DH(1,1)),
cosd(DH(1,1))*DH(1,3) ;
                0          ,          0          ,     0     ,          1       ];
    T12=[-cosd((Q(R,1)))          , -sind((Q(R,1)))          ,     0     , DH(2,2)          ;
        sind((Q(R,1)))*cosd(DH(2,1)), -cosd((Q(R,1)))*cosd(DH(2,1)), -
sind(DH(2,1)), -sind(DH(2,1))*DH(2,3) ;
        sind((Q(R,1)))*sind(DH(2,1)), -cosd((Q(R,1)))*sind(DH(2,1)),
cosd(DH(2,1)), cosd(DH(2,1))*DH(2,3) ;
                0          ,          0          ,     0     ,          1       ];
    T23=[cosd(Q(R,1))          , -sind(Q(R,1))          ,     0     , DH(3,2)          ;
        sind(Q(R,1))*cosd(DH(3,1)), cosd(Q(R,1))*cosd(DH(3,1)), -sind(DH(3,1)), -
sind(DH(3,1))*DH(3,3) ;
        sind(Q(R,1))*sind(DH(3,1)), cosd(Q(R,1))*sind(DH(3,1)), cosd(DH(3,1)),
cosd(DH(3,1))*DH(3,3) ;
                0          ,          0          ,     0     ,          1       ];
    T14=[ 0 1 0    0    ;
        -1 0 0    0    ;
         0 0 1 -DH(3,3)/2;
         0 0 0    1    ]*...
```

```
        [cosd(Q(R,2))          , -sind(Q(R,2))          ,     0     , DH(4,2)          ;
         sind(Q(R,2))*cosd(DH(4,1)), cosd(Q(R,2))*cosd(DH(4,1)), -sind(DH(4,1)), -
sind(DH(4,1))*DH(4,3) ;
         sind(Q(R,2))*sind(DH(4,1)), cosd(Q(R,2))*sind(DH(4,1)), cosd(DH(4,1)),
cosd(DH(4,1))*DH(4,3) ;
                  0         ,         0         ,     0     ,         1       ];
     T45=[cosd(DH(5,4))          , -sind(DH(5,4))          ,     0     , DH(5,2)       ;
         sind(DH(5,4))*cosd(DH(5,1)), cosd(DH(5,4))*cosd(DH(5,1)), -sind(DH(5,1)),
-sind(DH(5,1))*Q(R,4) ;
         sind(DH(5,4))*sind(DH(5,1)), cosd(DH(5,4))*sind(DH(5,1)), cosd(DH(5,1)),
cosd(DH(5,1))*Q(R,4) ;
                  0         ,         0         ,     0     ,         1   ];
     T26=[ 0 1 0     0;
         -1 0 0     0;
          0 0 1  -DH(2,3)/2;
          0 0 0     1]*...
         [cosd(Q(R,3))          , -sind(Q(R,3))          ,     0     , DH(6,2)          ;
         sind(Q(R,3))*cosd(DH(6,1)), cosd(Q(R,3))*cosd(DH(6,1)), -sind(DH(6,1)), -
sind(DH(6,1))*DH(6,3) ;
         sind(Q(R,3))*sind(DH(6,1)), cosd(Q(R,3))*sind(DH(6,1)), cosd(DH(6,1)),
cosd(DH(6,1))*DH(6,3) ;
                  0         ,         0         ,     0     ,         1       ];
     T67=[cosd(DH(7,4))          , -sind(DH(7,4))          ,     0     , DH(7,2)       ;
```

```
        sind(DH(7,4))*cosd(DH(7,1)), cosd(DH(7,4))*cosd(DH(7,1)), -sind(DH(7,1)),
-sind(DH(7,1))*Q(R,5) ;
        sind(DH(7,4))*sind(DH(7,1)), cosd(DH(7,4))*sind(DH(7,1)), cosd(DH(7,1)),
cosd(DH(7,1))*Q(R,5) ;
                0       ,       0       ,   0   ,       1       ];
    if (R==3)||(R==6)||(R==9)
      r=0;
    else
      r=1;
    end
    T53=[sind(Q(R+r,1))         , -cosd(Q(R+r,1))         ,   0   , DH(8,2)         ;
        cosd(Q(R+r,1))*cosd(DH(8,1)), sind(Q(R+r,1))*cosd(DH(8,1)), -
sind(DH(8,1)), -sind(DH(8,1))*DH(8,3) ;
        cosd(Q(R+r,1))*sind(DH(8,1)), sind(Q(R+r,1))*sind(DH(8,1)),
cosd(DH(8,1)), cosd(DH(8,1))*DH(8,3)+DH(2,2)/2 ;
                0       ,       0       ,   0   ,       1       ];
    T70=[cosd(DH(9,4))         , -sind(DH(9,4))         ,   0   , DH(9,2)         ;
        sind(DH(9,4))*cosd(DH(9,1)), cosd(DH(9,4))*cosd(DH(9,1)), -sind(DH(9,1)),
-sind(DH(9,1))*DH(9,3) ;
        sind(DH(9,4))*sind(DH(9,1)), cosd(DH(9,4))*sind(DH(9,1)), cosd(DH(9,1)),
cosd(DH(9,1))*DH(9,3)-DH(3,2)/2 ;
                0       ,       0       ,   0   ,       1       ];
  % Link0
```

```
    p=[0;0;0;1];
    p(:,3)=T01(:,4); % O1
    p(:,2)=p(:,3)+[0 0 -DH(1,3) 0]';
% Link1
    p(:,5)=T01*T12(:,4); % O2
    p(:,4)=T01*(T12(:,4)+[0 0 -DH(2,3) 0]');
    p(:,10)=T01*T14(:,4); % O4
% Link2
    p(:,7)=T01*T12*T23(:,4); % O3
    p(:,6)=p(:,7)+[0 0 -DH(3,3) 0]';
    p(:,13)=p(:,5)+[-DH(2,2)/2 0 0 0]';
    p(:,14)=T01*T12*T26(:,4); % O6
% Link3
    p(:,8)=p(:,1)+[0 0 DH(1,3) 0]';
    p(:,9)=T01*(T12(:,4)+[-DH(3,2)/2 0 -DH(2,3) 0]');
% Link4
    p(:,11)=T01*T14*T45(:,4); % O5
% Link5
    p(:,12)=T01*T14*T45*[DH(4,2) DH(3,3)/2 0 1]';
% Link6
    p(:,15)=T01*T12*T26*T67(:,4); % O7
% Link7
    p(:,16)=T01*T12*T26*T67*[DH(6,2) DH(2,3)/2 0 1]';
```

```
    mT0right=T01*T14*T45*T53;
    mT0top=T01*T12*T26*T67*T70;
end
function link=plotRSS(p,link,azimuth,v)
% plotting/animation for RSS
    h=findobj('Type','figure','Name','RSS');
    if v>0;
     x0=-37;y0=-20;z0=-62.5;range=125;linewidth=2;pointwidth=0.1;
     %x0=-10;y0=-22;z0=-50;range=100;linewidth=2;
     if isempty(h)&&v==1
        figure('Name','RSS','NumberTitle','Off','BackingStore','Off','Color','k');
     end
     hold on;
     set(gca,'DrawMode','Fast');
     set(gca,'color','k','xcolor','b','ycolor','g','zcolor','r');
     axis('square'); axis([x0 x0+range y0 y0+range z0 z0+range]);
     grid on; xlabel('x');ylabel('y');zlabel('z');
     view(azimuth,azimuth);

     link(1)=line(p(1,[1 2]),p(2,[1 2]),p(3,[1 2]),'Color','c','LineStyle','-
','LineWidth',linewidth,'EraseMode','xor');
     link(2)=line(p(1,[2 3]),p(2,[2 3]),p(3,[2 3]),'Color','c','LineStyle','-
','LineWidth',linewidth,'EraseMode','xor');
```

```
        link(3)=line(p(1,[3 4]),p(2,[3 4]),p(3,[3 4]),'Color','b','LineStyle','-
','LineWidth',linewidth,'EraseMode','xor');
        link(4)=line(p(1,[4 5]),p(2,[4 5]),p(3,[4 5]),'Color','b','LineStyle','-
','LineWidth',linewidth,'EraseMode','xor');
        link(10)=line(p(1,[9 10]),p(2,[9 10]),p(3,[9 10]),'Color','b','LineStyle','-
','LineWidth',linewidth,'EraseMode','xor');
        link(5)=line(p(1,[5 6]),p(2,[5 6]),p(3,[5 6]),'Color','m','LineStyle','-
','LineWidth',linewidth,'EraseMode','xor');
        link(6)=line(p(1,[6 7]),p(2,[6 7]),p(3,[6 7]),'Color','m','LineStyle','-
','LineWidth',linewidth,'EraseMode','xor');
        link(7)=line(p(1,[13 14]),p(2,[13 14]),p(3,[13 14]),'Color','m','LineStyle','-
','LineWidth',linewidth,'EraseMode','xor');
        link(8)=line(p(1,[7 8]),p(2,[7 8]),p(3,[7 8]),'Color','r','LineStyle','-
','LineWidth',linewidth,'EraseMode','xor');
        link(9)=line(p(1,[8 1]),p(2,[8 1]),p(3,[8 1]),'Color','r','LineStyle','-
','LineWidth',linewidth,'EraseMode','xor');
        link(11)=line(p(1,[10 11]),p(2,[10 11]),p(3,[10 11]),'Color','g','LineStyle','-
','LineWidth',linewidth,'EraseMode','xor');

        link(12)=line(p(1,[11 12]),p(2,[11 12]),p(3,[11 12]),'Color','r','LineStyle','-
','LineWidth',linewidth,'EraseMode','xor');
        link(13)=line(p(1,[14 15]),p(2,[14 15]),p(3,[14 15]),'Color','y','LineStyle','-
','LineWidth',linewidth,'EraseMode','xor');
```

