2012-06-13

Development of a Model and Simulation Framework for a Modular Robotic Leg

Katiegrace Youngsma
Worcester Polytechnic Institute

Follow this and additional works at: https://digitalcommons.wpi.edu/etd-theses

Repository Citation
https://digitalcommons.wpi.edu/etd-theses/890

This thesis is brought to you for free and open access by Digital WPI. It has been accepted for inclusion in Masters Theses (All Theses, All Years) by an authorized administrator of Digital WPI. For more information, please contact wpi-etd@wpi.edu.
DEVELOPMENT OF A MODEL AND SIMULATION FRAMEWORK FOR A MODULAR ROBOTIC LEG

Authored by
Katiegrace Youngsma
A Thesis
Submitted to the Faculty of
WORCESTER POLYTECHNIC INSTITUTE
in partial fulfillment of the requirements for the
Degree of Master of Science
in
Robotics Engineering

June 11, 2012

Approved by:

________________________________________
Dr. Taskin Padir, Assistant Professor, Robotics Engineering Program

________________________________________
Dr. William Michalson, Professor, Robotics Engineering Program

________________________________________
Dr. Michael Gennert, Professor, Robotics Engineering Program
Abstract

As research in the field of mobile robotics continues to advance, legged robots in different forms and shapes find a variety of applications on rough terrain where wheeled robots fail to operate in practice. For this reason, a modular legged robot platform is being developed at WPI. This research focuses on developing a mathematical model and then building a simulation to verify the model for a single leg for this platform. The robot platform is modular in the sense that leg modules can be removed and added to predetermined ports on the robot chassis. The modularity of a legged robot is a significant advancement in mobile robotics technology as it enables a single robot to take on different body configurations depending on circumstances and environment to achieve its goals. It also poses a challenge in terms of overall design as it requires autonomous operation of the leg. The goal for this research is to in part fulfill the need for a mathematical model for an autonomous leg. This research investigates the development of a kinematic and dynamic model for the leg, a step trajectory for walking, a simulation of the system to verify the dynamic model, and various functions and scripts to identify shortcomings within the model. This research uses Mathworks Matlab and Wolfram Mathematica to develop the mathematical model, and Matlab Simulink SimMechanics and Matlab functions to build a simulation. Both the mathematical model and simulation follow the classic design of other legged robots, utilizing Lagrangian dynamics, the Jacobian, and simulation tools. The result is a project that is unique in that it drives a robot leg almost independently with very limited communication to a central controller.
Contents

1 Introduction 1
  1.1 Purpose ................................................. 1
  1.2 Report Overview ....................................... 2
  1.3 Background ............................................. 3
  1.4 Accomplishments ....................................... 10

2 Kinematic and Dynamic Modeling of a Robot Leg 12
  2.1 Mechanical Design ....................................... 12
  2.2 Forward Kinematics ..................................... 18
  2.3 SimMechanics Model ..................................... 20
  2.4 Inverse Kinematics ..................................... 25
  2.5 The Full Robot Body ..................................... 26
  2.6 Chapter 2 Summary ....................................... 27

3 Robot Leg Dynamics 28
  3.1 Calculating the Jacobian Matrix ....................... 28
  3.2 Dynamics ................................................. 30
    3.2.1 Components of the Manipulator Lagrangian .......... 31
    3.2.2 Methods of Computing the Euler-Lagrange for a Man-
          nipulator ........................................... 33
  3.3 Using User Defined Blocks ............................. 37
  3.4 Chapter 3 Summary ....................................... 37

4 Simulating the Leg 38
  4.1 Leg Trajectory .......................................... 40
    4.1.1 Workspace ............................................ 40
    4.1.2 Set Points ........................................... 41
    4.1.3 Generating Smooth Trajectories ..................... 43
    4.1.4 Trajectory Generation Methods in the Simulation ... 45
    4.1.5 Stability Polygon .................................... 47
    4.1.6 Checking for Singularities and Manipulability .. 48
Chapter 1

