DESIGN AND CONTROL OF A ROBOTIC LEG WITH BRAIDED PNEUMATIC ACTUATORS

by

ROBB WILLIAM COLBRUNN

Submitted in partial fulfillment of the requirements

For the degree of Master of Science

Thesis Advisor: Dr. Roger D. Quinn

Department of Mechanical and Aerospace Engineering

CASE WESTERN RESERVE UNIVERSITY

May, 2000

To JoNancy and Boyd Colbrunn

CONTENTS

LIST OF TABLES

LIST OF FIGURES

Acknowledgements

I would like to thank my wife Joni for being supportive of this crazy idea to go get a masters degree. Without her support, I never would have been able to do it. I would also like to thank my wonderful new son Boyd for helping me put priorities into perspective.

I want to thank Roger Quinn for providing an excellent work environment. I have thoroughly enjoyed my experience working in the Biorobotics Lab. I also appreciate his patient and supportive leadership style.

Gabe Nelson has given me invaluable counsel in robotics as well as life. In this thesis, you will see evidence of all the work he has done before me to make my work possible. For this, I would like to thank him.

I want to thank Rich Bachmann, Matt Birch, Jim Berilla, and Yuan Dao Zhang for using their time and talents to help me build the robot.

I would also like to express my gratitude to the other guys in the lab for being sounding boards for my half-baked ideas and providing a good political discussion at the drop of a hat.

I want to express my appreciation to my parents for raising me and giving me direction. I was taught many of the fundamentals of engineering while tinkering in the garage with my dad.

I want to thank Dr. Joe Mansour and Dr. Steve Phillips for serving on my thesis committee.

The work was supported by the Office of Naval Research under grant number N0014-99-1-0378, and DARPA under contract number DAAN02-98-C-4027.

Finally, I want to thank God for life, and all these wonderful creatures that we study. Even with all our great technology, our understanding of the world is so limited. As an engineer, His infinite wisdom and designs humble me. Without Him, we are nothing.

Design and Control of a Robotic Leg
With Braided Pneumatic Actuators

Abstract

by

ROBB WILLIAM COLBRUNN

A Braided Pneumatic Actuator is a device developed in the 1950's by J.L. McKibben. In recent years, robotics engineers have begun to rediscover these fascinating devices, and use them as actuators for their robots. These actuators exhibit non-linear force-length properties similar to skeletal muscle, and have a very high strength-to-weight ratio. In this thesis, emphasis was placed on understanding the actuator properties so that this knowledge could be used in simulation and control of legged robots. Static and dynamic mathematical models were developed for the actuators, and verified through testing and simulation. A four-degree of freedom robotic leg was designed, constructed, and controlled. The leg provided stable, sensible forward walking for the robot, and was capable of operating 94% passively. Though these actuators have a few limitations, their muscle-like properties including high strength-to-weight ratio, passive characteristics, and self-limiting force properties make them ideal for legged robots.

Chapter I: Introduction

1.1 Background

Much ink has been used to explain the benefits of legged locomotion over wheeled locomotion for robots. As with most things, there is always a trade off. A wheeled robot might go faster and be easier to build, but tell it to climb a flight of stairs and its shortcomings become apparent quickly. Upon observation of all the creatures around us, we see the legged design as the locomotion method of choice. The purpose of this thesis, however, is not to show why it should be done, but how it can be done.

Legged locomotion is the beginning of the biologically inspired approach to robotics. However, putting legs on a frame is not enough to have a walking robot. Arguably, the most important part of any motion device is the mechanism that does the moving: the actuators. In the past, CWRU walking robots have used DC motors to generate leg movement. With the move toward physical autonomy (operation without an umbilical), the poor strength to weight ratio of DC motors became far too prohibitive. The next generation of robot used pneumatic cylinders to increase this ratio. Air cylinders' strength to weight ratio is much better than electric motors, but there is still room for improvement. The total weight of Robot III was 29.5 lbs. with 76% of that being actuator and valve weight (Bachmann, 2000). A lighter weight actuator that was just as strong would increase the robot's payload capacity. The biologically inspired approach led the group to look at the prime movers that are designed into natural walking "machines": muscles. Using real muscle tissue is technologically not practical at this time (Shimoyama, 1997), so an artificial muscle is necessary. This actuator is known as a Braided Pneumatic Actuator or a McKibben Artificial Muscle. These actuators can have

a power-to-weight ratio as high as 25 times greater than DC motors (Shadow, 2000). These actuators also have experimentally been shown to have force-length properties similar to skeletal muscle (Klute et al.,1999).

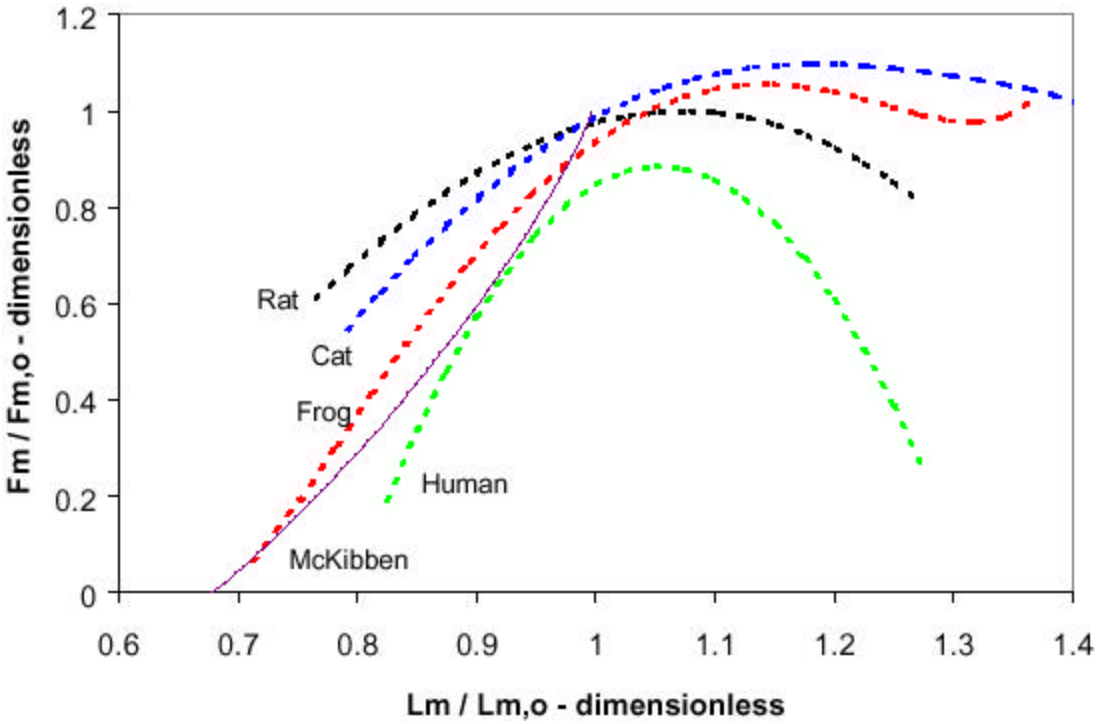


Figure 1-1: The dimensionless relationship between force and length under isometric conditions at maximal activation for various animals as well as a McKibben actuator pressurized to 5 bar (Klute et al.,1999).

This muscle-like stiffness and their structural flexibility present real advantages over air cylinders. In this thesis, these braided pneumatic actuators will be referred to as "McKibbens" or BPA.

1.2 Purpose and Overview

The focus of this project involves the investigation of using McKibbens as the prime mover in a walking robot. This work is divided into six sections. Each chapter builds upon the other to explain the process and methods of using BPA to make a robotic leg walk. Chapter two explains all the properties of the BPA as well as the mathematical

model formulated for simulation and control. Chapter three describes the simple spring-mass-damper simulation that was written to dynamically validate the model. This simulation included a model of the actuator, and a model of a solenoid valve for controlling the mass of air in the actuator. The valve states were the only inputs, and the simulation calculated the remaining system properties. The next chapter describes all the hardware involved in making the robot walk. A system overview, the valves, the sensors, and the leg design are all presented. Chapter five delves into the control program and how the sensor feedback is turned into valve commands. Then, chapter six presents the results or quantification of the walking. The last chapter discusses the limits and shortcomings of using McKibbens. Finally, possible solutions and improvements are addressed as topics of future work.

Chapter II: Braided Pneumatic Actuator

2.1 Physical Characteristics

A BPA is a pneumatic device developed in the 1950's as an orthotic appliance for polio patients by J.L. McKibben (Nickel et al., 1963). It consists of a rubber bladder encompassed by a tubular braided mesh. When the bladder is inflated, the actuator expands radially and undergoes a lengthwise contraction. Figure 2-1 is a picture of an inflated (bottom) and uninflated (top) actuator.

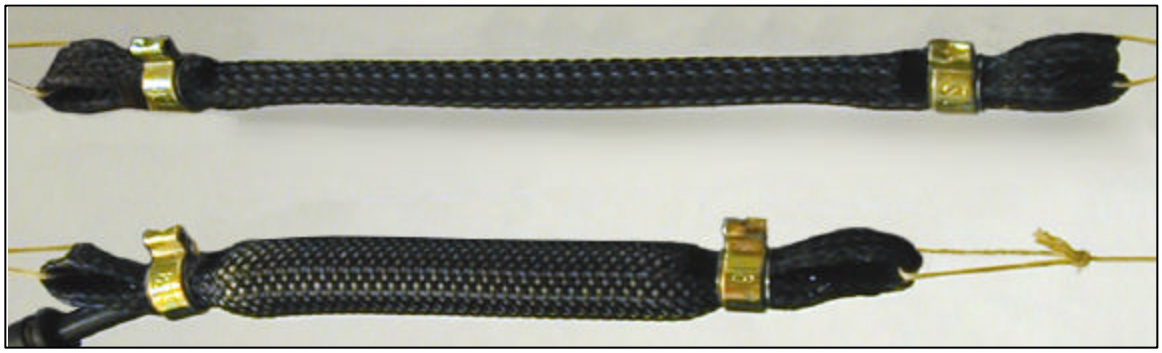


Figure 2-1

The mesh of the actuators in Figure 2-1 is made of nylon and is ¼" in diameter. It is the same mesh that is used for flexible electrical conduit. The bladder is a 0.22"∅ID x 0.035" wall-thickness latex rubber tube. The proximal end of the tube contains the 5/32" hose that connects the actuator to the valves. The distal end of the bladder is sealed with a small Plexiglas rod. The mesh is then looped on both ends, dipped in an epoxy, and clamped tight to provide an air tight seal. Shadow Robot Company of London, England manufactured the actuators used in this work. Figure 2-2 contains the overall dimensions for these actuators.

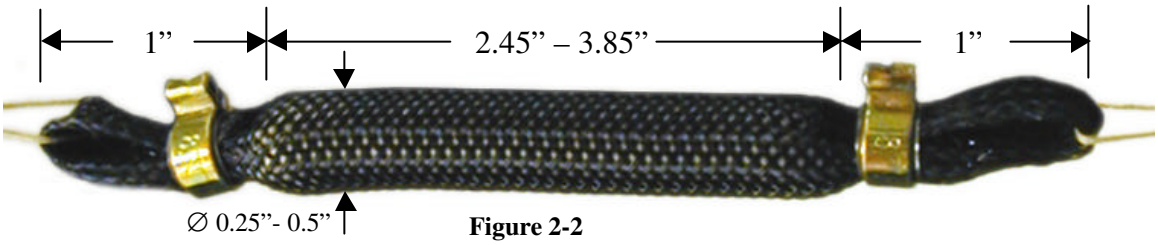


Figure 2-2

2.2 Geometric and Static Model

The formulation work described below builds upon the work of Ching-Ping Chou and Blake Hannaford (1996). It also uses a geometric method similar to the one presented by Darwin Caldwell, Gustavo Medrano-Cerda, and Mike Goodwin (1995). Chou and Hannaford developed a static model for the actuators using a virtual work argument. A similar argument will be used, but the final form of the equations will better suit the needs of modeling and controlling them. This will provide a relationship between actuator force, pressure, and length. Next, these equations will be used to derive further relationships between force, pressure, length, and stiffness. These relationships will then be verified and fine-tuned with empirical data. From there, a simple dynamic model of the actuators will be presented for use with a dynamic simulation.

A BPA can be modeled as a cylinder. The non-cylindrical end effects are ignored, and the wall thickness is assumed to be zero. The dimensions of this cylinder are the length, L, and diameter, D. Neither of these dimensions remain constant. Assuming inextensibility of the mesh material, the geometric constants of the system are the thread length, b, and n, the number of turns for a single thread. The final dimension used for this formulation is the interweave angle, θ. Theta is the angle between the thread and the long axis of the cylinder. The interweave angle changes as the length of the actuator changes. The relationship between these parameters is shown in Figure 2-3.

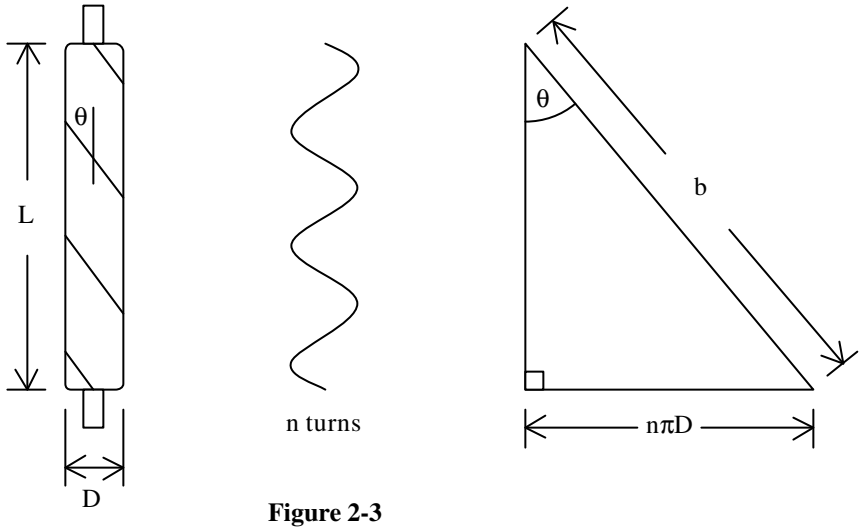


Figure 2-3

The mathematical relationships for length and diameter are as follows:

$$L = b\cos\theta \qquad [1]$$

$$D = \frac{b\sin\theta}{n\pi} \qquad [2]$$

The volume of any cylinder is equal to its length times the cross sectional area.

$$V = \frac{\pi D^2}{4} L \qquad [3]$$

Substituting equations [1] and [2] into [3] yields:

$$V = \frac{b^3}{4\pi n^2}\sin^2\theta\cos\theta \qquad [4]$$

The maximum contracted length (minimum length) occurs when the actuator volume is at its greatest. This results in equilibrium of the system. Taking a derivative of the volume with respect to theta, $dV/d\theta$, and setting it equal to zero yields the maximum interweave angle.

$$\frac{dV}{d\theta} = \frac{b^3}{4\pi n^2}\left(2\sin\theta\ \cos^2\theta - \sin^3\theta\right) = 0 \qquad [5]$$

$$\theta_{max} = 54.7^\circ \qquad [6]$$

Now that the geometry is established, a force as a function of pressure and length will be developed. This approach uses a simple energy analysis. The assumption is made that it is a conservative system in which the work in (W_{in}) is equal to the work out (W_{out}). The losses will be accounted for later. Work is input to the actuator when the air pressure moves the inner bladder surface.

$$dW_{in} = \int_{S_i} \left(P_{abs} - P_{atm}\right) dl_i \cdot ds_i = \left(P_{abs} - P_{atm}\right) \int_{S_i} dl_i \cdot ds_i = P_g \, dV \qquad [7]$$

Where: P_{abs} = Absolute internal gas pressure
P_{atm} = Atmospheric pressure (14.7 psi)
P_g = Gage pressure
S_i = Total inner surface
ds_i = Area vector
dl_i = Inner surface displacement
dV = Volume change

The output work occurs when the actuator shortens due to the change in volume.

$$dW_{out} = -F \, dL \qquad [8]$$

The ideal system assumption can now be applied. The work input to the system should be equal to the work done by the actuator.

$$dW_{in} = dW_{out} \qquad [9]$$

Substituting [7] and [8] into [9],

$$P_g \, dV = -F \, dL \qquad [10]$$

$$F = -P_g \frac{dV}{dL} \qquad [11]$$

Using the geometry that was established above, an equation for force as a function of pressure and interweave angle can now be developed.

$$F = -P_g \frac{dV}{dL} = -P_g \frac{dV/d\theta}{dL/d\theta} = \frac{P_g b^2 \left(2\cos^2\theta - \sin^2\theta\right)}{4\pi n^2} \qquad [12]$$

Thus, we have an equation for force as a function of P_g and θ,

$$F = \frac{P_g b^2 \left(3\cos^2\theta - 1\right)}{4\pi n^2} \qquad [13]$$

Note that at the maximum interweave angle, 54.7°, the force output of the actuator is zero. The geometric variables used above provide a straightforward formulation, but to use the resulting equations in practice, they first need to be modified. The first step is to develop a method to accurately measure the braid length, b, and count the non-integer number of thread wraps. This method was first developed by Caldwell, Medrano-Cerda, and Goodwin (1995). If the cylindrical mesh is opened and laid flat, the trapezoidal geometry is easily observed (Figure 2-4). The shape of the trapezoid is governed by the interweave angle, θ, and the length of the trapezoid side, ℓ.

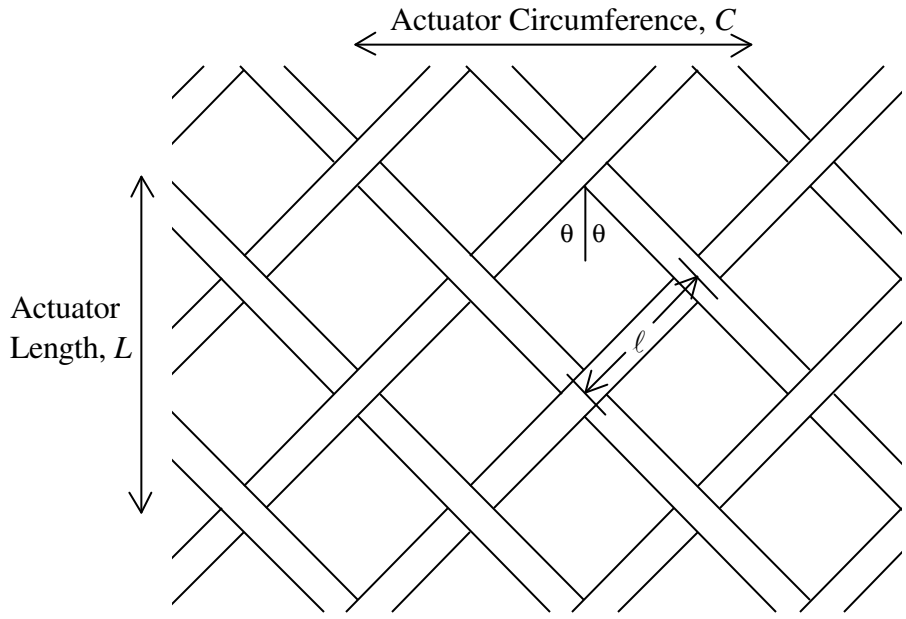


Figure 2-4

The overall length and circumference of the actuator are given by the following relationships:

$$L = 2A\ell\cos\theta \quad [14]$$

$$C = 2B\ell\sin\theta \quad [15]$$

Where: A = Number of lengthwise trapezoids
B = Number of circumferencial trapezoids (around actuator)

Since the diameter is proportional to the circumference, this becomes:

$$D = \frac{2B\ell}{\pi}\sin\theta \quad [16]$$

Recall equations [1] and [2]

$$L = b\cos\theta$$

$$D = \frac{b\sin\theta}{n\pi}$$

Setting the length ([1] and [14]) and diameter ([2] and [16]) equations equal yields,

$$b = 2A\ell \quad [17]$$

$$n = \frac{A}{B} \quad [18]$$

Therefore, to practically characterize an actuator only the trapezoid size and count are necessary. The next step in making these equations practical in control is to remove θ from the equations. It is difficult to sense the interweave angle during operation of the actuator. It is much easier to measure the length. If the equations can be rewritten in terms of force, pressure, and length, then they will be more useful, because these variables can be measured most easily. Recall the triangle from Figure 2-3 and note that the equation for the side opposite θ has been rewritten in terms of b and L.

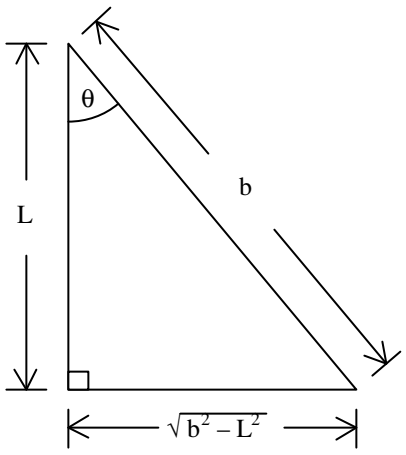


Figure 2-5

Now relationships can be expressed between θ, L, and b.

$$\cos\theta = \frac{L}{b} \qquad [19]$$

$$\sin\theta = \frac{\sqrt{b^2 - L^2}}{b} \qquad [20]$$

Substituting these relationships into the volume [4] and force [13] equations,

$$V = \frac{L\left(b^2 - L^2\right)}{4\pi n^2} \qquad [21]$$

$$F = \frac{P_g b^2}{4\pi n^2}\left(\frac{3L^2}{b^2} - 1\right) \qquad [22]$$

BPA exhibit the properties of a variable stiffness spring. This is beneficial to the control of a legged robot. Joint angle is a relatively easy quantity to measure. However, stiffness is not. Therefore a model of actuator stiffness was developed. Stiffness is simply a derivative of force with respect to length.

$$k = \frac{dF}{dL} \qquad [23]$$

Differentiating [22] with respect to L,

$$k = \frac{b^2}{4\pi n^2}\left(\frac{3L^2}{b^2} - 1\right)\frac{dP_g}{dL} + \frac{3P_g L}{2\pi n^2} \qquad [24]$$

The first term (dP_g/dL) is the most difficult to formulate. When the valves are closed, the pressure changes proportionally with volume according to gas laws. As a result, the relationship dV/dL would need to be developed. However, when the valves are opened, this relationship (dP_g/dL) is even more difficult to model. Fortunately, the pressure change as a function of length is small, and can be neglected. If the external volume to the actuator (e.g. in the inlet hose) is roughly the same or more than the maximum internal actuator volume, then the total volume is enough to cause the pressure change to remain minimal through the actuators' range of motion. In this case we can assume:

$$\frac{dP_g}{dL} \approx 0 \qquad [25]$$

Actuator stiffness is now given by:

$$k = \frac{3P_g L}{2\pi n^2} \qquad [26]$$

Or solving [22] for P_g and substituting the result into [26],

$$k = \frac{6F}{\left(3L - \frac{b^2}{L}\right)} \qquad [27]$$

Figure 2-6 is a plot of theoretical actuator force and stiffness as a function of internal pressure ([22] and [26]) for the range from 20 to 110 psi. Force is a nonlinear function of length, tending to zero at maximum contraction, and stiffness is a linearly increasing function of length and pressure.

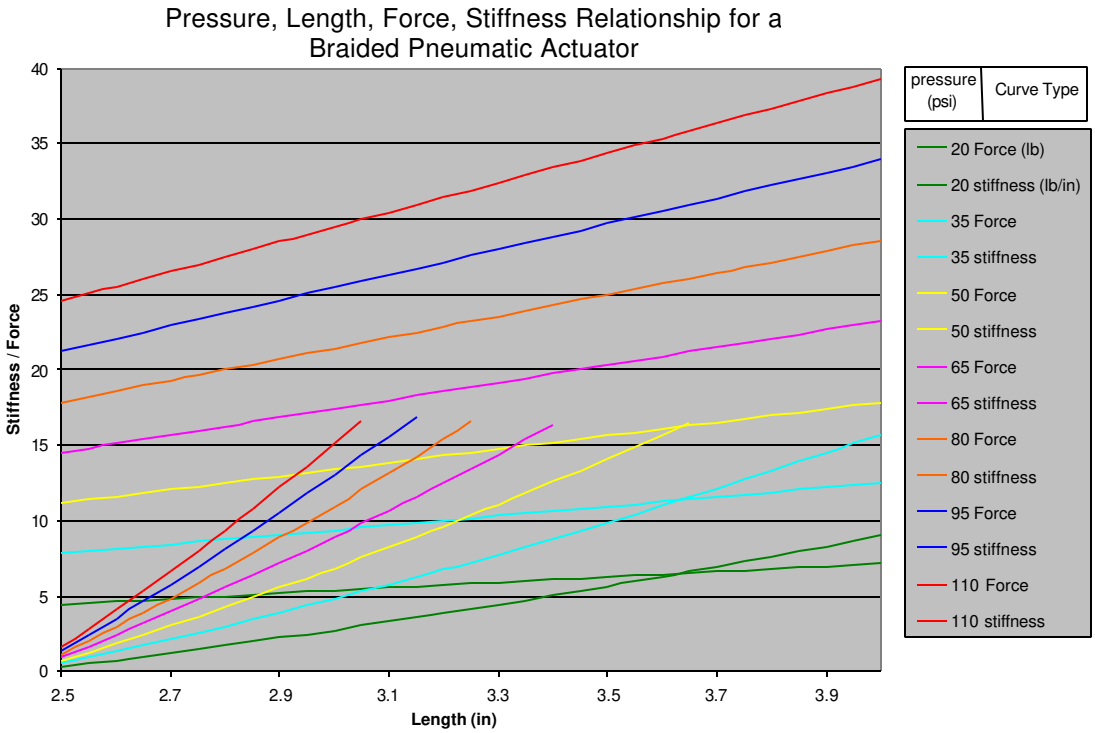
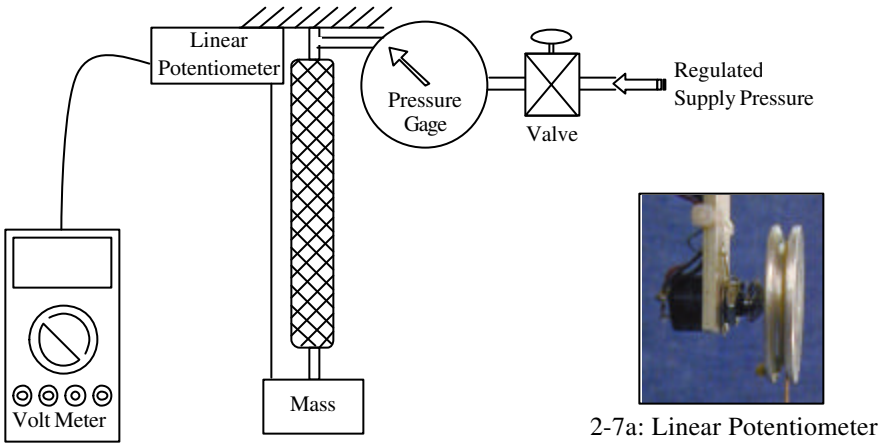


Figure 2-6

To determine actuator state, only two of the following pieces of information are needed: length, pressure, force, or stiffness. The other two will be defined by Figure 2-6.

2.3 Static Model Verification

Chou and Hannaford (1996), using constant pressure testing, have verified the model of force as a function of pressure and length [13]. In the control of the robotic leg described in this thesis, internal air mass is the variable being controlled instead of pressure. Therefore, it was decided to take data for a constant mass system. Admittedly, the mass-pressure relationship may simply be related by the ideal gas law, but this became another assumption to verify. Also, the dynamic simulation and its verification were performed using the same constant mass assumption. It is much easier to close a valve than to build an infinitely fast pressure regulator. Figure 2-7 contains the block diagram for the static testing configuration.



2-7a: Linear Potentiometer

Figure 2-7

The linear potentiometer was a custom built transducer consisting of a rotary potentiometer, an aluminum wheel, a torsional spring, and a Kevlar cord (Figure 2-7a). The error in the force measurement due to the linear potentiometer was negligible. The actuator was inflated to some nominal pressure, and the valve was then closed. Mass was then added in 1 lb increments to a maximum of 15 lbs and then decreased to zero in 1 lb increments. The suspended mass, actuator length, and internal pressure were all recorded during each step of the test. Four separate tests were run, each with a different nominal pressure. Figure 2-8 is a plot of the force-length relationship at each nominal pressure.

The force output of a BPA is a non-linear function of length. The closed curves in Figure 2-8 also show the hysteretic behavior of the actuators. This hysteresis is a by-product of the coulomb friction between the bladder and the mesh (Chou and Hannaford, 1996). The next step is to develop a model that matches the empirical data. To incorporate this constant mass model, the effect of the change in internal pressure must be included. The increased internal pressure may be predicted according to the ideal gas law.

$$PV = nRT \qquad [28]$$

Considering mass is constant and assuming T is constant,

$$PV = \text{constant} \qquad [29]$$

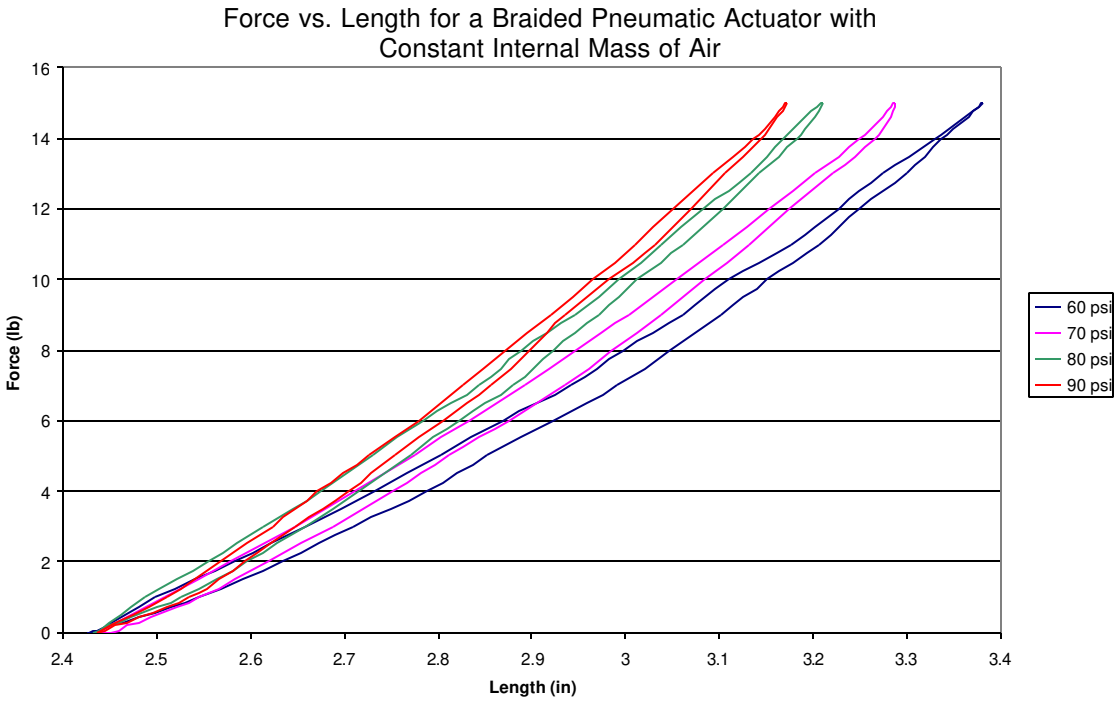


Figure 2-8

Figure 2-9 is a comparison of the measured internal pressure and the pressure curve predicted by [29].

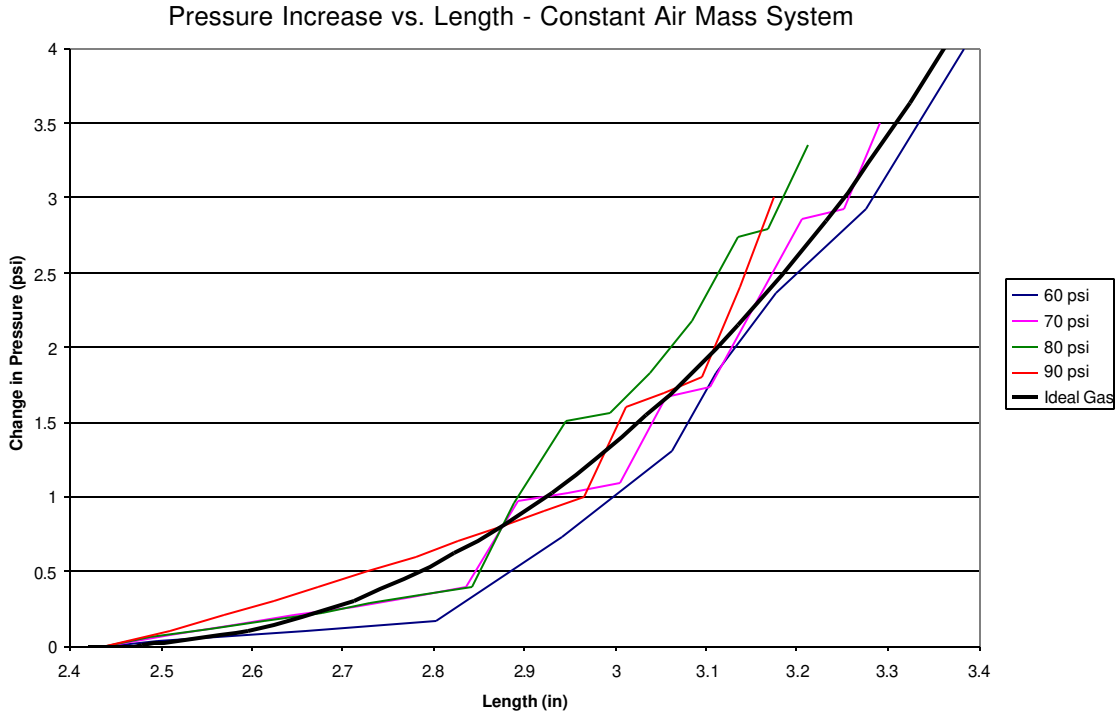


Figure 2-9

Figure 2-9 suggests the predicted pressure increase is valid. The imperfection of the empirical data is likely due to the coarseness of the pressure gage measurements. Now that the pressure change is verified, the next step is to compare the force output. Figure 2-10 is a plot of predicted vs. experimental force output at the nominal pressures of 60 and 90 psi.

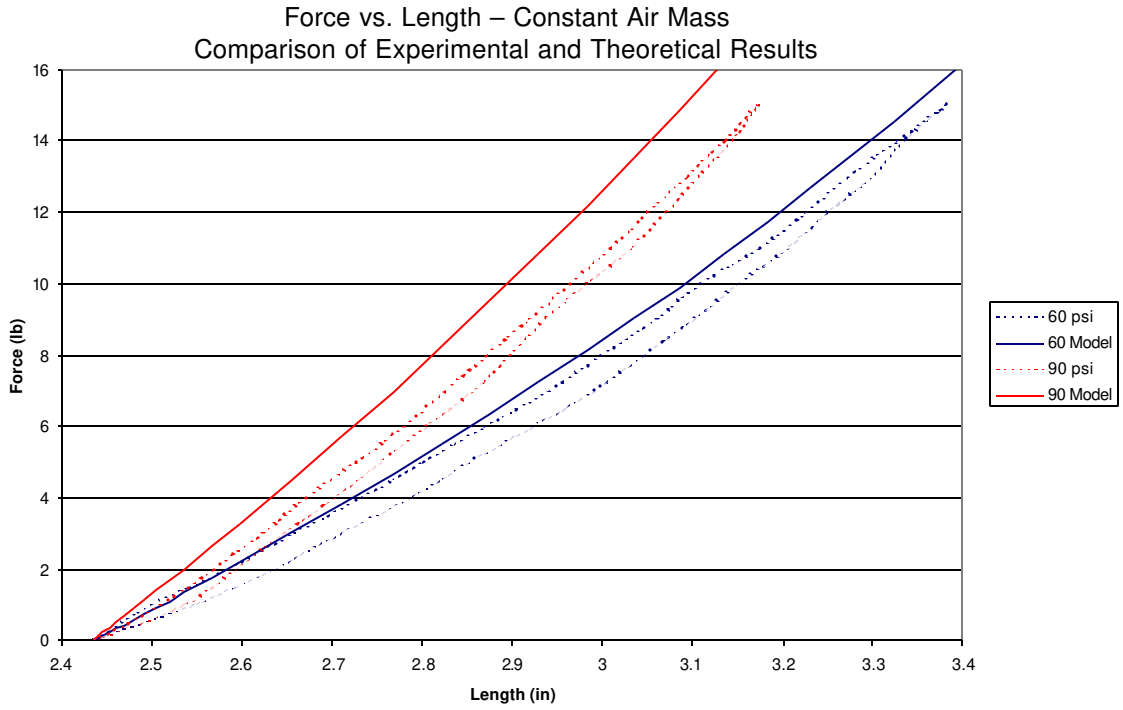


Figure 2-10

The model does not predict the hysteresis nor does it perfectly match the experimental data. The model predicts a higher force output than what was experimentally measured. Therefore, an effectiveness term is introduced into the model. Effectiveness is the measure of what percentage the actual force is to the predicted. Figure 2-10 also suggests that effectiveness is a function of pressure.

$$F_{act} = Eff\left(P_g\right) \cdot F_{theoretical} \qquad [30]$$

Effectiveness was determined for each nominal pressure, and a relationship for effectiveness as a function of pressure was created. (Figure 2-11)

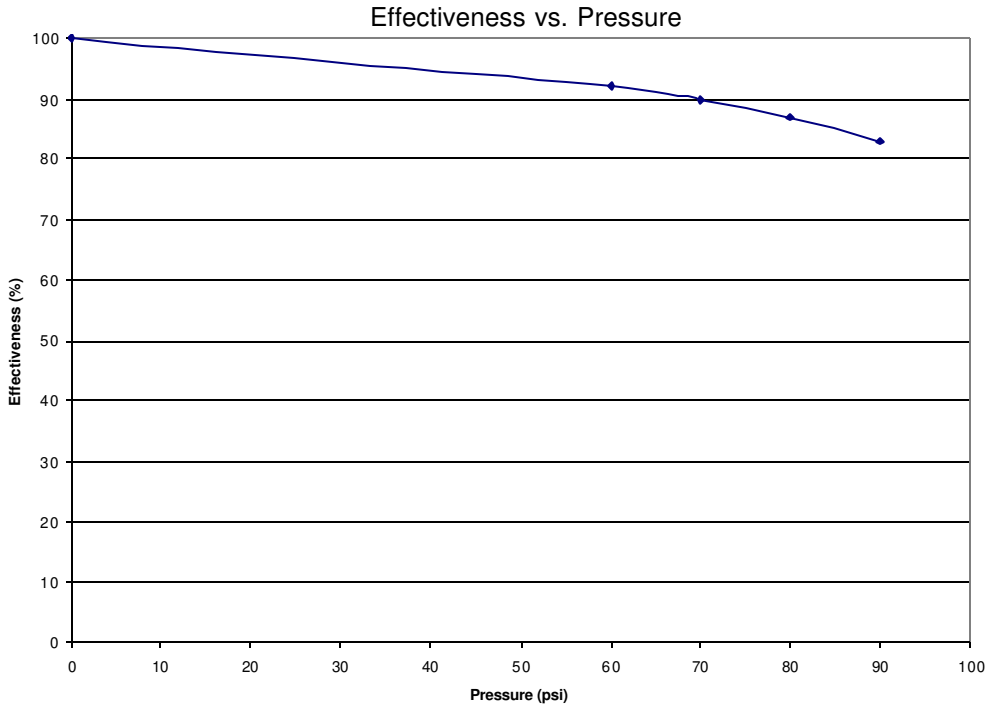


Figure 2-11

After applying the effectiveness function to the model, the experimental and theoretical data sets were compared again (Figure 2-12).