```
        link(14)=line(p(1,[15 16]),p(2,[15 16]),p(3,[15 16]),'Color','c','LineStyle','-
','LineWidth',linewidth,'EraseMode','xor');
        return;
      end;
      view(azimuth,azimuth);
      set(link(1),'Xdata',p(1,[1 2]),'Ydata',p(2,[1 2]),'Zdata',p(3,[1 2]));
      set(link(2),'Xdata',p(1,[2 3]),'Ydata',p(2,[2 3]),'Zdata',p(3,[2 3]));
      set(link(3),'Xdata',p(1,[3 4]),'Ydata',p(2,[3 4]),'Zdata',p(3,[3 4]));
      set(link(4),'Xdata',p(1,[4 5]),'Ydata',p(2,[4 5]),'Zdata',p(3,[4 5]));
      set(link(5),'Xdata',p(1,[5 6]),'Ydata',p(2,[5 6]),'Zdata',p(3,[5 6]));
      set(link(6),'Xdata',p(1,[6 7]),'Ydata',p(2,[6 7]),'Zdata',p(3,[6 7]));
      set(link(7),'Xdata',p(1,[13 14]),'Ydata',p(2,[13 14]),'Zdata',p(3,[13 14]));
      set(link(8),'Xdata',p(1,[7 8]),'Ydata',p(2,[7 8]),'Zdata',p(3,[7 8]));
      set(link(9),'Xdata',p(1,[8 1]),'Ydata',p(2,[8 1]),'Zdata',p(3,[8 1]));
      set(link(10),'Xdata',p(1,[9 10]),'Ydata',p(2,[9 10]),'Zdata',p(3,[9 10]));
      set(link(11),'Xdata',p(1,[10 11]),'Ydata',p(2,[10 11]),'Zdata',p(3,[10 11]));
      set(link(12),'Xdata',p(1,[11 12]),'Ydata',p(2,[11 12]),'Zdata',p(3,[11 12]));
      set(link(13),'Xdata',p(1,[14 15]),'Ydata',p(2,[14 15]),'Zdata',p(3,[14 15]));
      set(link(14),'Xdata',p(1,[15 16]),'Ydata',p(2,[15 16]),'Zdata',p(3,[15 16]));
      figure(h);
      drawnow;
end
end
```

Forward kinematics solver for the rhombic surface:

```
% Robotic Rhombic Surface
    clear all;close all;clc;
%% RSS
   Dia=1.75*2.54; H=(0.75+0.68)*2.54;
% alpha(i-1) , A(i-1),   D    , theta(degrees)
    DH=[  0 , 3*Dia ,   H    ,  pi/2 ;  % link 1
          0 , 3*Dia ,  -H    ,  pi/2 ;  % link 2
          0 , 3*Dia ,   H    ,  pi/2 ;  % link 3
         pi/2 ,  Dia  ,-1.5*Dia,   0 ;  % link 4
          0 ,   0   ,   0    ,   0 ;  % link 5
         pi/2 ,  Dia  ,-1.5*Dia,   0 ;  % link 6
          0 ,   0   ,   0    ,   0 ;  % link 7
        -pi/2 ,  Dia  ,  -H/2  ,   0 ;  % link 8 (between modules right)
        -pi/2 ,  Dia  ,  -H/2  , -pi/2 ]; % link 9 (between modules top)
% joint range
   % range1=30;range2=150; %theta1
   % range3=-45;range4=45; %theta4 and theta6
   % range5=-10;range6=10; %D5 and D7
syms th11 th12 th13 th14 th15 th16 th17 th18 th19
syms th41 th42 th43 th44 th45 th46 th47 th48 th49
syms th61 th62 th63 th64 th65 th66 th67 th68 th69
syms D51 D52 D53 D54 D55 D56 D57 D58 D59
```

```
syms D71 D72 D73 D74 D75 D76 D77 D78 D79
syms DH21 DH22 DH23 DH24 DH25 DH26 DH27 DH28 DH29
syms DH31 DH32 DH33 DH34 DH35 DH36 DH37 DH38 DH39
% alpha(i-1) ,A(i-1),  D   , theta(degrees)
 DHsym=[  0 ,  DH21  ,  DH31  , pi/2 ;  % link 1
         0 ,  DH22  ,  DH32  , pi/2 ;  % link 2
         0 ,  DH23  ,  DH33  , pi/2 ;  % link 3
        pi/2 ,DH24  ,  DH34  ,  0 ;  % link 4
         0 ,  DH25  ,  DH35  ,  0 ;  % link 5
        pi/2 ,DH26  ,  DH36  ,  0 ;  % link 6
         0 ,  DH27  ,  DH37  ,  0 ;  % link 7
       -pi/2 ,DH28  ,  DH38  ,  0 ;  % link 8 (between modules right)
       -pi/2 ,DH29  ,  DH39  ,-pi/2 ]; % link 9 (between modules top)
   border=0;
   %     theta1  theta4  theta6   D5     D7
   Qsym =[th11,th41,th61, D51  , D71  ;  % module 1
        th12,th42,th62, D52  , D72  ;  % module 2
        th13,border ,th63,border, D73  ;  % module 3
        th14,th44,th64, D54  , D74  ;  % module 4
        th15,th45,th65, D55  , D75  ;  % module 5
        th16,border ,th66,border, D76  ;  % module 6
        th17,th47,border , D57  ,border;  % module 7
        th18,th48,border , D58  ,border;  % module 8
```

```
    th19,border ,border ,border,border]; % module 9
[mT12 mT14 p1]=positionsym(DHsym,Qsym,1);
[mT23 mT25 p2]=positionsym(DHsym,Qsym,2);
[mT3border mT36 p3]=positionsym(DHsym,Qsym,3);
[mT45 mT47 p4]=positionsym(DHsym,Qsym,4);
[mT56 mT58 p5]=positionsym(DHsym,Qsym,5);
[mT6border mT69 p6]=positionsym(DHsym,Qsym,6);
[mT78 mT7border p7]=positionsym(DHsym,Qsym,7);
[mT89 mT7border p8]=positionsym(DHsym,Qsym,8);
[mT9border mT9border p9]=positionsym(DHsym,Qsym,9);
p2=mT12*p2;
p3=mT12*mT23*p3;
p4=mT14*p4;
p5=mT14*mT45*p5;
p6=mT14*mT45*mT56*p6;
p7=mT14*mT47*p7;
p8=mT14*mT47*mT78*p8;
p9=mT14*mT47*mT78*mT89*p9;
M15a=mT12*mT25;
M15b=mT14*mT45;
M15=(M15a(1:3,:)-M15b(1:3,:));
M26a=mT23*mT36;
M26b=mT25*mT56;
```

```
  M26=(M26a(1:3,:)-M26b(1:3,:));
  M48a=mT45*mT58;
  M48b=mT47*mT78;
  M48=(M48a(1:3,:)-M48b(1:3,:));
  M59a=mT56*mT69;
  M59b=mT58*mT89;
  M59=(M59a(1:3,:)-M59b(1:3,:));
%
% controlled joint variables for 4 leg mode (non-walking)
  %   theta1  theta4  theta6   D5     D7
  Qleg =[th11, th41 , th61 ,  D51 ,  D71 ;  % module 1
       pi/2, th42 , pi/6 ,  D52 ,   0  ;  % module 2
       th13,border, th63 ,border,  D73 ;  % module 3
       pi/2, pi/6 , th64 ,   0  ,  D74 ;  % module 4
       pi/2, pi/6 , pi/6 ,   0  ,   0  ;  % module 5
       pi/2,border, th66 ,border,  D76 ;  % module 6
       th17, th47 ,border,  D57 ,border;  % module 7
       pi/2, th48 ,border,  D58 ,border;  % module 8
       th19,border,border,border,border]; % module 9
%}
  %  theta1 theta4 theta6  D5      D7
  Q =[ pi/2  ,   0  ,   0  ,  0   ,   0  ;  % module 1
     pi/2  ,   0  ,   0  ,  0   ,   0  ;  % module 2
```

```
    pi/2  ,border,   0  ,border,   0  ;  % module 3
    pi/2  ,   0  ,   0  ,  0   ,   0  ;  % module 4
    pi/2  ,   0  ,   0  ,  0   ,   0  ;  % module 5
    pi/2  ,border,   0  ,border,   0  ;  % module 6
    pi/2  ,   0  ,border,  0   ,border;  % module 7
    pi/2  ,   0  ,border,  0   ,border;  % module 8
    pi/2  ,border,border,border,border]; % module 9
  %digits 10
  %M15
  %One=simple((M15))
  syms c12 c14 c15 c16 c18 c44 c45 c62 c65 COS41 COS61
  syms s12 s14 s15 s16 s18 s44 s45 s62 s65 SIN41 SIN61
  COS=[cos(th12) cos(th14) cos(th15) cos(th16) cos(th18) cos(th44) cos(th45)
cos(th62) cos(th65)];
  SIN=[sin(th12) sin(th14) sin(th15) sin(th16) sin(th18) sin(th44) sin(th45)
sin(th62) sin(th65)];
  c=[c12 c14 c15 c16 c18 c44 c45 c62 c65];
  s=[s12 s14 s15 s16 s18 s44 s45 s62 s65];
  a=(subs(simple(M15),[COS SIN],[c s]));
```

Appendix B: XC Control

Manual Actuator Control

The following code is written in the XC language for the XMOS controller. It is used to manually actuate each joint of the mechanism using the buttons on the microcontroller device. Because there are no feedback sensors, this is necessary to correct any mismatch caused by losing steps when the device is used. The code is included only for reference.