Introduction

1.1 Purpose

Every robot, every being, has a purpose for its existence. The motivation for a reconfigurable robot, is that for every purpose, there will be a configuration which the robot can take on to accomplish that purpose. The proposed solution is a robot where the sensors, the actuators, and the processing dedicated to each peripheral are modular. Meaning each part is simply an attachment that can be used or removed on the robot to suit its mission and environment. This project is a part of the process of developing a fully functioning reconfigurable multi-legged robot. This is a significant development because, in previous developments research investigated legged walking for robots which were fully defined. Meaning basic parameters such as the number of legs, size, overall mass, and which sensors were available were known. However this robot is, by its nature, physically undefined until it is placed together in its desired configuration. This robot could potentially take on configurations that would enable it to accomplish a wide variety of tasks. While this research focuses on designing a leg to enable the robot to walk, there is no reason why it could not have wheel modules instead, giving it the ability to navigate over flat surfaces very quickly. One could even place wheels on the end of legs, making it a wheeled robot of variable height and able to lift its wheels over obstacles. The reconfigurable robot being developed at WPI has 12 multi-use ports for attachments which have yet to be designed, but has a wide range of possible applications.

The reconfigurable robot has been designed such that the central body for the robot would simply act as a physical frame on which to bolt the components, a power bus for the modules (since the power supplies themselves should also be modular, allowing the user to balance the weight to stored
energy ratio for a given mission), and an information bus for data that needs to be shared among the modules. The ports could be used for a variety of sensors, as simple or as advanced as the robot needs to accomplish its purpose. For basic navigation something as inexpensive or lightweight as a sonar range finder could be used, however for outdoor navigation through rubble something more advanced like a video array may be required. Either one can be mounted provided it can attach to the universal ports on the robot body. This particular robot design seeks to move the central processing out to its modular attachments, so that the body is not weighed down with a complicated controller. This is ideal since different type and grades of sensors require varying amounts of processing. On this robot, the sensor and dedicated sensor processing would be handled on the peripheral attachment, such that it only sends useful information through the robot body. The goal for this particular research is to model and simulate a single leg for reconfigurable robot. Specifically to design a modular leg with its own processing and control that needs minimal communication with the robot body to achieve stable walking.

1.2 Report Overview

This paper has been introduced with an explanation of the purpose and need for a primarily autonomous leg for a reconfigurable robot. The report then discusses the history of advances in legged mechanisms, legged robots, and reconfigurable robots. This includes a short literature review of papers presented by those whose study has contributed to the work done in this project. We then discuss the overall design and construction of the robot leg. Beginning with a proposed mechanical design, and a discussion of how to physically model the leg using Matlab’s Simulink software with the SimMechanics package. This section also guides the reader through the forward and inverse kinematics for this robot leg, using the mathematical techniques outlined in the textbook, “Robot Modeling and Control” [1]. The third section continues the mathematical model and demonstrates calculations for the Jacobian and Euler-Lagrange dynamics for the single robot leg. Section 4 discusses the leg as simulated in the Simulink software. It begins with a lengthy discussion on how to generate the trajectory for a single robot leg and compares two different methods. It then discusses the details of the construction and flow of the Simulink simulation: how the mathematical model outlined in Section 3 is incorporated into the simulation, how the trajectory feeds into the system, and how these components interface with the SimMechanics physical model. It then discusses how the simulation is used to
analyze verify the mathematical model. Section 5 discusses the results of the data collected from the simulation and identifies ways to define errors that came up in the results from the simulation. This section later describes future work that could be conducted with this project. The discussion on future work goes as far to propose ideas for the full operating reconfigurable robot.

1.3 Background

Many technological advancements utilize wheels to achieve faster and more energy efficient mobility, however, soon after their development man must have realized that only creatures with legs could properly navigate over dramatically uneven terrain, the way an ant navigates thick grass or a mule navigates a canyon. Only legged beings could step over objects and maintain stability on surfaces with varying heights. These scenarios demonstrate some motivation for legged robots as opposed to those on wheels or tracks.