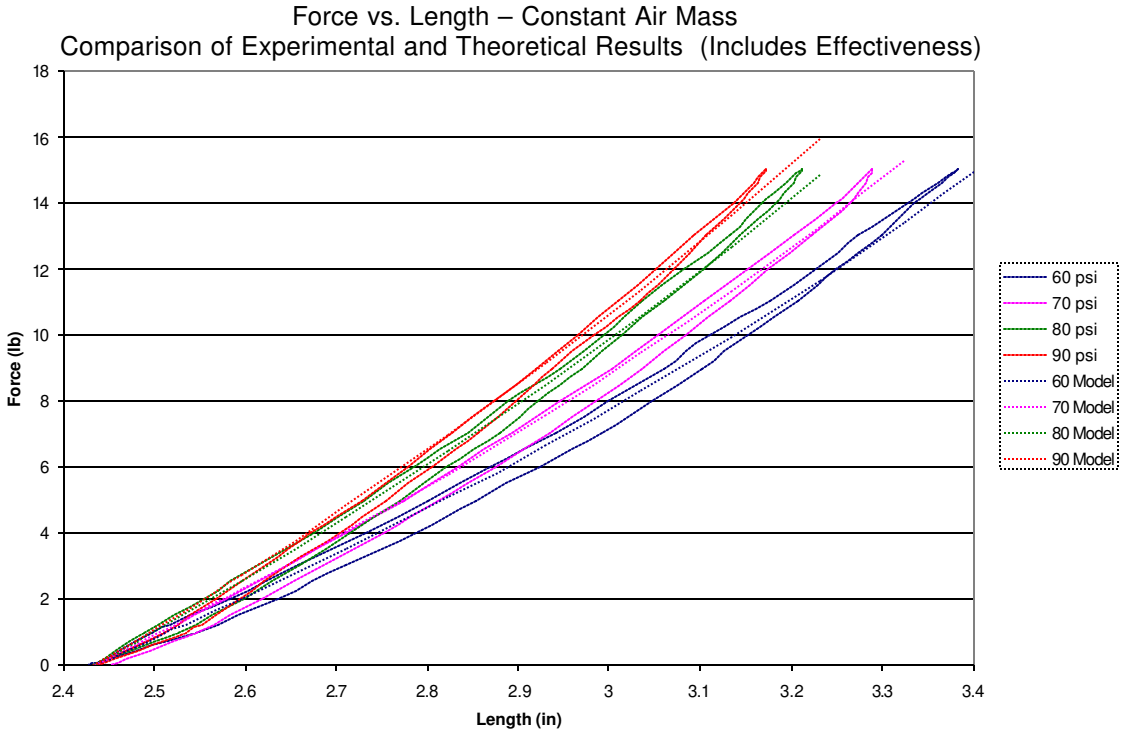


Figure 2-12

Figure 2-12 suggests the static model presented may be an adequate representation of the actuator force output.

The final additions to the model are the end effects. The end effects are changes in force output at the length limits of the actuator. The long end effect occurs when the actuator is at its maximum length. At this point, any increase in length would cause the braids to stretch. Since the stiffness of the braiding material is very high compared to the stiffness of the actuator, the long end effect can be modeled as a spring of very high stiffness when the actuator reaches a maximum length. Figure 2-13 is a plot of force vs. length for a constant pressure system. This data was taken by Shadow Robot Company (1999) for a larger actuator, but it shows the long end effect.

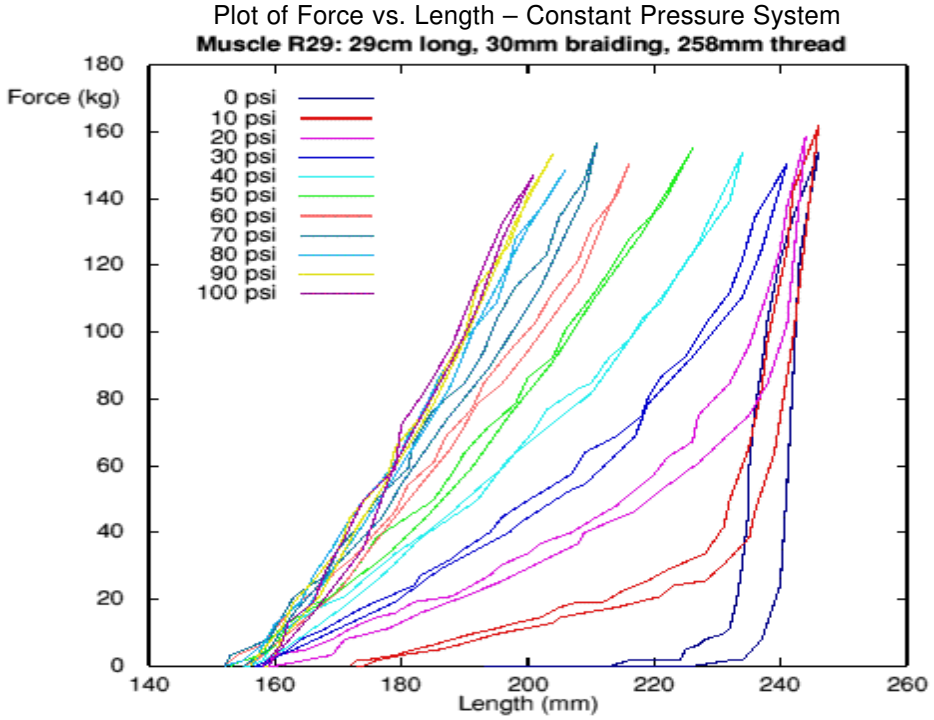


Figure 2-13

The actuator, like a muscle, can only pull. The model [22] predicts that if the length is less than the maximum contracted length, then the force output will be negative (pushing). Thus, a short end effect is added to the model. It simply states that if the

length is less that the minimum length, the force output becomes zero. Incorporating the effectiveness and the end effects into [22] yields:

$$F = \begin{cases} \dfrac{P_g b^2}{4\pi n^2}\left(\dfrac{3L^2}{b^2}-1\right)\cdot Eff\left(P_g\right)+ F_{\max\,\text{limit}} & if\left(L > L_{\min}\right) \\ 0 & if\left(L < L_{\min}\right) \end{cases} \quad [31]$$

Where:

$$F_{\max\ \text{limit}} = \begin{cases} K_{braid}\left(L - L_{\max}\right) & if\left(L > L_{\max}\right) \\ 0 & if\left(L < L_{\max}\right) \end{cases} \quad [32]$$

K_{braid} = Braid material stiffness

Equation [31] is the static model for a BPA. This model ignores bladder thickness, bladder thickness variation, friction-induced hysteresis, and non-linear elastic energy storage of the bladder. Klute and Hannaford (1999) have developed a more accurate model using these factors, but the cost of the more accurate model is much longer equations and therefore more computational time. This model is adequate and yet simple enough that it is suitable for dynamic simulations and control.

2.4 Dynamic Model

The dynamic model of the BPA is broken into three parts. It is modeled as a spring in parallel with a viscous damper and coulomb friction. The spring has a nonlinear force-length relationship given by [31]. The viscous damper models the viscous effects of the fluid flow losses in the system. The coulomb friction models the losses due to the sliding contact between the mesh and bladder (Figure 2-14).

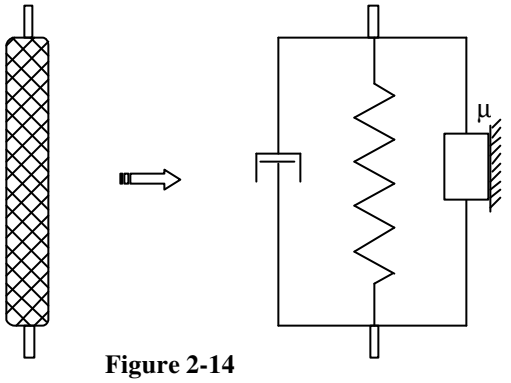


Figure 2-14

$$F = \begin{cases} \dfrac{P_g b^2}{4\pi n^2}\left(\dfrac{3L^2}{b^2}-1\right)\cdot Eff\left(P_g\right)+F_{\max\,\text{limit}}+c\cdot v\pm Q\cdot k & if\left(L>L_{\min}\right) \\ 0 & if\left(L<L_{\min}\right) \end{cases} \quad [33]$$

$$k = \frac{\left(\dfrac{P_g b^2}{4\pi n^2}\left(\dfrac{3L^2}{b^2}-1\right)\cdot Eff\left(P_g\right)\right)-F_{\min}}{L-L_{\min}} \quad [34]$$

Where: c = Viscous damping constant
v = Actuator tip velocity
k = Actuator stiffness (linear assumption)
$Q = \dfrac{\mu N}{k}$ = Coulomb damping constant

Equation [34] is simply a linear stiffness equation (ΔF/ΔL). F_{min} is the force output at L_{min}. Which, according to [13], is always zero. Viscous damping of the mesh material was also added through its inclusion in the $F_{max\,limit}$ equation [32].

$$F_{\max\ \text{limit}} = \begin{cases} K_{braid}\left(L-L_{\max}\right)+C_{braid}v & if\left(L>L_{\max}\right) \\ 0 & if\left(L<L_{\max}\right) \end{cases} \quad [35]$$

Where: C_{braid} = Braid viscous damping constant

All the damping constants were experimentally determined. To verify this dynamic model and determine the damping constants, a computer simulation was developed and compared with experimental results.

Chapter III: Simulation

3.1 Simulation Overview

The purpose of the dynamic simulation was not only to verify the dynamic model of a Braided Pneumatic Actuator, but also to create a generic dynamic model of a Braided Pneumatic Actuator working in conjunction with a solenoid valve. The input to the model was simply the valve state (i.e. open to supply, open to atmosphere, chamber sealed), and the output was actuator force. Figure 3-1 is a schematic of the simulation flow.

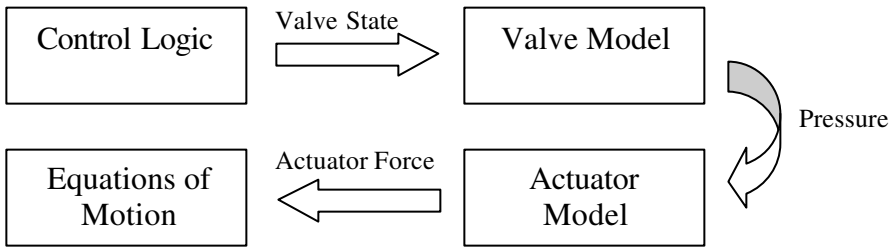


Figure 3-1

The right side of Figure 3-1 is a very valuable tool for use in any dynamic simulation of a robot using Braided Pneumatic Actuators. This model can be integrated into the equations of motion to determine joint torques and provide accurate results. It also is a very efficient model, which makes it useful for genetic algorithms and other iterative problem solving.

Figure 3-1 is an overview of the simulation but a more detailed flowchart is needed to understand the method (Figure 3-2). The simulation has four state variables: length, L, velocity, $\dot{L}$, pressure, P, and air mass, m. To determine the next system state, the Runge-Kutta numerical integration algorithm was applied to the equations of motion. The valve state is calculated at each time step according to the system state or the time. For all the simulations performed for this thesis, the control system was a feed-forward

time dependent control law. However, the simulation is set up for feedback if that is desired.

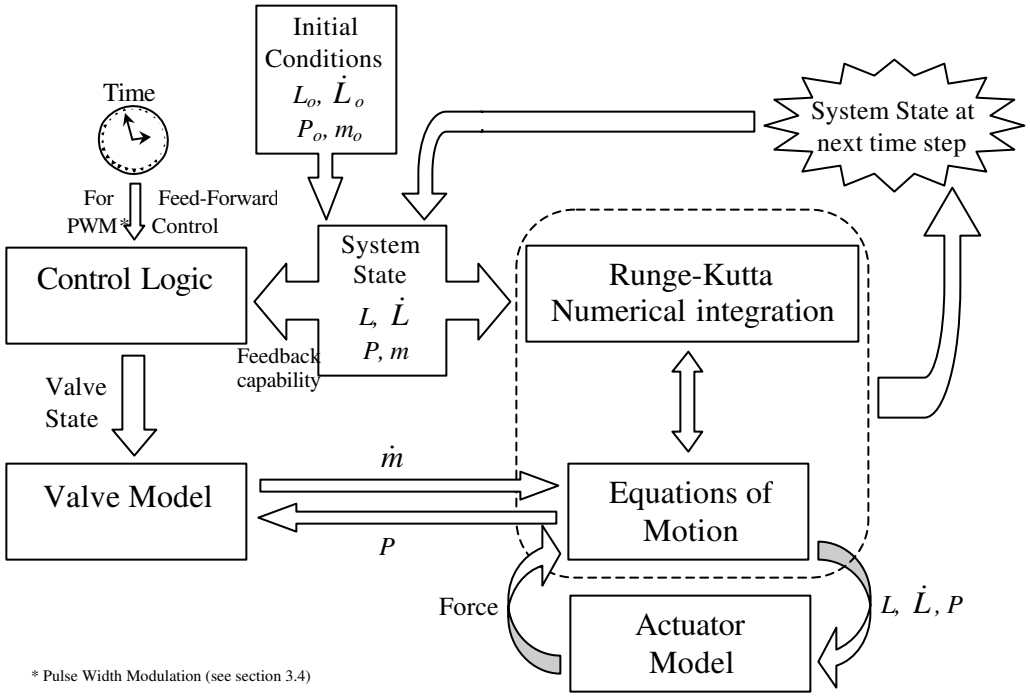


Figure 3-2

The equations of motion are used to determine the time derivatives of the state variables for Runge-Kutta integration. Each Runge-Kutta iteration re-calculates the equations of motion using both the actuator and valve models.

3.2 Equations of Motion

The system controlled in this dynamic model verification was a one DOF mass-spring-damper system where the spring force is a function of the mass of air in the actuator. Therefore, there are two different equations of motion for this system. One equation defines the movement of the mass, M, and the other defines the movement of the air. The first equation of motion was formulated for the motion of the mass, M, as a function of actuator and gravity forces (Figure 3-3).

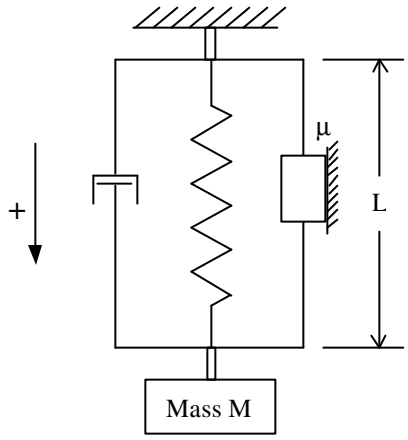


Figure 3-3

The equation of motion for this one dimensional system is quite simple. The first step is to sum the forces and apply Newtons 2nd law.

$$\sum F = -F_{actuator} + Mg = M\ddot{L} \qquad [36]$$

Solving for $\ddot{L}$,

$$\ddot{L} = g - \frac{F_{actuator}}{M} \qquad [37]$$

The second equation of motion is a state relationship of the air in the system (i.e. time derivative of pressure as a function of air mass, mass flowrate, volume, and volume flowrate). Gabe Nelson (in press) developed this formulation for use in his simulation of Robot III at the Case Western Reserve University Biorobotics Laboratory, and it is true for any variable volume and variable mass system. The first step is to take the time derivative of the standard mass-volume-density relationship.

$$\frac{d}{dt}\left(\rho = \frac{m}{V}\right) \rightarrow \dot{\rho} = \frac{\dot{m}}{V} - m\frac{\dot{V}}{V^2} \rightarrow \dot{\rho} = \frac{\dot{m}}{V} - \rho\frac{\dot{V}}{V} \qquad [38]$$

Where: ρ = Density
V= Volume
m = Air Mass

The next step is to take the time derivative of the constant pressure-density

relationship. This is based on an adiabatic flow assumption.

$$\frac{d}{dt}\left(P\rho^{-\gamma} = \text{constant}\right) \rightarrow \frac{\dot{P}}{\rho^{\gamma}} - \frac{\gamma P\dot{\rho}}{\rho^{\gamma+1}} = 0 \rightarrow \dot{P}\rho = \gamma P\dot{\rho} \quad [39]$$

Where: P = Absolute Pressure
$\gamma = 1.4$ specific heat ratio of air

Substituting [38] into [39],

$$\dot{P} = \gamma \frac{P}{\rho}\left(\frac{\dot{m}}{V} - \rho\frac{\dot{V}}{V}\right) \rightarrow \dot{P} = \gamma P\left(\frac{\dot{m}}{m} - \frac{\dot{V}}{V}\right) \quad [40]$$

Where: $\dot{V}$ = Volume flowrate [41]
$\dot{m}$ = Mass flowrate (given by valve model)

The volume flowrate is the time derivative of the geometry formulated volume equation [21].

$$\dot{V} = \frac{\left(b^2 - 3L^2\right)}{4\pi n^2}\dot{L} \quad [41]$$

It is important to note that the volume in [40], V, is a summation of the actuator volume and the external volume (i.e. hoses), not just the volume given by [21].

The two equations of motion, [37] and [40], provide the relationships necessary for numerical integration in the simulation. To complete [40], however, the valve model needs to be formulated.

3.3 Valve Model

The formulation of the valve model presented below is included for completeness, but much of the work was done by Gabe Nelson (in press). In addition to the equation of motion presented above, Nelson also incorporated this model into the Robot III simulation. The valve model is used to determine the mass flowrate $\dot{m}$. The mass flowrate through a valve can be expressed by the following equation (McCloy et al., 1980;Ye et al., 1992).

$$\dot{m} = \frac{A_p P_{up} C_q C_m}{\sqrt{T_{am}}} \qquad [42]$$

Where: A_p = Air passage area of solenoid valve
P_{up} = Upstream air pressure (absolute)
C_q = Flow rate coefficient
C_m = Flow rate parameter
T_{am} = Temperature of upstream air

C_q is an empirically determined efficiency term. It can be expressed as,

$$C_q = \frac{\dot{m}\, measured}{\dot{m}\, calculated\ with\ C_q = 1} \qquad [43]$$

C_m changes depending on whether the flow through the valve is choked or not. It is defined by the following;

$$C_m = \begin{cases} \sqrt{\dfrac{\gamma}{R}\left(\dfrac{2}{\gamma+1}\right)^{\frac{\gamma+1}{\gamma-1}}} & \dfrac{P_{down}}{P_{up}} \le 0.528 \left(choked\right) \\ \sqrt{\dfrac{2\gamma}{R(\gamma-1)}\left[\left(\dfrac{P_{down}}{P_{up}}\right)^{\frac{2}{\gamma}} - \left(\dfrac{P_{down}}{P_{up}}\right)^{\frac{\gamma+1}{\gamma}}\right]} & \dfrac{P_{down}}{P_{up}} > 0.528 \left(not\ \ choked\right) \end{cases} \qquad [44]$$

Where: P_{down} = Downstream air pressure (absolute)
$R = 53.34\ \dfrac{ft\ lbf}{lbm\ ^{\circ}R}$ air constant

Nelson used the above model and evaluated the empirically determined constants. The valve, an eight-channel three-way 758 series valve manufactured by Matrix S.p.A. of Italy, was used in Robot III and in the verification of the dynamic actuator model. Due to the complex geometry of the solenoid valves, the A_p term was lumped with the C_q term and backed out of the pressure-flowrate plot given in valve literature specifications (Figure 3-4). Nelson concluded that $A_pC_q=1.8x10^{-5}$ ft^2 for these valves.

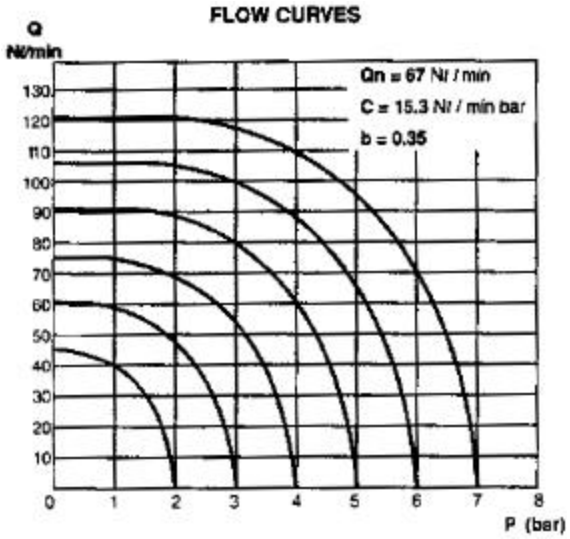


Figure 3-4

3.4 Dynamic Model Verification

The dynamic model was verified using the setup shown in Figure 3-5.

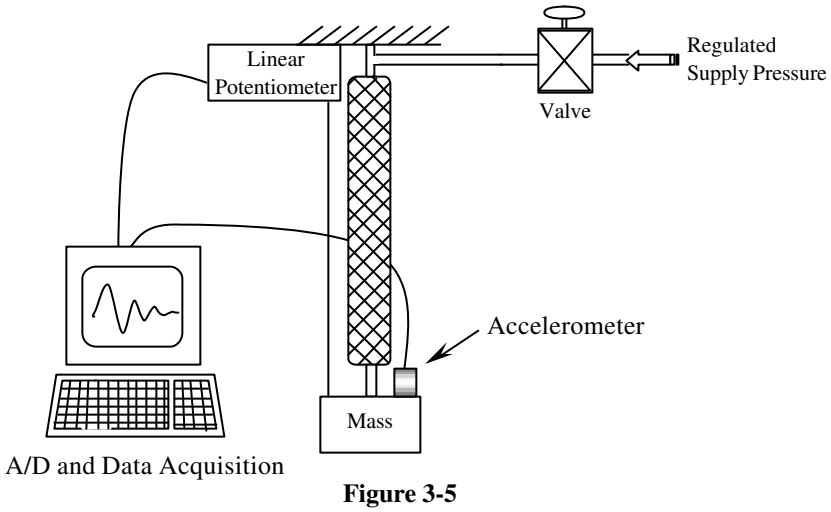


Figure 3-5

This setup was very similar to the static verification configuration. An accelerometer was added to the mass, and the data was recorded with a PC and the commercial package Data Physics. For the first part of the verification, a trapped air, constant mass system was tested. This provided a means to determine the damping constants, without using the valve model. The second part of the verification incorporated the valve model and compared the results. The three-way valve was simulated and controlled using PWM at a 50% duty cycle and various frequencies.

There are multiple flow control techniques for solenoid valves. The method chosen for this robot was Pulse Width Modulation (PWM). PWM control technique is the generation of a constant frequency square wave with variable pulse duration. The valve is commanded open at constant intervals, and then closed at a variable time later. The length of this pulse divided by the total time between open commands is termed the duty cycle. Duty cycle values range from 0-100%, where 0% is when the valve is never opened, and 100% commands the valve to always remain open.

The constant mass tests were executed by inflating the actuator to the nominal pressure, and closing the valve. Then, the mass was given an initial displacement and released. Multiple tests were run at each configuration, and the better tests were analyzed to determine the damping constants. (i.e. less horizontal swaying than another. This is why some plots below are labeled trial 1, 2, or 3). To determine the damping constants, a form of the damping equation was developed. It was assumed that the damping had two components: viscous and coulomb. Equation [45] defines the relationship between the decrease in amplitude over one period and the damping components (Rao, 1995).

$$\Delta x = 4\frac{\mu N}{k} + X\left(1 - \frac{1}{e^{\zeta \omega_n T_d}}\right) \qquad [45]$$

Where: Δx = change in amplitude over one period

$\frac{\mu N}{k} = Q$ = Coulomb damping constant

X = Initial Amplitude

ζ = Damping ratio

ω_n = Natural Frequency

T_d = Damped Period

There is one equation and two unknowns. The two damping constants can be solved by measuring the change in amplitude at the beginning of the oscillation, and then again later in the waveform. The equations can then be solved simultaneously. Equation [45] can be rewritten more simply in terms of the damping constants for each measurement point.

$$\Delta x_1 = 4Q + X_1(1-b) \quad [46]$$

$$\Delta x_2 = 4Q + X_2(1-b) \quad [47]$$

Subtracting [47] from [46] and solving for b,

$$b = 1 - \frac{\Delta x_1 - \Delta x_2}{X_1 - X_2} \quad [48]$$

Solve [46] for Q considering b as known.

$$Q = \frac{\Delta x_1 - X_1(1-b)}{4} \quad [49]$$

This method was applied to solve for the damping constants. The standard viscous damping constant c is a function of the damping constant b from above. Given that,

$$c = 2\zeta m\omega_n \quad [50]$$

And,

$$b = \frac{1}{e^{\zeta \omega_n T_d}} \quad [51]$$

Therefore,

$$c = \frac{2m}{T_d} \ln\left(\frac{1}{b}\right) \quad [52]$$

To solve for the damping ratio ζ, the coulomb and viscous damping are combined into an equivalent damping ratio. ζ_{eq} is given by the following relationship (Rao, 1995):

$$\zeta_{eq} = \frac{\ln\left(\frac{X}{X - \Delta x}\right)}{2\pi} \quad [53]$$

The average damping constants determined by the above empirical analysis are:

$$Q = \frac{\mu N}{k} = 0.0044 \text{ in}$$

$$c = 0.02 \text{ lb-s/in}$$

$$\zeta = 0.035$$

These values were then entered into the simulation, and the output of the simulation was compared to the empirical data. Figures 3-6 and 3-7 are comparison plots of actuator length vs. time for a 60 psi nominal pressure and 6 lb suspended mass system.

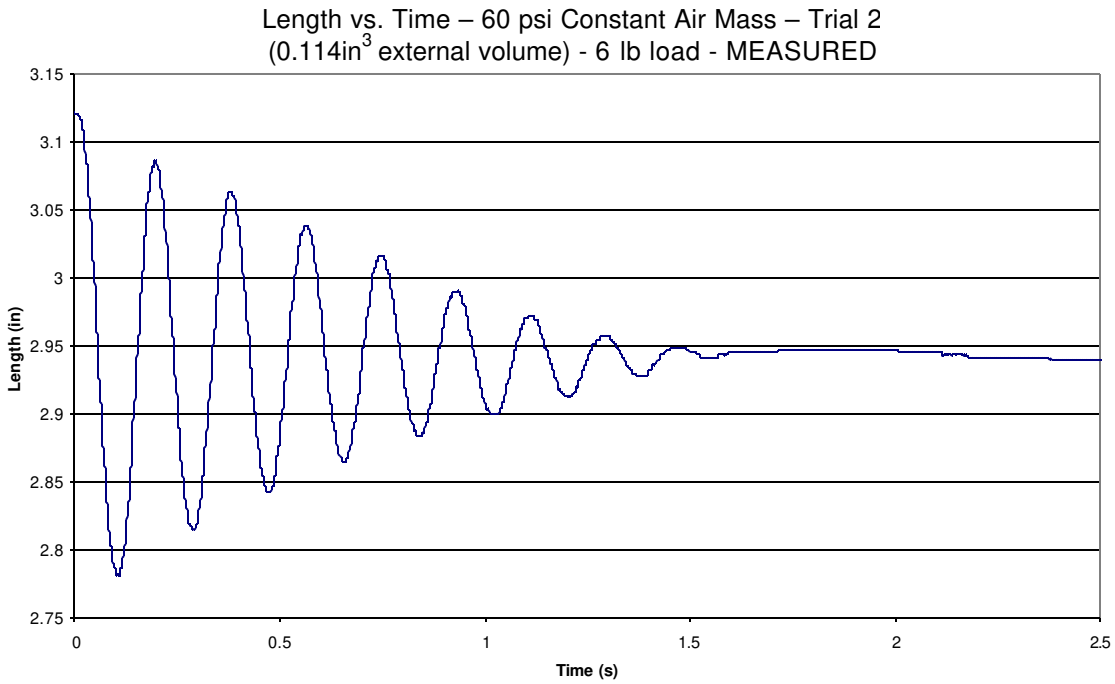


Figure 3-6

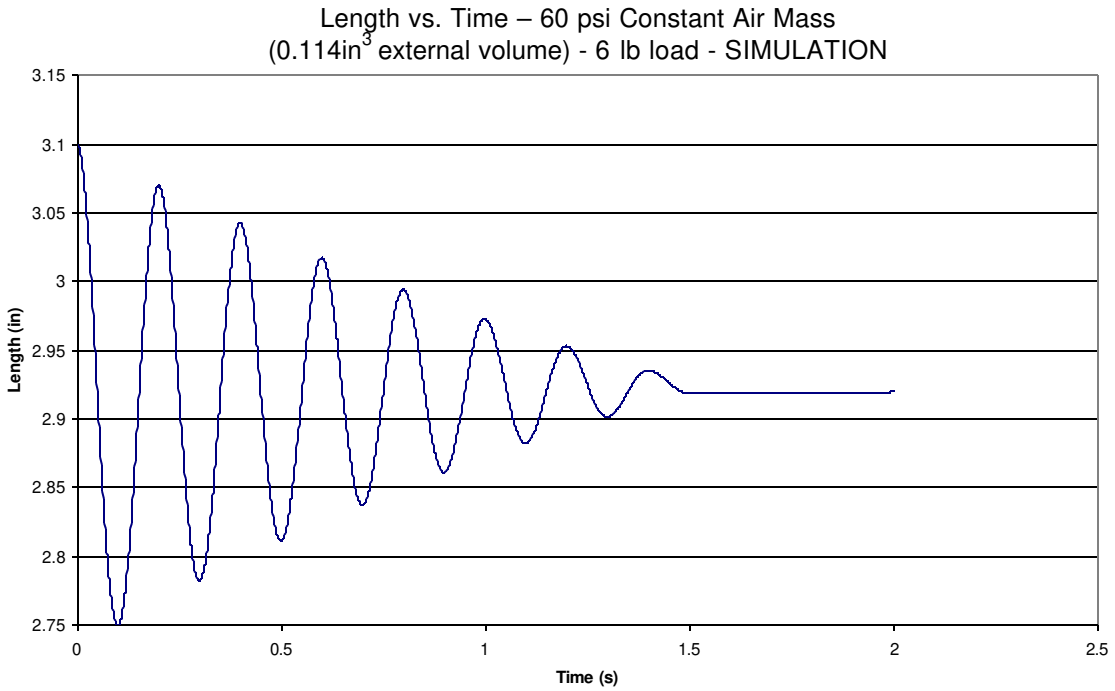


Figure 3-7

Figures 3-8 and 3-9 are comparison plots of actuator length vs. time for an 80 psi – 11 lb system.

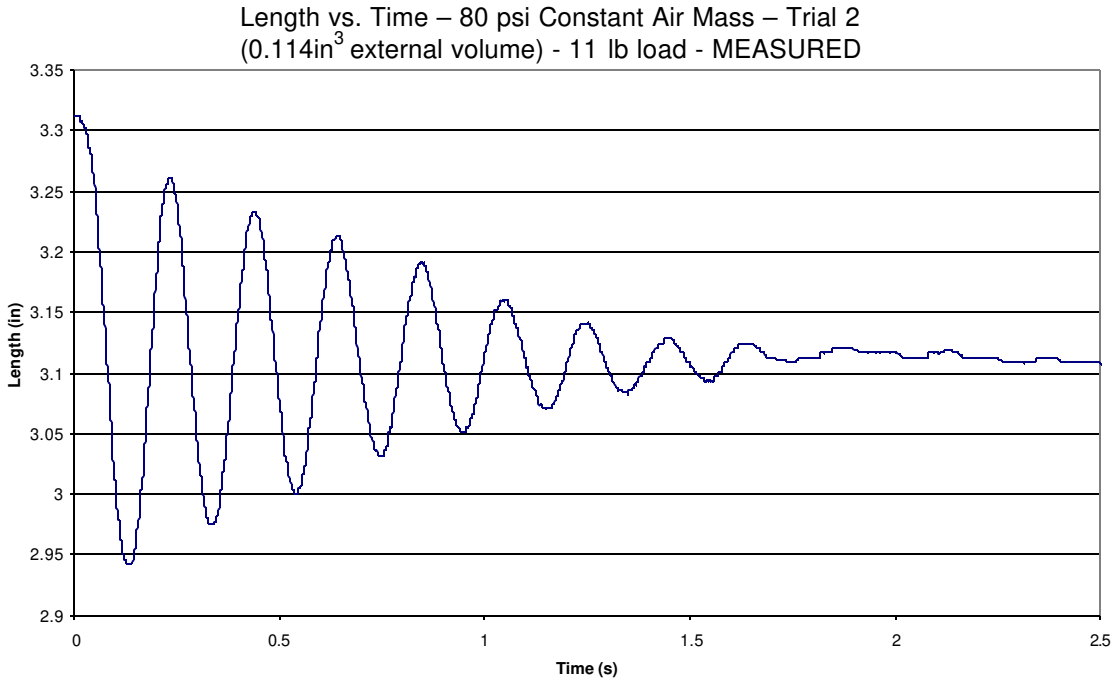


Figure 3-8

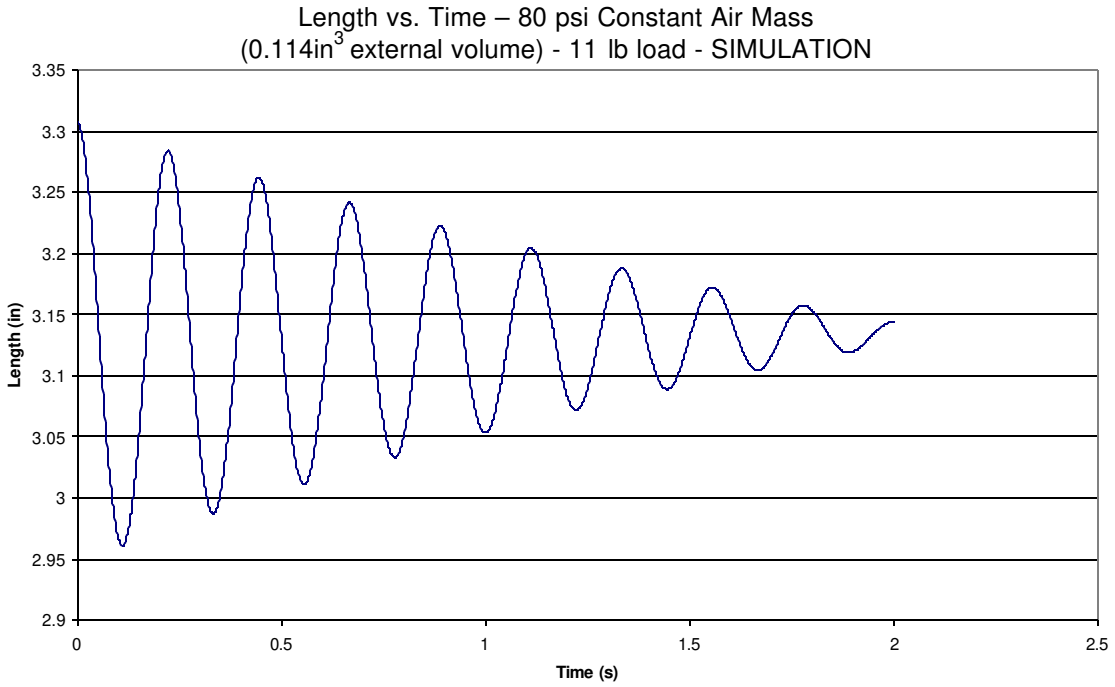


Figure 3-9

Figures 3-10 and 3-11 are comparison plots of actuator length vs. time for a 60 psi – 11 lb system.

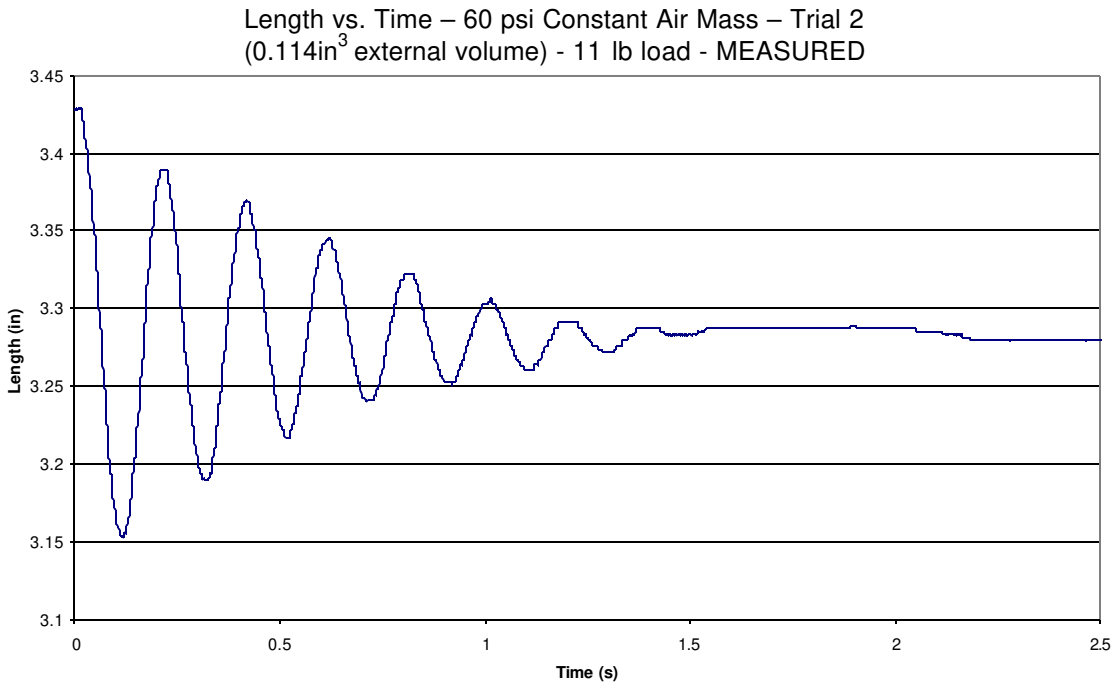


Figure 3-10

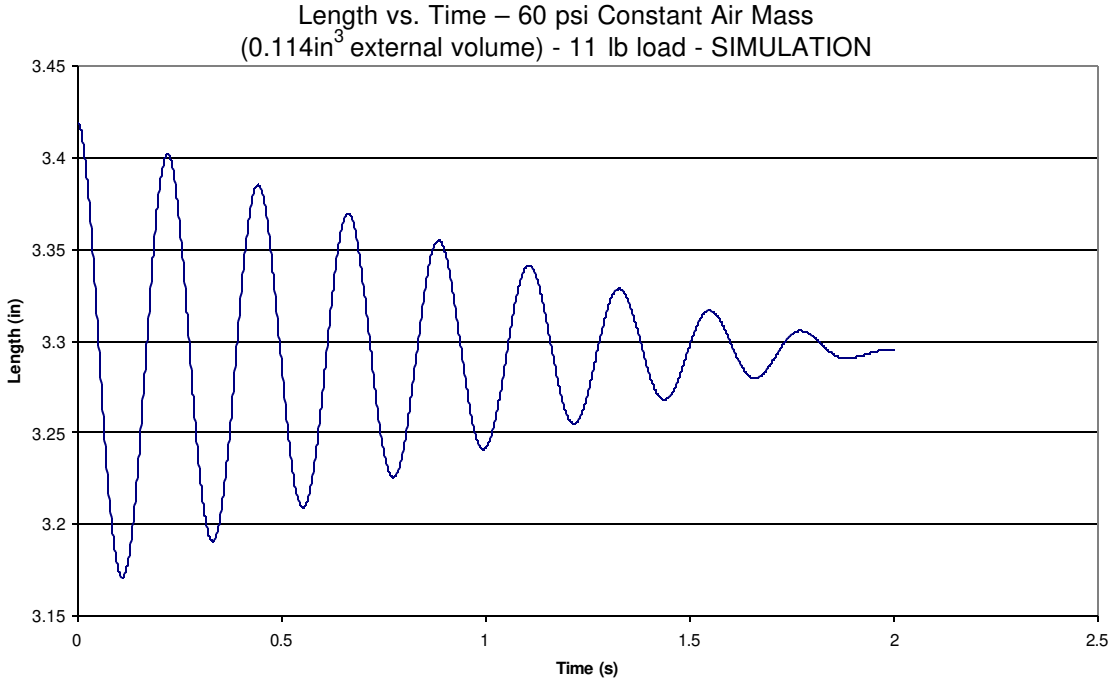


Figure 3-11

Figures 3-6 and 3-7 show excellent agreement. However, Figures 3-8 through 3-11 show some imperfections in the model. On all plots, the damped natural frequencies are quite close. This is important because it is very much a function of how accurate the static force model is. On the other hand, if the damping did not match, then it would mean that the empirical damping values are not correct, and they would need to be adjusted. This, in fact, was the case. The model was under-damped compared to the real data when the 11 lb load was used. With the current damping model, no amount of tuning will cause all three plots to match. This suggests that the damping may not be a constant. Another possibility is that the linear stiffness equation [34], needs to be replaced with [26]. The important thing to notice is that even though the damping model was a small error, the model is under-damped and therefore is better for simulation use. If it was over-damped, compared to the system, then a simulation may not turn up any potential problems because the actuator model wrongly damps them out.