```
// System headers
#include <xs1.h>
#include <platform.h>
#include <print.h>
#define SEC 100000000
#define steptime SEC*3/16/ 1000 //rpm
/* Port declarations */
// Buttons and Button-leds
out port p_kled = PORT_BUTTONLED;//4 bits for 4 green leds near buttons
in port p_key  = PORT_BUTTON;       //4 bits for 4 buttons
// Motor Out
on stdcore[0] : out port Core0Port4F = XS1_PORT_4F;  //2 rotary motors
on stdcore[1] : out port Core1Port8A = XS1_PORT_8A;  //4 rotary motors
on stdcore[2] : out port Core2Port8A = XS1_PORT_8A;  //4 linear motors
on stdcore[3] : out port Core3Port16A = XS1_PORT_16A;//8 linear motors
void motorSlave(chanend motor, out port m); //prototype
```

```
void motorSlave(chanend motor, out port m)
{
	int x;
	while(1)
	{
		motor:>x;
		m<:x;
	}
}
//main function that runs on core0
void masterThread(chanend rmotor1, chanend rmotor2, chanend lmotor1, chanend
lmotor2)
{
	//variables
	char buttonValue;
	int b=0xF,time,shift=0;
	timer t;
	p_kled <: b; //turn on green lights near buttons
	while(1){ //loop forever
		//TurnAllLEDsOff(cLED); //turn off all LEDS
		p_key :> buttonValue; //read buttons
		//motor <: 0;//no steps 0b0000 0
		if (buttonValue == 0b1110) //button 'A' (upside-down left)
```

```
                {
                        b^=1; //alternate button led so you know if button push was
registered
                        p_kled <: b;
                        if(shift<8)
                        {
                                lmotor1 <: (1<<(2*shift));//0x5555;//direction right
0b0101010101010101 21845 0x5555
                                lmotor1 <: (3<<(2*shift));//0xFFFF;//step right
0b0000000011111111 65535 0xFFFF
                        }
                        if((shift>7)&(shift<12))
                        {
                                lmotor2 <: (1<<(2*(shift-8)));//0x55;//direction right
0b0101010101010101 21845 0x5555
                                lmotor2 <: (3<<(2*(shift-8)));//0xFF;//step right
0b0000000011111111 65535 0xFFFF
                        }
                        if((shift>11)&(shift<16))
                        {
                                rmotor1 <: (1<<(2*(shift-12)));//0x55;//direction right
0b0101010101010101 21845 0x5555
```

```
                        rmotor1 <: (3<<(2*(shift-12)));//0xFF;//step right
0b0000000011111111 65535 0xFFFF
                    }
                    if((shift>15)&(shift<18))
                    {
                        rmotor2 <: (1<<(2*(shift-16)));//0x5;//direction right
0b0101010101010101 21845 0x5555
                        rmotor2 <: (3<<(2*(shift-16)));//0xF;//step right
0b0000000011111111 65535 0xFFFF
                    }
                t :> time;
                t when timerafter(time + steptime) :> void;
            }
            else if (buttonValue == 0b1101) //button 'B'
            {
                b^=1<<1;//alternate button led so you know if button push
was registered
                p_kled <: b;
                if(shift<8)
                {
                    lmotor1 <: (0<<(2*shift));
                    lmotor1 <: (2<<(2*shift));
                }
```

```
                if((shift>7)&(shift<12))
                {
                        lmotor2 <: (0<<(2*(shift-8)));
                        lmotor2 <: (2<<(2*(shift-8)));
                }
                if((shift>11)&(shift<16))
                {
                        rmotor1 <: (0<<(2*(shift-12)));
                        rmotor1 <: (2<<(2*(shift-12)));
                }
                if((shift>15)&(shift<18))
                {
                        rmotor2 <: (0<<(2*(shift-16)));
                        rmotor2 <: (2<<(2*(shift-16)));
                }
            t :> time;
            t when timerafter(time + steptime) :> void;
        }
        else if (buttonValue == 0b1011)//button 'C'
        {
                b^=1<<2;//alternate button led so you know if button push
was registered
                p_kled <: b;
```

```
                shift+=1;
                if (shift>17)
                {
                        shift=0;
                }
                t :> time;
            t when timerafter(time + SEC/2) :> void;
        }
        else if (buttonValue == 0b0111) //button 'D' (upside-down right)
        {
                b^=1<<3;//alternate button led so you know if button push
was registered
                p_kled <: b;
                shift-=1;
                if (shift<0)
                {
                        shift=17;
                }
                t :> time;
            t when timerafter(time + SEC/2) :> void;
        }
    }
}
```

```
//program entry point for all cores
int main()
{
	chan rmotor1, rmotor2, lmotor1, lmotor2; //communication channels
	par
	{
		on stdcore[0]: masterThread(rmotor1,rmotor2,lmotor1,lmotor2);
		on stdcore[0]: motorSlave(rmotor2, Core0Port4F); //2 rotary
motors
		on stdcore[1]: motorSlave(rmotor1, Core1Port8A); //4 rotary
motors
		on stdcore[2]: motorSlave(lmotor2, Core2Port8A); //4 linear
motors
		on stdcore[3]: motorSlave(lmotor1, Core3Port16A);//8 linear
motors
	}
	return 0;
}
```

Preprogrammed Offline Control

The code in this section is an example motion with each point along the path of each vertex embedded on the microcontroller. This limits the quality of the motion to whatever will fit on the memory of the microcontroller. In this particular

example, the device forms a pyramid and then returns to the neutral planar position. All positions in this code were precalculated with the Matlab control code given in Appendix C. This code is included only as a reference.

```
// System headers
#include <xs1.h>
#include <platform.h>
#include <print.h>
#define SEC 100000000
#define steptime SEC*3/16/  1000 //rpm
/* Port declarations */
// Buttons and Button-leds
out port p_kled = PORT_BUTTONLED;//4 bits for 4 green leds near buttons
in port p_key  = PORT_BUTTON;       //4 bits for 4 buttons
// Motor Out
on stdcore[0] : out port Core0Port4F = XS1_PORT_4F;  //2 rotary motors
on stdcore[1] : out port Core1Port8A = XS1_PORT_8A;  //4 rotary motors
on stdcore[2] : out port Core2Port8A = XS1_PORT_8A;  //4 linear motors
on stdcore[3] : out port Core3Port16A = XS1_PORT_16A;//8 linear motors
void motorSlave(chanend motor, out port m); //prototype
//catches all out port writes
void motorSlave(chanend motor, out port m)
{
        int x;
```

```
        while(1)
        {
                motor:>x;
                m<:x;
        }
}
//main function that runs on core0
void masterThread(chanend rmotor1, chanend rmotor2, chanend lmotor1, chanend
lmotor2)
{
        int a, i, j, k, time, linearmotor1, linearmotor2, rotarymotor1, rotarymotor2;
        short
        lsteps1[8][782]=
        {
        {**782 element vector calculated by code in Appendix C**},
        {**782 element vector calculated by code in Appendix C**},
        {**782 element vector calculated by code in Appendix C**},
        {**782 element vector calculated by code in Appendix C**},
        {**782 element vector calculated by code in Appendix C**},
        {**782 element vector calculated by code in Appendix C**},
        {**782 element vector calculated by code in Appendix C**},
        {**782 element vector calculated by code in Appendix C**}
        },
```

```
lsteps2[4][782]=
{
{**782 element vector calculated by code in Appendix C**},
{**782 element vector calculated by code in Appendix C**},
{**782 element vector calculated by code in Appendix C**},
{**782 element vector calculated by code in Appendix C**}
},
rsteps1[4][782]=
{
{**782 element vector calculated by code in Appendix C**},
{**782 element vector calculated by code in Appendix C**},
{**782 element vector calculated by code in Appendix C**},
{**782 element vector calculated by code in Appendix C**}
},
rsteps2[2][782]=
{
{**782 element vector calculated by code in Appendix C**},
{**782 element vector calculated by code in Appendix C**}
};
timer t;
for(i=0;i<782;i++)
{
        for(j=0;j<266;j++)
```

```
{
        //initialize motors for each new step
        linearmotor1=0;linearmotor2=0;
        rotarymotor1=0;rotarymotor2=0;
        for(k=0;k<8;k++)
        {
            a=((lsteps1[k][i])<0)?(0-lsteps1[k][i]):(lsteps1[k][i]);
            if(a>j)
            {
                if (lsteps1[k][i]<0)
                {
                    linearmotor1+=(2<<(2*k));//shorter
                }
                if (lsteps1[k][i]>0)
                {
                    linearmotor1+=(3<<(2*k));//longer
                }
            }
        }
        lmotor1<:linearmotor1&0x5555;//send just direction first
        lmotor1<:linearmotor1;//then direction + step to ensure
direction is registered
        for(k=0;k<4;k++)
```

```
{
        a=((lsteps2[k][i])<0)?(0-lsteps2[k][i]):(lsteps2[k][i]);
        if(a>j)
        {
            if(lsteps2[k][i]<0)
            {
                linearmotor2+=(2<<(2*k));//shorter
            }
            if(lsteps2[k][i]>0)
            {
                linearmotor2+=(3<<(2*k));//longer
            }
        }
    }
    lmotor2<:linearmotor2&0x55;//send just direction first
    lmotor2<:linearmotor2;//then direction + step to ensure
direction is registered
    for(k=0;k<4;k++)
    {
        a=((rsteps1[k][i])<0)?(0-rsteps1[k][i]):(rsteps1[k][i]);
        if(a>j)
        {
            if(rsteps1[k][i]<0)
```

```
                {
                        rotarymotor1+=(2<<(2*k));//concave
down
                }
                if(rsteps1[k][i]>0)
                {
                        rotarymotor1+=(3<<(2*k));//concave up
                }
            }
        }
        rmotor1<:rotarymotor1&0x55;//send just direction first
        rmotor1<:rotarymotor1;//then direction + step to ensure
direction is registered
        for(k=0;k<2;k++)
        {
            a=((rsteps2[k][i])<0)?(0-rsteps2[k][i]):(rsteps2[k][i]);
            if(a>j)
            {
                if(rsteps2[k][i]<0)
                {
                        rotarymotor2+=(2<<(2*k));//concave
down
                }
```

```
                                        if(rsteps2[k][i]>0)
                                        {
                                                rotarymotor2+=(3<<(2*k));//concave up
                                        }
                                }
                        }
                        rmotor2<:(rotarymotor2&0x5);//send just direction first
                        rmotor2<:rotarymotor2;//then direction + step to ensure
direction is registered
                        t :> time;//store the timer in variable time
                        t when timerafter(time + steptime) :> void;//wait until
time+=steptime before moving on
                }
        }
}
//program entry point for all cores
int main()
{
        chan rmotor1, rmotor2, lmotor1, lmotor2; //communication channels

        par
        {
                on stdcore[0]: masterThread(rmotor1,rmotor2,lmotor1,lmotor2);
```

```
                on stdcore[0]: motorSlave(rmotor2, Core0Port4F); //2 rotary
motors
                on stdcore[1]: motorSlave(rmotor1, Core1Port8A); //4 rotary
motors
                on stdcore[2]: motorSlave(lmotor2, Core2Port8A); //4 linear
motors
                on stdcore[3]: motorSlave(lmotor1, Core3Port16A);//8 linear
motors
        }
        return 0;
}
```

Appendix C: Matlab Control

This code sets up the object mesh used by the device to sample shapes, calculates and plans the desired equation driven (rather than dynamic shape driven) path based on a maximum actuator speed, calculates all joint motions, and then generates the actual steps that each actuator will have to take in order to perform the desired motion. There are also extra sections of code to calculate different implementations of the mechanism included as programming comments. This code is included as a reference only.