Ideas for mechanical walking creatures may go back as far as 480 BC, and many images and ideas for various artificial animals can be found throughout the centuries [2]. The first real documented attempt for a walking machine are concepts for basic linkage driven machines. In 1893, Rigg took out a patent for a mechanical horse, proving that although legged mobility was available through domestic animals, a machine that could accomplish the same task was desired [3]. The first documented designs for a walking robots arrive in the 1970’s, with WABOT 1, a statically stable robot developed by I. Kato, and an active exoskeleton developed independently by M. Vukobratovic. Vukobratovic’s work was not statically stable, but rather developed the concept of the Zero Moment Point (ZMP) for dynamic walking. This work inspired the need for research in dynamics of mobile legged robots.[4].

While investigating ideas to support this project, a wide variety of papers and concepts were reviewed [4] [5] [6] [7] [8] [9] [10] [11] [12] [13]. As mentioned, Vukobratovic developed the idea of the ZMP control method for dynamic walking. The ZMP method involves taking readings from force sensors placed on the legged robot’s feet. By measuring the reaction forces on the “ground”, the controller calculates the zero-moment point.

It is named thus since it is the single point around which no force is generating a moment on the robot. That is, the sum of all the moments on the robot body (gravity, reaction forces, etc.) sum to zero at this point (See Figure 1.1). To maintain stability, this point must lie within the robot’s support polygon for dynamically stable movement. Using Newtonian dynamics, one can predict the location of the ZMP based on expected movement. In Vuko-
bratovic’s paper he suggests the use of a moving counterweight to maintain a ZMP within the stability polygon as the robot walks.[4].

This method is one of the more popular and a very useful method, however it is not used in this project due to the autonomous nature of the robotic leg as the full configuration of the entire robot must be known to compute the ZMP. While a full control scheme for the reconfigurable robot in this project has not been developed, it is planned that the robot walk using minimal communication between the modular legs. The ZMP method however, requires almost constant feedback from foot force sensors and plans its motion holistically (considering all the forces on the robot combined) while this project seeks to design a robot in which the motions are planned almost independently by each foot.

Another strategy for dynamic motion control is by developing a cyclic motion that maintains stability through support patterns. Some classic examples are Raibert’s “hopping” robots developed at the MIT LegLab shown in Figure 1.2 [4]. Raibert’s research focuses on running robots, in which the strategy for stability is broken into three parts: cyclic hopping, forward motion, and posture.[3]. Cyclic hopping occurs when the leg(s) propel the body upward to a determined height, and while in the air the robot follows a predictable ballistic path. Forward motion is achieved for the control system.
by moving the foot or feet to a desired position while the robot is in the air, such that when the robot lands it will be propelled forward in the next bounce. Posture is the angles at which the foot or feet need to be located to maintain stability in three dimensional space over the given terrain. The method is not used for this project because it is specifically designed for running robots, which at this time the goal for this robot is stable walking.

![Figure 1.2: The MIT LegLab’s Dynamically Stable “hopping” robots](image)

This project instead uses a more basic strategy for stability, and that is by maintaining its center of mass well within its stability polygon such that it is statically stable, and keeping dynamic forces from the legs to a minimum to avoid upsetting the balance. Should further developments to this robot arise that require the legs to move quickly and thus exert more dynamic forces on the body, a more advanced control system like those described above would need to be developed.

One control development that is very useful for this project is that of adaptive locomotion over uneven terrain as described by McGhee and Iswandhi [10]. McGhee and Iswandhi describe how adaptive locomotion can be realized by developing a sequence of support states as the robot steps over uneven terrain. It describes how the robot may optimally place its steps to avoid interference with other legs, move within its reachable area, and maintain stability of the robot body.[10]. It stresses the importance of maintaining a support pattern for the robot’s center of mass throughout all stages of the walking pattern. This study was considered for this project while developing...
the leg trajectories. Later we will show how a Matlab script was developed to demonstrate that the robot center of mass (CoM) remains within its support polygon throughout the leg step trajectory. When McGhee’s robot walks in an adaptive manner (meaning the leg trajectory changes) an algorithm determines what “cells” or areas of space are acceptable for leg placement to maintain stability.