The next step in the dynamic model verification process was to test the valve model. A three-way solenoid valve was placed into the system, and opened and closed under PWM control at 25 or 50 Hz and at a 50% duty cycle. The input signal to the valve was a 0 to 12 V square wave. The signal was created by a 0 to 5 V function generator connected to an optical relay that switched the 12V signal to the valve. (Figure 3-12)

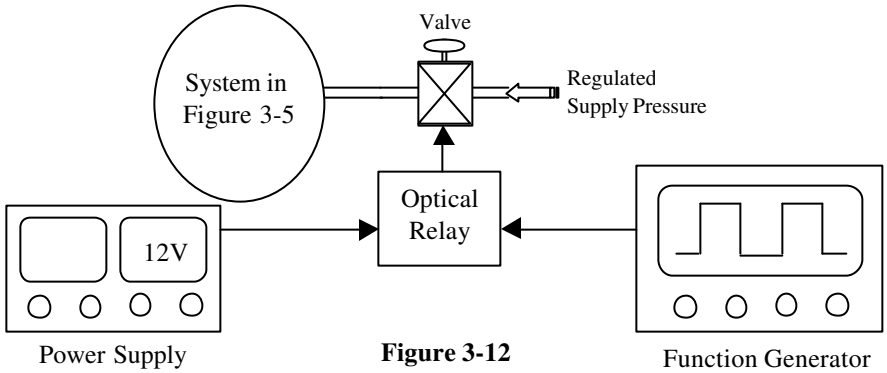


Figure 3-12

The valve does not open instantaneously, and the opening and closing times of the valve are not the same. Therefore, the PWM frequency was adjusted for the simulation. Based on testing done by Nelson (in press), the valve opens in approximately 4 ms, and the valve closes in approximately 1 ms. This means that any duty cycle that commands the valve to be open less than 4 ms will not actually open the valve. Also, any duty cycle that does not allow at least 1 ms for closing will never fully close the valve. Figure 3-13 provides a graphical relationship between the commanded and actual duty cycles.

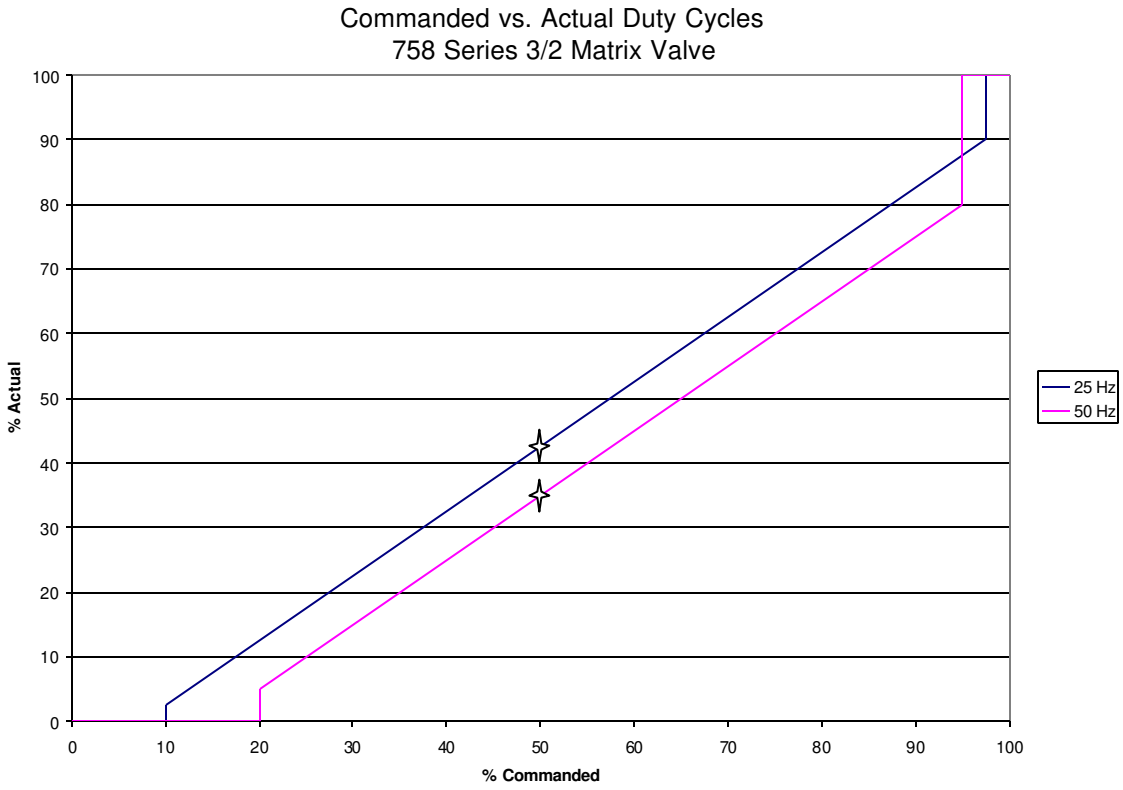


Figure 3-13

Based on Figure 3-13, the duty cycles used in the simulation were 43% for 25 Hz, and 35% for 50 Hz. Actual and simulation plots for each configuration are presented below. Figure 3-14 is a plot of actuator length vs. time for a 100 psi, 25 Hz, 5 lb system.

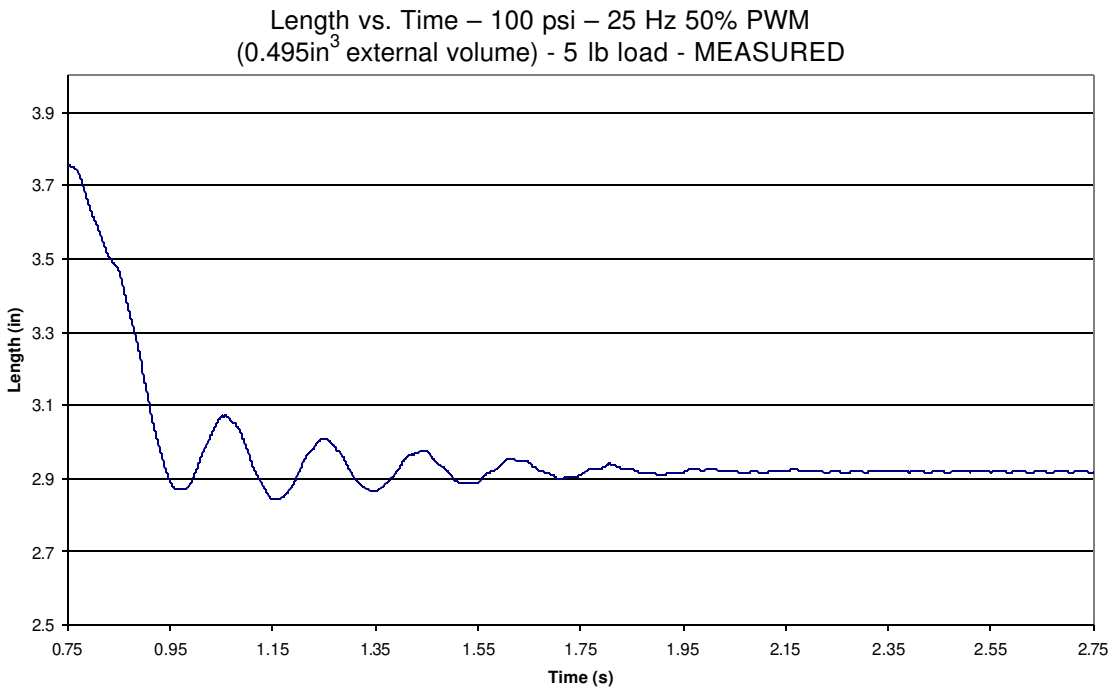


Figure 3-14

Figure 3-15 is the simulation output for the same system.

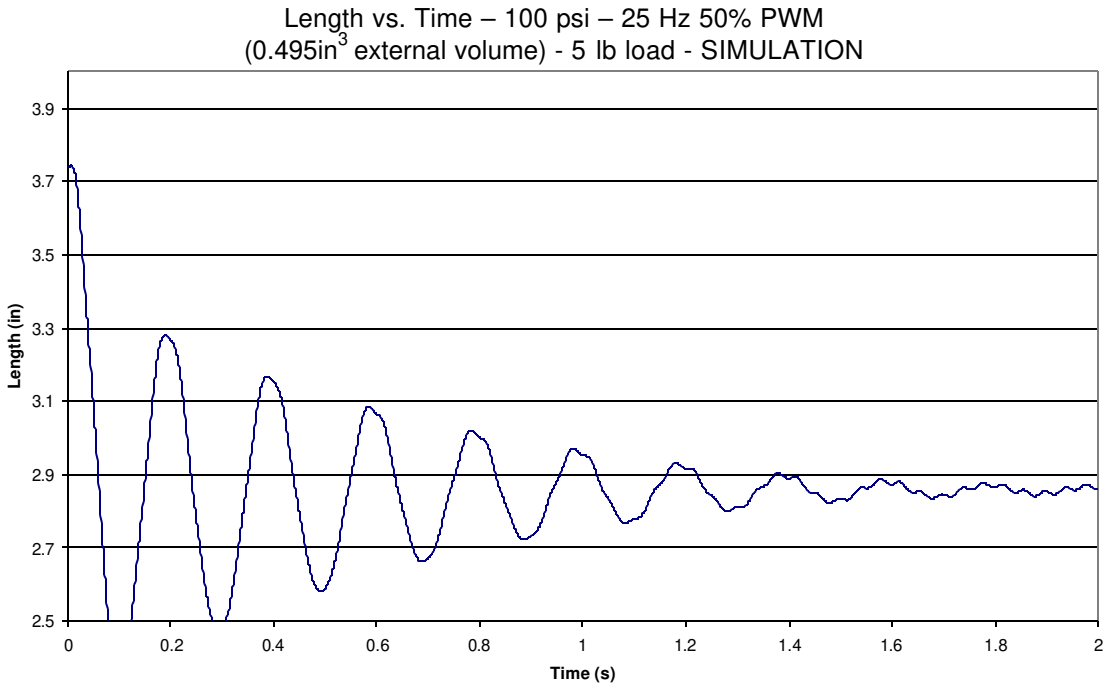


Figure 3-15

Figure 3-16 is a plot of actuator length vs. time for a 100 psi, 25 Hz, 1 lb system.

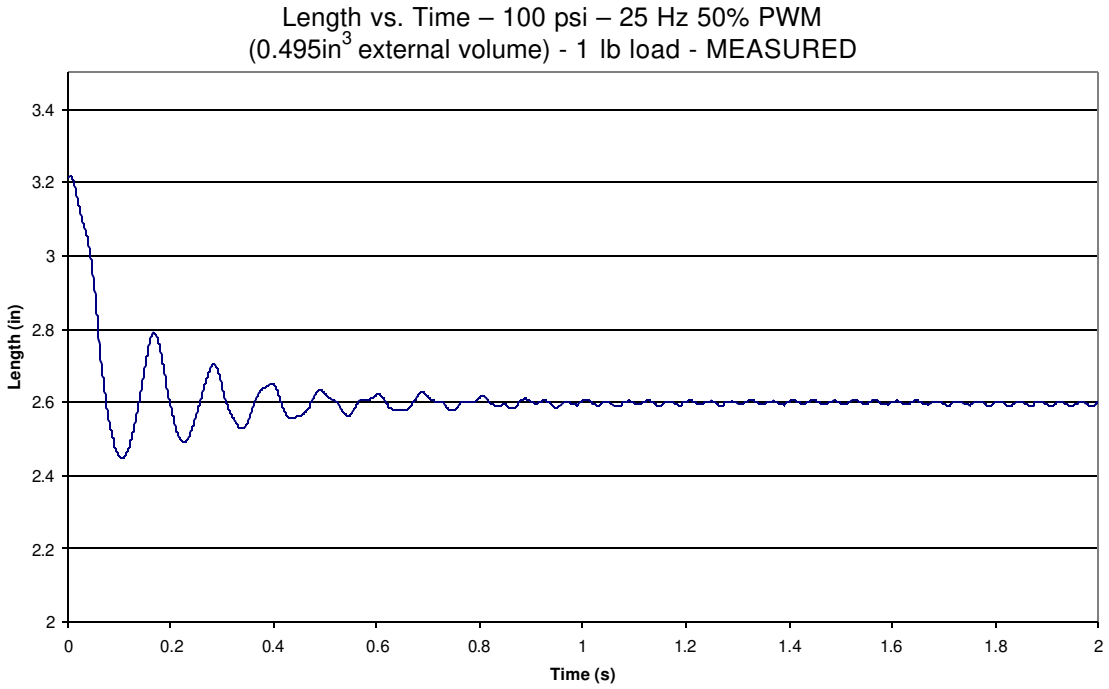


Figure 3-16

Figure 3-17 is the simulation output for the same system.

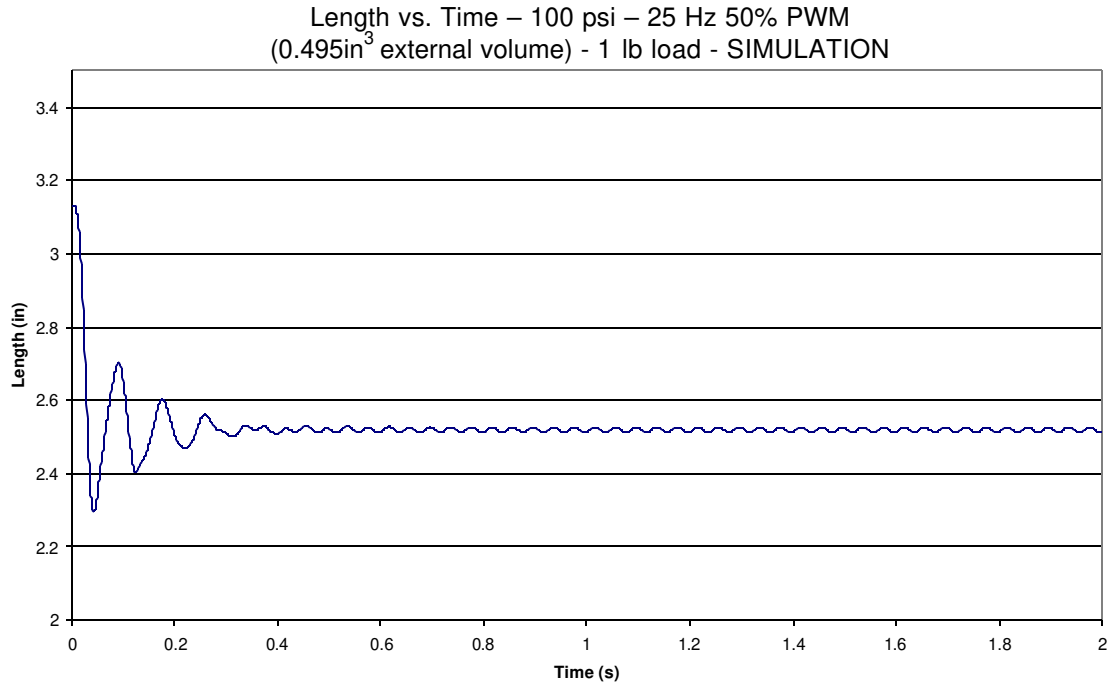


Figure 3-17

Figure 3-18 is a plot of actuator length vs. time for a 100 psi, 25 Hz, 15 lb system.

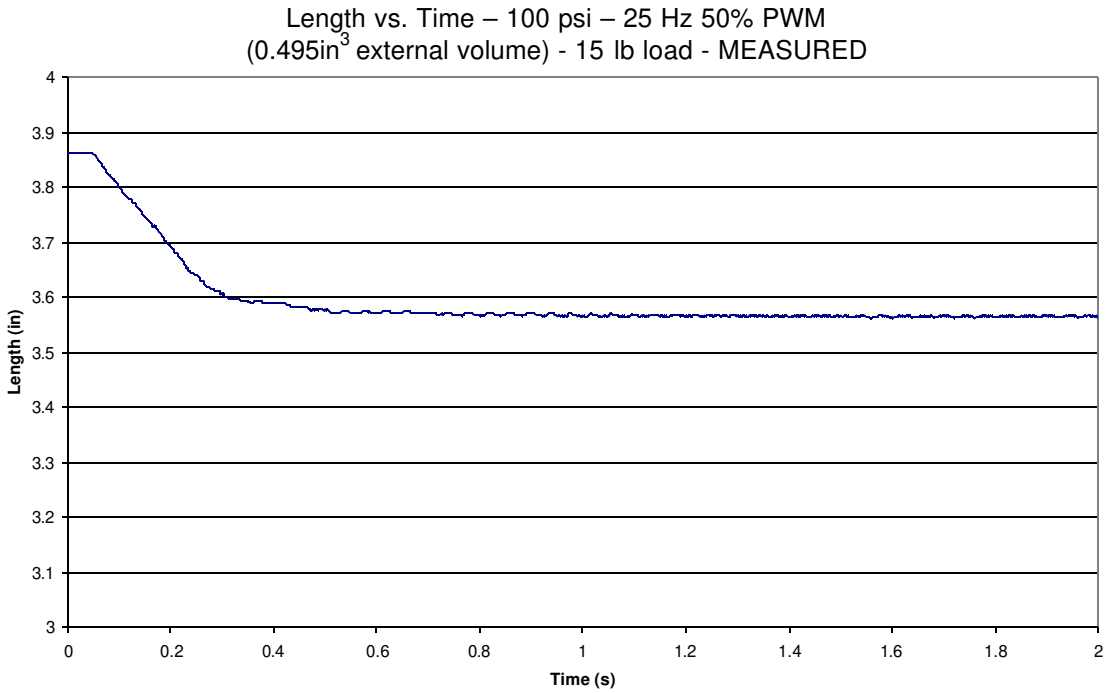


Figure 3-18

Figure 3-19 is the simulation output for the same system.

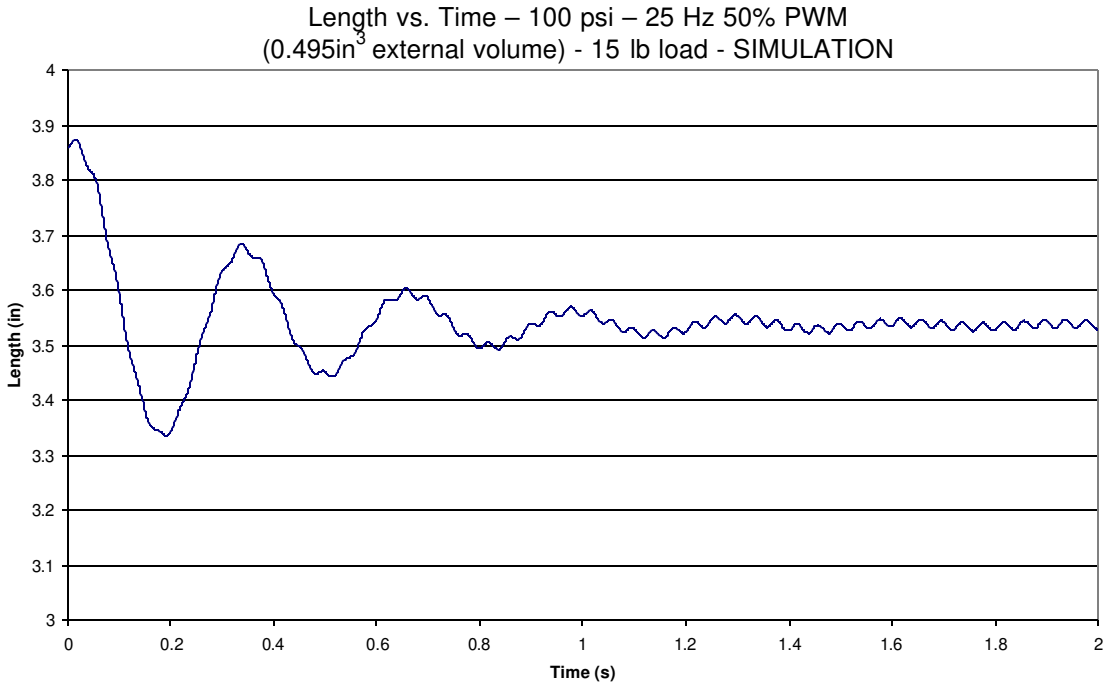


Figure 3-19

Figure 3-20 is a plot of actuator length vs. time for a 100 psi, 50 Hz, 1 lb system.

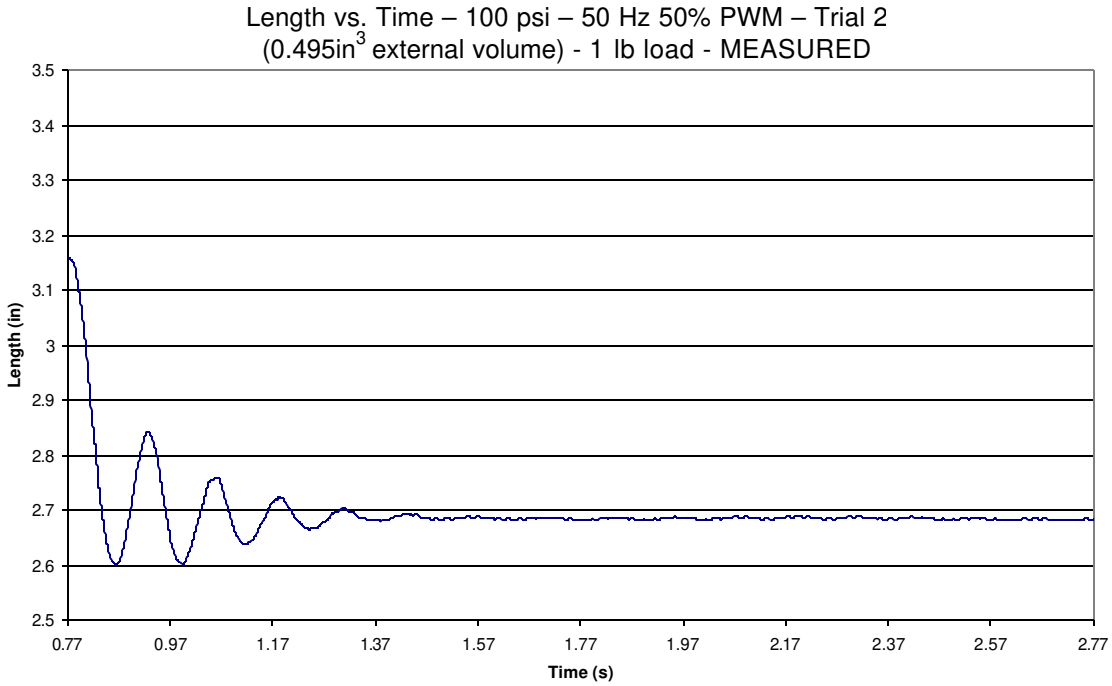


Figure 3-20

Figure 3-21 is the simulation output for the same system.

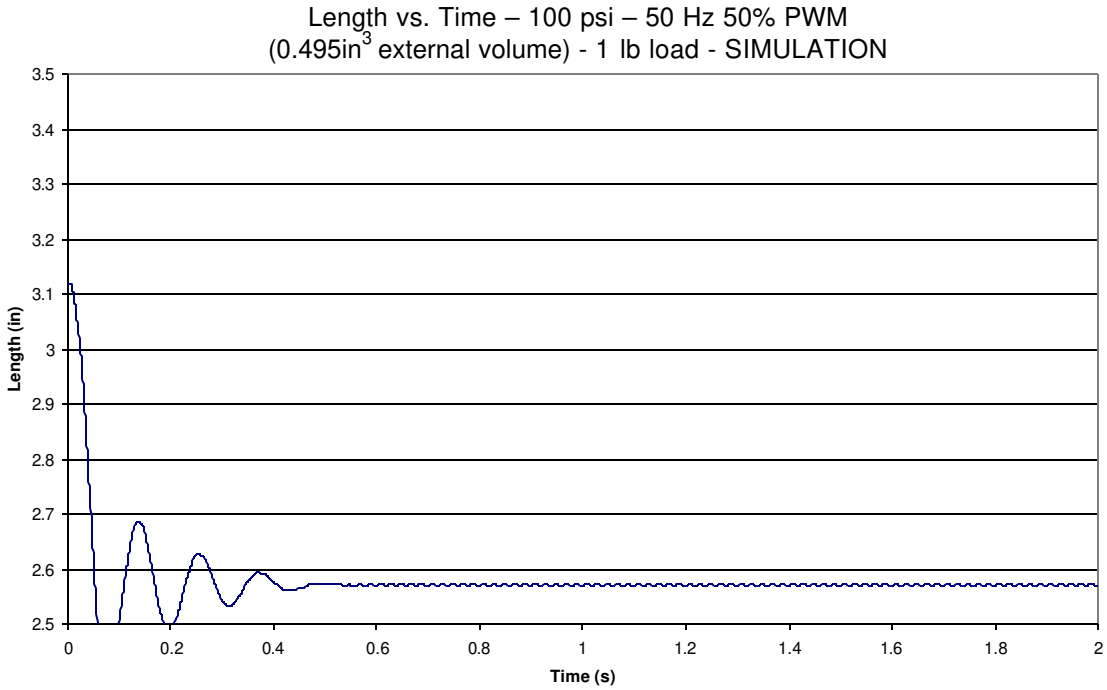


Figure 3-21

Figure 3-22 is a plot of actuator length vs. time for a 100 psi, 50 Hz, 1 lb system.

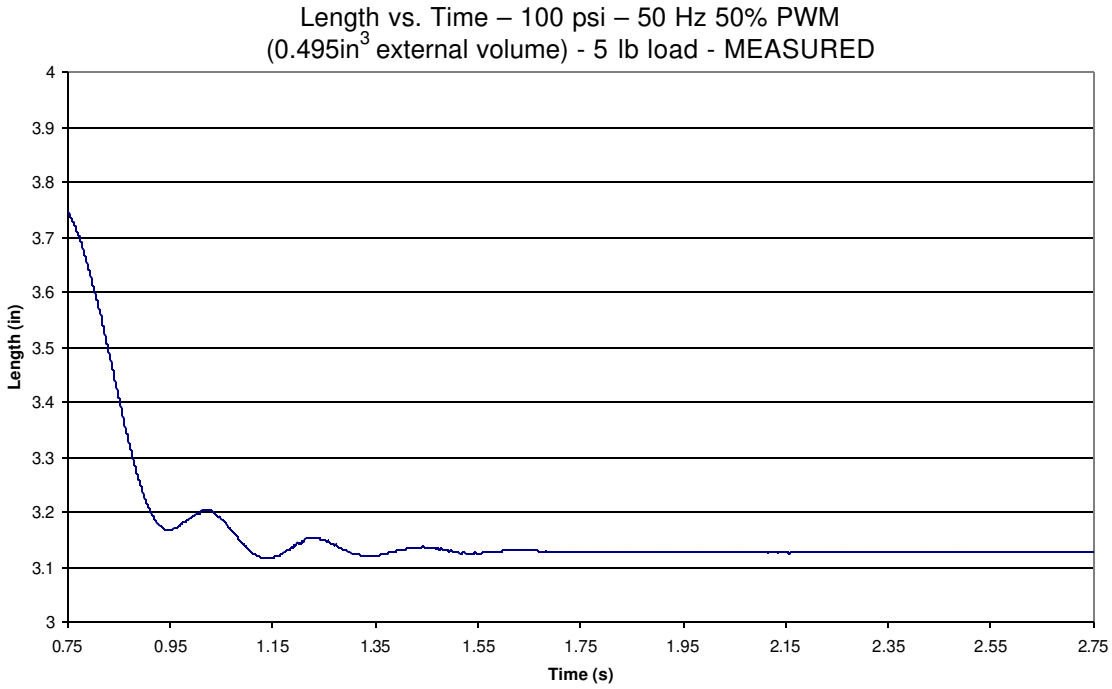


Figure 3-22

Figure 3-23 is the simulation output for the same system.

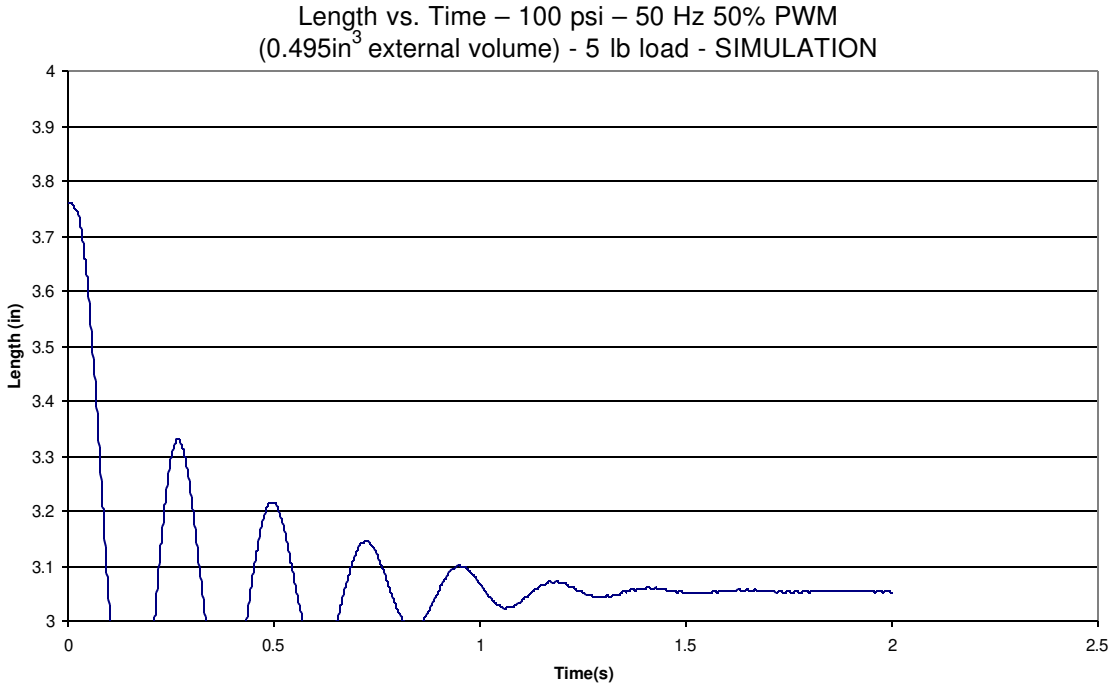


Figure 3-23

These plots show that the valve model is adequate. The under-damped actuator model is evident. For the output of the simulation, an efficiency term was added to the valve model mass flowrate equation [42]. This efficiency was set for 85% in all the simulation plots above. For the systems with large mass (Figure 3-18), the measured steady state actuator length was close to the simulation output. As the mass was reduced, the error increased. This suggests that a better efficiency term could be chosen. The need for this efficiency factor and the under-damped waveforms can be attributed to the hoses. The viscous losses that occur when the air flows through the plumbing are not modeled in the system. If they were modeled, then there would be more damping, and also the mass flowrate into the actuator would decrease. This is exactly the phenomena that the above plots suggest. A hose loss model could improve the dynamic actuator/valve model. However, the current model is accurate and efficient which makes it ideal for robot simulations and control. Appendix A contains the simulation code.

Chapter IV: Robot Hardware

4.1 System Overview

A robotic leg was built to provide a platform for testing braided pneumatic actuator control theories. An important part of the design was that the leg had to alternate between swing and stance to model a leg on a mobile robot. A four-degree of freedom planar motion leg was designed to move the leg through swing and stance and walk across a table. The control technique, described in chapter 6, required three feedback signals for each joint: Joint angle and force output of each tensile actuator. Potentiometers and strain gage based force transducers were designed and mounted on the leg for this purpose. Sensor signals were sent to a computer where the A/D and control decisions were done. The output of the control program sent signals on DIO lines that were connected to optical relays. The relays commanded the opening and closing of the valves, which moved air in and out of the actuators, and moved the leg. Figure 4-1 shows a hardware system overview.

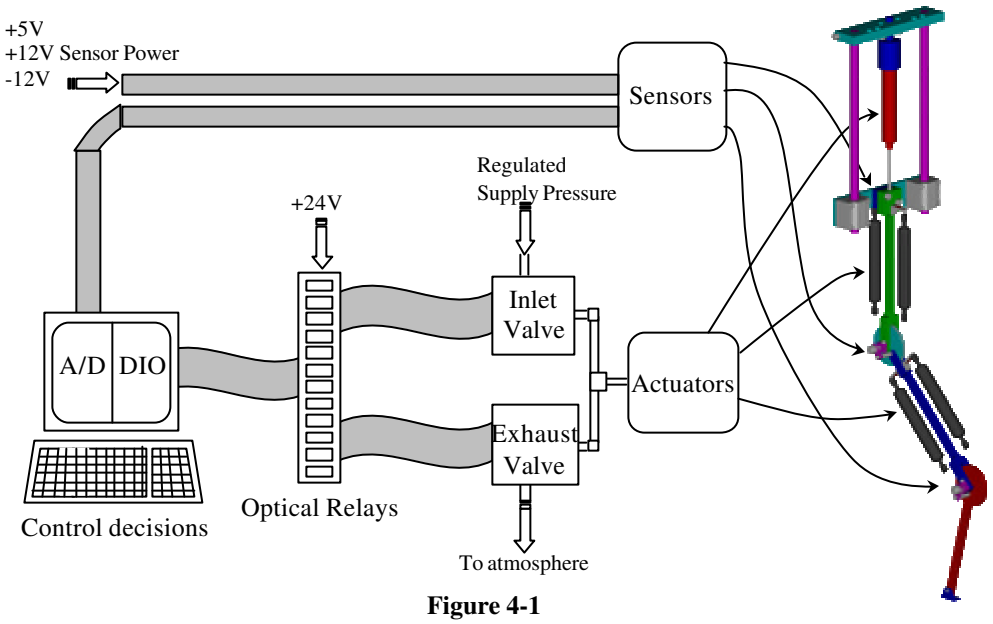


Figure 4-1

4.2 Leg Design

The leg was to have femur-tibia proportions similar to a cockroach. However, the leg motion was not cockroach like at all. This configuration was much more like a biped leg with planar motion and four degrees of freedom. Due to this humanoid configuration, the joints and links are referred to as the hip, knee, femur and tibia. The leg was to be able to alternate between swing and stance and carry a load. BPA were used to drive the two rotational joints, and an air cylinder was used to drive the prismatic vertical degree of freedom at the hip. This way, a regulator could be used to supply a known pressure to the air cylinder, and cause the leg to lift a known force. The leg was designed so that if desired, the cylinder could be placed in a control loop and position control could be performed. The leg was mounted to a trolley that rolled along a track to allow for forward motion of the robot. Early in testing it was noticed that the leg was much stronger than originally anticipated, so a 10 lb counterweight and friction roller were added to keep the rollers on the track and the speed down. The four degrees of freedom were the unactuated linear trolley motion on the track (1), the vertical linear motion due to the air cylinder (2), the rotational hip motion (3), and the rotational knee motion (4). A 50 conductor twisted pair ribbon cable connected the leg to the controller. This included all valve signals, sensor power, and sensor signals. To reduce noise and interference, the sensor power and valve signals were separated from the sensor signals by using either side of the ribbon cable (i.e. Sensor: channels 1-14, Power and valves: channels 23-50). The free channels also provide a way to easily implement more sensors in the future. The air was supplied to the leg through a 10mm umbilical hose that was connected to a 100 gallon tank. Figure 4-2 is a labeled photograph of the completed leg.