```
clear all
close all
clc
L=15 ; % shortest link length
Dia=2; % link thickness limited by actuator diameter
% Large...
%{
X=[(L:L:2*L)',zeros(2,1),zeros(2,1);
  (L/2:L:5/2*L)',sind(60)*L*ones(3,1),zeros(3,1);
  (0:L:3*L)',sind(60)*2*L*ones(4,1),zeros(4,1);
  (L/2:L:5/2*L)',sind(60)*3*L*ones(3,1),zeros(3,1)];
%}
X=[(L/2:L:3/2*L)',zeros(2,1),zeros(2,1);
  (0:L:2*L)',sind(60)*L*ones(3,1),zeros(3,1);
  (L/2:L:3/2*L)',sind(60)*2*L*ones(2,1),zeros(2,1)];
```

```
% switched x and y
%{
X=[zeros(2,1),(L/2:L:3/2*L)',zeros(2,1);
  sind(60)*L*ones(3,1),(0:L:2*L)',zeros(3,1);
  sind(60)*2*L*ones(2,1),(L/2:L:3/2*L)',zeros(2,1)];
%}
 Tri = DelaunayTri(X(:,1:2));
Face=Tri.Triangulation;
Vertex=Tri.X;
%% Temporary Shape to Find Unique Edges
% find edge indices
% assuming x and y distribution is not inversely quadratic, this gives
% linearly independent edge lengths - other functions can easily be chosen
Vertex(:,3)=(1:length(Vertex)).^0.5;
for i=1:length(Face)
   for j=1:3
      P(j,:)=Vertex(Face(i,j),:);
   end
   tempVertex(:,:)=[P(1,:);P(2,:);P(3,:);P(1,:)];
   x=diff(tempVertex(:,1));% link lengths in x direction
   y=diff(tempVertex(:,2));% link lengths in y direction
   z=diff(tempVertex(:,3));% link lengths in z direction
   d(i,:)=hypot(hypot(x,y),z);% overall link lengths
```

```
end
[Unused M ~]=unique(d,'first');% M is an index vector for unique edges
[Unused1 M1 ~]=unique(d);% M1 is for the same edges, but attached to adjacent
faces
 %% Calculate Edge Indices
for i=1:length(M)
   if M(i)>(2*length(d))
      N(i,1)=M(i)-2*length(d);
   else
      N(i,1)=M(i)+length(d);
   end
end
Edge=[Face(M) Face(N)];% index for endpoints of unique edges
 clear x y z d i j Unused Unused1 P tempVertex
 %% Initialize Graph Objects
Vertex(:,3)=0;direction=zeros(3,18);Sign=ones(18);
[Normal newVertex newFace d]=NormAndOffset(Face,Vertex,Dia);
[FaceAngle direction Sign]=CalculateAngle(M,M1,d,Normal,direction,Sign);
% Cell arrays used to shorten some lines for improved readability
%{
GeometryData={Face,Edge,Vertex,newVertex,newFace,Normal,FaceAngle};
GraphObjectData=plotvariables(GeometryData,cell(1,5));
%}
```

```
%% Plot
% center (H,K)
H=(min(Vertex(:,1))+max(Vertex(:,1)))/2;
K=(min(Vertex(:,2))+max(Vertex(:,2)))/2;
R=4;% radius
tempd=L; tempA=0;% initialize temp joint variables
S=0; TempChangeofLinkLength=0;
last=0; last1=0; pStep=0;
linearstep=7.9375/128000; % cm (6.20117187 × 10-5)
rotarystep=360/(312785+5/11); % deg (1.15094866 × 10-3)
 MaxVrpm=1e3; % rpm
MaxV=MaxVrpm*16/3; % steps/second
%dT=1/MaxV % length of time slot in seconds
 %
Amplitude=3; % one-d sinusoid
dT=0.05 % length of time slot in seconds
fIncrement=1e-1; % increment of function argument
%}
%{
Amplitude=2; % two-d sinusoid
dT=5e-1 % length of time slot in seconds
fIncrement=1e-1; % increment of function argument
%}
```

```
tic
count=0;
for H=0:fIncrement:R
   %Vertex(:,3)=(R^2-(Vertex(:,1)-H).^2-(Vertex(:,2)-K).^2).^0.5; % sphere
   %Vertex(:,3)=(R^2-(Vertex(:,1)-H).^2).^0.5; % 2-D cylinder
   %Vertex(:,3)=(R^2-(Vertex(:,1)-H).^2+(Vertex(:,2)-K).^2).^0.5; % spheroid
cylinder
   %Vertex(:,3)=((Vertex(:,1)-H).*(Vertex(:,2)-K))/R; % hyperbola
   %Vertex(:,3)=R-abs(Vertex(:,1)-H); % 1D absolute value
   %Vertex(:,3)=R-abs(Vertex(:,1)-H)-abs(Vertex(:,2)-K); % 2D absolute value
   Vertex(4,3)=H;
   %{
   Vertex(:,3)=((abs(sin(Vertex(:,1)-H))<fIncrement)|last).*sin(Vertex(:,1)-H); % 1D
sinusoid
      last=last|(abs(sin(Vertex(:,1)-H))<fIncrement);
   %}
   %{
   Vertex(:,3)=((sin(Vertex(:,1)-H)<fIncrement)|last)*R/2.*sin(Vertex(:,1)-H)+...
         ((sin(Vertex(:,2)-H)<fIncrement)|last1)*R/2.*sin(Vertex(:,2)-H); % 2D
sinusoid
     last=last|(abs(sin(Vertex(:,1)-H))<fIncrement);
     last1=last1|(abs(sin(Vertex(:,2)-H))<fIncrement);
   %}
```

```
  %{
  Vertex(:,3)=((sin(Vertex(:,1)-H)<fIncrement)|last)*R/2.*sin(Vertex(:,1)-H)+...
    ((cos(Vertex(:,2)-H)<fIncrement)|last1)*R/2.*cos(Vertex(:,2)-H); % 2D offset
sinusoid
    last=last|(abs(sin(Vertex(:,1)-H))<fIncrement);
    last1=last1|(abs(cos(Vertex(:,2)-H))<fIncrement);
  %}
    % Change all points not on function to zero
  Vertex(:,3)=(imag(Vertex(:,3))==0).*Vertex(:,3);
      % set low point to 0
  Vertex(:,3)=Vertex(:,3)-min(Vertex(:,3));
  %Vertex(:,3)=Vertex(:,3)+2.5;
    % make sure links do not exceed max length - x and y are fixed
  Vertex(:,3)=Amplitude*Vertex(:,3);
  while max(Vertex(:,3))>((((1+1/3)*L)^2-L^2)^0.5)
    Vertex(:,3)=Vertex(:,3)*0.999
  end
  %TODO:GlobalVertex=GlobalCoordinates(Edge,Vertex);
  %F=CalculateLinkForces(Edge,Vertex);
  %TODO: Check F to make sure max actuator force is not exceeded
  [Normal newVertex newFace d LinkAngle]=NormAndOffset(Face,Vertex,Dia);
  [FaceAngle direction Sign]=CalculateAngle(M,M1,d,Normal,direction,Sign);
```

```
count=count+1;
maxcheckangle(count)=max((FaceAngle));
mincheckangle(count)=min((FaceAngle));
%{
GeometryData={Face,Edge,Vertex,newVertex,newFace,Normal,FaceAngle};
GraphObjectData=plotvariables(GeometryData,GraphObjectData);
%}
ChangeofLinkLength=d(M)-tempd;% for output to microcontroller
ChangeofAngle=FaceAngle-tempA;% for output to microcontroller
ChangeofAngle=ChangeofAngle(:,[2,3,6:8,12])';
max(ChangeofAngle);
tempd=d(M); tempA=FaceAngle;
numSteps=[ChangeofLinkLength/linearstep;ChangeofAngle/rotarystep];
%{
  if ~min(min(sign(ChangeofLinkLength)~=sign(TempChangeofLinkLength)))
     count=count+1
  end
  TempChangeofLinkLength=ChangeofLinkLength;
%}
%{
  max(max(abs(FaceAngle)))
  max(max(abs(d)))
  pause
```

```
  %}
  if max(abs(numSteps))~=0
    MaxNumSteps=max(max(abs(numSteps)));
    V=MaxV*numSteps/MaxNumSteps;
    Slots=ceil(MaxNumSteps./max(abs(V))/dT);
    d0=fix(V*dT);
    for n=1:Slots
      pStep=pStep+V*dT-d0;
      if(abs(pStep)>=1)
        StepArray(:,n)=d0+sign(d0);
        pStep=pStep-sign(pStep);
      else
        StepArray(:,n)=d0;
      end
    end
    if (round(abs(pStep))==1) %account for round-off error
      StepArray(:,n)=StepArray(:,n)+sign(pStep);
    end
    StepOut(:,S+1:S+size(StepArray,2))=StepArray;
    S=size(StepOut,2);
  end
end
```