Waldron and McGhee later develop an adaptive walking hexapod robot. Waldron is known for exploring motion and force management for each robotic leg as it interacts with the ground on a robot with quasi-static stability [14]. Waldron’s methods were not explored in depth for this project since they deal primarily with balancing overall robot forces to avoid slip. Since we use the Lagrangian to determine the necessary joint torques to achieve desired forces for the robot feet, further equations for balancing the forces should not be necessary. Instead, the torques provided by joint motors are specifically specified to maintain the desired leg position throughout the stepping trajectory.

In 1992, Boissonnat, Devillers, Preperata, and Donati outlined proper walking patterns for their own spider robot. This group set out their own algorithms for robot foot placement while insuring the robot maintains a stable configuration [6]. This method is almost identical to that used by McGhee to determine areas that are acceptable for foot placement, only that Boissonnat uses sets and unions to determine or eliminate areas in which the robot could step. Once again, for this project we use a single cyclic trajectory which repeats for each step. However it would be of great benefit to be able to adapt the trajectory using the methods of either McGhee or Boissonnat [10][6]. Kimura, Maufroy, and Takase also expand upon the ideas of McGhee and Iswandi to develop a robot with adaptive walking using a very different strategy that is more appealing for rough terrain. Rather than adjusting where the robot’s foot can be placed, Maufroy focuses on when the step cycle for the robotic steps can be broken. For example, if there were a rock on the ground, McGhee or Boissonnat’s method would make an effort to not step in that area, whereas Kimura’s former robot would merely rest its foot atop the rock and end the step cycle there. [9][10]. What makes Maufroy’s method possible is the addition of some kind of sensor on the robot foot.

Kimura also developed a robot for dynamic walking with Fukuoka and Cohen called Tekken 2. Tekken 2 uses similar gait and walking concepts to that developed with Maufroy, utilizing a step cycle. However the leg mechanical design utilizes springs such that its gait is more like the hopping motion of Raibert’s robots than those of McGhee’s hexapod [8]. Another major difference of note between the hexapods of McGhee and Waldron and the quadrupeds of Kimura and Maufroy is the overall construction of the leg.
Figure 1.3: This is a photo of the mammal-like quadruped with springed legs developed by Kimura’s team. [8]

Figure 1.4: This is a photo of an insect-like hexapod developed by McGhee’s team. [10]

All have three degrees of freedom, however, those of Kimura and Maufroy are like those of a mammal as seen in Figure 1.3, having a hip that allows the body to roll and a knee and ankle that produce the step’s up and down motion. Those of McGhee and Waldron are more like that of an insect, that pivot along a vertical axis and have two more bends outward from the body as in Figure 1.4.

For our walking robot design, an overall structure similar to that of McGhee’s robot has been chosen to provide more stability while walking. Both designs place the robots feet further out from the center of the robots body while walking. Having the feet placed further apart creates a larger stability polygon, which provides a larger area to place the center of mass and reduces chances of tipping. Our design also favors a lower center of mass to further reduce the risk of tipping. Stability during running and the use of springlike legs are not covered in this research. This research also seeks to be capable of using an adaptive approach to a walking gait rather than a deterministic path. The adaptive method means that the robot could shorten, lengthen, or adjust the height of each step (within limits) depending on terrain, whereas a deterministic path would either repeat the same step (as on level ground) or dead reckon its steps (set a different step height in advance to step onto a known object). Since the legs are designed to operate autonomously from the body, adaptivity with in the leg is desired. Kimura’s step cycle which ends when the foot encounters an object is one proven method for an adaptive gait. Another can be found in a paper by Espenschied et. al. which presents an insect like robot with an adaptive (in his case reflexive) gait for walking on uneven terrain [15]. Like Kimura, rather than focusing on planning the locations which the foot should take its next step, the robot actually uses its foot to sense good foot placement.
locations. Inspired by insects biology, the robot “feels around”, or rather reads the reaction forces of placing its foot in a given space where it may want to step, until a satisfactory resting place is found and the robot can transfer its weight onto the leg.