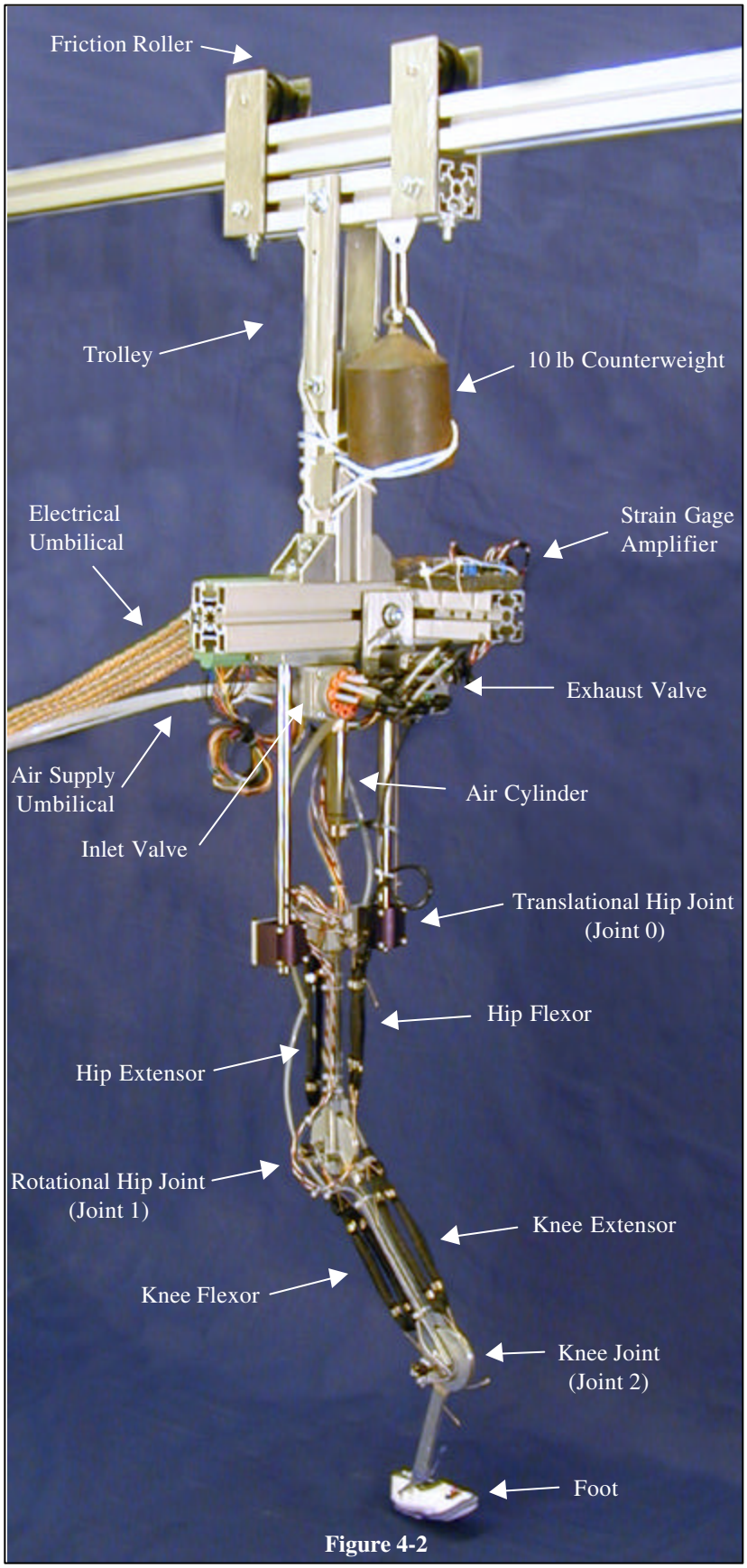

Friction Roller
Trolley
10 lb Counterweight
Electrical Umbilical
Strain Gage Amplifier
Exhaust Valve
Air Supply Umbilical
Air Cylinder
Inlet Valve
Translational Hip Joint (Joint 0)
Hip Flexor
Hip Extensor
Rotational Hip Joint (Joint 1)
Knee Extensor
Knee Flexor
Knee Joint (Joint 2)
Foot

Figure 4-2

Joint 0 was actuated with an Airpel air cylinder. Airpel uses a graphite piston and a Pyrex cylinder to create a very minimal friction actuator. Model E16D2.0U is a ∅0.627" x 2.0" stroke air cylinder. The cylinder was connected to the trolley with a misalignment coupler to prevent harmful internal stresses in the cylinder. The rod was attached to two linear bearings that allowed the leg to translate vertically while preventing rotation. The bearings translated on two ∅0.5" hardened steel shafts. The air cylinder rod threaded into the body link. The actuators on the body rotated the femur at Joint 1, and Joint 2 was the connection of the femur and the tibia. The distal actuator tendons (Kevlar cord) were wrapped around a 2" cylindrical cam at each joint. This was done to keep the actuator moment arm constant. Admittedly, a constant moment arm is not a biologically correct technique, but in a study of the actuators, including a changing moment arm would only complicate the issue. The 1" radius also was beneficial in that a moment arm of 1 essentially dropped the moment arm term out of the equations. The cam was notched to keep the tendon aligned regardless of joint angle. The bearings at the rotational joints were thermoplastic bushings from Berg pressed into the distal segment that rotated on a steel shoulder screw mounted to the proximal segment. This configuration allowed a minimal friction joint to be packaged in a very small area. At each joint there were also actuator force and angle sensors. These are explained in further detail in sections 4.4 and 4.5.

The last part of the leg was the foot. The foot was connected to the tibia at a passive ankle joint. A small aluminum foot was attached to the tibia, and then a baby shoe was attached to provide passive damping at the joint as well as to make the leg aesthetically pleasing. The leg was designed using the commercial CAD package

Mechanical Desktop and was then machined from 6061-T6 aluminum stock. Figure 4-3a is the rendered CAD model. Figure 4-3b is a photograph of the leg from the same perspective.

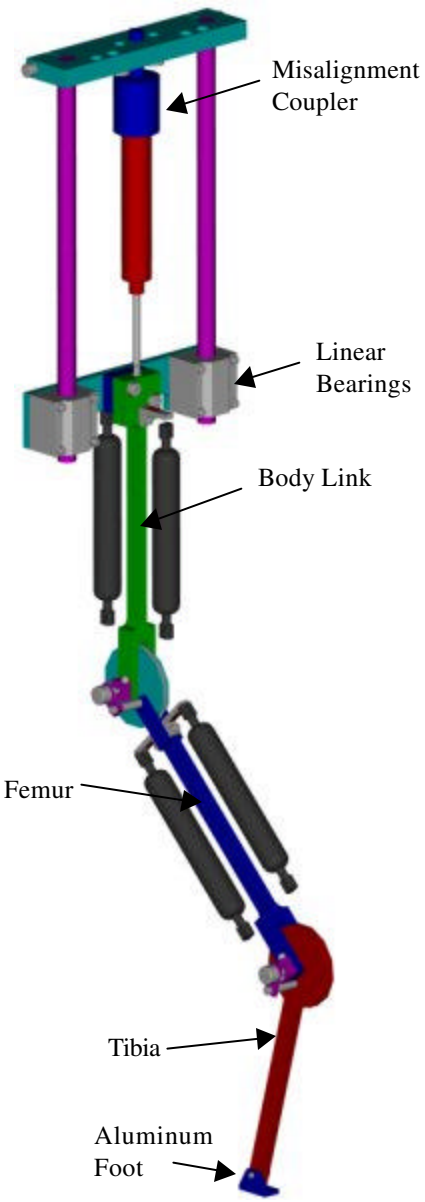


Figure 4-3a

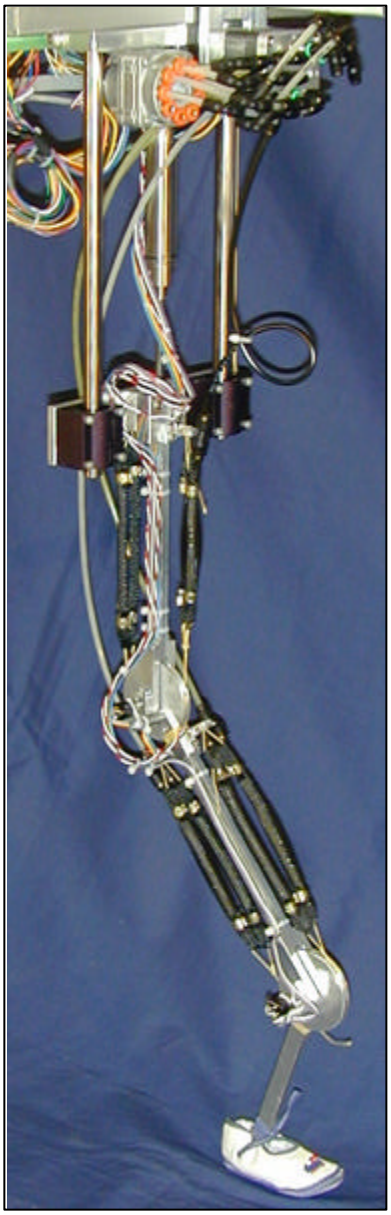


Figure 4-3b

All of the plumbing on the robot, except for the air umbilical, was 5/32" nylon tubing connected with quick disconnect Legris fittings. Figure 4-4 is a CAD drawing and photograph close-up of the hip translational joint.

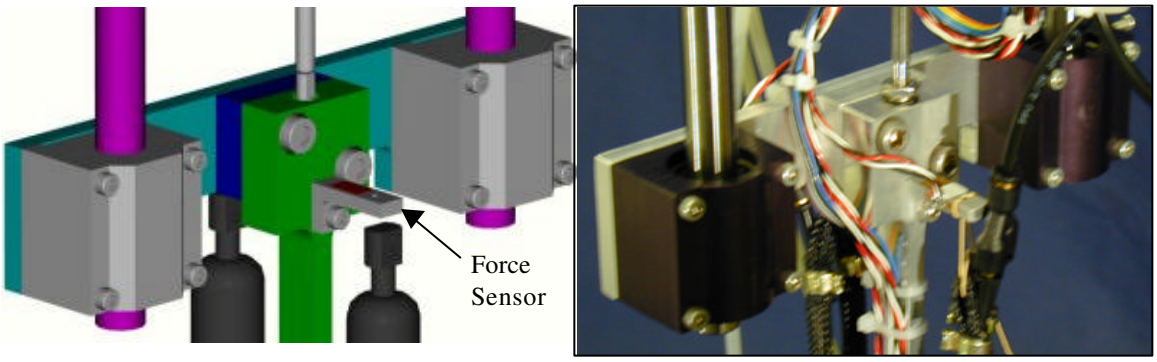


Figure 4-4

Figure 4-5 is a CAD drawing and photograph of the hip rotational joint. Note the machined trough for the tendon, and the packaging of the angle and force sensors.

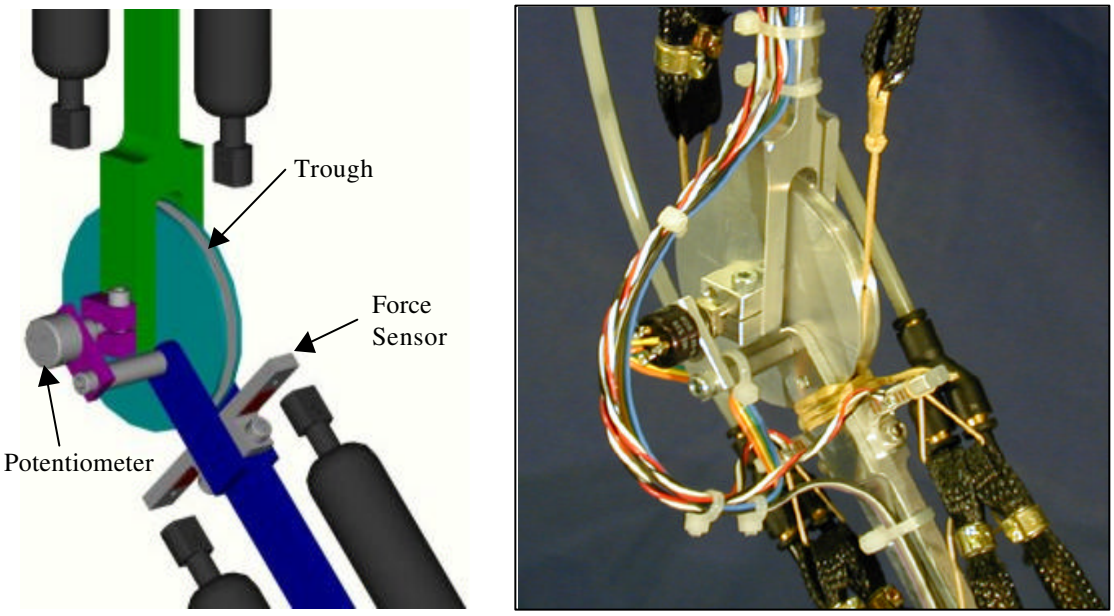


Figure 4-5

Each joint was designed to be actuated by an antagonistic pair of braided pneumatic actuators. This allowed for a variable stiffness joint, which is one feature of these actuators that is of great benefit to walking robots. Simply increasing the pressure in both actuators increases the joint stiffness (equation [26]). This is analogous to increasing activation level in an antagonistic pair of skeletal muscles. Watson et al. (1998) has shown that in cockroach locomotion, there is antagonistic muscle activation

just before foot plant. His work suggests that joint stiffness control could be beneficial to walking.

In choosing a cam radius, there was a trade off between maximum joint torque and joint range of motion. The maximum force output of an actuator was about 15 lb. Based on the 2" diameter cam design, a quick calculation revealed that more force output was needed from the actuators. The decision was made to simply double-up the actuators so that each joint used four actuators instead of two. Therefore, the maximum force output of each "actuator" was roughly 30 lbs. The range of motion of these actuators was about 1.4". With a 1" moment arm, this equates to a 1.4 rad angular range of motion, or about 80°. This range of motion was sufficient for the desired excursions of the leg. Thus, the 2" diameter cam and doubled-up actuator design was implemented.

To achieve this 80° joint range of motion, the actuator free length needed to be set to the optimum value. By increasing or decreasing the free length of the actuator from the optimum length, the range of motion decreased. Either one actuator was fully taught before the other one was able to fully contract, or when one was fully contracted the other had not been stretched to its maximum length. The optimum length was 3.15", half the range of motion. Actually setting the actuator free length to this value was not as easy as calculating it. The total tendon length was determined, and marks were placed on the tendon at the appropriate points. The leg was placed at the desired free angle, and the tendons were wrapped around the distal segments and tightly tied. The beauty of this configuration was that changing the resting angle was as simple as untying and retying the tendons at the desired rest configuration. Appendix C contains all the mechanical drawings for the leg.

4.3 Valves

The solenoid valves were manufactured by Matrix S.p.A. of Italy. The inlet valve was an 850 series, 9 channel, 2-way, in a speed-up configuration. (Figure 4-6)

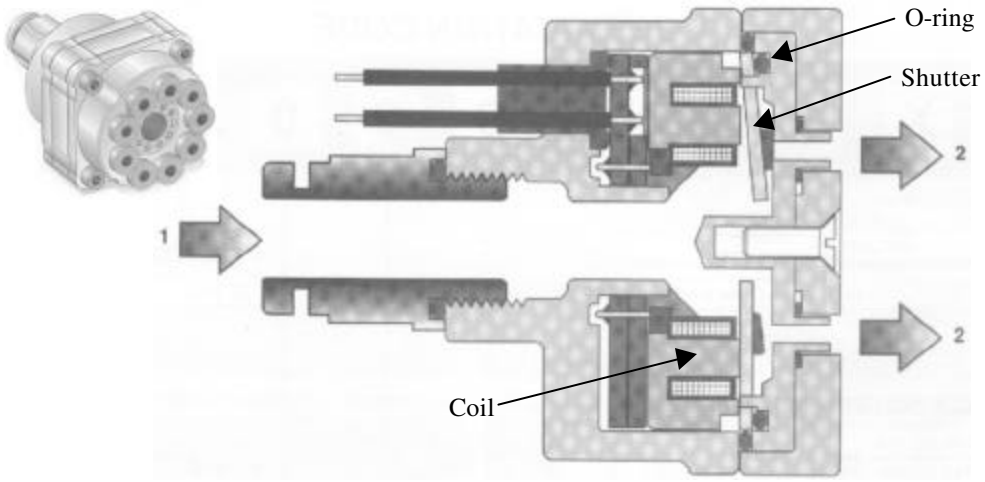


Figure 4-6

Supply pressure air enters at port 1 and exits at port 2 depending on which shutter is activated. When the coil is charged, it attracts the steel shutter, and opens the port. When the current is cut from the coil, the shutter is released, and the restoring force of the O-ring returns the shutter to seal the port. This valve is designed to run in speed-up mode. The manufacturer claims that this option will decrease the opening and closing response time to less than 1 ms. The standard model response times for opening and closing are 2 ms and 5 ms respectively.

The exhaust was a 750 series, 8 channel, 2-way valve, in a speed up configuration. (Figure 4-7) When the valve is used in the standard configuration, (how it was designed to be used) supply pressure air enters at port 1 and exits at port 2 depending on which shutter is activated. However, for the control of the robotic leg, this valve was operated in the opposite direction: High pressure air enters port 2, and port 1 is open to the atmosphere. This caused a few problems. Normally, when the coil is charged, it

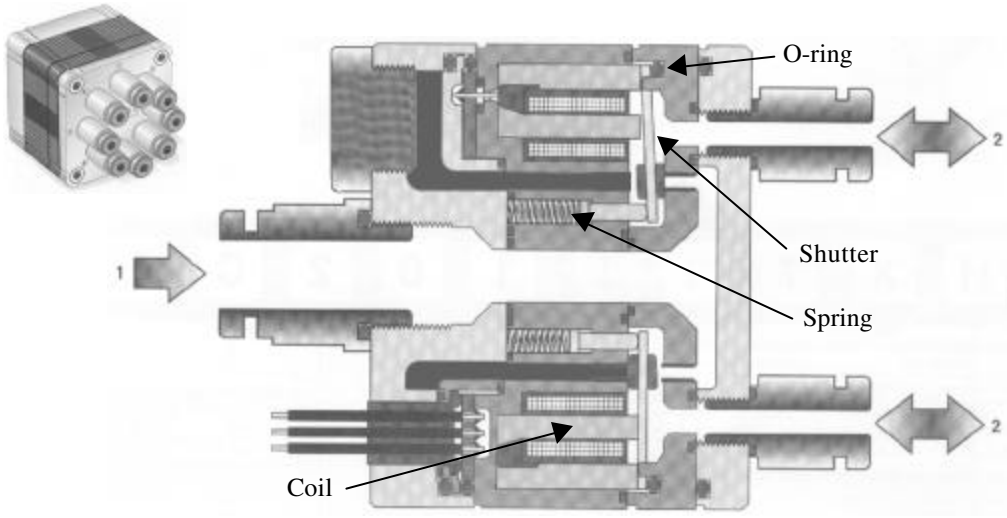


Figure 4-7

attracts the steel shutter, compresses the spring, and opens the port. When the current is cut from the coil, the shutter is released, and the restoring forces of the O-ring and spring return the shutter to seal the port. Matrix designed the spring to provide the correct force to seal the port in a normally closed configuration. A problem, however, occurs when supply pressure is on the wrong side (side 2). The activated coil does not create a strong enough magnetic field to overcome the spring force and the force on the shutter due to the supply pressure. A quick test revealed that the valve would only operate with supply pressures less than 35 psi. The spring was removed, and the O-ring was relied upon to return the shutter. Another test revealed that this was a good solution. The valve operated flawlessly to a pressure of 100 psi, the limit of the compressor. This valve was also designed to run in speed-up mode. However, with the removal of the spring, the manufactured specified response times were no longer valid.

Valve speed is very important in the implementation of pneumatic control. Therefore, testing was done to determine the true speed of the valves, and also to determine the maximum operating frequency. A small resistor was placed in series with

the coil, and the voltage drop across the resistor was measured on an oscilloscope. This current measurement circuit provided an easy way to determine the response time of the valves. Figure 4-8 is a plot of current vs. time for the inlet valve. The supply pressure was 100 psi, and a 50% duty cycle, 125Hz PWM frequency, square wave was the command signal.

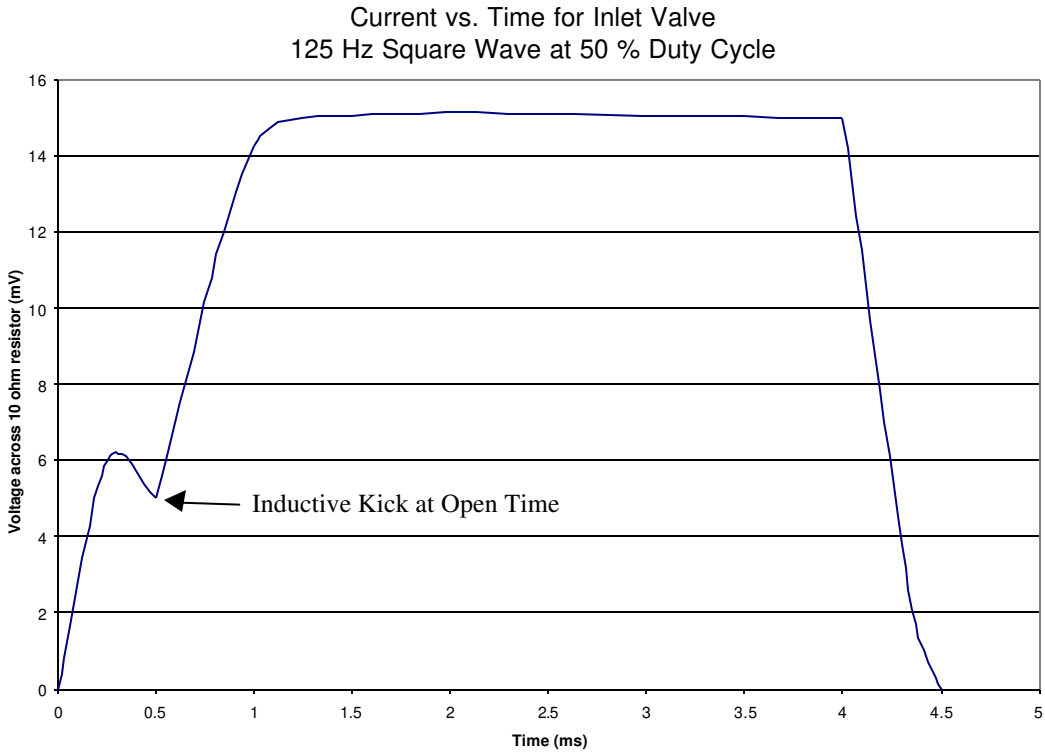


Figure 4-8

The shutter opened in about 0.5 ms. The open time was independent of supply pressure. The close response time appeared to be about 0.5 ms also, but this was not a good way to measure the close time. All this plot suggested was that the current had left the system about 0.5 ms after the valve was commanded close. There was not an inductive kick to indicate a time when the valve actually closed. Figure 4-8 suggests a very quick response time for the valve, and the audible maximum frequency was found to exceed 400Hz.

Figure 4-9 is a plot of current vs. time for the exhaust valve. The supply pressure was varied with a 100Hz PWM frequency, 50% duty cycle commanded signal.

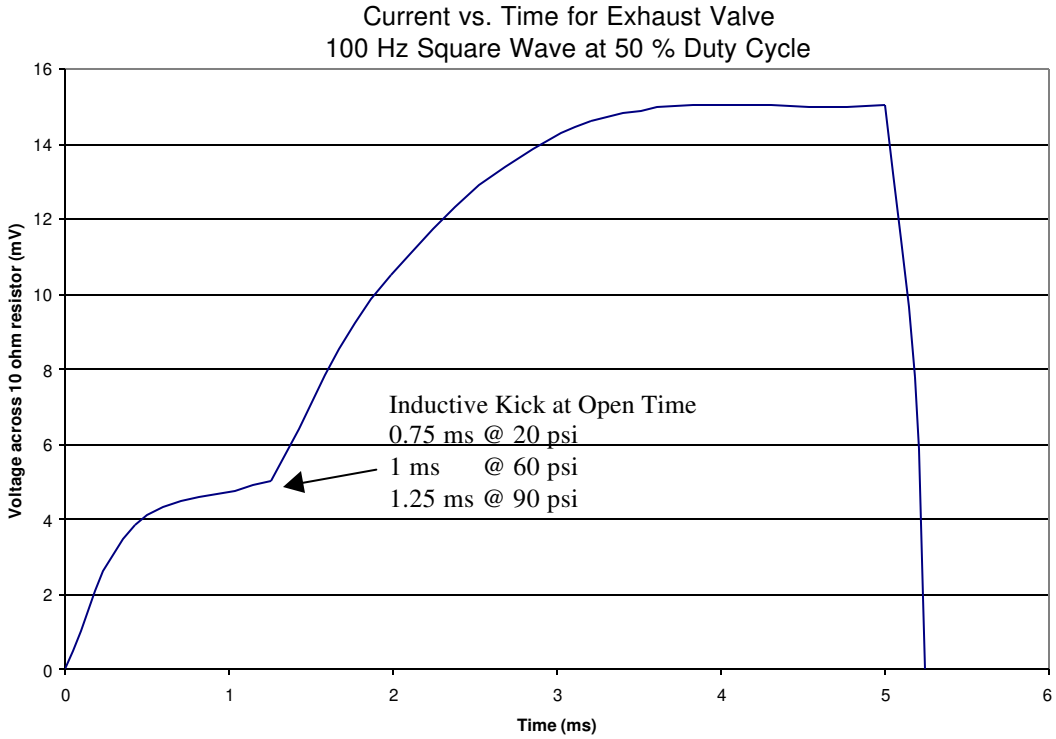


Figure 4-9

The exhaust valve response time was slower than the inlet valve. The opening time increased as supply pressure increased, as was expected. However, the audible maximum operating frequency increased as the supply pressure increased. This at first seems contradictory, but it is most likely due to the increased pressure decreasing the closing time faster than it increases the opening time. The maximum operating frequency at 90 psi was about 350 Hz.

4.4 Force Sensors

Force sensors were designed to accurately measure the force output of each actuator. The maximum force output of an actuator was about 15 lb. Based on the doubled-up actuator design, the maximum load on the transducer was roughly 30 lbs.

Classic stress-strain beam analysis was performed to size the transducer. 6061-T6 aluminum was chosen for the transducer material, and the desired strain at the gage location (A) was 1000με. The stress at the mount point (B) was designed to be less than the 20ksi, which corresponds to a factor of safety (FOS) of 2. The design also needed to have enough room to mount the gages properly. Figure 4-10 is a schematic of the beam.

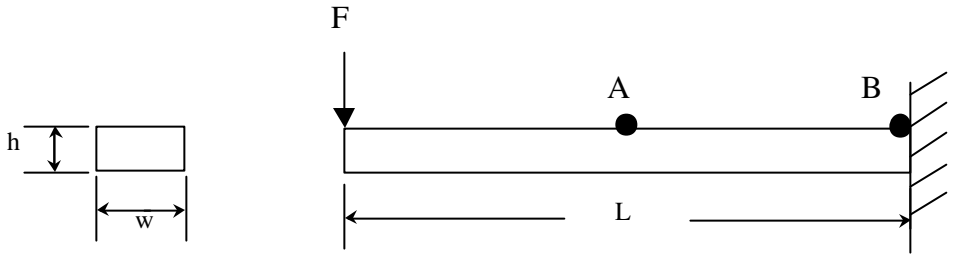


Figure 4-10

The length L was already set by the geometry of the leg to be 0.75". The width, w, and height, h, were designed to be 0.3" and 0.15", respectively.

This was a good first iteration, but the final transducer design wouldn't be as simple as the beam above. To mount the transducer, a flange was added, and therefore a stress concentration was created. A model was created in ALGOR and a finite element stress analysis was performed. The design was iterated to find the minimum radius necessary for a FOS of 2. The minimum radius was important because the smaller the radius, the closer the gage can be placed to the area of high strain. The radius of the flange was designed to be 0.063". Figure 4-11 is the stress contour plot of the final design.

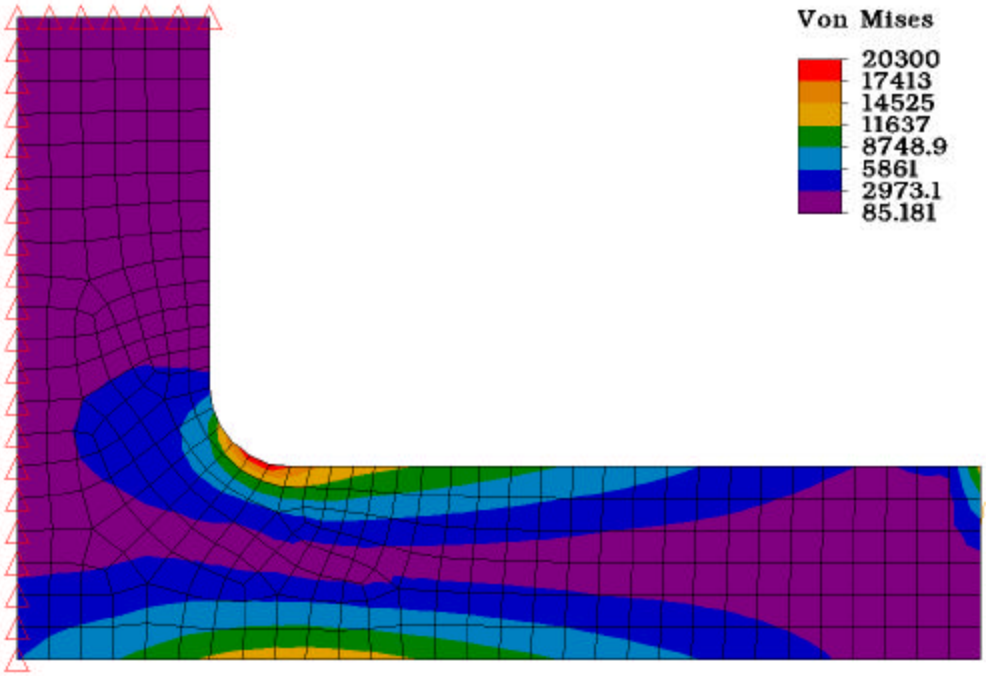


Figure 4-11

The transducers were machined and Jim Berilla (Biomechanics Technician) mounted the strain gages. Mounting holes were also placed in the transducer. Figure 4-12 is a photograph of the completed transducers.

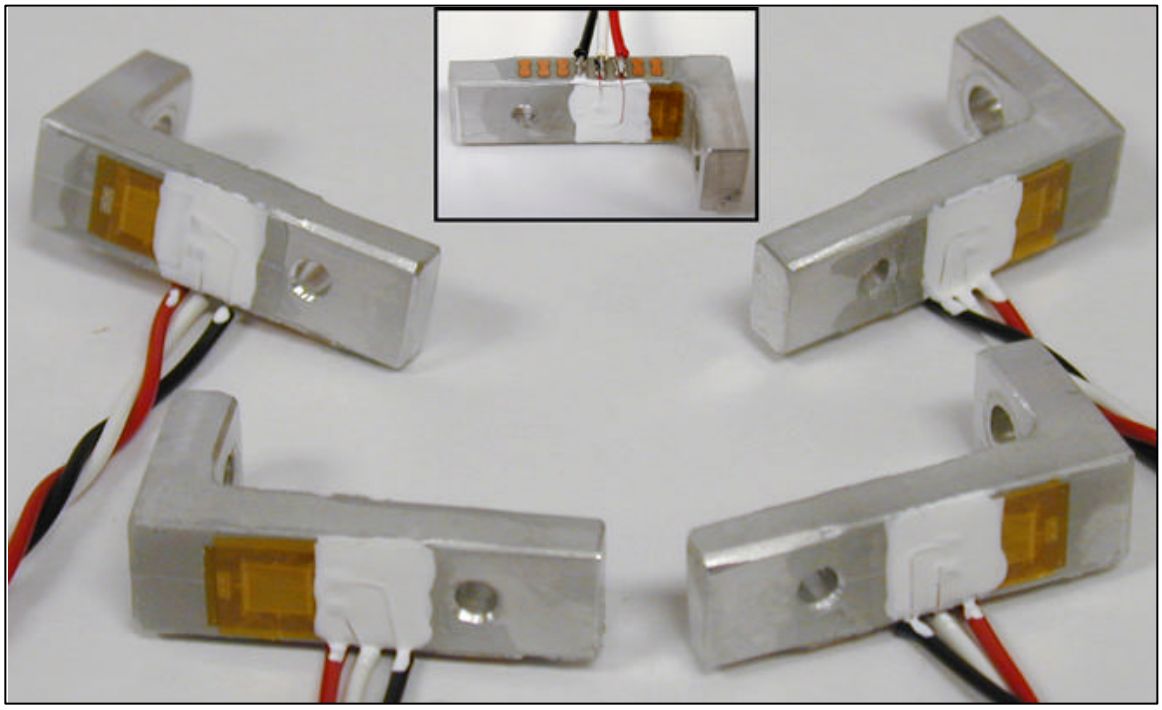


Figure 4-12

Strain gage outputs are notoriously small, and are very sensitive to noise, temperature, and lead length variation. To minimize sensor error, a small strain gage amplifier, designed by Yuandao Zhang (CAISR Technician), was mounted to the leg, which provided a high level signal for the A/D converter. Figure 4-13 is a photograph of the four-channel strain gage amplifier.

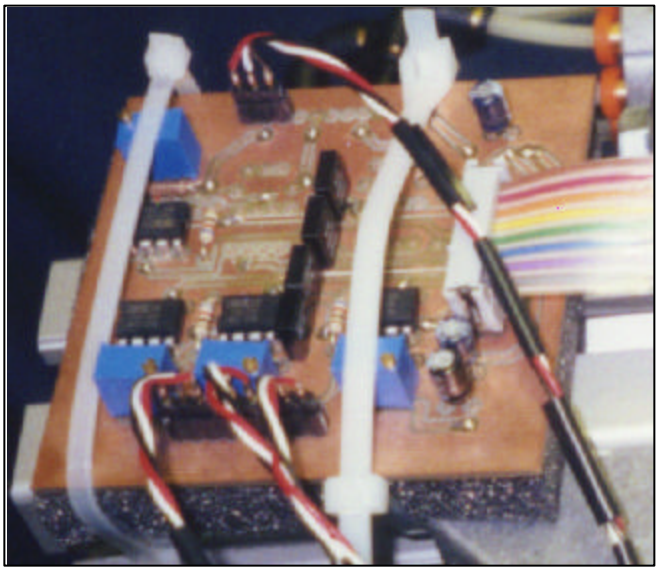


Figure 4-13

The circuit was supplied with +5V for the strain gage bridge, and ±12V for the amplifiers. The gain was set to 800 so that the output of the amplifier would be about 5V at 30lbs. Appendix C includes the pin-outs for this board.

The next step in the implementation of any transducer is the calibration. Each transducer was marked, and a small fixture was designed to orient the transducers in the proper direction. Then, dead weights were hung from the tendon attachment points and the output was recorded. The maximum load applied was only 16 lb. It would have been better to use the full-scale load, but that amount of dead weight was not readily available. This introduced some error into the calibration, however, the absolute accuracy of the transducers was not imperative. Figure 4-14 is a plot of the calibration data.

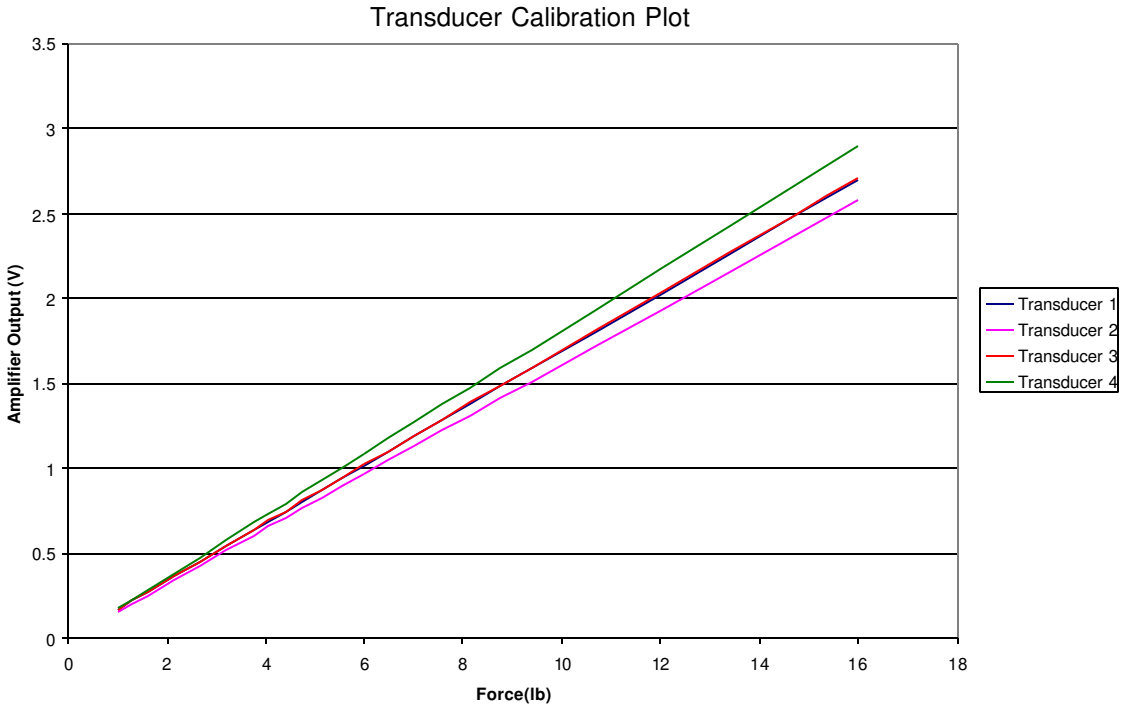


Figure 4-14

Transducers 1 and 3 had similar sensitivities. Transducers 2 and 4 were not as close. Table 4-1 shows the relationships between the transducer scale factors, and the error from the average.

Table 4-1: Transducer Sensitivity and Error

transducer #	Scale factor (V/lb)	% error from average
1	0.1689	0.7%
2	0.1610	5.6%
3	0.1694	0.4%
4	0.1809	-6.0%
Average	0.1700	

If the scale factors from table 4-1 were used in the control program, the force error would be minimal. However, in the interest of computational time in the control program, the average was the only value used. Again, the absolute accuracy of the force measurement was not important, and a 6% error was acceptable. Upon completion of the calibration, the transducers were mounted on the leg, and the actuators were attached with kevlar tendons. Figure 4-15 shows the completed force measurement system.

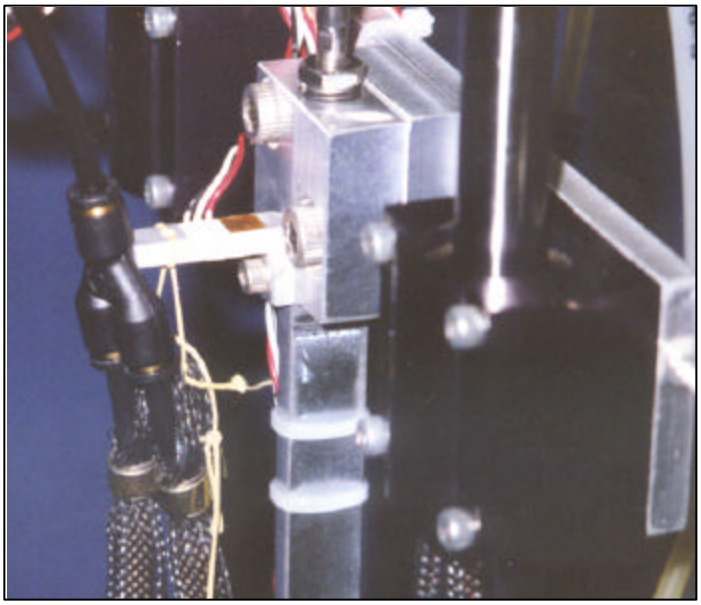


Figure 4-15

4.5 Angle Sensors

The angle sensors were single-turn 10kΩ potentiometers manufactured by Spectrol. The body of the potentiometer was fixed to the distal segment, and the shaft was mounted to the proximal segment. A power supply provided +5V across the fixed resistance part of the potentiometer. When the actuators rotated the joint, the voltage between the fixed pin and the variable resistance pin changed. This voltage was read by an A/D card in the PC and used as the measure of joint angle in the control program. The calibration of the angle sensors was done using a simple two-point method. The joint was set to some nominal position, and the voltage was read. Then the joint was rotated through 90° and the voltage was read again. The scale factor for the angle sensors was 1.147 rad/V. Figure 4-16a is a photograph of the completed joint angle measurement system.