```
last=0;last1=0;
for H=R:-fIncrement:0
   %Vertex(:,3)=(R^2-(Vertex(:,1)-H).^2-(Vertex(:,2)-K).^2).^0.5; % sphere
   %Vertex(:,3)=(R^2-(Vertex(:,1)-H).^2).^0.5; % 2-D cylinder
   %Vertex(:,3)=(R^2-(Vertex(:,1)-H).^2+(Vertex(:,2)-K).^2).^0.5; % spheroid
cylinder
   %Vertex(:,3)=((Vertex(:,1)-H).*(Vertex(:,2)-K))/R; % hyperbola
   %Vertex(:,3)=R-abs(Vertex(:,1)-H); % 1D absolute value
   %Vertex(:,3)=R-abs(Vertex(:,1)-H)-abs(Vertex(:,2)-K); % 2D absolute value
   Vertex(4,3)=H;
   %{
   Vertex(:,3)=(~((abs(sin(Vertex(:,1)-H))<fIncrement)|last)).*sin(Vertex(:,1)-H); %
1D sinusoid
      last=(last|(abs(sin(Vertex(:,1)-H))<fIncrement));
   %}
   %{
   Vertex(:,3)=(~((sin(Vertex(:,1)-H)<fIncrement)|last))*R/2.*sin(Vertex(:,1)-H)+...
         (~((sin(Vertex(:,2)-H)<fIncrement)|last1))*R/2.*sin(Vertex(:,2)-H); % 2D
sinusoid
     last=last|(abs(sin(Vertex(:,1)-H))<fIncrement);
     last1=last1|(abs(sin(Vertex(:,2)-H))<fIncrement);
   %}
   %{
```

```
  Vertex(:,3)=((sin(Vertex(:,1)-H)<fIncrement)|last)*R/2.*sin(Vertex(:,1)-H)+...
    ((cos(Vertex(:,2)-H)<fIncrement)|last1)*R/2.*cos(Vertex(:,2)-H); % 2D offset
sinusoid
    last=last|(abs(sin(Vertex(:,1)-H))<fIncrement);
    last1=last1|(abs(cos(Vertex(:,2)-H))<fIncrement);
  %}
  % Change all points not on function to zero
  Vertex(:,3)=(imag(Vertex(:,3))==0).*Vertex(:,3);
  % set low point to 0
  Vertex(:,3)=Vertex(:,3)-min(Vertex(:,3));
  %Vertex(:,3)=Vertex(:,3)+2.5;
  % make sure links do not exceed max length - x and y are fixed
  Vertex(:,3)=Amplitude*Vertex(:,3);
  while max(Vertex(:,3))>((((1+1/3)*L)^2-L^2)^0.5)
    Vertex(:,3)=Vertex(:,3)*0.999
  end
  %TODO:GlobalVertex=GlobalCoordinates(Edge,Vertex);
  %F=CalculateLinkForces(Edge,Vertex);
  %TODO: Check F to make sure max actuator force is not exceeded
  [Normal newVertex newFace d LinkAngle]=NormAndOffset(Face,Vertex,Dia);
  [FaceAngle direction Sign]=CalculateAngle(M,M1,d,Normal,direction,Sign);
  count=count+1;
  maxcheckangle(count)=max((FaceAngle));
```

```
mincheckangle(count)=min((FaceAngle));
%{
GeometryData={Face,Edge,Vertex,newVertex,newFace,Normal,FaceAngle};
GraphObjectData=plotvariables(GeometryData,GraphObjectData);
%}
ChangeofLinkLength=d(M)-tempd;% for output to microcontroller
ChangeofAngle=FaceAngle-tempA;% for output to microcontroller
ChangeofAngle=ChangeofAngle(:,[2,3,6:8,12])';
tempd=d(M); tempA=FaceAngle;
numSteps=[ChangeofLinkLength/linearstep;ChangeofAngle/rotarystep];
%{
  if ~min(min(sign(ChangeofLinkLength)~=sign(TempChangeofLinkLength)))
     count=count+1
  end
  TempChangeofLinkLength=ChangeofLinkLength;
%}
%{
  max(max(abs(FaceAngle)))
  max(max(abs(d)))
  pause
%}
if max(abs(numSteps))~=0
  MaxNumSteps=max(max(abs(numSteps)));
```

```
      V=MaxV*numSteps/MaxNumSteps;
      Slots=ceil(MaxNumSteps./max(abs(V))/dT);
      d0=fix(V*dT);
      for n=1:Slots
        pStep=pStep+V*dT-d0;
        if(abs(pStep)>=1)
          StepArray(:,n)=d0+sign(d0);
          pStep=pStep-sign(pStep);
        else
          StepArray(:,n)=d0;
        end
      end
      if (round(abs(pStep))==1) %account for round-off error
        StepArray(:,n)=StepArray(:,n)+sign(pStep);
      end
      StepOut(:,S+1:S+size(StepArray,2))=StepArray;
      S=size(StepOut,2);
    end
end
max(max(maxcheckangle));
min(min(mincheckangle));
StepOut;
size(StepOut)
```

```
toc
%% Check
%{
xyzdistance=(Vertex(Face(N),:)-Vertex(Face(M),:));
distances=hypot(hypot(xyzdistance(:,1),xyzdistance(:,2)),xyzdistance(:,3));
%make sure distances calculated from endpoints is the same
check=distances-d(M);% should equal zero
%}
%}
function [Normal newVertex newFace d
LinkAngle]=NormAndOffset(Face,Vertex,Dia)
% Show Link Thickness by Offsetting Vertices
% redefine faces to accommodate extra vertices
newFace=zeros(length(Face),3);
for i=1:length(Face)
   for j=1:3
      x(j,:)=Vertex(Face(i,j),:);
      newFace(i,j)=3*i+j-3;
   end
   tempVertex(:,:,i)=[x(1,:);x(2,:);x(3,:);x(1,:)];
end
% offset vertices to show link thickness and calculate face normals
for k=1:length(Face)
```

```
  x=diff(tempVertex(:,1,k));% link lengths in x direction
  y=diff(tempVertex(:,2,k));% link lengths in y direction
  z=diff(tempVertex(:,3,k));% link lengths in z direction
  d(:,k)=(x.^2+y.^2+z.^2).^0.5;% overall link lengths
  A(1,1)=acosd((-d(2,k)^2+d(1,k)^2+d(3,k)^2)/(2*d(1,k)*d(3,k)));%angle from L1
to L3
  A(2,1)=acosd((-d(3,k)^2+d(2,k)^2+d(1,k)^2)/(2*d(2,k)*d(1,k)));%angle from L1
to L2
  A(3,1)=acosd((-d(1,k)^2+d(2,k)^2+d(3,k)^2)/(2*d(2,k)*d(3,k)));%angle from L2
to L3
  d1=Dia./2./sind(A);% offset distance in direction of links
  sx=x./d(:,k);% scale factors in x direction
  sy=y./d(:,k);% scale factors in y direction
  sz=z./d(:,k);% scale factors in z direction
  dx=(sx-[sx(3);sx(1);sx(2)]).*d1;% offset distance of vertices in x
  dy=(sy-[sy(3);sy(1);sy(2)]).*d1;% offset distance of vertices in y
  dz=(sz-[sz(3);sz(1);sz(2)]).*d1;% offset distance of vertices in z
  newVertex(3*k-2:3*k,:)=tempVertex(1:3,:,k)+[dx dy dz];% new vertices
  LinkAngle(1:3,k)=A;
  temp=cross([x(1),y(1),z(1)],[x(2),y(2),z(2)]);% compute vectors normal to faces
  Normal(k,:)=temp/(hypot(hypot(temp(1),temp(2)),temp(3)));% use unit normal
vectors
end
```

```
d=d';
end
function [FaceAngle,direction,Sign]=CalculateAngle(M,M1,d,Normal,direction,Sign)
% calculate angles between faces
   Angle=zeros(length(M),1);
   for i=1:length(M)
      if M(i)~=M1(i)
         temp=M(i);
         temp1=M1(i);
         while temp>length(d)
            temp=temp-length(d);
         end
         while temp1>length(d)
            temp1=temp1-length(d);
         end
         newdirection=cross(Normal(temp,:),Normal(temp1,:));
acosd(dot(direction(:,i),newdirection)/(sum(direction(:,i).^2).^0.5)/(sum(newdirec
tion.^2).^0.5));
if((acosd(dot(direction(:,i),newdirection)/(sum(direction(:,i).^2).^0.5)/(sum(newdi
rection.^2).^0.5)))>90)
            Sign(i)=-Sign(i);
         end
         direction(:,i)=newdirection;
```

```
        FaceAngle(i)=Sign(i)*acosd(dot(Normal(temp,:),Normal(temp1,:)));
      end
   end
end
function [GraphObjectData]=plotvariables(GeometryData,GraphObjectData)
% plot edges, face numbers, face normals, and edge angles or edge numbers
Face=GeometryData{1};
Edge=GeometryData{2};
Vertex=GeometryData{3};
newVertex=GeometryData{4};
newFace=GeometryData{5};
Normal=GeometryData{6};
link=GraphObjectData{1};
linknum=GraphObjectData{2};
NormalLine=GraphObjectData{3};
facenum=GraphObjectData{4};
TM=GraphObjectData{5};
linewidth=1;
h=findobj('Type','figure','Name','Triangular Surface');
if isempty(h)
   h=figure('Name','Triangular
Surface','NumberTitle','Off','BackingStore','Off','Color','k');
   hold on;
```

```
   set(gca,'DrawMode','Fast');
   set(gca,'color','k','xcolor','b','ycolor','g','zcolor','r');
   %% Initialize Plot Variables
   %Edges
   for i=1:length(Edge)
      P=Vertex(Edge(i,:),:);
      link(i)=line(P(:,1),P(:,2),P(:,3),'Color',[i-1 0 length(Edge)-i]/(length(Edge)-
1),'LineStyle','-','LineWidth',linewidth);
linknum(i)=text(sum(P(:,1))/2,sum(P(:,2))/2,sum(P(:,3))/2,num2str(i),'Horizontal
Alignment','center','Color',[i-1 0 length(Edge)-i]/(length(Edge)-1),'LineStyle','-
','LineWidth',linewidth);% Edge numbers and angles
   end
   %Faces
   for j=1:length(Face)
      P=Vertex(Face(j,:),:);
      NormalLine(j)=line(sum(P(:,1))/3+[0 2*Normal(j,1)],sum(P(:,2))/3+[0
2*Normal(j,2)],sum(P(:,3))/3+[0 2*Normal(j,3)],'Color',[j-1 length(Face)-j
0]/(length(Face)-1),'LineStyle','-','LineWidth',linewidth);
      facenum(j)=text(sum(P(:,1))/3,sum(P(:,2))/3,sum(P(:,3))/3,{''
num2str(j)},'HorizontalAlignment','center','Color',[j-1 length(Face)-j
0]/(length(Face)-1),'LineStyle','-','LineWidth',linewidth);
   end
```