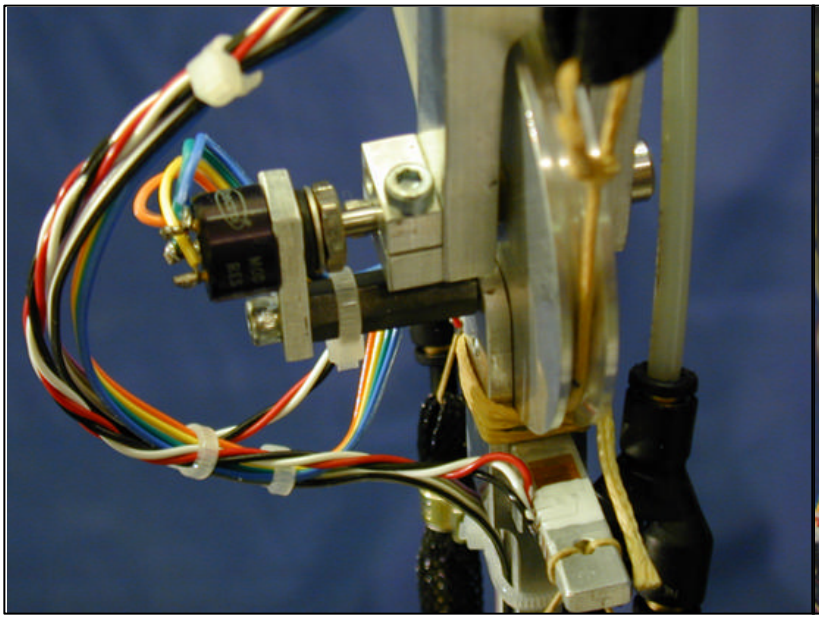


Figure 4-16a

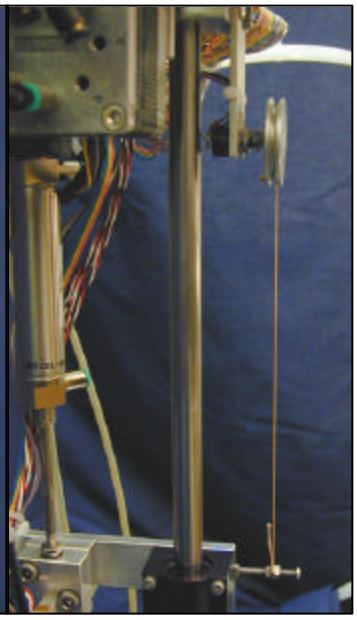


Figure 4-16b

In addition to angle sensors, a linear potentiometer was built for use in the static and dynamic actuator verifications as well as for measuring the position of the air cylinder. The linear distance sensor utilized the same potentiometer as the rotary transducers. However, the shaft was threaded to allow a notched wheel to be attached to it. Then, a small torsional spring was made by winding light gage music wire and attaching it to both the wheel and the potentiometer case. A multi-point calibration was done and provided a scale factor of 0.57 in/V when powered in the same manner as the angle sensors. Figure 4-16b is a photograph of the linear potentiometer mounted on the leg.

Chapter V: Control

5.1 Control Architecture and Control Laws

Much of this work built upon what had been learned in the construction and control of Robot III. The valves on Robot III were 3-way valves, and they were controlled using pulse width modulation, or PWM. In this manner, the variation in the duty cycle was essentially varying pressure. A downside to using 3-way valves is that the actuator chambers can never be sealed. Therefore, the robot consumed a lot of air during operation. For our long-term goal of autonomy, we sought to decrease the amount of air necessary to operate a robot. Instead of using one 3-way valve to control each actuator, two 2-way valves were implemented in the leg described in this thesis. One valve being the inlet valve, and the other being the exhaust valve. This way, air could be trapped when actuator activation was not needed. The next step was to develop a control scheme that only opened one valve at a time, either the inlet, or the exhaust. In doing this, the control idea was shifted from that of controlling air pressure to controlling air mass. If actuator activation was desired, then the inlet valve would be opened to increase the mass of air in the actuator, which increased the pressure. To relieve the actuator, then the same thing was done with the exhaust valve. This mass control scheme shifts air mass to and from the actuator to achieve the desired results.

Control of the robot was done with two controllers. There was a high-level controller that generated the trajectory, performed all the inverse kinematics which generated all the desired joint values, and was capable of making modulations on control gains all as a function of the sensory feedback signals. This controller, on a very simple level, is analogous to the brain. There was also a low-level controller that made control

decisions based on the desired values given to it by the high-level controller. Included in this controller were the error based proportional control laws whose outputs were duty cycle values that were communicated to the valves. This controller is, again on a very simple level, analogous to spinal cord reflexes.

In the low-level controller, there were two overlaid control laws. One law controlled joint angle, and the other controlled joint stiffness. These laws were designed to interact in a way that minimized the consumption of air. Figure 5-1 is a schematic of a joint to aid in the explanation of the control theory.

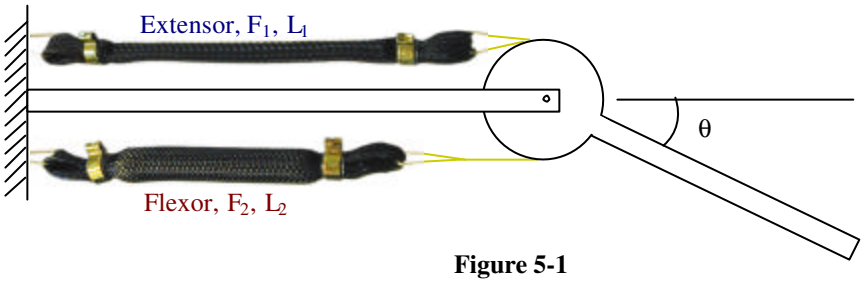


Figure 5-1

The extensor and flexor actuators were numbered 1 and 2, respectively. When the flexor contracted, θ increased. Figure 5-2 is a block diagram of the control algorithm.

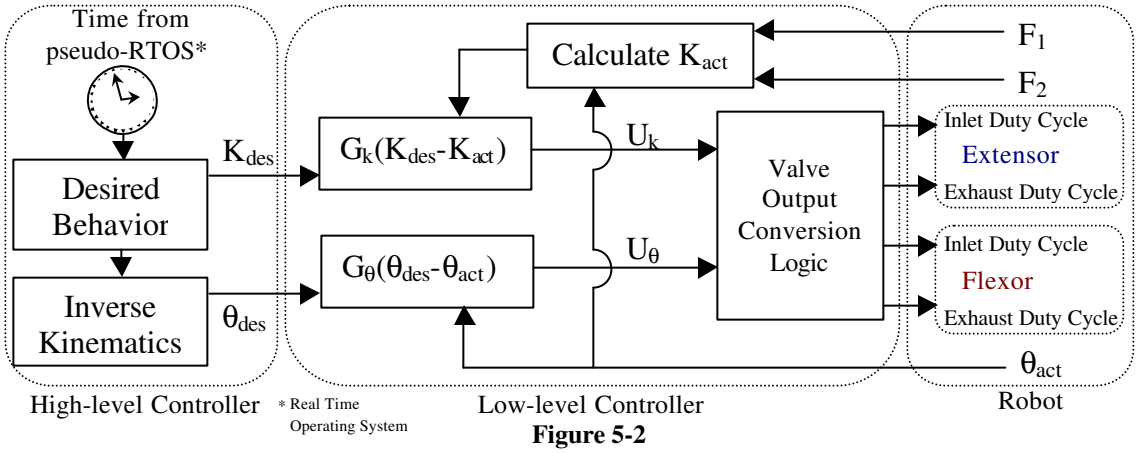


Figure 5-2

The two most important parts of this controller were the error based proportional control laws, and the valve output conversion logic. The valve output conversion logic is what converted the two-space control outputs, U_k and U_θ, into the four-space signals to

the valves. The conversion method chosen not only quickly converted U_k and U_θ to valve duty cycles, but it also was an excellent method for conserving air. The valve output conversion logic is most easily explained by the following equations.

$$\text{Extensor Inlet Duty Cycle} = U_k - U_\theta \quad [54]$$

$$\text{Extensor Exhaust Duty Cycle} = -U_k + U_\theta \quad [55]$$

$$\text{Flexor Inlet Duty Cycle} = U_k + U_\theta \quad [56]$$

$$\text{Flexor Exhaust Duty Cycle} = -U_k - U_\theta \quad [57]$$

When an increase in θ is desired, the flexor inlet valve will operate at a duty cycle governed by U_θ. The extensor exhaust valve will also operate at the same duty cycle. For a decrease in θ, just the opposite happens. The extensor inlet will operate at the same duty cycle as the flexor exhaust. When an increase in stiffness is desired, both inlets operate at a duty cycle governed by U_k. When a decrease in stiffness is needed, both the exhausts operate at U_k. When U_k and U_θ are added according to [54]-[57], both flexor valves have the same duty cycle, just a different sign. The same thing occurs with the extensor. To prevent both valves from opening at the same time, and wasting air, all negative duty cycles are set to zero, as a negative duty cycle does not make sense. In this way, U_k and U_θ are converted to valve control signals. The control laws are:

$$U_k = G_k\left(K_{des} - K_{act}\right) \quad [58]$$

$$U_\theta = G_\theta\left(\theta_{des} - \theta_{act}\right) \quad [59]$$

Where: G_k = Stiffness control gain
G_θ = Angle control gain
θ_{act} = Measured angle
θ_{des} = Desired angle
K_{act} = Measured stiffness
K_{des} = Desired stiffness

K_{act} was calculated using [27] and applying it to both actuators. The antagonist pair places the two actuators in parallel. Therefore, the stiffness at the joint is the summation of the stiffness of the two actuators.

$$K_{act} = K_1 + K_2 \qquad [60]$$

Normally a moment arm term would be included, but the cam radius of 1" allows it to drop out for the units used in this thesis. This was done to increase calculation speed in the control program. Even though [27] was written for a single actuator, no change needed to be made for the doubled-up actuator configuration. The stiffness of the two actuators is just twice the stiffness of one actuator. The denominator of [27] will be the same for both actuators, and the only thing that would change would be the force in the numerator. This force should be twice that of one actuator. When the force is doubled, it doubles the stiffness. The beauty of this configuration is that no matter how many actuators are added, the stiffness equation will always hold true. Therefore, more actuators can be added if increased joint torque is desired, and no change will need to be made to the control program.

Another interesting thing about these control laws is that even though they appear to be just proportional control, when they are combined with the valve output conversion logic, they have an imbedded integral control property. Suppose the leg is physically restrained not to rotate from an angle of 15°, and yet the desired angle is 20°. The flexor inlet valve will put more air into the actuator while the extensor exhaust valve also opens at a rate given by U_θ. The next time the control calculation is made, there will still be a 5° error. Therefore, the same U_θ will again open both the flexor inlet and extensor exhaust valves. This loop will continue until the flexor actuator is filled to the maximum

pressure, and the extensor contains atmospheric pressure air. This configuration provides the maximum joint torque possible. Therefore, as error accumulates over time, the joint torque is increased in an effort to correct this. This example was for an angle error, but the same holds true for stiffness errors. Hence, this control algorithm can correctly be referred to as PI control even though the laws are just error based proportional control.

5.2 Control Program

The control program, written in C, was an adaptation of the program written by Gabe Nelson for the control of Robot III. Nelson (in press) designed a low-level controller to run an interrupt service routine (ISR) at a constant frequency. He designed the high-level controller to run on a separate machine, and then communicate with the low-level controller through serial communications. The amount of data to be processed necessitated this for him, but for the leg presented in this thesis, it was not necessary. In this program, the high-level controller ran on the same processor to generate the desired trajectory. The two programs were synced together with a pseudo-real time operating system (RTOS). This was done so that the desired trajectory could be generated at a constant rate regardless of processor speed.

The method chosen for controlling the valves on this robot was Pulse Width Modulation (PWM). When the PWM frequency was chosen, a trade-off needed to be resolved. A higher PWM frequency will provide smoother control, yet the valves have finite opening and closing times, which puts a limit on the PWM frequency. For this robot, 100 Hz was the frequency chosen for the PWM control. PWM frequency is important, but so is duty cycle resolution.

To be able to command the duty cycle to 1% resolution, the program needed to be able to change the valve state 100 times every 10 ms. Therefore, the interrupt service routine, ISR, needed to run once every 100 µs. It also needed to be efficient enough so that the high-level controller had enough processor cycles to run when the ISR was finished. Fortunately, a complete PWM period, 10 ms, was available to perform all calculations. The output of the ISR was a valve duty cycle to be used for that period. Therefore, to calculate it any quicker or more than once would not have been useful. The beginning of each ISR opened or closed the valves based on the duty cycles calculated in that period. Then, the ISR read a sensor channel and performed control calculations based on that channel. Every other subsequent ISR read and performed calculations on a new channel until it ran out of channels, at which point the ISR just commanded valve states. The PWM period for each channel began in the ISR that the duty cycle calculation was made. This provided the least delay between when the control decision was made and the valve carried out the action. This also meant that the PWM period for different valves began at different times. The reason that the A/D calculations occurred every other ISR was that 25 µs was needed to let the amplifier in the A/D multiplexer settle, and the next 12-bit conversion also needed to be initiated. One ISR was not enough time to do it all. Figure 5-3 is a graphical representation of the ISR scheme.

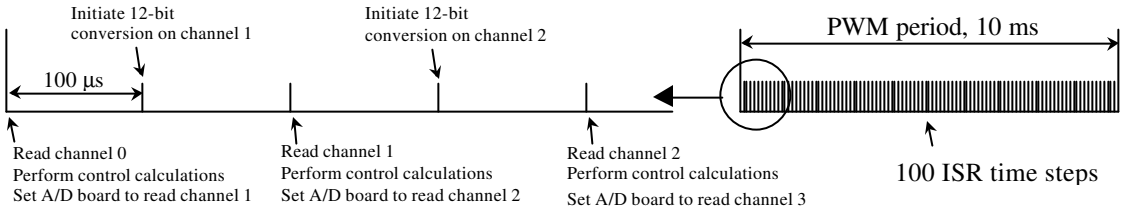


Figure 5-3

The interrupts were generated by a timing control board in the computer. Nelson designed the program so that the PWM modulation frequency could easily be adjusted in the code with the alteration of two lines. This provided a good method to see if it was possible to increase the PWM frequency and improve the robot performance. Some initial testing showed that frequencies above 100 Hz caused the computer to occasionally lock-up, probably due to one ISR not completing before the next one should start. The code for the control program is included in Appendix B.

5.3 Inverse Kinematics

The desired motion of the trolley was just passive horizontal sliding due to the frictional force at the foot. During walking motion, the air cylinder was held in the extended position. Therefore, the inverse kinematics were developed for a 2-DOF planar arm. Figure 5-4 is a schematic of the leg.

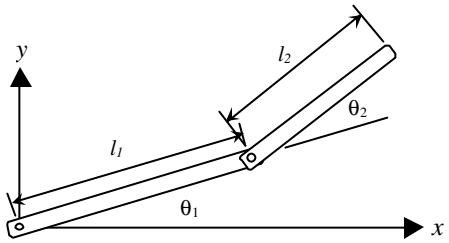


Figure 5-4

The forward kinematics for this 2-DOF leg are given by:

$$x_{des} = l_1 \cos\theta_1 + l_2 \cos(\theta_1 + \theta_2) \quad [61]$$

$$y_{des} = l_1 \sin\theta_1 + l_2 \sin(\theta_1 + \theta_2) \quad [62]$$

Where: x_{des} = Desired x-coordinate of foot
y_{des} = Desired y-coordinate of foot

Solving [61] and [62] for θ_1 and θ_2,

$$\theta_2 = \pm \text{acos}\left(\frac{x_{des}^2 + y_{des}^2 - l_1^2 - l_2^2}{2 l_1 l_2} \right) \quad [63]$$

$$\theta_1 = \text{atan}\left(\frac{l_1 + l_2 \cos\theta_2}{l_2 \sin\theta_2} \right) \pm \text{acos}\left(\frac{y_{des}}{\sqrt{l_1^2 + l_2^2 + 2 l_1 l_2 \cos\theta_2}} \right) \quad [64]$$

The desired trajectory was generated as a function of time, then [63] and [64] were used to calculate the desired angles. The leg lengths, l_1 and l_2, were 8.875" and 6.25", respectively. In the interest of processor time, the angles were left in terms of the 12-bit numbers generated by the A/D board. The high-level controller converted the desired angles in [63] and [64] from radians to a comparable 12-bit number. The forces were also left in 12-bit numbers, but a conversion factor was placed in the stiffness equation to convert both the 12-bit force and the 12-bit angle to yield stiffness units of lb-in/rad. Therefore, the stiffness control gain modulated stiffness units of lb-in/rad, and the angle control gain modulated units of angle counts (each integer in a 12-bit number is a count).

Chapter VI: Results and Discussion

6.1 Desired Walking Behavior

The ultimate goal of the project was to make a robotic leg walk using braided pneumatic actuators as the prime movers. Therefore, a desired trajectory was developed to tell the leg how to walk. Before the trajectory could be generated, however, the robots' range of motion needed to be determined. In Chapter 4, a theoretical joint range of motion was discussed. However, the minimum length of an actuator only occurred when the force on the actuator was zero. Also, the effect of gravity on the mass of the leg segments was not accounted for. Therefore, the full range of motion of an actuator was not achievable with an antagonistic configuration. Table 6-1 shows the actual joint excursion limits.

Table 6-1: Joint Range of Motion

Joint	Maximum Extension Angle	Relaxed Joint Angle	Maximum Flexion Angle	Range of Motion
Hip	249°	278°	307°	57°
Knee	343°	314°	289°	54°

Table 6-1 suggests the actual maximum range of motion was about 60°, depending on the weight and configuration of the segments. The lack of precision in which the actuator free lengths were assured during assembly also had an effect on the range of motion. This explains why the hip joint, with more leg mass attached, had a larger excursion than the knee joint, which just had the tibia attached.

With the range of possible joint angles determined, the next step was to develop a set of equations for the desired foot trajectory of the walking motion. A simple arc for the swing phase and straight lines for the plant/propulsion phase were chosen. To create the arc, three pass-through points were chosen, and a second order curve fit was

generated. The curve was y_{des} as a function of x_{des}. x_{des}, was chosen as a linear function of time.

The next phase of the walking motion was the foot plant. This short motion was used to begin moving the foot in the negative x direction before the foot actually touched the ground. This is known as velocity matching. Had the foot hit the ground while heading in the positive x direction, it would create a friction force that could send the robot backward. This trajectory was chosen to be linear in x, y. Again, x_{des} was generated by a linear time based function.

In the propulsion phase, y_{des} was held constant and x_{des} was a linear function of time. Figure 6-1 is a plot of the desired foot position.

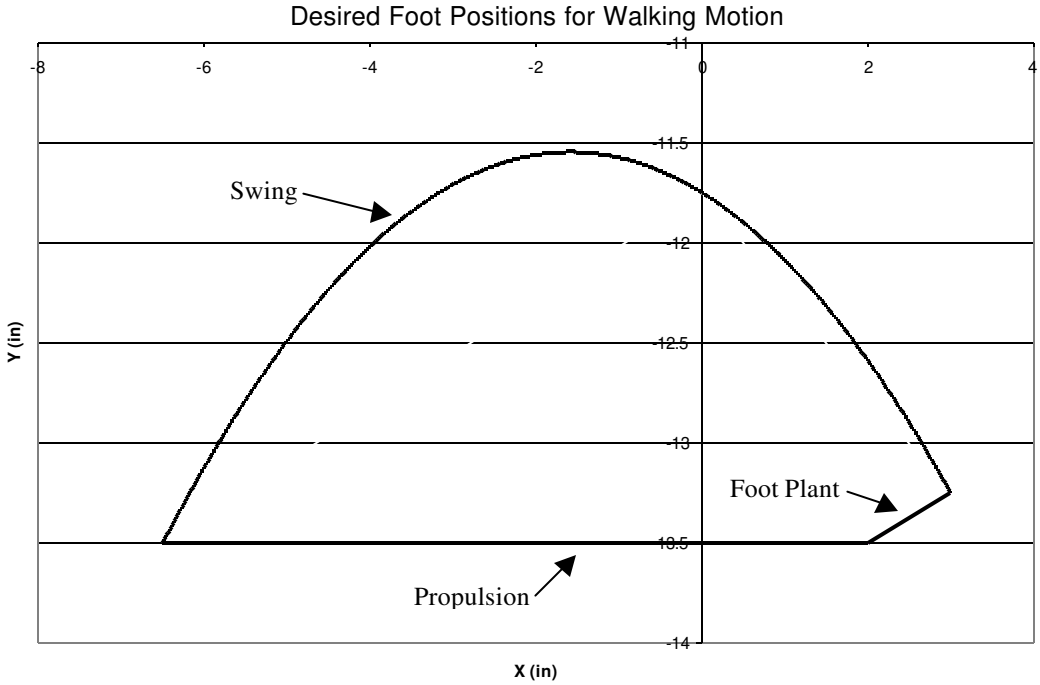


Figure 6-1

The equations for each section were:

Swing: $$x_{des} = -6.5 + 9.5\frac{time}{\Delta t_s} \qquad [65]$$

$$y_{des} = -0.081x_{des}^2 - 0.2571x_{des} - 11.75 \quad [66]$$

Foot Plant: $$x_{des} = 3 - 1.0\frac{time}{\Delta t_{fp}} \quad [67]$$

$$y_{des} = 0.25x_{des} - 14 \quad [68]$$

Propulsion: $$x_{des} = 2 - 8.5\frac{time}{\Delta t_p} \quad [69]$$

$$y_{des} = -13.5 \quad [70]$$

Where: Δt_s = Total time chosen for swing phase
Δt_{fp} = Total time chosen for foot plant phase
Δt_p = Total time chosen for propulsion phase

All of the coordinates were measured from the center of the hip rotational joint. The time variable was generated by incrementing a counter in the ISR. After every 100 ISR's, the time was incremented by 10 ms. This way, if the program had enough time to calculate the desired trajectory multiple times before the next PWM time period, then it would just calculate the same number again, and not generate new positions so fast that the robot could not physically attain them. Also, when one walking phase was completed, the time counter was reset to zero. This timing system is what is referred to as the pseudo real time operating system.

Two tests were run using the walking motion described above. Table 6-2 shows the time parameters used for each test.

Table 6-2: Time Parameters for Walking Motion

Test	Δt_s	Δt_{fp}	Δt_p
1	1.5 s	30 ms	1.5 s
2	0.75 s	30 ms	0.75 s

6.2 Tuning

The goal of the project was to make the robotic leg walk well. For example, it is not good walking if it just thrashes around and happens to cause forward motion of the trolley. Good walking must include stable, sensible motions. Tuning the control gains was a very important part of making the leg walk well. To tune the system, the leg was lifted up so that the foot would not touch the ground, and Test 1 walking motion was implemented. The behavior was observed and, through trial and error, appropriate control gains were converged upon. Table 6-3 shows the control gains that were determined.

Table 6-3: Walking Motion Control Gains

Joint	G_θ	G_k
Hip	0.035	6.5
Knee	0.12	5.0

Table 6-3 shows that the tuning for each joint was different. This is interesting because the actuators, cams, and sensors, are the same at both joints. The difference can be attributed to the different rotational inertia at each joint. The hip joint has a much higher inertia than the knee joint, and became unstable at much lower angle gains. Also note the higher stiffness gain at the hip joint. It was observed that a higher stiffness gain increased the dynamic stability limit of the joint as G_θ was increased.

Even though the desired stiffness of each joint could be changed each time period, during all the tests, they were held constant. Desired joint stiffness values at the hip and knee joints were 35 and 20 lb-in/rad, respectively. The selection of these values also had a significant effect on the motion of the joint. It was observed that as the desired joint stiffness was increased, the stability limit of the joint increased, to a point. Increasing G_k solved some problems, but created others. When the desired stiffness was increased, the

joint range of motion was decreased. This was due to the need for the greater mass of air in the actuators. Remember, a high stiffness correlated to a high force output of both actuators. If both actuators were filled with near supply pressure air to achieve the desired stiffness, not much could be done to rotate the joint while maintaining the stiffness. The balancing act between G_θ , G_k, desired stiffnesses, and joint inertia's suggest the intricacy of the control problem.

6.3 Walking Results

It is difficult to write a paper about a moving object and show how well it works. Therefore, the walking results are presented in two different ways. The first method is to show a sequence of frames from a video of the leg walking. Unfortunately, this only gives a reader a general idea as to how the leg moved. A reader could get a better feel for the motion through a quantifiable method that would also include any oscillations or quick motions which cannot be conveyed through this sequence of frames. While the leg was walking, the joint angles, actuator forces, calculated stiffnesses, and commanded duty cycles were all recorded to a data file. This output file was post processed and other information, such as ground reaction forces, were extracted.

Up to this point, the leg was simply moved around in the air. To test if the leg was able to effectively walk or not, it was lowered so that it could contact the table. The Test 1 program was then executed. The leg took two beautiful steps across the table before the trolley ran out of track. A piece of foam was wedged in one of the trolley rollers to reduce the coast length, and 4 steps were now possible. Figure 6-2a-h are the sequential frames of the leg during the walking motion.

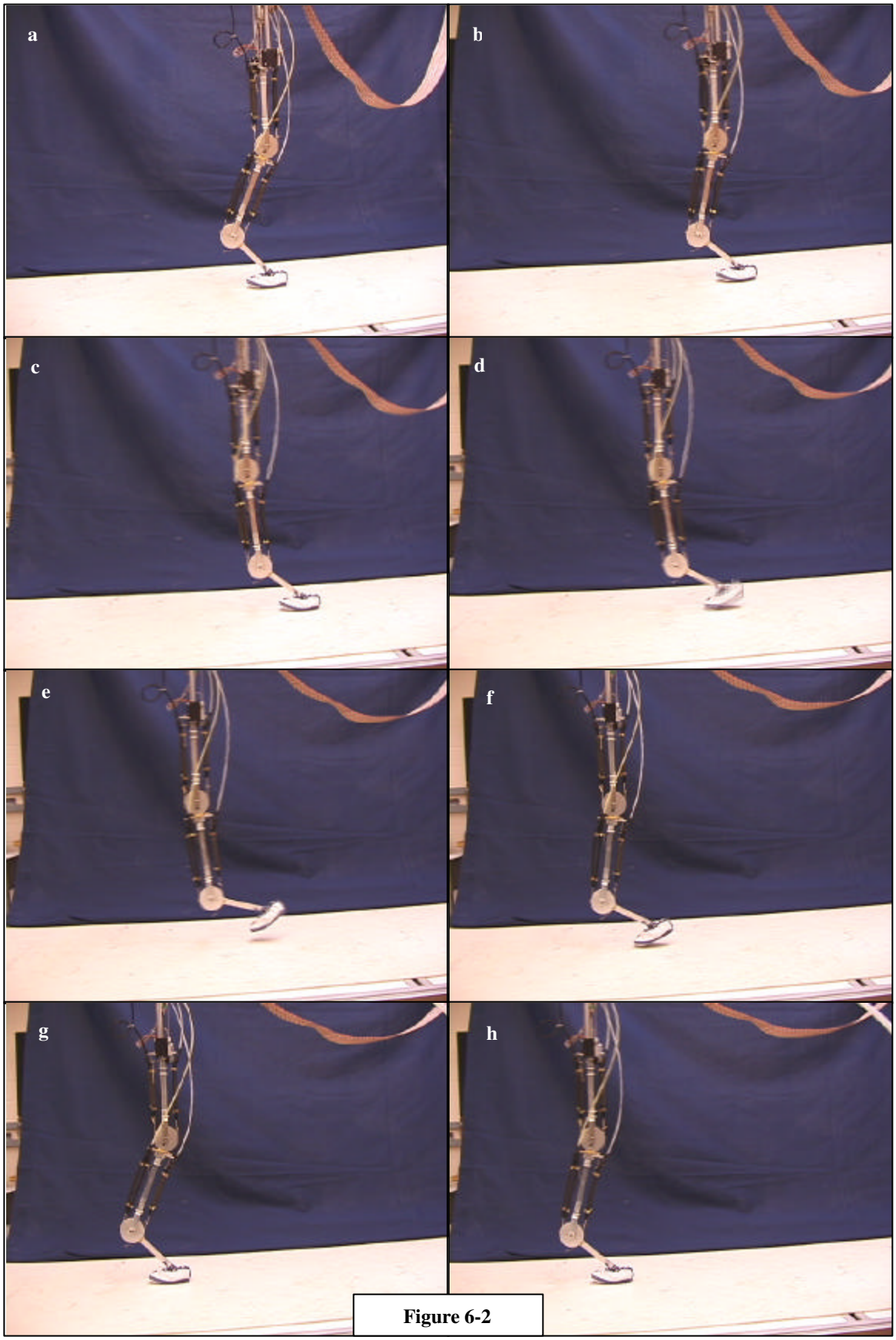


Figure 6-2

In the post-processing of the data file, the joint angles were converted from angle ticks to degrees. Forward kinematics were applied to the joint angles, and the desired x-y foot positions were compared with the actual. Figure 6-3 is a plot of the two foot paths.

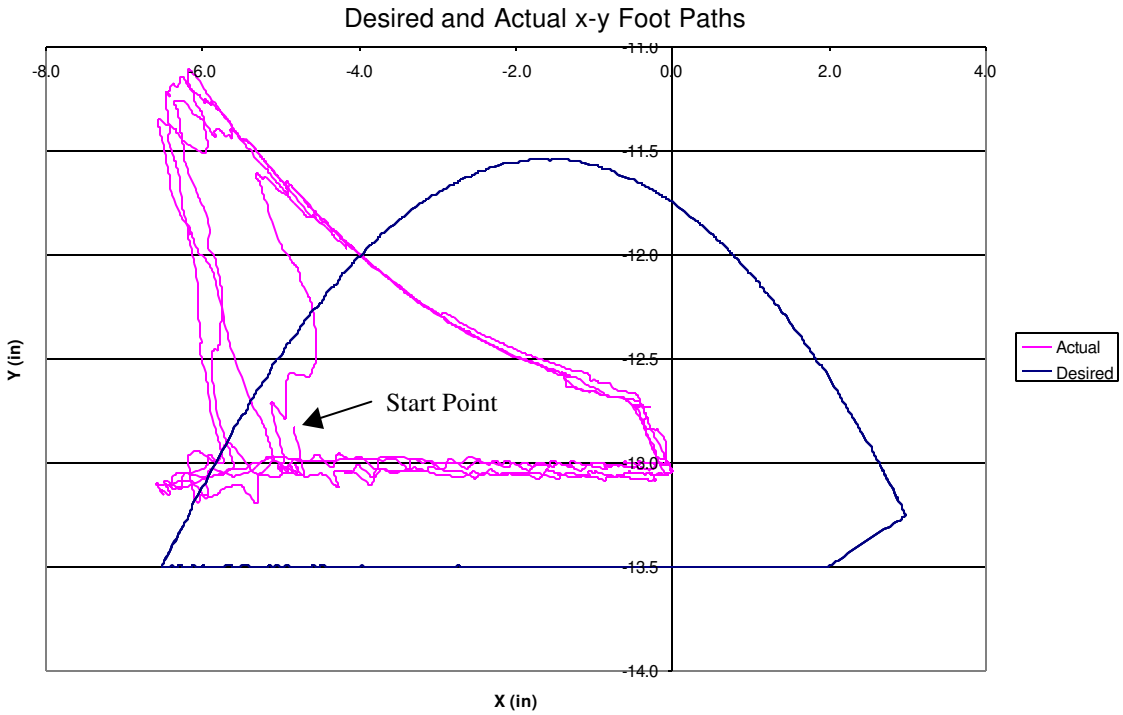


Figure 6-3

Figure 6-3 shows the y coordinate of the table to be about –13". The 0.5" difference between the desired and actual possible was an intentional gap. This gap caused the leg to create the necessary amount of force to propel the trolley forward. Remember, error causes more actuator force output in this feedback loop. When the gap was reduced to zero, the foot simply slid across the table without providing any forward motion. Figure 6-3 suggests a poor end effector position control algorithm. This may be true, but the important thing to remember is that during walking, precise position control is not as important as stable, sensible actions that cause forward motion. Figure 6-3 contains data from four steps overlaid. The actual foot path was a fairly smooth, stable motion. It was also very repeatable. The one deviant path in the middle was the initial

path that the foot took from the resting position to move towards the desired trajectory. It is also important to note that it was joint angles, not x-y coordinates that were being controlled. Figure 6-4 is a comparison plot of desired and actual joint angles.

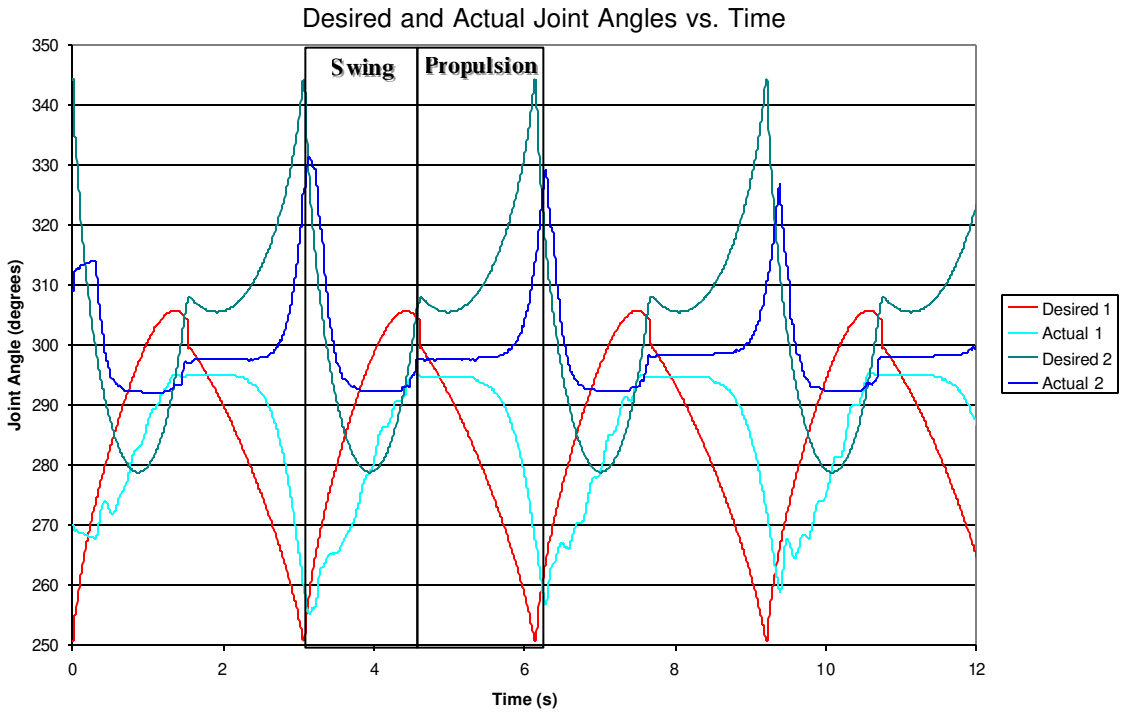


Figure 6-4

Figure 6-4 suggests that the position control algorithm was not as inefficient as Figure 6-3 suggested. The slight time delay shift characteristic of proportional control was apparent. With the exception of Joint 2 during a small section of the swing phase, all of the desired joint angles were within the joint range of motions given by Table 6-1. However, Figure 6-4 shows that the joints were not even able to reach these limit positions. This can be attributed to the joint stiffness/range of motion trade-off described in section 6.2. This explains why Figure 6-3 was not an exact match during the swing phase. There was also a larger than normal angle error during the propulsion phase. This was simply due to the fact that the table stopped the foot 0.5" before it was able to reach its desired propulsion coordinates. Figure 6-4 shows an adequate position control

scheme. It could be improved by putting the desired values within the stiffness dependent attainable range of motion. This figure also shows that there were a few small oscillations in the hip joint at the beginning of each swing phase.

Other data was recorded from the actuator force sensors. The correct scale factors were used in post processing, so the values in Figure 6-5 are as close as possible to the actual values.

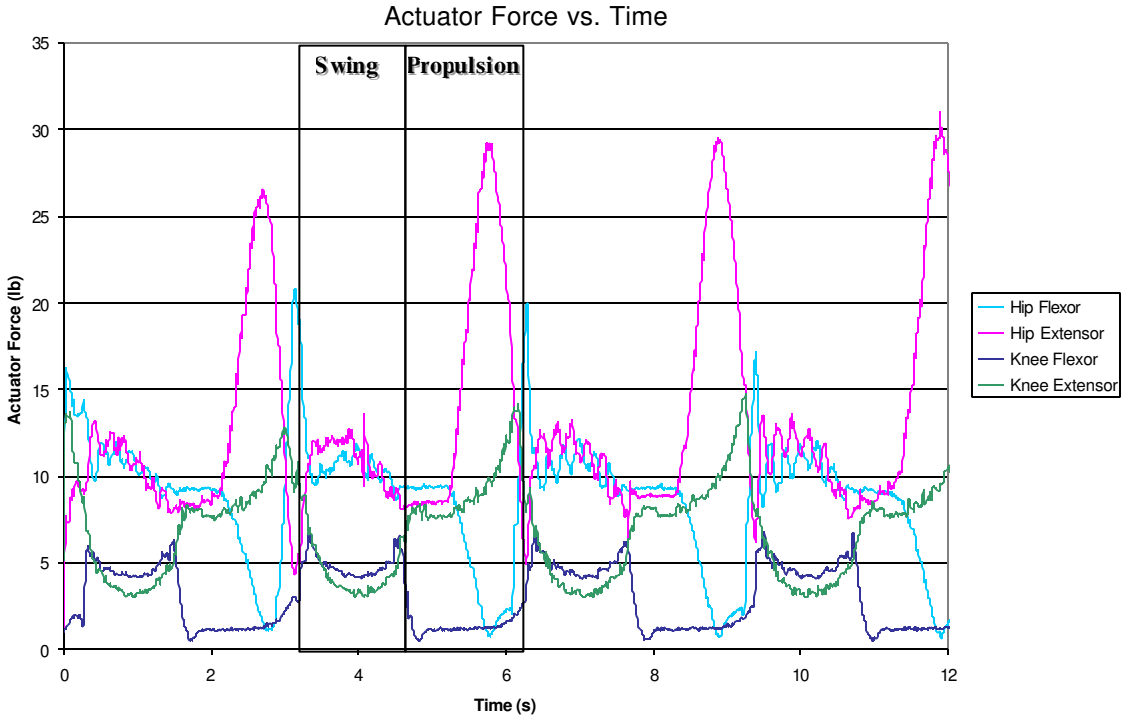


Figure 6-5

The maximum force was near the predicted 30 lbs. As was expected, during the swing phase the antagonistic actuator forces were quite close. During the propulsion phase, the actuator forces deviated in opposite directions as the ground applied external forces. Figure 6-5 also suggests that only the Hip Extensor actuator was being used to the maximum capacity. This meant that simply putting another actuator in the hip flexor bundle could increase maximum load capacity of the leg. This type of plot would be a good tool to optimize any leg design with BPA even after the leg was designed and built.

The second control law in the control program was the stiffness control. Figure 6-6 is a comparison plot of desired and actual stiffness.

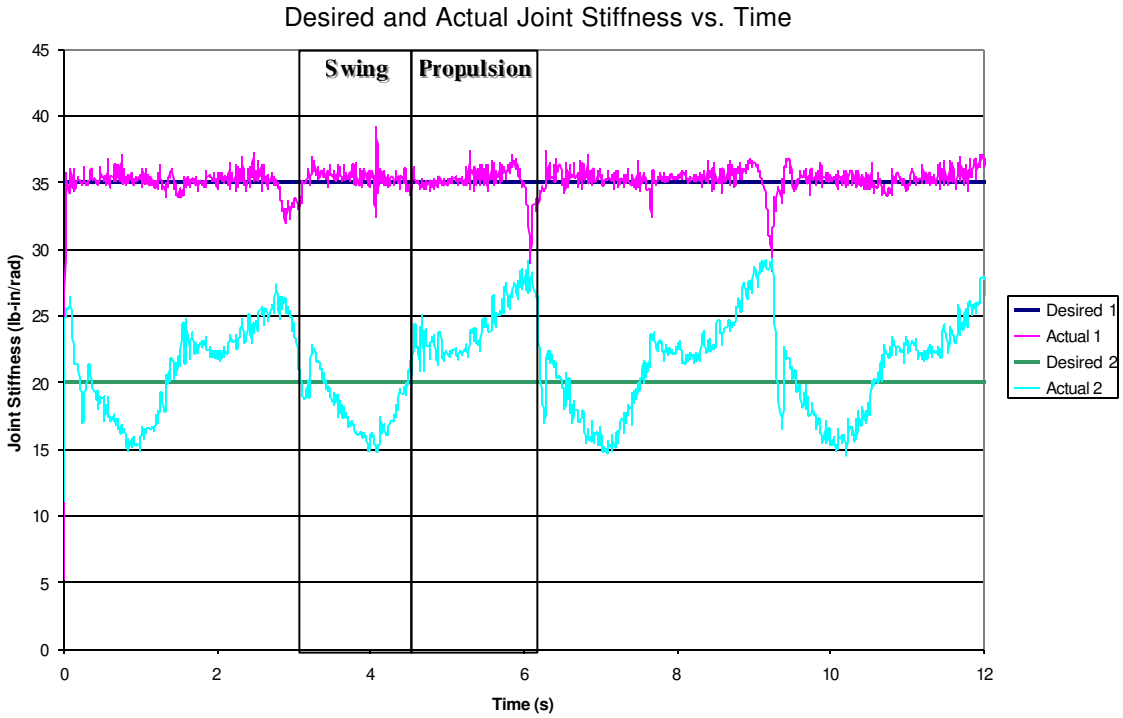


Figure 6-6

The hip joint stiffness was much more tightly controlled than the knee. This was due to the higher control gain at the hip. The knee joint was not as tightly controlled, but it really wasn't that important. The motion of the knee joint was very sensible already. Recall that a higher stiffness control gain increased stability. The control gain at the hip was increased to control the swing oscillations that were evident in Figure 6-4. Due to the small rotational inertia of the tibia, there were no noticeable oscillations at the knee joint. This allowed the stiffness control gain, and desired stiffness value to be lower while still maintaining stable, sensible motion.

The joint torque was calculated by subtracting the output of the antagonistic force sensors. Figure 6-7 is a plot of joint torque as a function of time.

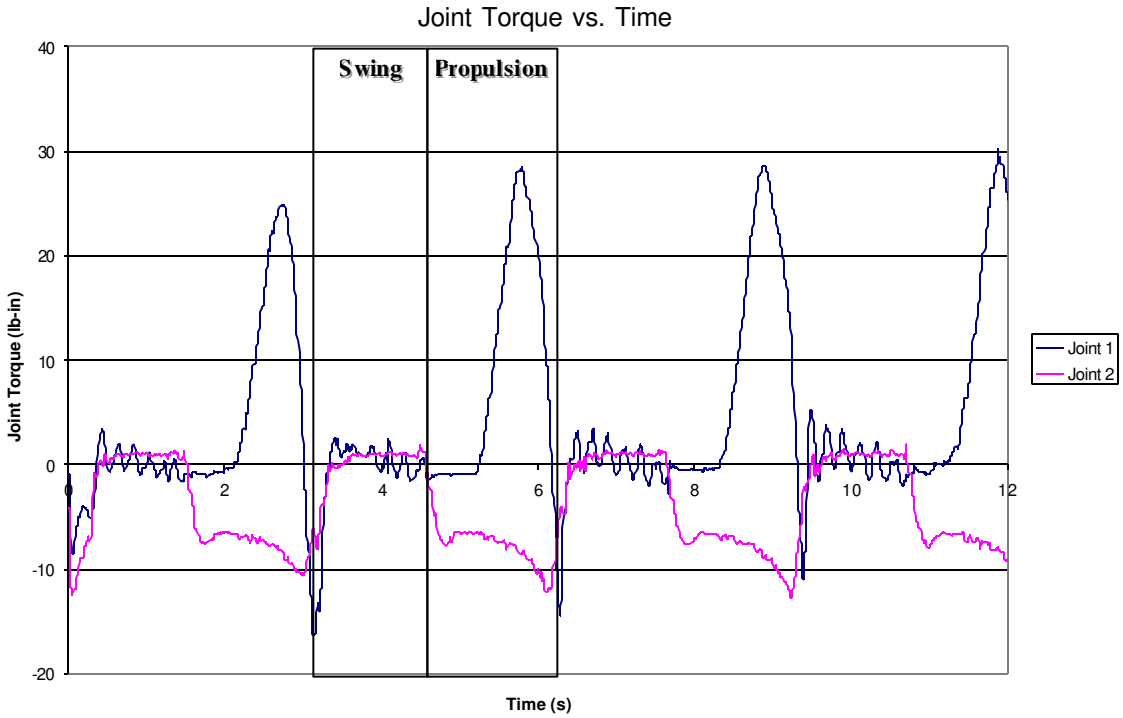


Figure 6-7

During the swing phase, the torque should have been, and was, near zero. Only the inertial and gravity effects caused a difference in actuator force. The slight oscillation of the hip joint was apparent in the torque data. Also, notice the slope of the torque signal during the swing phase. This was due to the change in gravity induced torque as the femur and tibia orientations changed. When the foot was set on the ground, the normal and friction forces at the ground applied an external moment to the leg. The actuators at the joints carried these moments with magnitudes given in the propulsion phase of Figure 6-7. Not only is the joint torque interesting, but so is what can be gleaned from it. Given the joint positions and the joint torques, the friction and normal forces at the foot can be determined as a function of time. Assuming the inertial and gravitational moments are negligible, the joint torques and foot forces are related by the following equation.

$$\underset{\sim}{T} = \underset{\sim}{J}^T \underset{\sim}{F} \qquad [71]$$

Where: T = 2x1 Joint Torque Matrix
J = 2x2 Jacobian Matrix
F = 2x1 Foot Force Matrix

$$J = \begin{bmatrix} -l_1 \sin\theta_1 - l_2 \sin(\theta_1 + \theta_2) & -l_2 \sin(\theta_1 + \theta_2) \\ l_1 \cos\theta_1 + l_2 \cos(\theta_1 + \theta_2) & l_2 \cos(\theta_1 + \theta_2) \end{bmatrix} \quad [72]$$

$$F = \begin{bmatrix} F_{friction} \\ F_{normal} \end{bmatrix} \quad [73]$$

$$T = \begin{bmatrix} T_{hip} \\ T_{knee} \end{bmatrix} \quad [74]$$

Solving [71] for Force,

$$\underset{\sim}{J}^{T^{-1}} \underset{\sim}{T} = \underset{\sim}{F} \quad [75]$$

Equation [75] was applied to the data in Figures 6-4 and 6-7, and the ground reaction forces are shown in Figure 6-8.

Figure 6-8

Fy is the normal force, and Fx is the friction force. These calculations ignored inertial and gravity effects, which caused some false reaction forces during the swing phase. However, they were minimal in relation to the forces during propulsion. Therefore, if the ground reaction forces are desired as feedback in a control algorithm for this leg, ignoring the inertial terms will provide quick and accurate results. The minimal effect inertia had during swing also suggests that the ground reaction forces during propulsion were accurate. The initial spike in the normal force during the propulsion phase occurred when the foot was first set on the ground and tried to reach the desired y position of –13.5". Then, as the desired x position traveled backwards, the foot began to push more and more to propel the trolley forward. This is why the friction force did not begin as soon as the foot hit the ground. This behavior was also observed in Figure 6-4. At the beginning of propulsion, the joints did not rotate. It wasn't until about the middle of propulsion that the joints began to rotate.

The strength of the leg was impressive. Figure 6-8 suggests that a total of 5 lbs of mass could have been transported with this leg during the walking motion above. Even more impressive than that was the fact that the leg was strong enough to propel the trolley along the track. The total weight of the whole system was a substantial 23.8 lbs with a 0.8 lb drag force due to the friction roller and umbilicals.

With the mass and external forces known, trolley acceleration as a function of time was calculated. The acceleration was then integrated twice to determine trolley velocity and position as a function of time. A linear numerical integration was used. During the swing phase, the friction forces were ignored, and the deceleration was just due to the drag forces. Figure 6-9 is a plot of calculated trolley position and velocity.

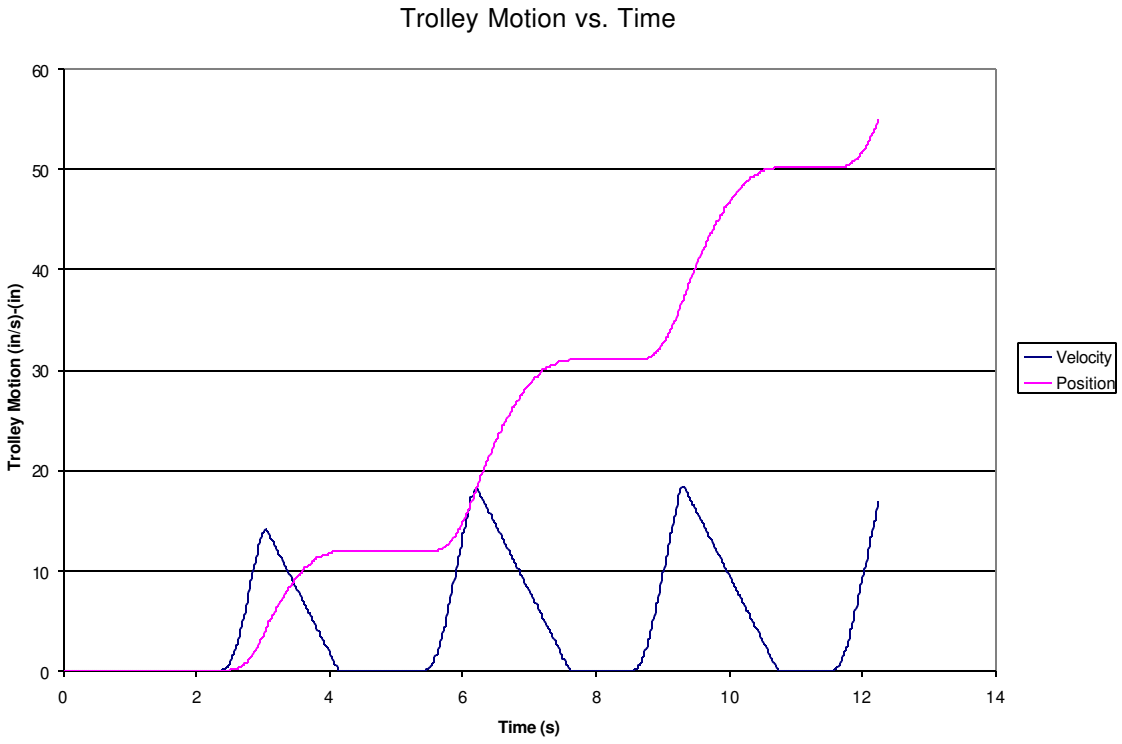


Figure 6-9

Figure 6-9 shows the total distance traveled to be about 5 ft. These are very good results, because the track on which the trolley ran was about 5 ft long. The average velocity was 4.5 in/s. Obviously, if the drag forces were minimized, the average velocity would be greater. In fact, when the friction was removed from the roller, the leg ran out of track after 2 steps. This suggests that the average velocity could be about 9 in/s.

An important goal of this project was to have to the motion and control be as passive as possible. This meant opening the valves for as little time as possible. Not only did this reduce electrical power consumption, but it also minimized the air consumption. These were two characteristics that are very important for autonomy. The commanded duty cycles were recording during the walking motion, and Figure 6-10 contains these values for the hip joint as a function of time.

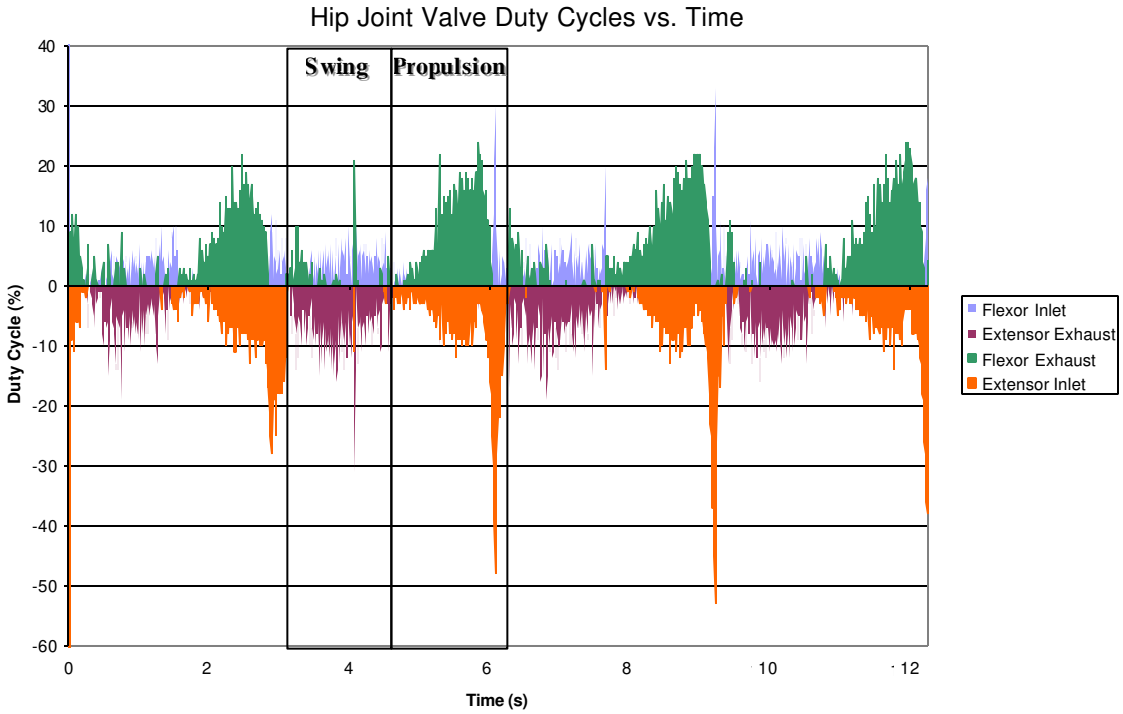


Figure 6-10

Figure 6-11 is a plot of the knee joint commanded duty cycles during the walking motion.

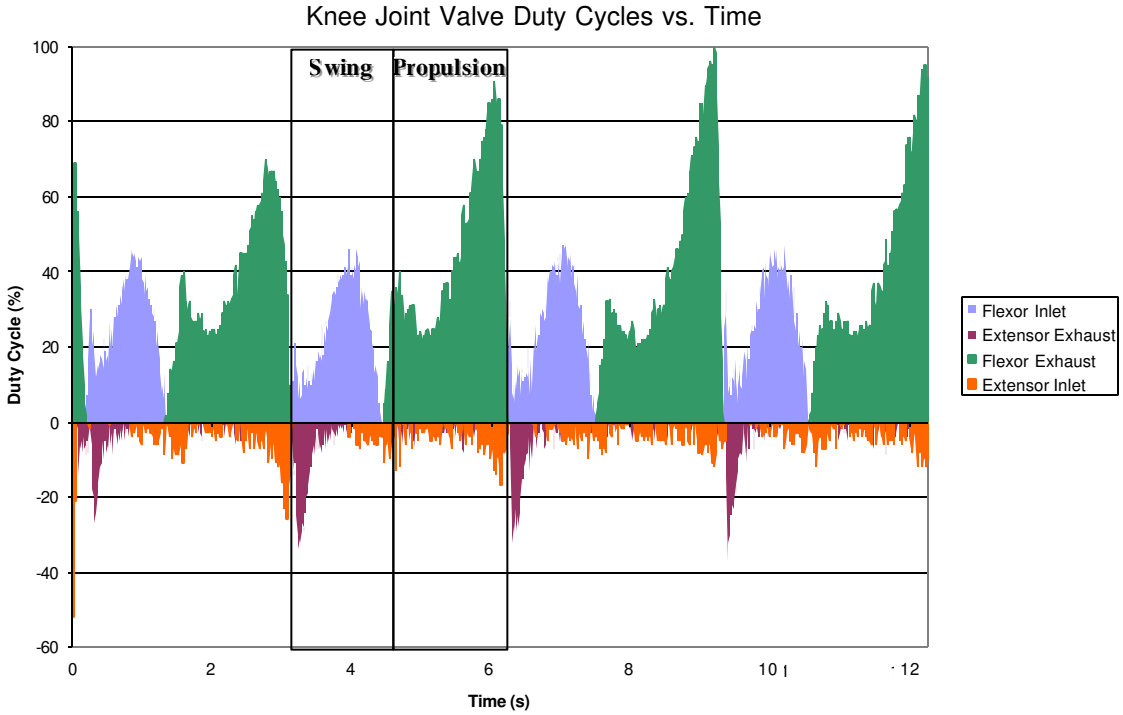


Figure 6-11

In both plots, the duty cycles that were plotted in the negative domain were not actually negative duty cycles, as this does not make sense. To make the plots easier to read, the flexor valve values were plotted in the positive domain, and the extensor valve values were plotted in the negative domain. Figure 6-10 shows the hip joint duty cycle values to be small. The flexor duty cycles in Figure 6-11 were much larger than those at the hip joint. Fortunately, this was not a function of the control algorithm. The increased flexor inlet duty cycles were a by-product of the fact that during the swing phase the desired joint angle was beyond the possible range of motion. If the desired angle had been achievable, the flexor inlet duty cycles would have been much smaller. The highest duty cycle value in Figure 6-11 is the flexor exhaust. The exhaust valve was opened in order to reduce the joint stiffness (Figure 6-6). However, air was not wasted and autonomy was not compromised because the flexor inlet duty cycles were much less than the flexor exhaust. Therefore, for most of the propulsion phase the flexor actuator contained atmospheric pressure air even though the exhaust valve was still being operated. This phenomenon is also apparent in Figure 6-10. The flexor inlet valve duty cycles were less than the exhaust.

The duty cycle values for the valves were combined to create a quantification of passiveness of the system. An average duty cycle value can provide a general idea as to how much air was used. It also provides an estimate of how much of the control was active, which was a measure of how often did the actuators just “do the right thing” without the control system responding to external disturbances to keep the system stable. Table 6-4 shows the average duty cycle for each valve. By design, the inlet and exhaust valves never operate at the same time. Therefore, the passiveness for each actuator is

given by subtracting the sum of these values from 100%. Then, the passiveness at each joint is given by the least passive of the flexor or extensor. According to Figures 6-10 and 6-11, the flexor and extensor valves tend to operate at the same time. If they operated at different times, then the joint passiveness would be the sum of the passiveness of the two actuators. The total robot passiveness was given by the least passive of the joints.

Table 6-4: Passiveness and Average Duty Cycles of Each Valve

	Flexor		Extensor		
Joint	Inlet	Exhaust	Inlet	Exhaust	Passiveness
Hip	2%	4%	3%	3%	94%
Knee	11%	23%	2%	3%	66%
			Total Passiveness		66%

With the current configuration, the robot passiveness was 66%. Table 6-4 also suggests that if the knee flexor valve's duty cycles were reduced, in accordance with the discussion above, the robot passiveness could be as high as 94%. This very high passiveness value can be attributed to three things. The two 2-way valves provide a convenient method to trap air in the actuators. However, just using the valves is not enough, the valve output conversion logic, which does not allow both inlet and exhaust valves to be open at the same time, minimizes air consumption. The final factor is the excellent passive properties of the braided pneumatic actuators. Similar to biological muscles, they are self-limiting in their force output. The more they contract, the less force they can generate (Figure 2-8). Also, minor length disturbances have a minimal effect on the force output. This goes back to the idea presented in Chapter 2 that, in practice, dP_g/dL is negligible for these actuators. If the pressure changes minimally with length, then the force output change is also minimal according to [22]. This is quite different than an air cylinder, where, when the rod is compressed the ideal gas law predicts a large internal pressure change, and thus a large force change.

6.4 Robot Limitations

Up to this point, the leg has performed very well at the given tasks. However, there will be limits to the walking motion. How fast can the leg walk? What is the speed stability limit? Are quick motions possible? Does a more stable motion require a smaller range of motion? Is there a rotational inertia limit for stability? If the desired angles are generated too fast, does it cause unwanted oscillations? There were many justifiable concerns that could only be answered by changing some parameters and observing the effect.

The first test was to increase the rate at which the desired trajectory was generated. The time parameters from Test 2 were used. The goal was to find the speed stability limit. Figure 6-12 is a comparison plot of desired and actual joint angles.

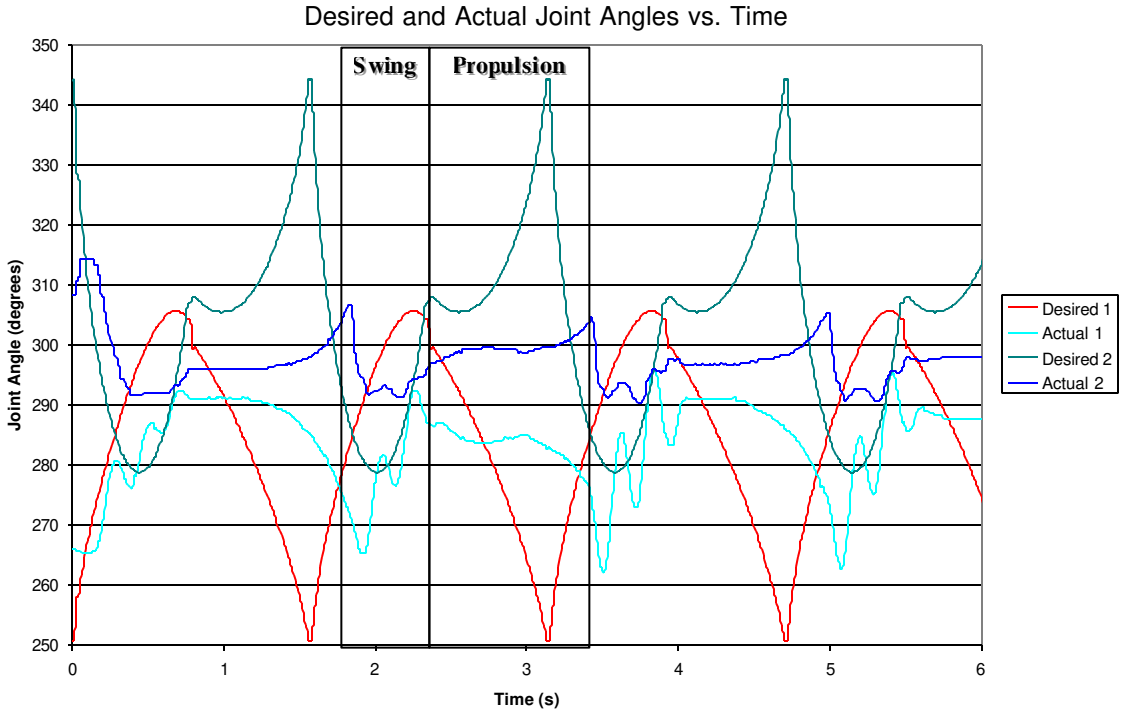


Figure 6-12

Figure 6-12 shows the walking motion to be not as stable as that of the previous test. There were oscillations at both joints, with substantial ones at the hip joint.

However, the leg did not go unstable. This was due to the passiveness of the system. There was also another downside to the faster walking. The forward travel of the trolley was only 21", less than half that of Test 1. The average velocity for the walking was calculated to be about 3 in/s. The passiveness for this walking was 92%, slightly less than Test 1. There were two hypotheses to explain why the fast trajectory generation degraded the walking.

The first hypothesis is that the actuators are not capable of performing quick motions. To test this hypothesis, a desired kicking trajectory was developed. Figure 6-13 is an x-y plot of the desired and actual foot position. Each marker on the actual curve is a data point taken every 10 ms.

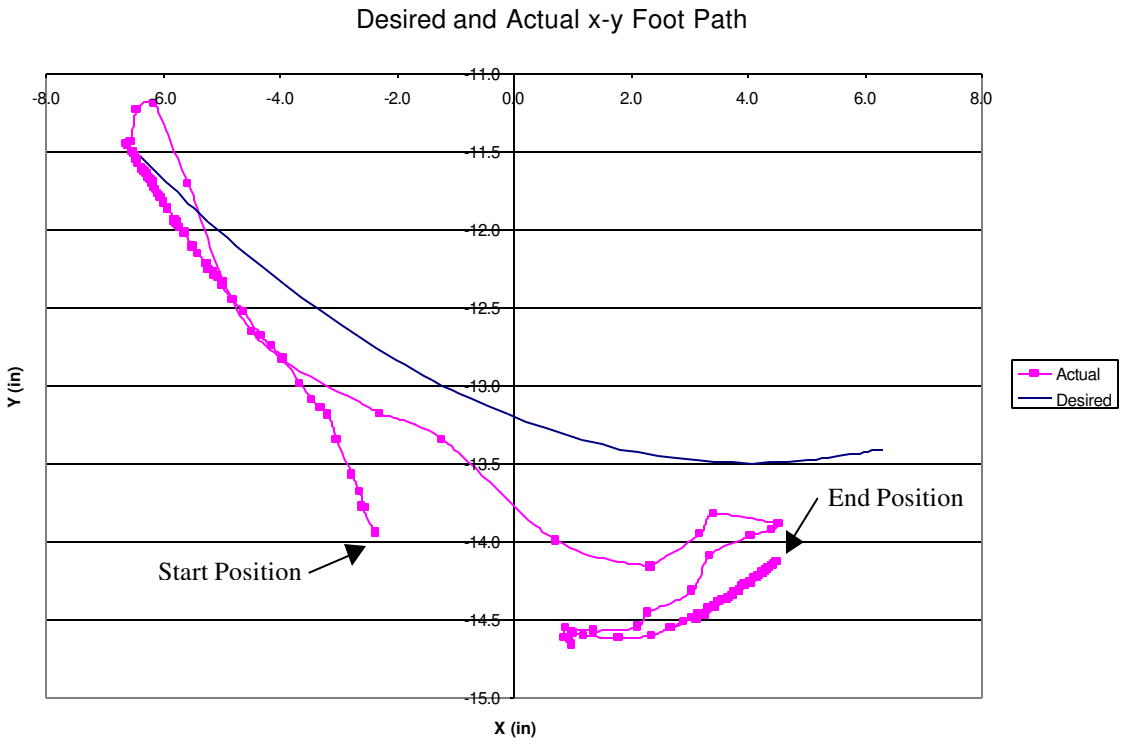


Figure 6-13

Figure 6-13 suggests that the leg response time is excellent. From the retracted position to the extended position, the forward motion took place in about 90 ms. To achieve such a quick response time, the desired stiffness and control gains were modulated during the motion. The control gains for the walking motion were too low to allow the leg to respond well to this trajectory. This suggested that it was beneficial to design the high level controller to be capable of modulating all the system parameters in real time. Figure 6-14 contains the angles for this trajectory.

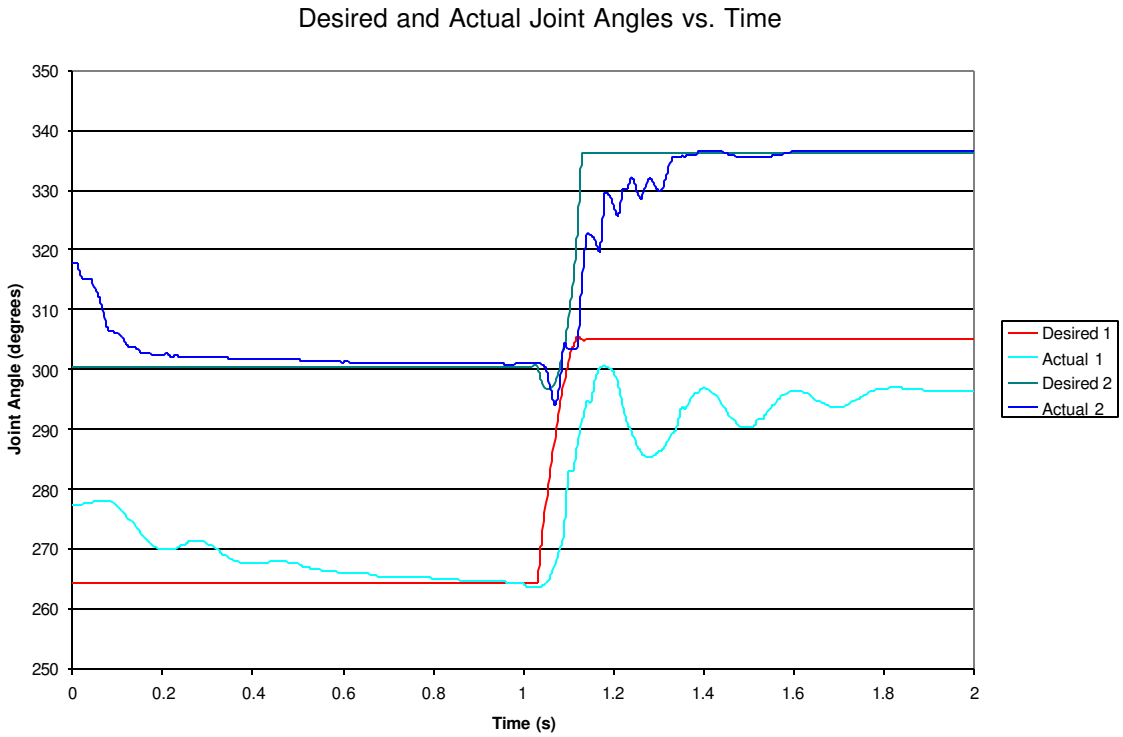


Figure 6-14

The quick response time was also apparent in Figure 6-14. However, the oscillations were still present in the system. This suggests that the reason the fast trajectory generation degraded the walking was not the speed capability of the actuators and valves, rather, there was not enough damping in the system. In fact, this is the problem that has prevented the leg from achieving its' full capability. Higher angle

control gains would produce better motion, but higher control gains cause the leg to oscillate and/or go unstable. It is simply a matter of not enough damping in the system. There are two different methods to increase the apparent damping at the joint. One is to physically place a damping mechanism on the leg, and the other is to put derivative control in software.

Another limitation of the robot is the fatigue life of the actuators. A total of about 200 steps were taken with this robot. In that time, three actuators failed. The actuators failed when the latex tubing ruptured, and the air leaked out the hole. It was not clear what caused the bladders to rupture. It could have been the constant rubbing of the bladder and the mesh which developed a weak spot. There also could have been a slight defect in the latex tubing, which caused a stress concentration. All of the failures seemed to be in the middle section of the actuator.

6.5 Derivative Control

Derivative control was implemented in the control code by storing both the previous joint angle and the current joint angle. The two angles were subtracted, and then divided by 10 ms, the time between samples. This value was the estimated velocity of the leg. The velocity was multiplied by a velocity control gain, and subtracted from U_θ to decrease the duty cycle as the velocity increased. This method works well in theory, but in practice it is usually not feasible. The angle sensor data is noisy, and the A/D assigns discrete values to the signal. Velocity at one time step may be calculated to be 0 angle counts/sec, and at the next be 1200 counts/sec. Figure 6-15 is a plot of the calculated angular velocity of the hip joint during the swing and propulsion of one step.

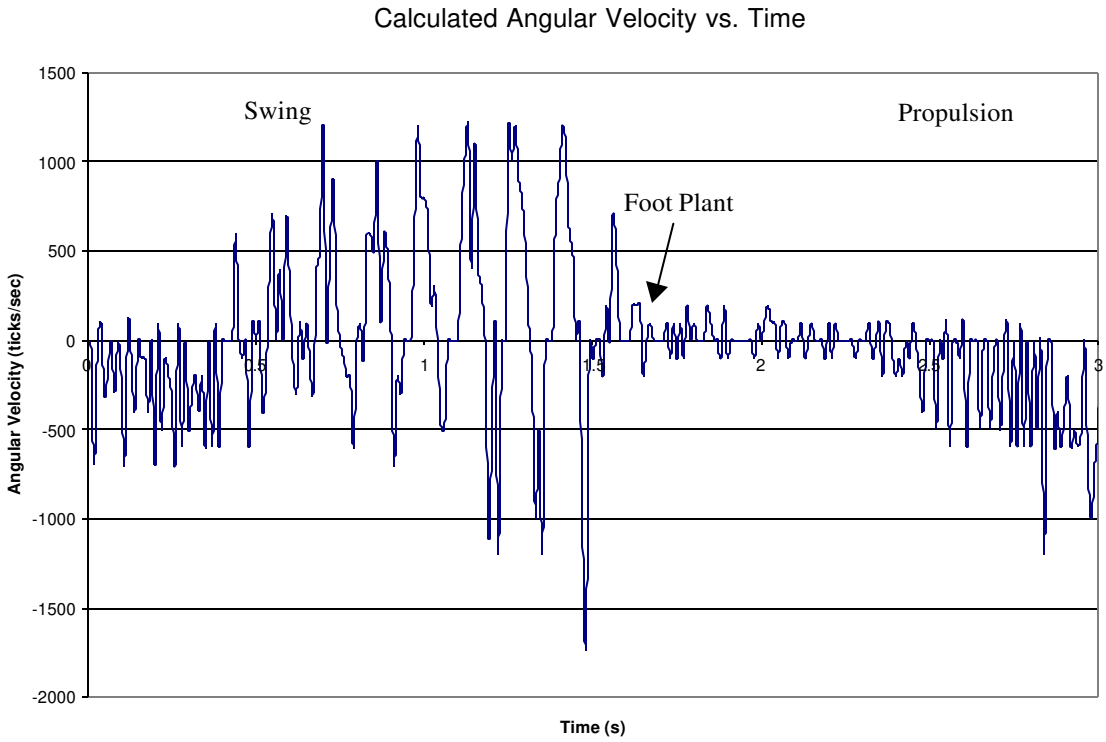


Figure 6-15

The signal is very noisy. Imagine taking this signal and using it to command the valves. It would probably cause the system to go unstable. Even though this damping solution seemed daunting, derivative control was implemented and the leg was commanded to walk. There seemed to be three different results depending on the control gain. If the gain was low enough, the derivative control was useless, and the walking motion was no different. The gain could also be set higher, and the leg would go grossly unstable. The third configuration was where the gain was high enough to change the leg motion, but low enough to still be quasi-stable. Figure 6-16 is a plot of the joint angles vs. time.

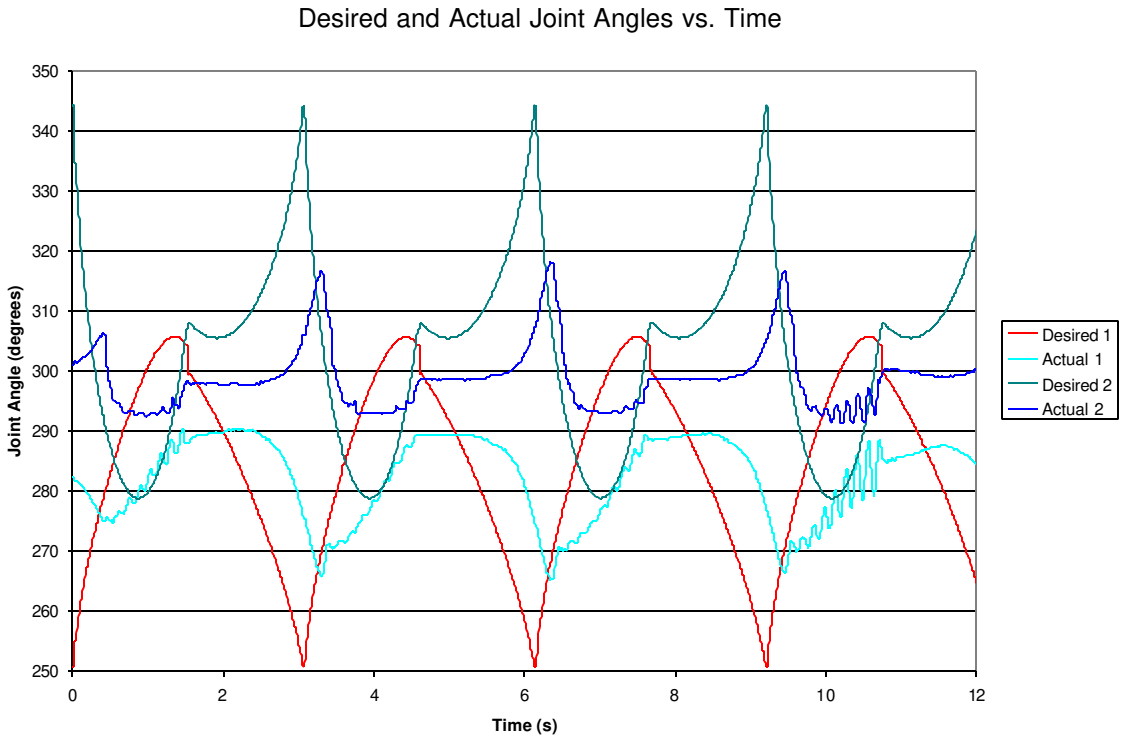


Figure 6-16

During all swing phases, oscillations at both joints were apparent. The fourth swing motion, in particular, appeared to be headed for instability. Fortunately, the foot was set on the ground, and the oscillations stopped. The joint excursions were also much smaller than those in Figure 6-4 were. Figure 6-16 suggests that derivative control is not a feasible option if the velocity is calculated from the position sensors with a single point history calculation.

Chapter VI: Conclusions

7.1 It Walks!

The ultimate goal of the project was to make a robotic leg walk using braided pneumatic actuators as the prime movers. The information and techniques developed in this thesis will be used in the design and control of Robot IV; a six legged cockroach-like robot outfitted with McKibbens. Though Robot IV will not be autonomous, eventually an autonomous robot is desired. The work presented here is a very useful tool for robot designers. Accurate static and dynamic models of the actuators were presented. An accurate model of a solenoid valve working in conjunction with an actuator for use in robotic simulations was also presented. The usefulness of these simulation models extends beyond Robot IV. The simulation can be tailored to fit any size robot. In fact, the models are being used to design and control a cricket-micro robot that will fit into a two inch cube.