```
TM=trimesh(newFace,newVertex(:,1),newVertex(:,2),newVertex(:,3),'FaceAlpha',0);
   axis equal vis3d;
   axis([0 max(Vertex(:,1)) 0 max(Vertex(:,2)) 0 20]);
   grid on;
   xlabel('x');ylabel('y');zlabel('z');
   camorbit(10,-30);
   rotate3d on;
end
%Edges
for i=1:length(Edge)
   P=Vertex(Edge(i,:),:);
   set(link(i),'Xdata',P(:,1),'Ydata',P(:,2),'Zdata',P(:,3));
set(linknum(i),'Position',[sum(P(:,1))/2,sum(P(:,2))/2,sum(P(:,3))/2],'String',num2
str(i));% Edge numbers and angles
end
%Faces
for j=1:length(Face)
   P=Vertex(Face(j,:),:);
   set(NormalLine(j),'Xdata',sum(P(:,1))/3+[0
2*Normal(j,1)],'Ydata',sum(P(:,2))/3+[0 2*Normal(j,2)],'Zdata',sum(P(:,3))/3+[0
2*Normal(j,3)]);
```

```
   set(facenum(j),'Position',[sum(P(:,1))/3,sum(P(:,2))/3,sum(P(:,3))/3],'string',{''
num2str(j)});
end
% small offset triangles
set(TM,'Vertices',newVertex);
figure(h);
drawnow
GraphObjectData={link,linknum,NormalLine,facenum,TM};
end
```

Appendix D: Online Communication

Simplex Communication

The following code was written to test one-way online communication from a computer to the XMOS microcontroller. This code sends musical signals to the speaker on the microcontroller. For the purposes of this project, the speaker port simply needs to be changed to the mechanism port, and the musical signal needs to be changed to the mechanism control signal. This is a very simple change, and was only not implemented due to time and licensing constraints. The code is included only as a reference for future work.

Simplex C++

```
#include "stdafx.h"
#include <windows.h>
#include "ftd2xx.h"
#include <iostream>
#include <cstdio>
#include <ctime>
#define TXBUFFERSIZE 256
#define RXBUFFERSIZE 1024
using namespace std;// for cout
//forward declaration of helper functions
FT_STATUS InitXMOSComm(FT_HANDLE &myHandle);
//main program
```

```
int _tmain(int argc, _TCHAR* argv[])
{
	//variables
	FT_HANDLE fthandle;		//special datatype to reference device to open, passed by reference then overwritten
	FT_STATUS ftStatus; //another special datatype which indicates status of device

	DWORD BytesWritten,BytesReceived,EventDWord; //return variables from FT_Write or FT_Read
	DWORD RxBytes,TxBytes; //how many bytes are available in the buffer, used in an FT_GetStatus call
	char TxBuffer[3],RxBuffer[1];
	int Notes[26]={7,5,3,5,7,7,7,5,5,5,7,10,10,7,5,3,5,7,7,7,7,5,5,7,5,3};
	int	beat[26]={1,1,1,1,1,1,2,1,1,2,1,1,2,1,1,1,1,1,1,1,1,1,1,1,1,4};
	double Note=220;
	int note,loop=1;
	//open the device
	ftStatus=InitXMOSComm(fthandle);
	while(1)
	{
		//wait for bytes
		FT_GetStatus(fthandle,&RxBytes,&TxBytes,&EventDWord);
```

```
if (RxBytes > 0)
{
	FT_Read(fthandle,RxBuffer,RxBytes,&BytesReceived);//dump
start bit
	for(int i=0;i<26;i++)
	{
		//*
		for(int j=0;j<Notes[i];j++)
		{
			Note*=1.059463094; //2^(1/12)=scaling factor
for piano scale
		}
		note=Note;
		Note=220;
			//cout<<note<<"\n";
		//*/
		TxBuffer[0]=(note>>8)&0xFF; //MSByte
		TxBuffer[1]=note&0xFF; //LSByte
		TxBuffer[2]=beat[i]&0xFF; //beat
		//cout<<Notes[i]<<"\n";
```

```
		cout<<(TxBuffer[0]&0xFF)<<":"<<(TxBuffer[1]&0xFF)<<":"<<(TxBuffer[2]&
0xFF)<<"\n";
                                        FT_Write(fthandle, TxBuffer, 3, &BytesWritten);
                                FT_Read(fthandle,RxBuffer,RxBytes,&BytesReceived);
		cout<<(RxBuffer[0]&0xFF)<<":"<<((TxBuffer[0]+TxBuffer[1]+TxBuffer[2])&
0xFF)<<"\n\n";
                                        if
((RxBuffer[0]&0xFF)!=((TxBuffer[0]+TxBuffer[1]+TxBuffer[2])&0xFF))
                                        {
                                                cout<<"Tx error. Trying again"<<"\n";
                                                i--;
                                        }
                                }
                        }
                }
        }
        return 0;
}
//helper functions
FT_STATUS InitXMOSComm(FT_HANDLE &fthandle)
{
        //variables
        FT_STATUS ftStatus;
```

```
	double timeDiff;
	int i;
	//open the FTDI Device based on Description -> change if using different
device
	ftStatus = FT_OpenEx("XC-1A 1V0 B",FT_OPEN_BY_DESCRIPTION,&fthandle);

	if (ftStatus == FT_OK)
	{ // success
		printf("Opened XMOS UART\n");
	}
	else
	{ // failure
		printf("Cannot open device.\n");
	}
	// Set read timeout of 5000ms, no write timeout
	ftStatus = FT_SetTimeouts(fthandle,5000,0);
	// Set RX and TX Buffer size, 64 to 64k, multiples of 64, bigger is better
transfer rate
	ftStatus = FT_SetUSBParameters(fthandle, RXBUFFERSIZE, TXBUFFERSIZE);
	// Set RX and TX Buffer size, 64 to 64k, multiples of 64, bigger is better
transfer rate
	//ftStatus = FT_SetUSBParameters(fthandle, 64, 64);
	ftStatus = FT_SetFlowControl(fthandle, FT_FLOW_NONE, 0x11, 0x13);
```

```
        // Set Baud Rate to 115,200 and 8n1
        ftStatus = FT_SetBaudRate(fthandle, 115200);
        ftStatus =
FT_SetDataCharacteristics(fthandle,FT_BITS_8,FT_STOP_BITS_1,FT_PARITY_NONE);
        // Purge both Rx and Tx buffers
        ftStatus = FT_Purge(fthandle, FT_PURGE_RX | FT_PURGE_TX);
        return ftStatus;
}
```

Simplex XC

```
// System headers
#include <xs1.h>
#include <platform.h>
#include <print.h>
#define BIT_TIME XS1_TIMER_HZ / 115200
#define MM 120 //tempo in beats per second ("Maelzel's Metronome")
buffered out port:1 p_spk = XS1_PORT_1K;
out port UART_TX_PORT = PORT_UART_TX; // 1bit port Tx
in  port UART_RX_PORT = PORT_UART_RX; // 1bit port Rx
// Functions
unsigned char getch(void);
void putch(unsigned char buffer);
void uart_configure(int baud_rate);
```

```
void play(int note, int beats, int octave);
// Global state
static unsigned bit_time = 0;
/** Initialize UART... bit_time
 *  Its fixed to, Data : 8bits, Parity : None, Stop : 1bit, Flow control : none.
 */
void uart_configure(int baud_rate)
{
  bit_time  = XS1_TIMER_MHZ * 1000000 / (unsigned) baud_rate;
  UART_TX_PORT <: 1;
}
/** UART receive a character  **/
unsigned char getch(void)
{
        unsigned data = 0, time;
        int i;
        unsigned char c;

            // Wait for stop bit
        UART_RX_PORT when pinseq (1) :> int _;

        // wait for start bit
        UART_RX_PORT when pinseq (0) :> int _ @ time;
```

```
        time += BIT_TIME + (BIT_TIME >> 1);


        // sample each bit in the middle.
        for (i = 0; i < 8; i += 1)
        {
                UART_RX_PORT @ time :> >> data;
          time += BIT_TIME;
        }


        // reshuffle the data.
        c = (unsigned char) (data >> 24);
        return c;
}
/** UART transmit a character.  This is blocking call for now. */
void putch(unsigned char buffer)
{
  unsigned time, data;
  data = buffer;
  // get current time from port with force out.
  UART_TX_PORT <: 1 @ time;
  // Start bit.
  UART_TX_PORT <: 0;
  // Data bits.
```

```
  for (int i = 0; i < 8; i += 1)
  {
    time += bit_time;
    UART_TX_PORT @ time <: >> data;
  }
  // Stop bit
  time += bit_time;
  UART_TX_PORT @ time <: 1;
  time += bit_time;
  UART_TX_PORT @ time <: 1;
}
void play(int note, int beats, int octave)
{
  int note_delay=100000000/(2*note);
  int time, spkVal = 0, x = 2, lowoctave=1;
  timer t;
  if (octave>0)
  {
          for (int k=0;k<(octave-1);k++)
          {
           x*=2;
          }
          octave=x;
```

```
}
if (octave<0)
{
        for (int k=(octave+1);k<0;k++)
        {
         x*=2;
        }
        lowoctave=x;
        octave=1;
}
if (octave==0)octave=1;
for (int i=0;i<beats;i++)
{
 for (int j=0;j<(100000000/note_delay*60/MM*octave/lowoctave);j++)
 {
   p_spk <: spkVal;
   t :> time;
   t when timerafter(time + note_delay/octave*lowoctave) :> void;
   spkVal = !spkVal;
 }
}
t :> time;
t when timerafter(time + 100000) :> void;//pause for a millisecond between notes
```

```
}
int main(void)
{
 int octave=0; // 0 is middle octave; -1 is one octave lower, and 1 is one octave
higher
 int note2,note1,note,beat;
 uart_configure(115200);
 putch(1);
 while(1)
 {
        note1=getch();
        note2=getch();
        beat=getch();
        note=(note1<<8)+note2;
        play(note,beat,octave);
        putch((note1+note2+beat)&0xFF);
 }
}
```

Duplex Communication

The next step is to test duplex (bidirectional) communication between the computer and the microcontroller. The example used is a simple binary calculator that utilizes the buttons on the microcontroller for input. This manual feedback can

easily be obtained from sensors rather than the buttons if and when sensors are introduced to the mechanism. This code is also given solely for future research efforts.

Duplex C++