In the position control of Robot III it was observed that the gains could be tuned to provide sensible swing motion, however, when the foot entered stance, the gains were too low to adequately move the robot. When the gains were increased so that the leg could move the robot while in stance, the higher gains caused the swing motion to become unstable. This was not even an issue in the control of this robotic leg. The leg walking motion was very stable and sensible regardless of whether or not the foot was on the ground.

The robot operated 66% passively. If the desired trajectory were changed, this number would probably be closer to 94%. This is very important to control, and also is very important to autonomy. Minimal valve operation implies that the controller is just

watching the system move, and then makes minor changes just to keep it in check. Minimal valve operation also implies that air consumption is being kept to a minimum.

Section 6.4 shows the quick motion capabilities of these actuators. This suggests that slow, deliberate motions of this leg are possible, but so are fast bursts of motion. Modulation of the system parameters in the controller make this possible.

In conclusion, braided pneumatic actuators are excellent devices for use as prime movers in legged locomotion. Their high strength to weight ratio, and the passive and self-limiting properties make them ideal actuators, provided that they can be made more reliable.

7.2 Semi-Observed Speculation

This section is simply an unconfirmed speculation. Due to the abundance of sensors necessary to provide stable and sensible motion with these actuators, two questions arise. Is the actuator force feedback necessary? Does just position control provide stable sensible walking? If the number of sensors on Robot IV can be minimized it would be beneficial to the speed of the control program. According to section 6.2, the joint stiffness, or actuator stiffness has an effect on joint stability. The same joint angle can be produced at an infinite number of stiffnesses. Therefore, even though the same angle can be achieved, each time it arrives at the desired angle, it will probably have a different stiffness. If desired trajectory was fixed, this robot is capable of 94% passiveness, so the motion of the leg is highly dependent on the passive properties of the actuators. For these reasons, stable and sensible walking is probably not achievable without force feedback. Even if it was stable, it would probably not be repeatable, and

therefore unpredictable. To test this theory, the stiffness control gains were set to zero, and the leg was commanded to walk. Figure 7-1 is a plot of x-y desired and actual foot positions.

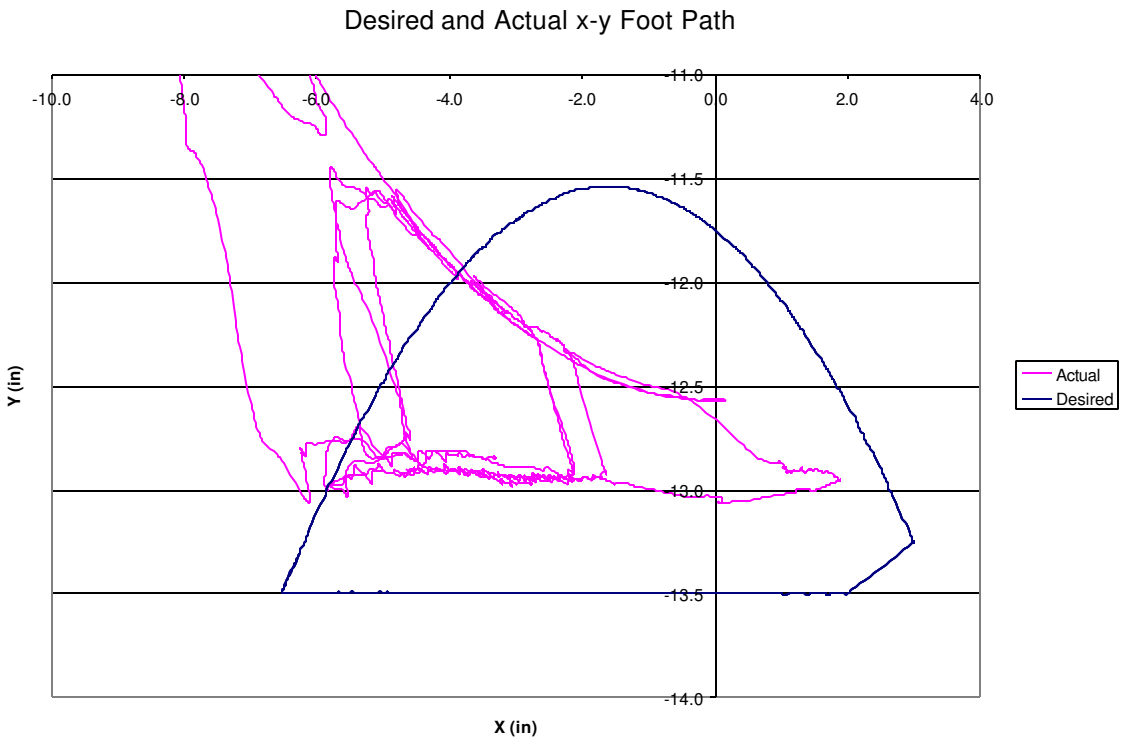


Figure 7-1

Figure 7-1 shows a non-repeating foot path. However, it is stable. The foot forces during this motion seemed to be much greater than Test 1 data. In fact, during this test, it was observed that the leg lifted the front trolley roller off the track in spite of the 10 lb mass hanging under it. Due to the inconclusive data, it was presented in this section and not included in Chapter 6. However, it would be prudent for someone to investigate this further.

7.3 Future Work

There are a few things that could be suggested to improve this robot, or control of these actuators in general. The two biggest problems are the light damping inherent in the actuators, and the frequent actuator failure.

Work needs to be done on understanding the actuator failures and increase the actuator life. Glenn Klute at the University of Washington Biorobotics Laboratory has performed a fatigue analysis of the actuators and found that a latex bladder has a fatigue limit 24 times greater than that of silicone rubber bladders (Klute et al., 1998). Unfortunately, the bladders in the actuators used in this project are already latex. Other bladder options need to be investigated. One idea is to use a mylar balloon that has a diameter larger than the fully contracted diameter, and a length longer than the maximum actuator length. This way, the mylar will just provide a seal, and all of the stresses will be in the mesh. This larger bladder idea could also be used to increase the fatigue life of the latex rubber bladder.

Derivative control was implemented in an effort to increasing damping, but it was unsuccessful. There are three possible solutions. Design a velocity sensor that can be used in derivative control, place a mechanical damper on each joint, or implement derivative control using the position sensors, but utilize a more complex algorithm.

The first option was the direction chosen by Pierre-Henry Marbot at the University of Washington Biorobotics Laboratory. He developed the first prototype of a mechanical muscle spindle that actively measures muscle position and velocity (Marbot, 1995). His work has been continued by Kristen Jaax. The problem with this option is

that there is now one more sensor to read at each joint. Computational time is very important.

The second option is to use a mechanical damper. Airpot, the company that manufactured the air cylinder for this project, also makes low friction, variable orifice viscous dampers. They utilize the same graphite/Pyrex technology in their dampers. One could be placed at each joint. The one drawback is that the maximum velocity and quick motion capability of the robot will no longer be possible. It may be able to walk faster than it currently does, but there will be a velocity limit.

The third option could probably be a thesis in and of itself. Many people have spent a lot of time and energy developing complex, robust algorithms for controlling with discrete sensors and discrete time hold data. The simplest would probably be a weighted average scheme that looks at a discrete set of previous positions to develop a derivative controller. Other ideas include the observer algorithm where a simple dynamic model is used to determine reasonable velocities, and the position based "calculated" velocities are not allowed to exceed them. These are all good ideas, and should be investigated. However, there is a computational time trade-off. The low-level controller needs to be as efficient as possible and a more complex algorithm will only slow down the controller. A robust, efficient algorithm is critical.

This problem could be slightly alleviated by increasing the speed of the processor on which the control program ran. The computer is currently a Pentium processor operating at 127 MHz. If the ISR was run any faster than 100 Hz, the computer would have a tendency to lock-up. This was because one ISR wanted to start before the last was

completed. Increasing the processor speed would also allow for these more accurate and complex control algorithms to be placed in the low-level controller.

This is a paraphrase of a famous quote, “The more you know, the more you realize you don’t know.” This was definitely the case with investigating these actuators. These actuators have many desirable properties. However, with each new discovery it seemed as though there were more questions. Much work will still need to be completed to gain a full understanding of the physical properties of the actuators. Much work also needs to be done in improving the actuator design. Both these goals are reasonable and achievable, for this work is definitely a step in the right direction. These actuators hold a promising future for robotics.

Appendix A: Simulation Code

A.1 Actuator.cpp

```
/*---------------------------------------------------------------*/
/* Robb Colbrunn */
/* Valve with Braided Pneumatic Actuator Simulation */
/* 7-30-99 */
/* this is a simulation of a simple mass and spring system where
the spring is a Braided Pneumatic Actuator.  The coordinate system
is positive down (direction of gravity).  The simulation also
includes a valve model.  This is used to determine pressures
in the actuator.*/
#include<stdio.h>
#include<stdlib.h>
#include<math.h>

#define L_trap 37.                    // number of lengthwise trapazoids on actuator
#define C_trap 16.                    // number of circumferencial trapazoids on actuator
#define Trap_size 0.057               // length of trapazoid side (in)
#define VIS_DAMP_CONST .02    // viscous damping constant
#define COULOMB_DAMP_CONST .0045 // pressure proportional Coulomb damping constant = (uN/k)
#define MESH_STIFFNESS 500    // mesh stiffness
#define MESH_DAMPING .75

#define EXT_VOL 0.495      // Volume external to actuator, only downstream of valve (in^3) (i.e. hoses)
#define VALVE_EFF .85              // Valve efficiency

#define Supply_p 112.7             // Supply pressure (psia)
#define GAS_CONST 1.4              // gas constant for air
#define DENSITY 0.000001349        // density of air at stnd conditions (slug/in^3)
#define MASS (5./386.088)          // deadweight applied mass (slugs)

#define g 386.088                  // gravitational constant (in/s^2)
#define PI 3.14159265

#define dt 0.001                   // simulation time step (s)
#define max_simulation_steps 2000
#define DEBUG_FORCE_OFF       // compile flag for printing forces
#define STND_PRINT_ON         // compile flag for printing standard output
#define DEBUG_SIMPLE_OFF      // compile flag for printing just time and length
#define DEBUG_LENGTH_OFF      // compile flag for printing just length

FILE *out;
static int eqn_order;
static double h,h2,h3,h6;
static double *x_mid1,*x_mid2,*x_end;
static double *xdot_init,*xdot_mid1,*xdot_mid2,*xdot_end;
double *make_dvector(int nl,int nh);
static int valve_state = 0, rkflip = 1;
static double B, N, length_min;

void setup_rk4(int n);
void fnc_xdot(double x[5], double x_dot[5]);
void rk4(double *x_in,double *t_ptr,double *x_out);
```

```
void pressures(double x[4], double xdot[4]);

void main(void)
{
        int i, j=5, k=0;
        int n = 4;          // number of state variables
        double t = 0.0;     // time
        double x[4] = {3.4, 0.0, 14.7, 0.0}; //state vector (lngth,lngth_dot,pressure,mass)(in,in/s,psia,slugs)
        double vol_initial;

        // convert actuator characteristics into dimensions
        B = 2 * L_trap * Trap_size;                         // length of braided thread
        N = L_trap / C_trap;                                // number of turns for thread
        length_min = sqrt(pow(2 * Trap_size * L_trap, 2.)/3); // actuator length at max contraction

        // calculate initial mass
                // calculate intial actuator volume
                vol_initial = x[0]/(4.*PI*N*N)*(B*B - x[0]*x[0]) + EXT_VOL;
        x[3] = vol_initial * DENSITY * pow(x[2]/14.7,(1./GAS_CONST)); // initial mass

        // call rk4 setup function
        setup_rk4(n);

        out = fopen("out.dat","w");                 /* open output file */

#ifdef STND_PRINT_ON
     fprintf(out,"Time Length Velocity Pressure Mass Valve_State\n");/* write stnd header to output file */
#endif

#ifdef DEBUG_FORCE_ON
        fprintf(out,"Length Force_act_w/o_damp Damping_forces Force_act_with_damp\n"); /* write
force header to output file */
#endif

#ifdef DEBUG_SIMPLE_ON
        fprintf(out,"Time Length\n"); /* write simple time and length header to output file */
#endif

#ifdef DEBUG_LENGTH_ON
        fprintf(out,"Length\n"); /* write time and length header to output file */
#endif

        j=16;  // set duty cycle for PWM
//      for ( j = 1 ; j < 10 ; j++)
//      {
                for ( i = 0 ; i < max_simulation_steps ; i++)
                {

                        // print to output file

                        #ifdef STND_PRINT_ON
                        fprintf(out,"%f %f %f %f %e %d \n", t, x[0], x[1], x[2], x[3], valve_state);
                        #endif

                        #ifdef DEBUG_SIMPLE_ON
                        fprintf(out,"%f %f\n", t, x[0]);
```

```
                        #endif

                        #ifdef DEBUG_LENGTH_ON
                        fprintf(out,"%f\n", x[0]);
                        #endif

                        // control loop

                        // loop PWM counter
                        if (k >= 40)
                                k=0;

                        if (k < j)
                                valve_state = 1;   /* open to supply pressure */
                        else
                                valve_state = 2;  /* open to atmosphere */

                        //valve_state = 0;  // override control loop

                        // call rk4
                        rk4(x,&t,x);

                        k++; // increment PWM counter
//                      }
        //fprintf(out,"%d %f\n", j, x[0]);
        }

}

void fnc_xdot(double x[4], double xdot[4])
{
        double fact, fact_damping, fext, act_eff;
        int sign;

        // determine sign of velocity for use in damping equations
        if (x[1] >= 0.0)  sign = 1;
        else sign = -1;

        // caclulate pressure proportional actuator efficiency (empirically determined constants)
        // this efficiency is based on a 12mm x 150mm shadow robot actuator(stiffness.xls / efficiency)
        act_eff =  -0.00002 * x[2]*x[2] + 0.0005 * x[2] + 0.9969;

        // calculate actuator force
        fact =     PI/4. * pow(B/(N*PI),2.) * fabs(x[2] - 14.7) * ((3 * x[0] * x[0])/(B*B) - 1) * act_eff;

        fact_damping = x[1]*VIS_DAMP_CONST + COULOMB_DAMP_CONST*fact/(x[0] -
length_min)*sign;

#ifdef DEBUG_FORCE_ON
        if (rkflip>4) rkflip=1;
        if (rkflip==4) fprintf(out,"%f %f %f ", x[0], fact, fact_damping);
#endif

        fact = fact + fact_damping;

#ifdef DEBUG_FORCE_ON
```

```
        if (rkflip==4) fprintf(out,"%f \n", fact);
        rkflip++;
#endif

        // long end effect  (0.3 and mesh stiffness are empirically determined constants)
        if (x[0] > (B - 0.3))
                fact = fact + (MESH_STIFFNESS * (x[0] - (B - 0.3))) + MESH_DAMPING * x[1] ;

        // short end effect  (5 and 10 are empirical guesses that match data. it should really be fact = 0.0;)
        if (x[0] < length_min)
//              fact = 5 * fact + 10 * fact_damping;
                fact = 0.0;

        // calculate external force
        fext = 0.0;

        // derivative of position is velocity
        xdot[0] = x[1];

        // calculate acceleration (length_dbl_dot) of mass - equation of motion goes here
        xdot[1] = g - (fact - fext)/MASS;

        // calculate pressure_dot and mass_dot
        pressures(x,xdot);
}

void pressures(double x[4], double xdot[4])
{
 double flowrate, highp, lowp;
 int flowdir;
 double act_vol, vol[2];

        // calculate actuator volume
        act_vol = x[0]/(4.*PI*N*N)*(B*B - x[0]*x[0]);

        if (x[0] >= B) act_vol = 0.;

        vol[0] = act_vol + EXT_VOL;

        // calculate time rate of change of volume
        vol[1] = x[1] * (pow(Trap_size * C_trap, 2.)/PI - 3./(4.*PI*N*N) * x[0] * x[0]);

  switch (valve_state)
  {
   case 0: //cylinder chamber sealed
                flowdir = 0 ;
                break ;

   case 1: //open to supply
                if (Supply_p >= x[2])
                {
                 highp = Supply_p ;
                 lowp = x[2] ;
                 flowdir = 1 ;
                }
                else
```

```
                        {
                         lowp = Supply_p ;
                         highp = x[2] ;
                         flowdir = -1 ;
                        }
                        break ;

     case 2: //open to atmosphere
                        if (x[2] >= 14.7)
                        {
                         lowp = 14.7 ;
                         highp = x[2] ;
                         flowdir = -1 ;
                        }
                        else
                        {
                         highp = 14.7 ;
                         lowp = x[2] ;
                         flowdir = 1 ;
                        }
                        break ;
            }

   if (flowdir)
   {
// see Gabe's "piston model" notes, pp. 16.1 - 16.3, which is based
// on Ye, et al, "Models of a Pneumatic PWM Solenoid Valve for
// Engineering Applications," Trans. ASME Vol 114, Dec. 1992
     if (highp >= (1.89 * lowp)) //choked
            flowrate = flowdir*83.66*0.0165*highp * VALVE_EFF;
     else  //not choked
            flowrate = flowdir*83.66*0.064*highp * VALVE_EFF * sqrt(pow(lowp/highp,2./1.4) -
pow(lowp/highp,2.4/1.4)) ;
   }
   else
    flowrate = 0.0 ;

   xdot[2] = x[2] * GAS_CONST * ((flowrate * DENSITY)/x[3]-vol[1]/vol[0]) ; // pressure_dot

   xdot[3] = flowrate * DENSITY ;  // mass_dot

}

void setup_rk4(int n)
{
            x_mid1 = make_dvector(0,n); /* create x_mid1[1 ... n] */
            x_mid2 = make_dvector(0,n);
            x_end = make_dvector(0,n);
            xdot_init = make_dvector(0,n);
            xdot_mid1 = make_dvector(0,n);
            xdot_mid2 = make_dvector(0,n);
            xdot_end = make_dvector(0,n);
            eqn_order = n;
            h = dt;
            h2 = dt/2.0;
            h3 = dt/3.0;
```

```
        h6 = dt/6.0;
}

double *make_dvector(int nl,int nh)
{
        double *v;
        v = (double *)malloc((unsigned)(nh-nl+1)*sizeof(double));
        if(!v)
          { printf("allocation error in dvector; I give up \n");
            exit(0);
          }
     /* subtracting nl allows the vector index to start at nl */
        return v-nl;
}

void rk4(double *x_in,double *t_ptr,double *x_out)
{
        int i;
        *t_ptr += h;  /* increment time: t <-- t+dt */

        /* first step */
        fnc_xdot(x_in,xdot_init); /* get xdot at initial x */

    /*first estimate of midpt */
        for (i=0;i<eqn_order;i++)
          x_mid1[i] = x_in[i] + h2*xdot_init[i];

  /* second step */
        fnc_xdot(x_mid1,xdot_mid1);/* get xdot at first guess midpt*/

    /*second estimate of midpt */
      for (i=0;i<eqn_order;i++)
        x_mid2[i] = x_in[i] + h2*xdot_mid1[i];

  /* third step */
      fnc_xdot(x_mid2,xdot_mid2);/* get xdot at 2nd guess midpt*/

    /* estimate of final pt, x(t+dt),  based on best est of avg xdot */
      for (i=0;i<eqn_order;i++)
        x_end[i] = x_in[i] + h*xdot_mid2[i];

  /* fourth step */
      fnc_xdot(x_end,xdot_end);/* get xdot at estimated x(t+dt) */

  /* net result using weighted sum of four slope estimates */
      for (i=0;i<eqn_order;i++)
            x_out[i] = x_in[i] + h6*(xdot_init[i]+xdot_end[i])
         + h3*(xdot_mid1[i]+xdot_mid2[i]);

          //fprintf(out,"rk444444 %f %f %f %f\n", xdot_end[0], xdot_end[1], xdot_end[2], xdot_end[3]);

}
```