```
#include "stdafx.h"

#include <windows.h>

#include "ftd2xx.h"

#include <iostream>

#include <cstdio>

#include <ctime>

#define TXBUFFERSIZE 256

#define RXBUFFERSIZE 1024

using namespace std;// for cout

//forward declaration of helper functions

FT_STATUS InitXMOSComm(FT_HANDLE &myHandle);

//main program

int _tmain(int argc, _TCHAR* argv[])

{
```

```
	//variables
	FT_HANDLE fthandle;		//special datatype to reference device to open, passed by reference then overwritten
	FT_STATUS ftStatus; //another special datatype which indicates status of device
	DWORD BytesWritten,BytesReceived,EventDWord; //return variables from FT_Write or FT_Read
	DWORD RxBytes,TxBytes; //how many bytes are available in the buffer, used in an FT_GetStatus call
	char TxBuffer[3],RxBuffer[3];
	int Total=0,loop=1;
	//open the device
	ftStatus=InitXMOSComm(fthandle);
	while(1)
	{
		//wait for bytes
		FT_GetStatus(fthandle,&RxBytes,&TxBytes,&EventDWord);
		if (RxBytes > 2)
```

```
{
    while(loop)
    {
        FT_Read(fthandle,RxBuffer,RxBytes,&BytesReceived);
        cout<<"Rx:
"<<(RxBuffer[1]&0xFF)<<":"<<(RxBuffer[0]&0xFF)<<(RxBuffer[2]&0xFF)<<"\n";
        if(RxBuffer[2]==(RxBuffer[0]+RxBuffer[1]))
        {
        Total+=(RxBuffer[0]&0xFF)+((RxBuffer[1]&0xFF)<<8);
            TxBuffer[0]=Total&0xFF; //LSByte
            TxBuffer[1]=((Total&0xFFFF)>>8)&0xFF;
//MSByte
            TxBuffer[2]=TxBuffer[0]+TxBuffer[1];
//checksum
            FT_Write(fthandle, TxBuffer, 3, &BytesWritten);
            cout << "Sum:
"<<(TxBuffer[1]&0xFF)<<":"<<(TxBuffer[0]&0xFF)<<"\n";
            loop=0;
```

```
                    }
                    else
if(((RxBuffer[0]+RxBuffer[1])==0)&&(RxBuffer[2]==1))
                    {
                        Total=0;
                        TxBuffer[0]=0&0xFF; //LSByte
                        TxBuffer[1]=0&0xFF; //MSByte
                        TxBuffer[2]=TxBuffer[0]+TxBuffer[1];
//checksum
                        FT_Write(fthandle, TxBuffer, 3, &BytesWritten);
                        cout<<"Total Cleared\n";
                        loop=0;
                    }
                    else
                    {
                        loop=1;
                    }
                }
```

```
                loop=1;
                }
        }
        return 0;
}
//helper functions
FT_STATUS InitXMOSComm(FT_HANDLE &fthandle)
{
        //variables
        FT_STATUS ftStatus;
        double timeDiff;
        int i;
                //open the FTDI Device based on Description -> change if using
different device
        ftStatus = FT_OpenEx("XC-1A 1V0 B",FT_OPEN_BY_DESCRIPTION,&fthandle);
        if (ftStatus == FT_OK)
        { // success
```

```
            printf("Opened XMOS UART\n");
        }
        else
        { // failure
            printf("Cannot open device.\n");
        }
        // Set read timeout of 5000ms, no write timeout
        ftStatus = FT_SetTimeouts(fthandle,5000,0);
        // Set RX and TX Buffer size, 64 to 64k, multiples of 64, bigger is better
transfer rate
        ftStatus = FT_SetUSBParameters(fthandle, RXBUFFERSIZE, TXBUFFERSIZE);
        // Set RX and TX Buffer size, 64 to 64k, multiples of 64, bigger is better
transfer rate
        //ftStatus = FT_SetUSBParameters(fthandle, 64, 64);
        ftStatus = FT_SetFlowControl(fthandle, FT_FLOW_NONE, 0x11, 0x13);
        // Set Baud Rate to 115,200 and 8n1
        ftStatus = FT_SetBaudRate(fthandle, 115200);
```

```
        ftStatus =
FT_SetDataCharacteristics(fthandle,FT_BITS_8,FT_STOP_BITS_1,FT_PARITY_NONE);
        // Purge both Rx and Tx buffers
        ftStatus = FT_Purge(fthandle, FT_PURGE_RX | FT_PURGE_TX);
        return ftStatus;
}
```

Duplex XC

```
// System headers
#include <xs1.h>
#include <platform.h>
#include <print.h>
#define SEC 100000000
#define BIT_TIME XS1_TIMER_HZ / 115200
/* Port declarations */
// Keys/Key-leds
out port p_kled = PORT_BUTTONLED;       //4 bits for 4 green leds near buttons
in port p_key  = PORT_BUTTON;     //4 bits for 4 buttons
// 'Clock' leds
out port p_cled_g = PORT_CLOCKLED_SELG;
out port p_cled_r = PORT_CLOCKLED_SELR;
```

```
out port p_cled_0 = PORT_CLOCKLED_0; //8 bit port bits 4,5,6 from right ->
0b01110000 turns all on
out port p_cled_1 = PORT_CLOCKLED_1;    //note this is on a different core
out port p_cled_2 = PORT_CLOCKLED_2;    //note this is on a different core
out port p_cled_3 = PORT_CLOCKLED_3;    //note this is on a different core
buffered out port:32 p_spk = PORT_SPEAKER;
out port UART_TX_PORT = PORT_UART_TX; // 1bit port Tx
in  port UART_RX_PORT = PORT_UART_RX; // 1bit port Rx
// Functions
unsigned char uart_getch(void);
void uart_putch(unsigned char buffer);
void uart_configure(int baud_rate);
void ledSlaves(chanend cLED, out port p);
void setLEDtoRed();
void setLEDtoGreen();
void TurnAllLEDsOff(chanend cLED[3]);
void LightUp(chanend cLED[3], int a);
// Global state
static unsigned bit_time = 0;
void wait(timer tmr, unsigned delay)
{
  unsigned t;
  tmr :> t;
```

```
 tmr when timerafter(t + delay) :> t;
}
/** Initialize UART... bit_time
 *  Its fixed to, Data : 8bits, Parity : None, Stop : 1bit, Flow control : none.
 */
void uart_configure(int baud_rate)
{
  bit_time  = XS1_TIMER_MHZ * 1000000 / (unsigned) baud_rate;
  UART_TX_PORT <: 1;
}
/** UART receive a character  **/
unsigned char uart_getch(void)
{
        unsigned data = 0, time;
        int i;
        unsigned char c;
                // Wait for stop bit
        UART_RX_PORT when pinseq (1) :> int _;
        // wait for start bit
        UART_RX_PORT when pinseq (0) :> int _ @ time;
        time += BIT_TIME + (BIT_TIME >> 1);
        // sample each bit in the middle.
        for (i = 0; i < 8; i += 1)
```

```
        {
                UART_RX_PORT @ time :> >> data;
          time += BIT_TIME;
        }
        // reshuffle the data.
        c = (unsigned char) (data >> 24);
        return c;
}
/** UART transmit a character.  This is blocking call for now. */
void uart_putch(unsigned char buffer)
{
  unsigned time, data;
  data = buffer;
  // get current time from port with force out.
  UART_TX_PORT <: 1 @ time;
  // Start bit.
  UART_TX_PORT <: 0;
  // Data bits.
  for (int i = 0; i < 8; i += 1)
  {
    time += bit_time;
    UART_TX_PORT @ time <: >> data;
  }
```

```
  // Stop bit
  time += bit_time;
  UART_TX_PORT @ time <: 1;
  time += bit_time;
  UART_TX_PORT @ time <: 1;
}
void LightUp(chanend cLED[3], int a)
{
        a&=0xFFF; //12-bit max, cut off extra
        p_cled_0 <: (a&7)<<4; //7 is binary 111
        cLED[0] <: ((a>>3)&7)<<4;
        cLED[1] <: ((a>>6)&7)<<4;
        cLED[2] <: ((a>>9)&7)<<4;
}
//slaves that are run on
void ledSlaves(chanend cLED, out port p)
{
 int x;
 while (1)
 {
        cLED :> x;
   p <: x;
 }
```

```
}
//helper functions
void setLEDtoRed()
{
        p_cled_g <: 0;
        p_cled_r <: 1;
}
void setLEDtoGreen()
{
        p_cled_g  <: 1;
        p_cled_r <: 0;
}
//help function, run from core0 only
void TurnAllLEDsOff(chanend cLED[3])
{
        p_cled_0 <: 0; //core0
        cLED[0] <: 0; //core 1
        cLED[1] <: 0; //core 2
        cLED[2] <: 0; //core 3

}
//main function that runs on core0
void masterThread( chanend cLED[3])
```

```
{
	//variables
	char buttonValue;
	int a,i=0,j=0,b=0xF,Total=0,loop=1;
	uart_configure(115200);
	setLEDtoGreen();
	p_kled <: b; //turn on green lights near buttons

	while(1){ //loop forever
		//TurnAllLEDsOff(cLED); //turn off all LEDS
		p_key :> buttonValue; //read buttons
		if (buttonValue == 0b0111) //button 1 (1)
		{
			b^=1<<3; //alternate button led so you know if button push
was registered
			p_kled <: b;
			i*=2;//shifts left by one to add a zero to the right side of the
number
			i+=1;//adds one to change the above mentioned zero to a one
			LightUp(cLED,i); //displays number as you type to make sure
you are typing what you think you are
			for (a=0;a<(SEC/20);a++);//debouncing
		}
```