Appendix B: Controller Code

B.1 Control.cpp

```
// this is robb's version of gabe's code.
// started 10-12-99
// Notes:
// These are the valve numberings in this code.
// NOTE: valves come in pairs for each joint, in the order ext, flex
// 0        thru       5          = inlet for lin, hip, knee
// 8        thru       13         = exhaust for lin, hip,


// For pot A/D, the joints are numbered accordingly
// 0        thru       2   = lin, hip, knee

// For gage A/D, the sensors are numbered accordingly
// 3        thru       6          = hip flex, hip ext, knee flex, knee ext

#include "def.h"
#define ACT_THD_LNGTH_SQD 10465225L //squared length of braided actuator
                                    // thread length (ticks^2)
FILE *out;
//---------------------------------------------------------------------------
// Function prototypes
void shut_down_valves(void) ;               // hardware.cpp
int hardware_start_up() ;                   // hardware.cpp
int hardware_shut_down(void) ;              // hardware.cpp
void interrupt pwm_isr(__CPPARGS) ;         // control.cpp
void interrupt (*old_isr)(__CPPARGS) ;      // control.cpp
void interrupt_initial(void) ;              // control.cpp
void interrupt_restore(void) ;              // control.cpp
void invkin(float p[2], float tdes[3]) ;    // control.cpp

//void interrupt test(__CPPARGS) ;

//---------------------------------------------------------------------------

int irq_mask, old_mask ;
int irq_chn = PWM_IRQ ;

//static short event_count = 0 ;

static short global_cycle_time = -1 ;
static int timer = 0 ;

// This is the counter
// 'cycle_time' gets set to 0 at the beginning of each cycle and is incremented
// to 100, at which, in the same cycle, it is reset to 0.
static short cycle_time[3] ;

// Prefixes
// 'ei' = extensor inlet
// 'ee' = extensor exhaust
// 'fi' = flexor inlet
```

```
// 'fe' = flexor exhaust

// Holds the current duty cycle value, as a percentage.
static short eiducyc[3] ;
static short fiducyc[3] ;
static short eeducyc[3] ;
static short feducyc[3] ;

static short feiducyc[3] ;
static short ffiducyc[3] ;
static short feeducyc[3] ;
static short ffeducyc[3] ;

// Holds flexor and extensor feed forward duty cycles for linear actuator.
static short effducyc ;
static short fffducyc ;

// Holds desired angles and stiffnesses
static short thetades[3] ;
static float stiffdes[3] ;

// The DIO bit which switches the valve
static short eibit[3], eebit[3], fibit[3], febit[3] ;

static short eipstate[3], fipstate[3], eepstate[3], fepstate[3] ;      // The previous valve state

static short *eiportcommand[3], *fiportcommand[3], *eeportcommand[3], *feportcommand[3] ;

// If any 'portcommand' bit was changed,
// 'portdelta' becomes nonzero positive
static short *eiportdelta[3], *fiportdelta[3], *eeportdelta[3], *feportdelta[3];

// valve timing curves, read from 'predict.dat'
// (see Lotus file 'predict.123')
static short closetiming[101] ;
static short opentiming[101] ;

// control law gains from gain.dat
static float tgain[3] ;
static float kgain[3] ;
static float vgain[3] ;

static int potreading[3] ;     // 12 bit pot reading
static int prevpotreading[3] ;           // previous state pot reading
static int gagereading[4] ; // 12 bit gage reading
static short gagenum ;
static short jointnum ;
static float stiffness[3];
static short length1 ;
static short length2 ;
static short free_length ;
static short potoffset[3] ;

static short Uk ;
static short Ut ;
```

```
static short adchannum[7] ;
static short chan ;

static short port0command = 255 ;
static short port1command = 255 ;

static short port0delta = 0 ;
static short port1delta = 0 ;

static short joint_enabled[3] ;

void invkin(float p[2], float tdes[3])
{
 /* calculate theta2 and theta1 values */
        tdes[2] = - acos( (p[0]*p[0] + p[1]*p[1] - L1*L1 - L2*L2) / (2.0 * L1 * L2) );
        if(p[0]<0.0)
                tdes[1] = atan2( L1 + L2*cos(tdes[2]) , L2*sin(tdes[2]) )
                                + acos( p[1] / sqrt( L1*L1 + L2*L2 + 2.0*L1*L2*cos(tdes[2]) ) );
                else
                {
                tdes[1] = atan2( L1 + L2*cos(tdes[2]) , L2*sin(tdes[2]) )
                                - acos( p[1] / sqrt( L1*L1 + L2*L2 + 2.0*L1*L2*cos(tdes[2]) ) );
                tdes[1] = 2.*PI + tdes[1];
                }
        tdes[2] = 2.*PI + tdes[2];
}
void interrupt_initial(void)
{
        int vect_no ;

        if (irq_chn==0)
                return ;

        if (irq_chn < 8)
                vect_no = 0x08 + irq_chn ;
        else if (irq_chn == 9)
                vect_no = 0x0A ;
        else
                vect_no = 0x70 + irq_chn - 8 ;

        disable() ;
        old_isr = getvect(vect_no) ;
        setvect(vect_no,pwm_isr) ;

        enable() ;

        // As far as I can tell, interrupt masking works like this.  If you
        // write a high bit ('1') to the PICU, that masks (turns off) that
        // interrupt from the PICU.  Thus, if you want to listen to an interrupt,
        // you unmask it by writing a low bit ('0') to the PICU.
        //
        // BTW, here are the interrupt priorities, highest to lowest:
        // IRQ 9,10,11,12,14,15,3,4,5,6,7
        //
        if (irq_chn < 8)
                {
```

```
                irq_mask = inp(IC8259_1 + 1) ;
                old_mask = irq_mask ;
                outp(IC8259_1 + 1 , irq_mask & (0xFF^(1<<irq_chn))) ;
                }
        else
                {
                irq_mask = inp(IC8259_1 + 1);
                // IRQs 8 and higher are cascaded from the second PICU to
                // the first through IRQ 2 !!!  Thus we had better unmask IRQ2
                // if we are using a IRQ higher than 7.
                outp(IC8259_1 + 1 , irq_mask & 0xFB); /* IRQ2 : 1111 1011 */
                // Notice, we access the second PICU here, 'IC8259_2'
                irq_mask = inp(IC8259_2 + 1);
                old_mask = irq_mask ;
                outp(IC8259_2 + 1 , irq_mask & (0xFF^(1<<(irq_chn-8))));
                }

        return;
}

//---------------------------------------------------------------------------

void interrupt_restore(void)
{
        int vect_no;

        if (irq_chn == 0)
                return ;

        if (irq_chn < 8)
                vect_no = 0x08 + irq_chn ;
        else if    (irq_chn == 9)
                vect_no = 0x0A ;
        else
                vect_no = 0x70 + irq_chn - 8 ;

        // Restore the old interrupt mask
        if (irq_chn < 8)
                outp(IC8259_1 + 1 , old_mask) ;
        else
                outp(IC8259_2 + 1 , old_mask) ;

        // Restore the old interrupt handler
        setvect(vect_no,old_isr) ;

        return ;
}

//---------------------------------------------------------------------------

void interrupt pwm_isr(__CPPARGS)
{
        register short i ;

        if (port0delta)
                {
```

```
            cbDOut(BOARDNUM,FIRSTPORTA,port0command) ;
            port0delta = 0 ;
            }
if (port1delta)
            {
            cbDOut(BOARDNUM,FIRSTPORTB,port1command) ;
            port1delta = 0 ;
            }

// enable interrupts
enable() ;

global_cycle_time++ ;
if (global_cycle_time >= 100)
{
            timer++;
            global_cycle_time = 0 ;
}

// On global_cycle_time, (chan):
// even < 14                  we initiate a 12 bit conversion
// odd < 14                   we read an A/D channel, set up next channel
// 0...6, (0...2)       we are working with pots
// 7...14, (3...6)      we are reading gages, controlling joints
// 8...99               we do nothing with A/D

switch ((chan = (global_cycle_time / 2)) < 7)
            {
// WE ARE...-------------------------------------------------------------
            case TRUE:
            switch (global_cycle_time % 2)      // odd or even cycle?
                        {
            // READING A...-----------------------------------------------
                        case TRUE:        // odd cycle
                        switch (chan < 3) // pot or gage this time?
                                    {
                        // POT AND CONTROLLING A JOINT.-------------------------
                                    case TRUE:        // pot
                                    // read pot channel
                                    potreading[chan] = (inpw(CB_BASEADDR) >> 4) ;

                                    switch (chan)
                                    {
                                                case 0:
                                                cycle_time[chan] = 0 ;

                                                eeducyc[0] = tgain[0] * (thetades[0] - potreading[0]) ;
                                                fiducyc[0] = fffducyc + eeducyc[0] ;
                                                feducyc[0] = fffducyc - eeducyc[0] ;
                                                eiducyc[0] = effducyc - eeducyc[0] ;
                                                eeducyc[0] = effducyc + eeducyc[0] ;

                                    /*          fiducyc[0] = ffiducyc[0];
                                                feducyc[0] = ffeducyc[0];
                                                eiducyc[0] = feiducyc[0];
                                                eeducyc[0] = feeducyc[0];
```

```
		*/

			// remove erroneous duty cycles
			if (eiducyc[chan] > 100)
				eiducyc[chan] = 100 ;
			else if (eiducyc[chan] < 0)
				eiducyc[chan] = 0 ;

			if (eeducyc[chan] > 100)
				eeducyc[chan] = 100 ;
			else if (eeducyc[chan] < 0)
				eeducyc[chan] = 0 ;

			if (fiducyc[chan] > 100)
				fiducyc[chan] = 100 ;
			else if (fiducyc[chan] < 0)
				fiducyc[chan] = 0 ;

			if (feducyc[chan] > 100)
				feducyc[chan] = 100 ;
			else if (feducyc[chan] < 0)
				feducyc[chan] = 0 ;

			if (eipstate[chan] & VALVEONNEXT)
				eipstate[chan] = VALVEON ;
			if (eepstate[chan] & VALVEONNEXT)
				eepstate[chan] = VALVEON ;
			if (fipstate[chan] & VALVEONNEXT)
				fipstate[chan] = VALVEON ;
			if (fepstate[chan] & VALVEONNEXT)
				fepstate[chan] = VALVEON ;

eipstate[chan] = (eipstate[chan] & 0xFE) + joint_enabled[chan] ;
eepstate[chan] = (eepstate[chan] & 0xFE) + joint_enabled[chan] ;
fipstate[chan] = (fipstate[chan] & 0xFE) + joint_enabled[chan] ;
fepstate[chan] = (fepstate[chan] & 0xFE) + joint_enabled[chan] ;

			break ;
		}

		break ;

// GAGE. ----------------------------------------------
		case FALSE:	// gage

		// find the gage number
		gagenum = chan - 3 ;

		// read gage channel
		gagereading[gagenum] = (inpw(CB_BASEADDR) >> 4);

		switch (gagenum % 2) // flexor or extensor gage reading
			{
			case TRUE:  // extensor (odd)

				jointnum = (gagenum + 1)/2 ;
```

```
                                cycle_time[jointnum] = 0 ;

// calculate actuator length
length1 = free_length + (potreading[jointnum] - potoffset[jointnum]) ;
length2 = free_length - (potreading[jointnum] - potoffset[jointnum]) ;

// calculate stiffness (lb/in)
stiffness[jointnum] = 5.5 * 6. * (gagereading[gagenum - 1]/(3. * length2
                                - ACT_THD_LNGTH_SQD/length2)
                                + gagereading[gagenum]/(3. * length1
                                - ACT_THD_LNGTH_SQD/length1)) ;

// calculate control outputs (duty cyle)
switch (jointnum) //flexor and extensor move pot differently at different joints
{
case 1: //hip
Ut = tgain[jointnum] * (thetades[jointnum] - potreading[jointnum])
            - vgain[jointnum] * ((potreading[jointnum]
            - prevpotreading[jointnum])/0.01);  //angle
           break;
case 2: // knee
Ut = tgain[jointnum] * (-thetades[jointnum] + potreading[jointnum])
            - vgain[jointnum] * ((prevpotreading[jointnum]
            -  potreading[jointnum])/0.01);  //angle
           break;
}
prevpotreading[jointnum] = potreading[jointnum];

Uk = kgain [jointnum] * (stiffdes[jointnum] - stiffness[jointnum]);  //stiffness

                                // calculate duty cycles
                                fiducyc[jointnum] =  Uk + Ut;
                                feducyc[jointnum] = -Uk - Ut;
                                eiducyc[jointnum] =  Uk - Ut;
                                eeducyc[jointnum] = -Uk + Ut;

            /*                  fiducyc[jointnum] = ffiducyc[jointnum];
                                feducyc[jointnum] = ffeducyc[jointnum];
                                eiducyc[jointnum] = feiducyc[jointnum];
                                eeducyc[jointnum] = feeducyc[jointnum];
             */
// remove erroneous duty cycles
if (eiducyc[jointnum] > 100)
           eiducyc[jointnum] = 100 ;
else if (eiducyc[jointnum] < 0) // change zero to some value for dead band
           eiducyc[jointnum] = 0 ;
if (eeducyc[jointnum] > 100)
           eeducyc[jointnum] = 100 ;
else if (eeducyc[jointnum] < 0)
           eeducyc[jointnum] = 0 ;
if (fiducyc[jointnum] > 100)
           fiducyc[jointnum] = 100 ;
else if (fiducyc[jointnum] < 0)
           fiducyc[jointnum] = 0 ;
if (feducyc[jointnum] > 100)
           feducyc[jointnum] = 100 ;
```

```
                    else if (feducyc[jointnum] < 0)
                            feducyc[jointnum] = 0 ;

                    if (eipstate[jointnum] & VALVEONNEXT)
                            eipstate[jointnum] = VALVEON ;
                    if (eepstate[jointnum] & VALVEONNEXT)
                            eepstate[jointnum] = VALVEON ;
                    if (fipstate[jointnum] & VALVEONNEXT)
                            fipstate[jointnum] = VALVEON ;
                    if (fepstate[jointnum] & VALVEONNEXT)
                            fepstate[jointnum] = VALVEON ;

                    eipstate[jointnum] = (eipstate[jointnum] & 0xFE) + joint_enabled[jointnum] ;
                    eepstate[jointnum] = (eepstate[jointnum] & 0xFE) + joint_enabled[jointnum] ;

                    fipstate[jointnum] = (fipstate[jointnum] & 0xFE) + joint_enabled[jointnum] ;
                    fepstate[jointnum] = (fepstate[jointnum] & 0xFE) + joint_enabled[jointnum] ;

                    break;

                                    // just reading flexor gages (even)
                                    case FALSE:
                                    break;

                                    }

                            break ;
                            }

                    // set multiplexer to next channel
                    outp(CB_BASEADDR + 2, adchannum[(chan == 6) ? 0 : chan+1]) ;
                    break ;

            // INITIATING A 12BIT CONVERSION.-------------------------------
                    case FALSE:     // (even cycle)
                    outp(CB_BASEADDR + 1, 0) ;
                    break;
                    }
            break ;

// WE ARE NOT DOING ANYTHING WITH A/D.---------------------------------

            case FALSE:
            break ;
            }

for (i=0;i<3;i++)
            {
            cycle_time[i]++ ;

            // check/change valve states

            switch (eipstate[i])
                    {
                    case VALVEON + JOINTENABLED:
                    // watch for valve-off time
```

```
                if (cycle_time[i] >= closetiming[eiducyc[i]])
                        {
                        *eiportcommand[i] |= eibit[i] ;         // set valve off bit
                        (*eiportdelta[i])++ ;                   // set the delta
                        eipstate[i] = VALVEOFF +                // the valve is now
                                        JOINTENABLED ;          // considered off
                        }
                break ;

                case VALVEOFF + JOINTENABLED:
                // watch for valve-on time
                if (cycle_time[i] >= opentiming[eiducyc[i]])
                        {
                        *eiportcommand[i] &= (~eibit[i]) ;// set valve on bit
                        (*eiportdelta[i])++ ;                   // set the delta
                        eipstate[i] = VALVEONNEXT +             // the valve is now
                                        JOINTENABLED ;          // considered on
                        }
                break ;

                case VALVEON:
                // turn the valve off immediately
                *eiportcommand[i] |= eibit[i] ;         // set valve off bit
                (*eiportdelta[i])++ ;                   // set the delta
                eipstate[i] = VALVEOFF ;                // the valve is now
                                                        // considered off
                break ;

                case VALVEOFF:
                // don't do anything - the valve is off and
                // should stay off until enabled
                break ;
                }

        switch (eepstate[i])
                {
                case VALVEON + JOINTENABLED:
                // watch for valve-off time
                if (cycle_time[i] >= closetiming[eeducyc[i]])
                        {
                        *eeportcommand[i] |= eebit[i] ;         // set valve off bit
                        //*eeportcommand[i] &= (~eebit[i]) ;// set valve on bit (3-way)
                        (*eeportdelta[i])++ ;                   // set the delta
                        eepstate[i] = VALVEOFF +                // the valve is now
                                        JOINTENABLED ;          // considered off
                        }
                break ;

                case VALVEOFF + JOINTENABLED:
                // watch for valve-on time
                if (cycle_time[i] >= opentiming[eeducyc[i]])
                        {
                        *eeportcommand[i] &= (~eebit[i]) ;// set valve on bit
                        //*eeportcommand[i] |= eebit[i] ;       // set valve off bit (3-way)
                        (*eeportdelta[i])++ ;                   // set the delta
                        eepstate[i] = VALVEONNEXT +             // the valve is now
```

```
                                        JOINTENABLED ;          // considered on
                }
        break ;

        case VALVEON:
        // turn the valve off immediately
        *eeportcommand[i] |= eebit[i] ;         // set valve off bit
        (*eeportdelta[i])++ ;                   // set the delta
        eepstate[i] = VALVEOFF ;                // the valve is now
                                                // considered off
        break ;

        case VALVEOFF:
        // don't do anything - the valve is off and
        // should stay off until enabled
        break ;
        }

switch (fipstate[i])
        {
        case VALVEON + JOINTENABLED:
        // watch for valve-off time
        if (cycle_time[i] >= closetiming[fiducyc[i]])
                {
                *fiportcommand[i] |= fibit[i] ;         // set valve off bit
                (*fiportdelta[i])++ ;                   // set the delta
                fipstate[i] = VALVEOFF +                // the valve is now
                                        JOINTENABLED ;          // considered off
                }
        break ;

        case VALVEOFF + JOINTENABLED:
        // watch for valve-on time
        if (cycle_time[i] >= opentiming[fiducyc[i]])
                {
                *fiportcommand[i] &= (~fibit[i]) ;// set valve on bit
                (*fiportdelta[i])++ ;                   // set the delta
                fipstate[i] = VALVEONNEXT +     // the valve is now
                                        JOINTENABLED ;          // considered on
                }
        break ;

        case VALVEON:
        // turn the valve off immediately
        *fiportcommand[i] |= fibit[i] ;         // set valve off bit
        (*fiportdelta[i])++ ;                   // set the delta
        fipstate[i] = VALVEOFF ;                // the valve is now
                                                // considered off
        break ;

        case VALVEOFF:
        // don't do anything - the valve is off and
        // should stay off until enabled
        break ;
        }
```

```
                switch (fepstate[i])
                        {
                        case VALVEON + JOINTENABLED:
                        // watch for valve-off time
                        if (cycle_time[i] >= closetiming[feducyc[i]])
                                {
                                *feportcommand[i] |= febit[i] ;         // set valve off bit
                                //*feportcommand[i] &= (~febit[i]) ;// set valve on bit (3-way)
                                (*feportdelta[i])++ ;                   // set the delta
                                fepstate[i] = VALVEOFF +                // the valve is now
                                                        JOINTENABLED ;  // considered off
                                }
                        break ;

                        case VALVEOFF + JOINTENABLED:
                        // watch for valve-on time
                        if (cycle_time[i] >= opentiming[feducyc[i]])
                                {
                                *feportcommand[i] &= (~febit[i]) ;// set valve on bit
                                //*feportcommand[i] |= febit[i] ;       // set valve off bit (3-way)
                                (*feportdelta[i])++ ;                   // set the delta
                                fepstate[i] = VALVEONNEXT +     // the valve is now
                                                        JOINTENABLED ;  // considered on
                                }
                        break ;

                        case VALVEON:
                        // turn the valve off immediately
                        *feportcommand[i] |= febit[i] ;         // set valve off bit
                        (*feportdelta[i])++ ;                   // set the delta
                        fepstate[i] = VALVEOFF ;                // the valve is now
                                                                // considered off
                        break ;

                        case VALVEOFF:
                        // don't do anything - the valve is off and
                        // should stay off until enabled
                        break ;
                        }

                }

        // Signal end of interrupt
        disable() ;
//      if (irq_chn > 8)
        outp(IC8259_2, EOI) ;
        outp(IC8259_1, EOI) ;


}

//---------------------------------------------------------------------------

void main(void)
{
        float p[2], tdes[3];  // position vector[ x,y], theta desired from inverse kinematics
```

```
        unsigned int counter_final ;
        short i, j, r, temp, timerold ;
//      long unsigned int timeout = 0 ;
        long unsigned int eventcount = 0 ;
        char junk[80] ;
        FILE * file1 ;
        FILE * file2 ;

        if (hardware_start_up() != EXIT_SUCCESS)
                {
                printf("\nProblems setting up hardware.\n") ;
                exit(EXIT_FAILURE) ;
                }

        free_length = 2416 ; // 3.15" in terms of length ticks
        potoffset[0] = 404 ;  // empirically determined constants
        potoffset[1] = 1895 ;
        potoffset[2] = 1175 ;

        for (i=0;i<3;i++)
                {
                eipstate[i] = VALVEOFF ;
                fipstate[i] = VALVEOFF ;
                eepstate[i] = VALVEOFF ;
                fepstate[i] = VALVEOFF ;
                eiducyc[i] = 0 ;
                fiducyc[i] = 0 ;
                eeducyc[i] = 0 ;
                feducyc[i] = 0 ;
                cycle_time[i] = 0 ;
                potreading[i] = 0 ;
                prevpotreading[i] = 0 ;
                gagereading[i] = 0 ;
                joint_enabled[i] = 0 ;
                thetades[i] = potoffset[i] ;
                stiffdes[i] = 0.0 ;
                }
                gagereading[3] = 0; // initialization not possible in loop
                effducyc = 0 ;
                fffducyc = 0 ;
                timer = 0;

        eiportcommand[0] = &port0command ; // inlet
        fiportcommand[0] = &port0command ;
        eiportcommand[1] = &port0command ;
        fiportcommand[1] = &port0command ;
        eiportcommand[2] = &port0command ;
        fiportcommand[2] = &port0command ;

        eeportcommand[0] = &port1command ; // exhaust
        feportcommand[0] = &port1command ;
        eeportcommand[1] = &port1command ;
        feportcommand[1] = &port1command ;
        eeportcommand[2] = &port1command ;
        feportcommand[2] = &port1command ;
```

```
        eiportdelta[0] = &port0delta ; // inlet
        fiportdelta[0] = &port0delta ;
        eiportdelta[1] = &port0delta ;
        fiportdelta[1] = &port0delta ;
        eiportdelta[2] = &port0delta ;
        fiportdelta[2] = &port0delta ;

        eeportdelta[0] = &port1delta ; // exhaust
        feportdelta[0] = &port1delta ;
        eeportdelta[1] = &port1delta ;
        feportdelta[1] = &port1delta ;
        eeportdelta[2] = &port1delta ;
        feportdelta[2] = &port1delta ;

        // Assign digital IO channels
        // inlet          channel on opto board
        fibit[0] = 0x10 ;   // 4
        eibit[0] = 0x20 ;   // 5
        fibit[1] = 0x1 ;    // 0
        eibit[1] = 0x2 ;    // 1
        fibit[2] = 0x4 ;    // 2
        eibit[2] = 0x8 ;    // 3

        // exhaust
        febit[0] = 0x10 ;   // 12
        eebit[0] = 0x20 ;   // 13
        febit[1] = 0x1 ;    // 8
        eebit[1] = 0x2 ;    // 9
        febit[2] = 0x4 ;    // 10
        eebit[2] = 0x8 ;    // 11

        // Assign A/D channel numbers
                        // pots
        adchannum[0] = 24 ; // lin
        adchannum[1] = 25 ; // hip
        adchannum[2] = 26 ; // knee

                        // force sensors
        adchannum[3] = 27 ; // hip flexor
        adchannum[4] = 28 ; // hip extensor
        adchannum[5] = 29 ; // knee flexor
        adchannum[6] = 30 ; // knee extensor

// open valve timing file
        if ((file1 = fopen(TIMING_FILE,"r")) == NULL)
                {
                printf("\nCannot open valve timing file, %s\n", TIMING_FILE) ;
                exit(1) ;
                }

        for (i=0;i<101;i++)
                {
                fgets(junk,80,file1) ;
                sscanf(junk,"%d,%d,%d\n", &temp,&closetiming[i],&opentiming[i]) ;
                }
```

```
        fclose(file1) ;

// open control gain data file
                if ((file2 = fopen(GAIN_FILE,"r")) == NULL)
                {
                printf("\nCannot open control gain file, %s\n", GAIN_FILE) ;
                exit(1) ;
                }

        for (i=0;i<3;i++)
                {
                fgets(junk,80,file2) ;
                sscanf(junk,"%f,%f,%f\n", &tgain[i],&kgain[i],&vgain[i]) ;
                }
        fclose(file2) ;

        // Install our interrupt handler and mask
        interrupt_initial() ;

        // Start the interrupts.  Once we do this, the PWM part of
        // this code is running.
        printf("begin isr\n");
// 200 Hz
//      _8454_Write_Counter( 5 , 3 , 25 ) ;
//      _8454_Write_Counter( 6 , 2 , 2 ) ;

// 151 Hz
//      _8454_Write_Counter( 5 , 3 , 33 ) ;
//      _8454_Write_Counter( 6 , 2 , 2 ) ;

// 100 Hz
        _8454_Write_Counter( 5 , 3 , 10 ) ;
        _8454_Write_Counter( 6 , 2 , 10 ) ;

// 90 Hz
//      _8454_Write_Counter( 5 , 3 , 55 ) ;
//      _8454_Write_Counter( 6 , 2 , 2 ) ;

// 80 Hz
//      _8454_Write_Counter( 5 , 3 , 25 ) ;
//      _8454_Write_Counter( 6 , 2 , 5 ) ;

// 50 Hz
//      _8454_Write_Counter( 5 , 3 , 20 ) ;
//      _8454_Write_Counter( 6 , 2 , 10 ) ;

// test Hz
//      _8454_Write_Counter( 5 , 3 , 25 ) ;
//      _8454_Write_Counter( 6 , 2 , 8 ) ;

//-------------------------------------------------------------------------
// This section is the high level controller
//-------------------------------------------------------------------------

//-----------------------kicking motion---------------------------------------
```

```
/*for (i=1;i<3;i++)
        joint_enabled[i] = 1 ;

stiffdes[1] = kgain[0] ;
stiffdes[2] = 20.0 ;
//effducyc = 45 ;
//fffducyc = 55 ;

out = fopen("out.dat","w");

fprintf(out,"mode timer pot[0] fi[0] ee[0] fe[0] ei[0] thetades[1] pot[1] gage[0] gage[1] stiffdes[1]
stiffness[1] fi[1] ee[1] fe[1] ei[1] thetades[2] pot[2] gage[2] gage[3] stiffdes[2] stiffness[2] fi[2] ee[2] fe[2]
ei[2]\n");


        //___________________pull leg back_____________________________________________________
        j=0;
        timer = 0;
        timerold = -1 ;
        while (timer <= 100)
        {
                // calculate desired x and y position
                p[0] = -6.5;
                p[1] = -11.5;

                // calculate desired angles
                invkin(p,tdes);

                //thetades[0] = 400 + 800 * ( p[1] + 13.5 ) ;
                thetades[1] = (tdes[1]-4.858)*767 + potoffset[1] ;
                thetades[2] = (tdes[2]-5.486)*767 + potoffset[2] ;

                if(timer > timerold)
                {
                timerold = timer;
                fprintf(out,"swing %d %d %d %d %d %d "
                                , timer, potreading[0], fiducyc[0], eeducyc[0], feducyc[0], eiducyc[0]);
                fprintf(out,"%d %d %d %d %f %f %d %d %d %d "
                                , thetades[1], potreading[1], gagereading[0], gagereading[1]
                                , stiffdes[1], stiffness[1], fiducyc[1], eeducyc[1], feducyc[1],
eiducyc[1]);
                fprintf(out,"%d %d %d %d %f %f %d %d %d %d\n"
                                , thetades[2], potreading[2], gagereading[2], gagereading[3]
                                , stiffdes[2], stiffness[2], fiducyc[2], eeducyc[2], feducyc[2],
eiducyc[2]);
                }

        }
        //_________________swing leg forward quickly_____________________________________
        j=0;
        timer = 0;
        timerold = -1 ;
        tgain[1] = tgain[0];
        tgain[2] = tgain[0];
        while (timer <= 10)
        {
```

```
                    // calculate desired x and y position
                    p[0] = -6.5 + timer * 0.01 * 12.8/0.1;
                    p[1] = 0.018*p[0]*p[0] - 0.1448*p[0] - 13.2;

                    // calculate desired angles
                    invkin(p,tdes);

                    //thetades[0] = 400 + 800 * ( p[1] + 13.5 ) ;
                    thetades[1] = (tdes[1]-4.858)*767 + potoffset[1] ;
                    thetades[2] = (tdes[2]-5.486)*767 + potoffset[2] ;

                    if(timer > timerold)
                    {
                    timerold = timer;
                    fprintf(out,"plant %d %d %d %d %d %d "
                                        , timer, potreading[0], fiducyc[0], eeducyc[0], feducyc[0], eiducyc[0]);
                    fprintf(out,"%d %d %d %d %f %f %d %d %d %d "
                                        , thetades[1], potreading[1], gagereading[0], gagereading[1]
                                        , stiffdes[1], stiffness[1], fiducyc[1], eeducyc[1], feducyc[1],
eiducyc[1]);
                    fprintf(out,"%d %d %d %d %f %f %d %d %d %d\n"
                                        , thetades[2], potreading[2], gagereading[2], gagereading[3]
                                        , stiffdes[2], stiffness[2], fiducyc[2], eeducyc[2], feducyc[2],
eiducyc[2]);
                    }

          }
          //_____________________________hold_____________________________________________
          j=0;
          timer = 0;
          timerold = -1 ;
          tgain[1] = 0.005;
          tgain[2] = 0.12;
          while (timer <= 100)
          {
                    // calculate desired x and y position
                    p[0] = 6.3;
                    p[1] = -13.4;

                    // calculate desired angles
                    invkin(p,tdes);

                    //thetades[0] = 400 + 800 * ( p[1] + 13.5 ) ;
                    thetades[1] = (tdes[1]-4.858)*767 + potoffset[1] ;
                    thetades[2] = (tdes[2]-5.486)*767 + potoffset[2] ;

                    if(timer > timerold)
                    {
                    timerold = timer;
                    fprintf(out,"stance %d %d %d %d %d %d "
                                        , timer, potreading[0], fiducyc[0], eeducyc[0], feducyc[0], eiducyc[0]);
                    fprintf(out,"%d %d %d %d %f %f %d %d %d %d "
                                        , thetades[1], potreading[1], gagereading[0], gagereading[1]
                                        , stiffdes[1], stiffness[1], fiducyc[1], eeducyc[1], feducyc[1],
eiducyc[1]);
                    fprintf(out,"%d %d %d %d %f %f %d %d %d %d\n"
```

```
                                        , thetades[2], potreading[2], gagereading[2], gagereading[3]
                                        , stiffdes[2], stiffness[2], fiducyc[2], eeducyc[2], feducyc[2],
eiducyc[2]);
                        stiffdes[1] -= 0.25 ;
                        }

                }

*/

//------------------------walking motion----------------------------------------


for (i=1;i<3;i++)
                joint_enabled[i] = 1 ;

stiffdes[1] = kgain[0] ;
stiffdes[2] = 20.0 ;
//effducyc = 45 ;
//fffducyc = 55 ;

out = fopen("out.dat","w");

fprintf(out,"mode timer pot[0] fi[0] ee[0] fe[0] ei[0] thetades[1] pot[1] gage[0] gage[1] stiffdes[1]
stiffness[1] fi[1] ee[1] fe[1] ei[1] thetades[2] pot[2] gage[2] gage[3] stiffdes[2] stiffness[2] fi[2] ee[2] fe[2]
ei[2]\n");

for(i=0;i<4;i++)
{
                //____________________swing phase_____________________________________________________
                j=0;
                timer = 0;
                timerold = -1 ;
                while (timer <= 150)
                {
                        // calculate desired x and y position
                        p[0] = -6.5 + timer * 0.01 * 9.5/1.5;
                        p[1] = -0.081*p[0]*p[0] - 0.2571*p[0] - 11.75;

                        // calculate desired angles
                        invkin(p,tdes);

                        //thetades[0] = 400 + 800 * ( p[1] + 13.5 ) ;
                        thetades[1] = (tdes[1]-4.858)*767 + potoffset[1] ;
                        thetades[2] = (tdes[2]-5.486)*767 + potoffset[2] ;

                        if(timer > timerold)
                        {
                        timerold = timer;
                        fprintf(out,"swing %d %d %d %d %d %d "
                                , timer, potreading[0], fiducyc[0], eeducyc[0], feducyc[0], eiducyc[0]);
                        fprintf(out,"%d %d %d %d %f %f %d %d %d %d "
                                , thetades[1], potreading[1], gagereading[0], gagereading[1]
                                , stiffdes[1], stiffness[1], fiducyc[1], eeducyc[1], feducyc[1], eiducyc[1]);
                        fprintf(out,"%d %d %d %d %f %f %d %d %d %d\n"
                                , thetades[2], potreading[2], gagereading[2], gagereading[3]
```

```
                        , stiffdes[2], stiffness[2], fiducyc[2], eeducyc[2], feducyc[2], eiducyc[2]);
                }
                /*if(j>15000)
                {
                for(r=1;r<3;r++)
                printf("swjt#%d|p %d|des %d|g f=%d e=%d st=%f|fi=%d ee=%d fe=%d ei=%d\n"
                        ,r,potreading[r],thetades[r],gagereading[r*2-2], gagereading[r*2-1], stiffness[r]
                        ,fiducyc[r], eeducyc[r], feducyc[r], eiducyc[r]);
                j=0;
                }
                j++;*/
        }
        //_____________foot plant phase of swing_____________________________________
        j=0;
        timer = 0;
        timerold = -1 ;
        while (timer <= 1)
        {
                // calculate desired x and y position
                p[0] = 3 - timer * 0.01 * 1.0/0.01;
                p[1] = 0.25*p[0] - 14.;

                // calculate desired angles
                invkin(p,tdes);

                //thetades[0] = 400 + 800 * ( p[1] + 13.5 ) ;
                thetades[1] = (tdes[1]-4.858)*767 + potoffset[1] ;
                thetades[2] = (tdes[2]-5.486)*767 + potoffset[2] ;

if(timer > timerold)
                {
                timerold = timer;
                fprintf(out,"plant %d %d %d %d %d %d "
                        , timer, potreading[0], fiducyc[0], eeducyc[0], feducyc[0], eiducyc[0]);
                fprintf(out,"%d %d %d %d %f %f %d %d %d %d "
                        , thetades[1], potreading[1], gagereading[0], gagereading[1]
                        , stiffdes[1], stiffness[1], fiducyc[1], eeducyc[1], feducyc[1], eiducyc[1]);
                fprintf(out,"%d %d %d %d %f %f %d %d %d %d\n"
                        , thetades[2], potreading[2], gagereading[2], gagereading[3]
                        , stiffdes[2], stiffness[2], fiducyc[2], eeducyc[2], feducyc[2], eiducyc[2]);
                }
                /*if(j>5000)
                {
                for(r=1;r<3;r++)
                printf("fpjt#%d|p %d|des %d|g f=%d e=%d st=%f|fi=%d ee=%d fe=%d ei=%d\n"
                        ,r,potreading[r],thetades[r],gagereading[r*2-2], gagereading[r*2-1], stiffness[r]
                        ,fiducyc[r], eeducyc[r], feducyc[r], eiducyc[r]);
                j=0;
                }
                j++;*/
        }
        //_____________________________stance or propulsion phase_____________________________
        j=0;
        timer = 0;
        timerold = -1 ;
        while (timer <= 150)
```

```
        {
                // calculate desired x and y position
                p[0] = 2.0 - (timer * 0.01) * 8.5/1.5;
                p[1] = -13.5;

                // calculate desired angles
                invkin(p,tdes);

                //thetades[0] = 400 + 800 * ( p[1] + 13.5 ) ;
                thetades[1] = (tdes[1]-4.858)*767 + potoffset[1] ;
                thetades[2] = (tdes[2]-5.486)*767 + potoffset[2] ;

                if(timer > timerold)
                {
                timerold = timer;
                fprintf(out,"stance %d %d %d %d %d %d "
                        , timer, potreading[0], fiducyc[0], eeducyc[0], feducyc[0], eiducyc[0]);
                fprintf(out,"%d %d %d %d %f %f %d %d %d %d "
                        , thetades[1], potreading[1], gagereading[0], gagereading[1]
                        , stiffdes[1], stiffness[1], fiducyc[1], eeducyc[1], feducyc[1], eiducyc[1]);
                fprintf(out,"%d %d %d %d %f %f %d %d %d %d\n"
                        , thetades[2], potreading[2], gagereading[2], gagereading[3]
                        , stiffdes[2], stiffness[2], fiducyc[2], eeducyc[2], feducyc[2], eiducyc[2]);
                }
         /*     if(j>15000)
                {
                //fprintf(out,"swing timer=%d thetades1=%d thetades2=%d xdes=%f ydes=%f\n"
                        , timer, thetades[1], thetades[2], p[0], p[1]);
                for(r=1;r<3;r++)
                printf("stjt#%d|p %d|des %d|g f=%d e=%d st=%f|fi=%d ee=%d fe=%d ei=%d\n"
                        ,r,potreading[r],thetades[r],gagereading[r*2-2], gagereading[r*2-1], stiffness[r]
                        ,fiducyc[r], eeducyc[r], feducyc[r], eiducyc[r]);
                j=0;
                }
                j++;*/
        }
}



//-------------------------control of cylinder---------------------------------
/*
effducyc = 45 ;
fffducyc = 55 ;
thetades[0] = 500 ;
joint_enabled[0] = 1 ;
for(i=0;i<50;i++)
{
while(eventcount++<1000000l);
        eventcount=0;
printf("p %d|des %d|fi=%d ee=%d fe=%d ei=%d\n"
         ,potreading[0],thetades[0],fiducyc[0], eeducyc[0], feducyc[0], eiducyc[0]);
thetades[0] += 30;
}
for(i=0;i<50;i++)
```

```
{
while(eventcount++<1000000l);
              eventcount=0;
printf("p %d|des %d|fi=%d ee=%d fe=%d ei=%d\n"
               ,potreading[0],thetades[0],fiducyc[0], eeducyc[0], feducyc[0], eiducyc[0]);
thetades[0] -= 30;
}
*/
//------------------------------------------------------------------------
/*
r=2;
joint_enabled[r] = 1 ;
thetades[r] = 900;
stiffdes[r] = 20.0;

for(j=0;j<2;j++)
{
              for (i=0;i<40;i++)
              {
                            while(eventcount++<1000000l);
                                          eventcount = 0;
                            printf("pot %d %d %d|gage f=%d e=%d stiff=%f|ducyc fi=%d ee=%d fe=%d ei=%d\n"
                                          ,potreading[0],potreading[1],potreading[2],gagereading[2], gagereading[3],
stiffness[r]
                                          ,fiducyc[r], eeducyc[r], feducyc[r], eiducyc[r]);
                            thetades[r] += 10 ;
              }
              for (i=0;i<40;i++)
              {
                            while(eventcount++<1000000l);
                                          eventcount = 0;
                            printf("pot %d %d %d|gage f=%d e=%d stiff=%f|ducyc fi=%d ee=%d fe=%d ei=%d\n"
                                          ,potreading[0],potreading[1],potreading[2],gagereading[2], gagereading[3],
stiffness[r]
                                          ,fiducyc[r], eeducyc[r], feducyc[r], eiducyc[r]);
                            thetades[r] -= 10 ;
              }
}
joint_enabled[r] = 0 ;
*/
//---------------feed forward cycling of joints-----------------------------------
/*  for (i=0;i<3;i++)
                            joint_enabled[i] = 1 ;

 for(i=0;i<3;i++)
 {
                            // feed forward cycling of joints
              printf("Joint #%d\n",i);
              ffiducyc[i] = 50;
                            while(eventcount++<7000000l);
                            eventcount=0;
                            ffiducyc[i] = 0;

              ffeducyc[i] = 50;
                            while(eventcount++<7000000l);
                            eventcount=0;
```

```
                ffeducyc[i] = 0;

        feiducyc[i] = 50;
                while(eventcount++<7000000l);
                eventcount=0;
                feiducyc[i] = 0;

        feeducyc[i] = 50;
                while(eventcount++<7000000l);
                eventcount=0;
                feeducyc[i] = 0;
        }
*/
//-------------------------------------------------------------------------
                // Sensor output
/*for(j=0;j<20;j++)
{
        for (i=0;i<3;i++)
        {
                printf("pot #%d %d\n",i, potreading[i]);
        }
        for (i=0;i<4;i++)
        {
                printf("gage #%d %d\n",i, gagereading[i]);
        }
}*/
//-------------------------------------------------------------------------
        // Shut down the interrupts
        _8454_Stop_Counter(6,&counter_final) ;

        // Restore the old handler and masks
        interrupt_restore() ;

        // Close all valves
        shut_down_valves() ;

}
```

B.2 Def.h

```
#ifndef PWM_DEF_FILE
        #define PWM_DEF_FILE

        #ifdef __cplusplus
          #define __CPPARGS ...
        #else
          #define __CPPARGS
        #endif

        #include <stdio.h>
        #include <stdlib.h>
        #include <conio.h>
        #include <math.h>
//      #include <string.h>
        #include <8454.h>
        #include <dos.h>
        #include <cb.h>

        #define TRUE                    1
        #define FALSE                   0
//      #define OTHER                   (-1)

        #define TIMING_FILE             "predict.dat"
        #define GAIN_FILE               "gain.dat"

        #define JOINTDISABLED           0
        #define JOINTENABLED            1
        #define VALVEOFF                2
        #define VALVEON                 4
        #define VALVEONNEXT             8

        #define PI 3.14159265
        #define L1 8.875  // length of link 1
        #define L2 6.25   // length of link 2

        // NOTE: DO NOT use a base address of 0x300.  This is
        // used by the digital I/O board, and is difficult
        // to change.  You need IOTEST.EXE from Computer Boards,
        // which I mistakenly deleted :(

        // digital I/O board stuff
        #define BOARDNUM                0

        // A/D board stuff
        #define CB_BASEADDR             (0x308)

        // interrupt board stuff
        #define INT_CARD_BASEADDR (0x240)
        #define PWM_IRQ                 IRQ10
        #define IC8259_1                0x20
        #define  IC8259_2               0xA0
        #define  EOI                    0x20
/*
        #define NUM_COMMAND_PRGS            12
        #define COMMAND_PRGS                { \
```

```
                                        {sendpot,               0,      2}, \
                                        {sendallpots,   3,      3}, \
                                        {sendgage,              5,      8}, \
                                        {sendallgages,  9,      9}, \
                                        {sendallsnrs,   10,     10}, \
                                        {enablejoint,   20,     22}, \
                                        {enableall,             23,     23}, \
                                        {disableall,    24,     24}, \
                                        {recvffdc,              25,     25}, \
                                        {recvthetades,  30,     32}, \
                                        {recvstiffdes,  41,     42}, \
                                        {recvalldes,    50, 50} \
                                        }
//                                      {stopvalves,    254,    254}, \
////                                     {stopall, 255,  255} \
//                                      }
        #define ISD     255 // Immediate Shutdown command

        // Status messages
        #define FINISHED        'F'
        #define BAD_COMMAND     'B'
        #define READY_FOR_16 'R'
//      #define READY_FOR_MORE      'M'

        typedef struct do_something
                {
                void (*prg)(short& i, short& offset) ;
                short offset ;
                short stop ;
                }
        do_something ;
*/
#endif //PWM_DEF_FILE
```

B.3 Hardware.cpp

```
#include "def.h"

//---------------------------------------------------------------

void shut_down_valves(void)
{
 long unsigned int eventcount = 0 ;

 // exhaust actuators
 cbDOut(BOARDNUM,FIRSTPORTA,255) ;
 cbDOut(BOARDNUM,FIRSTPORTB,0) ;
 while (eventcount++<5000000l);

 // close all valves
 cbDOut(BOARDNUM,FIRSTPORTA,255) ;
 cbDOut(BOARDNUM,FIRSTPORTB,255) ;
 cbDOut(BOARDNUM,FIRSTPORTCL,15) ;
 cbDOut(BOARDNUM,FIRSTPORTCH,15) ;
 cbDOut(BOARDNUM,SECONDPORTA,255) ;
 cbDOut(BOARDNUM,SECONDPORTB,255) ;
 cbDOut(BOARDNUM,SECONDPORTCL,15) ;
 cbDOut(BOARDNUM,SECONDPORTCH,15) ;
}

//---------------------------------------------------------------

int hardware_start_up()
//int hardware_start_up(short divisor)
{
 float RevLevel = (float) CURRENTREVNUM ;
 int direction  = DIGITALOUT ;
 int i = 0 ;

// setup DIO card
 cbDeclareRevision(&RevLevel) ;
 cbErrHandling(PRINTALL,STOPALL) ;

// configure the various DIO card ports
 if (cbDConfigPort(BOARDNUM,FIRSTPORTA,direction))
   {
   printf("\nDIO Board setup error on FIRSTPORT A.\n") ;
   return(EXIT_FAILURE) ;
   }
 if (cbDConfigPort(BOARDNUM,FIRSTPORTB,direction))
   {
   printf("\nDIO Board setup error on FIRSTPORT B.\n") ;
   return(EXIT_FAILURE) ;
   }
 if (cbDConfigPort(BOARDNUM,FIRSTPORTCL,direction))
   {
   printf("\nDIO Board setup error on FIRSTPORT CL.\n") ;
   return(EXIT_FAILURE) ;
   }
```

```
 if (cbDConfigPort(BOARDNUM,FIRSTPORTCH,direction))
   {
   printf("\nDIO Board setup error on FIRSTPORT CH.\n") ;
   return(EXIT_FAILURE) ;
   }
 if (cbDConfigPort(BOARDNUM,SECONDPORTA,direction))
   {
   printf("\nDIO Board setup error on SECONDPORT A.\n") ;
   return(EXIT_FAILURE) ;
   }
 if (cbDConfigPort(BOARDNUM,SECONDPORTB,direction))
   {
   printf("\nDIO Board setup error on SECONDPORT B.\n") ;
   return(EXIT_FAILURE) ;
   }
 if (cbDConfigPort(BOARDNUM,SECONDPORTCL,direction))
   {
   printf("\nDIO Board setup error on SECONDPORT CL.\n") ;
   return(EXIT_FAILURE) ;
   }
 if (cbDConfigPort(BOARDNUM,SECONDPORTCH,direction))
   {
   printf("\nDIO Board setup error on SECONDPORT CH.\n") ;
   return(EXIT_FAILURE) ;
   }

 shut_down_valves() ;

 // set up interrupt card
 _8454_Initial(INT_CARD_BASEADDR);

 // set up A/D card
 outp(CB_BASEADDR + 3, 3) ;     // set voltage level to 0-5V
 outp(CB_BASEADDR + 2, 0) ;     // set first channel to be read
 while (i < 10000) i++ ;     // wait a little bit

 return(EXIT_SUCCESS) ;
}

//-------------------------------------------------------------

int hardware_shut_down(void)
{
 shut_down_valves() ;

 return(EXIT_SUCCESS) ;
}

//-------------------------------------------------------------
```

B.4 Predict.dat

Junk Counter	Close Timing	Open timing
0	0	99
1	1	99
2	2	99
3	3	99
4	4	99
5	5	99
6	6	99
7	7	99
8	8	99
9	9	99
10	10	99
11	11	99
12	12	99
13	13	99
14	14	99
15	15	99
16	16	99
17	17	99
18	18	99
19	19	99
20	20	99
21	21	99
22	22	99
23	23	99
24	24	99
25	25	99
26	26	99
27	27	99
28	28	99
29	29	99
30	30	99
31	31	99
32	32	99
33	33	99
34	34	99
35	35	99
36	36	99
37	37	99
38	38	99
39	39	99
40	40	99
41	41	99
42	42	99
43	43	99
44	44	99
45	45	99
46	46	99
47	47	99
48	48	99
49	49	99
50	50	99
51	51	99
52	52	99
53	53	99
54	54	99
55	55	99
56	56	99
57	57	99
58	58	99
59	59	99
60	60	99
61	61	99
62	62	99
63	63	99
64	64	99
65	65	99
66	66	99
67	67	99
68	68	99
69	69	99
70	70	99
71	71	99
72	72	99
73	73	99
74	74	99
75	75	99
76	76	99
77	77	99
78	78	99
79	79	99
80	80	99
81	81	99
82	82	99
83	83	99
84	84	99
85	85	99
86	86	99
87	87	99
88	88	99
89	89	99
90	90	99
91	91	99
92	92	99
93	93	99
94	94	99
95	95	99
96	96	99
97	97	99
98	98	99
99	99	99
100	100	99

B.5 Gain.dat

During normal walking motion, the gain was file was as follows.

0.9, 35.0
0.035,6.5
0.12,5.0

During walking motion with damping control, the gain file was as follows.
0.9, 35.0, 0.0
0.035, 6.5, 0.02
0.12, 5.0, 0.0

Appendix C: Robot Hardware

C.1 Strain Gage Amplifier

The strain gage amplifier board was a six channel circuit. Only 4 channels were used, but it could be modified to handle 6 channels.

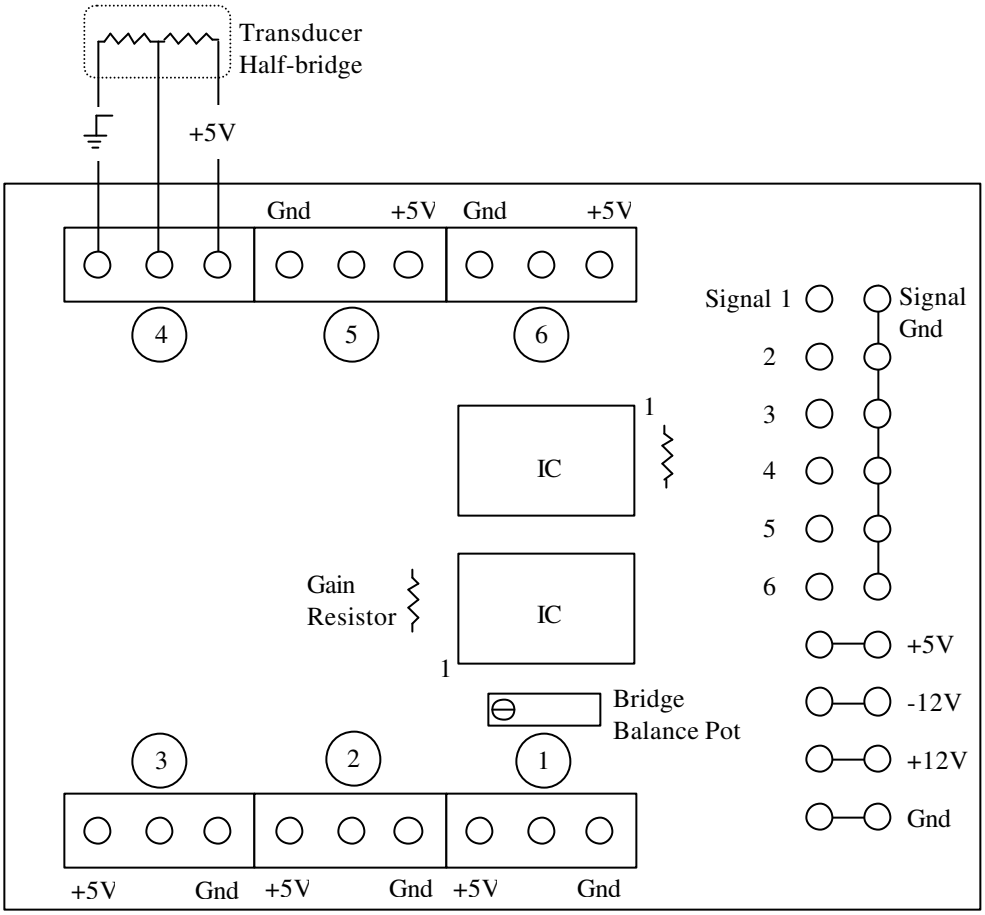


Figure C-1

C.2 Wiring

Table C-1: Wiring Connection Table					
Pin	Description	Color into Terminal Block	Color in Ribbon Cable	Plumbing Valve	Plumbing Description
1	Linear Pot Signal	Red	Brown	1	Up
2	Linear Signal gnd	Purple	tan	2	Down
3	Hip angle signal	Blue	Red	3	Hip flexor
4	Hip Signal gnd	Orange	tan	4	Hip extensor
5	Knee angle signal	Purple	Orange	5	Knee flexor
6	Knee Signal gnd	Grey	tan	6	Knee Extensor
7	Hip flexor force transducer		Yellow	**(input matches output)**	
8	Signal ground		tan		
9	Hip extensor force transducer		Green		
10	Signal ground		tan		
11	Knee flexor force transducer		Blue		
12	Signal ground		tan		
13	Knee extensor force transducer		Purple		
14	Signal ground		tan		
15	Hip flexor pressure transducer	NA	Grey		
16	Signal ground	NA	tan		
17	Hip extensor pressure transducer	NA	White		
18	Signal ground	NA	tan		
19	Knee flexor pressure transducer	NA	Black		
20	Signal ground	NA	tan		
21	Knee extensor pressure transducer	NA	Brown		
22	Signal ground	NA	tan		
23	+12V		Red		
24	-12V		tan		
25	+5 V	White(lin) Yellow(hip) White(knee)	Orange		
26	Power gnd (for +5)	Black(lin) Green(hip) Black(knee)	tan		
				Channel on Opto Board	
27	Input valve #1 control	Brown	Yellow	0	
28	Input valve common	Black	tan		
29	Input valve #2 control	Red	Green	1	
30	Input valve common	Black	tan		
31	Input valve #3 control	Orange	Blue	2	
32	Input valve common	Black	tan		
33	Input valve #4 control	Yellow	Purple	3	
34	Input valve common	jmpr	tan		
35	Input valve #5 control	Green	Grey	4	
36	Input valve common	jmpr	tan		
37	Input valve #6 control	Blue	White	5	
38	Input valve common	jmpr	tan		
39	output valve #1 control	Brown	Black	8	
40	Output valve common	Black	tan		
41	output valve #2 control	Red	Brown	9	
42	Output valve common	Black	tan		
43	output valve #3 control	Orange	Red	10	
44	Output valve common	Black	tan		
45	output valve #4 control	Yellow	Orange	11	
46	Output valve common	Black	tan		
47	output valve #5 control	Green	Yellow	12	
48	Output valve common	jmpr	tan		
49	output valve #6 control	Blue	Green	13	
50	Output valve common	jmpr	tan		

C.3 Mechanical Drawings

BIBLIOGRAPHY

Bachmann, R., A Cockroach-Like Hexapod Robot for Running and Climbing, M.S. Thesis, Case Western Reserve University, Cleveland OH, 2000.

Caldwell, D.G., G.A. Medrano-Cerda, and M.J. Goodwin, "Control of Pneumatic Muscle Actuators," "*IEEE Control Systems Journal*," Vol. 15, No. 1, pp. 40-48, Feb 1995.

Chou, C.P., B. Hannaford, "Measurement and Modeling of McKibben Pneumatic Artificial Muscles," "*IEEE Transactions on Robotics and Automation*," Vol. 12, No. 1, pp. 90-102, Feb 1996.

Klute, G.K., J.M. Czerniecki, and B. Hannaford, "McKibben Artificial Muscles: Pneumatic Actuators with Biomechanical Intelligence," Proceedings of the IEEE/ASME 1999 International Conference on Advanced Intelligent Mechatronics (AIM '99), Atlanta, GA, September 19-22, 1999.

Klute, G.K., B. Hannaford, "Modeling Pneumatic McKibben Artificial Muscle Actuators: Approaches and Experimental Results," Submitted to the ASME Journal of Dynamic Systems, Measurements, and Control, November 1998, revised March 1999.

Klute, G. K., B. Hannaford, "Fatigue Characteristics of McKibben Artificial Muscle Actuators." Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems. Victoria, B.C., Canada, October 13-17, 1998, pp.1776-1782.

Marbot, P.H., Ia Response of a Mechanical Spindle Replica, Ph.D. Dissertation, University of Washington, Department of Electrical Engineering, March 1995.

McCloy, D., H.R. Martin., *Control of Fluid Power: Analysis and Design 2nd (revised) Edition*, Ellis Horwood Limited, New York, 1980.

Nelson G.M., Simulation and Control of a Cockroach-Like Robot, Ph.D. Thesis, Case Western Reserve University, Cleveland OH, (in press).

Nickel, V.L., J. Perry, and A.L. Garrett, "Development of useful function in the severely paralyzed hand," "*Journal of Bone and Joint Surgery*," Vol. 45A, No. 5, pp. 933-952, 1963.

Rao, S.S., *Mechanical Vibrations 3rd Edition*, Addison-Wesley, USA, 1995.

Shimoyama, I. http://www.nando.net/newsroom/ntn/health/011097/health1_10531.html

Shadow Robot Company (2000). http://www.shadow.org.uk/products/airmuscles.shtml

Shadow Robot Company (1999). http://www.shadow.org.uk/muscles/performance

Watson, J. T., R. E. Ritzmann "Leg kinematics and muscle activity during treadmill running in the cockroach, *Blaberus discoidalis:* I. Slow running," "*J. Comp. Physiol. A*," Vol. 182: pp.11-22, 1998.

Ye, N., S. Scavarda, M. Betemps, A. Jutard, "Models of Pneumatic PWM Solenoid Valve for Engineering Applications," "*Journal of Dynamic Systems Measurement and Control, Transactions of the ASME*," Vol. 114, pp. 680-688, December 1992.