```
        else if (buttonValue == 0b1011) //button 2  (0)
        {
            b^=1<<2;//alternate button led so you know if button push
was registered
            p_kled <: b;
            i*=2;//shifts left by one to add a zero to the right side of the
number
            LightUp(cLED,i);//displays number as you type to make sure
you are typing what you think you are
            for (a=0;a<(SEC/20);a++);//debouncing
        }
        else if (buttonValue == 0b1101)//button 3 (+)
        {
            while(loop)
            {
                b^=1<<1;//alternate button led so you know if button
push was registered
                p_kled <: b;
                uart_putch(i&0xFF);//send two bytes because up to 11-
bit numbers can be added
                uart_putch((i>>8)&0xFF);
                uart_putch((i&0xFF)+((i>>8)&0xFF));//checksum
                i=uart_getch()&0xFF;
```

```
                j=uart_getch()&0xFF;
                if(((i+j)&0xFF)==(uart_getch()&0xFF))
                {
                        Total=i+(j<<8);
                        LightUp(cLED,Total);//displays current total
                        i=0;
                        loop=0;
                }
                else
                {
                        loop=1;
                }
                for (a=0;a<(SEC/20);a++);//debouncing
        }
        loop=1;
    }
    else if (buttonValue == 0b1110) //button 4 (clear)
    {
        while(loop)
        {
            b^=1;//alternate button led so you know if button push
was registered
            p_kled <: b;
```

```
                    uart_putch(0&0xFF);//send two bytes because up to
11-bit numbers can be added
                    uart_putch(0&0xFF);
                    uart_putch((0&0xFF)+(1&0xFF));//checksum
                    i=uart_getch()&0xFF;
                    j=uart_getch()&0xFF;
                    if((i+j)==uart_getch())
                    {
                        Total=i+(j<<8);
                        LightUp(cLED,Total);//displays current total
(zero)
                        i=0;
                        loop=0;
                    }
                    else
                    {
                        loop=1;
                    }
                    for (a=0;a<(SEC/20);a++);//debouncing
                }
                loop=1;
            }
        }
```

```
}
//program entry point for all cores
int main(){
	chan cLED[3]; //communication channels
//	uart_configure(UART_115200); //set up the uart
	par {
		on stdcore[0]: masterThread(cLED);
		on stdcore[1]: ledSlaves(cLED[0], p_cled_1);
		on stdcore[2]: ledSlaves(cLED[1], p_cled_2);
		on stdcore[3]: ledSlaves(cLED[2], p_cled_3);
	}
	return 0;
}
```

REFERENCES

[1] Hamlin, G.J.; Sanderson, A.C., "A novel concentric multilink spherical joint with parallel robotics applications," Robotics and Automation, 1994. Proceedings., 1994 IEEE International Conference on , vol., no., pp.1267-1272 vol.2, 8-13 May 1994

[2] Merlet, Jean-Pierre (2000). *Parallel Robots*. Norwell, Massachusetts: Kluwer Academic Publishers Group.

[3] A Y N Sofla; D M Elzey; H N G Wadley, "Shape morphing hinged truss structures, "Smart Materials and Structures, vol.18, no. 6, 8 pp., Jun 2009

[4] Hamlin, G.J.; Sanderson, A.C. (1998). *Tetrobot: A Modular Approach to Reconfigurable Parallel Robotics*. Norwell, Massachusetts: Kluwer Academic Publishers Group.

[5] P.C. Hughes; W.G. Sincarsin; K.A. Carroll, "Trussarm—A Variable-Geometry-Truss Manipulator, "Journal of Intelligent Material Systems and Structures, vol. 2, no. 2, pp. 148-160, April 1991

[6] Kei Senda; Hidefumi Kawano; Akihiro Ando; Yoshisada Murotsu, "Efficient formulation of inverse dynamics and control application to a planar variable geometry truss," Smart Materials and Structures, vol.8, no.6, pp.839-846, Dec 1999

[7] Wang, J., & Hy, Q. (1999). A Lagrangian Network for Kinematic Control of Redundant Robot Manipulators. IEEE Transactions on Neural Networks, 10(5), 1123.

[8] Wunderlich, Walter; Pilkey, Walter D. (2003). *Mechanics of Structures Variational and Computational Methods Second Edition.* Bora Raton, Florida: CRC Press LLC

[9] Athans, Michael; Falb, Peter L. (1966). *Optimal Control*. New York / St. Louis / San Francisco / Toronto / London / Sydney: McGRAW-HILL BOOK COMPANY.

[10] Kwakernaak, Huibert; Sivan, Raphael (1972). *Linear Optimal Control Systems*. New York / London / Sydney / Toronto: WILEY-INTERSCIENCE, a Division of John Wiley & Sons, Inc.

[11] Boltcheva, D., Yvinec, M., & Boissonnat, J. (2009). Feature preserving Delaunay mesh generation from 3D multi-material images. Computer Graphics Forum, 28(5), 1455-1464.

[12] Decayeux, C., & Semé, D. (2005). 3D Hexagonal Network: Modeling, Topological Properties, Addressing Scheme, and Optimal Routing Algorithm. IEEE Transactions on Parallel & Distributed Systems, 16(9), 875-884.

[13] Dyken, C., Reimers, M., & Seland, J. (2009). Semi-Uniform Adaptive Patch Tessellation. Computer Graphics Forum, 28(8), 2255-2263.

[14] Jilani, H., Bahreininejad, A., & Ahmadi, M. (2009). Adaptive finite element mesh triangulation using self-organizing neural networks. Advances in Engineering Software, 40(11), 1097-1103.

[15] Dereudre, D., & Lavancier, F. (2011). Practical simulation and estimation for Gibbs Delaunay–Voronoi tessellations with geometric hardcore interaction. Computational Statistics & Data Analysis, 55(1), 498-519.
[16] Peyré, G., & Cohen, L. (2006). Geodesic Remeshing Using Front Propagation. International Journal of Computer Vision, 69(1), 145-156.
[17] Tlili, S., & Mibar, H. (2009). Dynamic Output Feedback For Nonlinear Systems Using Linear Technique. AIP Conference Proceedings, 1107(1), 216-221.
[18] Anh, H. (2010). Online tuning gain scheduling MIMO neural PID control of the 2-axes pneumatic artificial muscle (PAM) robot arm. Expert Systems with Applications, 37(9), 6547-6560.
[19] Long, C., & Jin-chao, X. (2004). OPTIMAL DELAUNAY TRIANGULATIONS. Journal of Computational Mathematics, 22(2), 299-308.
[20] Jürgen Elstrodt, "The Life and Work of Gustav Lejeune Dirichlet (1805–1859), " Clay Mathematics Proceedings, vol. 7, 2007
[21] Kristof Van Laerhoven. (~1995). Voronoi Diagrams & Delaunay Triangulation. In Lancaster University Computer Science. Retrieved October 14, 2009, from http://www.comp.lancs.ac.uk/~kristof/research/notes/voronoi/index.html.

[22] H. Z. Li, Z. M. Gong, W. Lim, and T. Lippa , "Motion profile planning for reduced jerk and vibration residuals", *SIMTech Technical Reports*, Volume 8 Number 1, Jan-Mar 2007.

[23] Carl de Boor (1978). A Practical Guide to Splines. Springer-Verlag. pp. 113–114.
[24] Beauchemin, G., & Budimir, M. (2003). Microstepping myths. Machine Design, 75(19), 86.
[25] Hamlin, G.J.; Sanderson, A.C., "Tetrobot: a modular system for hyper-redundant parallel robotics ," Robotics and Automation, 1995. Proceedings., 1995 IEEE International Conference on , vol.1, no., pp.154-159 vol.1, 21-27 May 1995
[26] Hamlin, G.J.; Sanderson, A.C., "TETROBOT: a modular approach to parallel robotics," Robotics & Automation Magazine, IEEE , vol.4, no.1, pp.42-50, Mar 1997
[27] Hamlin, G.J.; Sanderson, A.C., "TETROBOT modular robotics: prototype and experiments," Intelligent Robots and Systems '96, IROS 96, Proceedings of the 1996 IEEE/RSJ International Conference on , vol.2, no., pp.390-395 vol.2, 4-8 Nov 1996
[28] Sangveraphunsiri, V., & Chooprasird, K. (2011). Dynamics and control of a 5-DOF manipulator based on an H-4 parallel mechanism. International Journal of Advanced Manufacturing Technology, 52(1-4), 343-364.
[29] Abdellatif, H., & Heimann, B. (2009). Computational efficient inverse dynamics of 6-DOF fully parallel manipulators by using the Lagrangian formalism. Mechanism & Machine Theory, 44(1), 192-207.

[30] Yasuda, G. (2003). Distributed autonomous control of modular robot systems using parallel programming. Journal of Materials Processing Technology, 141(3), 357.
[31] Bamberger, H., & Shoham, M. (2007). A Novel Six Degrees-of-Freedom Parallel Robot for MEMS Fabrication. IEEE Transactions on Robotics, 23(2), 189-195.
[32] W. L., X., Pap, J. S., & Bronlund, J. J. (2008). Design of a Biologically Inspired Parallel Robot for Foods Chewing. IEEE Transactions on Industrial Electronics, 55(2), 832-841.
[33] Guilbert, M. M., Joly, L. L., & Wieber, P. B. (2008). Optimization of Complex Robot Applications under Real Physical Limitations. International Journal of Robotics Research, 27(5), 629-644.
[34] Choi, J., Mori, O., Tsukiai, T., & Omata, T. (2004). Self-reconfigurable planar parallel robot in the horizontal plane. Advanced Robotics, 18(1), 45-60.

VITA

Graduate College

University of Nevada, Las Vegas

Christopher James Salisbury

Degrees:

Bachelor of Science, Mechanical Engineering, 2009

University of Nevada, Las Vegas

Publications:

Chris Salisbury. (2011). Dynamic Finite Element Analysis of a Highly Parallel Robotic Surface. Proceedings from SMASIS2011: The ASME 2011 Conference on Smart Materials, Adaptive Structures and Intelligent Systems, Paper 4974. Scottsdale, AZ.

Chris Salisbury, Woosoon Yim. (2011). Finite Element Analysis of a Highly Parallel Robotic Surface. Proceedings from IMECE2011: The ASME 2011 International Mechanical Engineering Congress & Exposition, Paper 63960. Denver, CO.