While researching robot leg control methods we find a paper by Barreto, Trigo, Menezes, Diaz, and Almeida demonstrates the full forward and inverse kinematics for a hexapod robot. The kinematics provide a mapping and coordinate change from the location and orientation of the center of the robot to the location and orientation at the end of each leg. The inverse provides the same from the end of the leg to the center of the robot body. Rather than focusing on robot gait like earlier researchers, Barreto’s team focuses on joint control, and uses free-body diagram methods to determine the robot dynamics part by part. This enables them to discuss the torques applied to leg joints needed to produce the robots motion in a way that previous research did not.[5] Instead, other researchers simply use some kind of closed loop PD or PID control to produce desired joint torques in the legs to produce a desired robot posture.

One of the most famous developments in legged robots, and one that has received much recognition is BigDog developed by Boston Dynamics[12]. Building on the ideas of McGhee and Waldron, the company successfully developed a robot that navigates over rough terrain with great agility in a way not seen before in former developments. It utilizes a combination of sensor readings, gait control, dynamic balance, and a control system that enables it to recover from disruptions in its movement. [12]. While their work is well above and beyond the scope of our research for a single leg, concepts from their research were considered. Big Dog’s computational design, which begins with trajectory planning and follows through to generating desired joint angles, velocities and accelerations, is similar to the one used for this single leg model. In contrast, BigDog has many more modes and states than our research has used, and requires over fifty sensors to control the robot, an option that our research does not consider [12].

In recent years, the study of legged robots has branched so widely that one must focus on those which are relevant to this study. In terms of reconfigurability, it is a rather new development in the field of robotics. The most well known type is like “Super-Bot”, which consists of multiple identical autonomous robotic modules which can assemble themselves in different configurations to achieve different tasks[13]. Each module can operate independently and work together to form a single robot.[13]. Our reconfigurable robot differs from Super-Bot in that its components cannot operate independently; rather, different components of this robot serve different purposes (such as the leg for walking), and while the components have nearly indepen-
dent control, they will not function unless manually attached to the central robot body. In this sense, our reconfigurable robot design is more like that of MiniQuad shown in Figure 1.5[7], which has body section modules so the user can change the number of legs on the robot. We will also briefly present a mechanical design for a multi-legged robot, which uses a compacted version of the architecture used by McGhee. MiniQuad’s developers also outline the control scheme for the robot. Using a tiered control approach, they use a full computer as a master controller, a “body level controller” to determine gait and foot position, and individual unit controllers for each actuator and sensor.

The robot design presented in here is reconfigurable in the sense that it has a body with multiple ports, each of which is universal so that a manipulator or sensor can be attached. With this advantage, one can add capabilities to the robot for specific missions, or remove extraneous components to reduce weight for others. In this paper, we will take the term reconfigurable to mean that components of the robot can be removed and added without inhibiting the operability of the robot as a whole. Specifically, we focus on a leg as a module that could be added or removed from a robot while maintaining overall operability, and so long that there are more than four legs, the robot is able to walk. The reconfigurable robot model presented in this paper and shown in Figure 1.6 will build off the reconfigurable robot platform previously developed at WPI which sought to develop a robot body with 12 ports for attachable legs each with an independent controller on each. The platform was presented at the 2010 ICRA Workshop by Professor Taskin Padir and students as their Major Qualifying Project (MQP). [16]

Figure 1.5: This is a photo of MiniQuad, a modular robot with insect-like legs designed by Chen [7]

Figure 1.6: This is a rendering of the mechanical design previously developed for this research [16]
In order to achieve this goal, we develop the necessary kinematics and dynamics as outlined by Spong in his book, *Robot Modeling and Control* [1]. The design is then verified using a software program with knowledge of the robot’s construction. In 1990, Micheal McKenna and David Zeltzer present a paper on Dynamic Simulation of Autonomous Legged Locomotion. Similar to this study, McKenna and Zeltzer use a software package which uses its knowledge of the robot kinematics and dynamics to complete this simulation. Zeltzer developed a gait controller for their hexapod robot influenced by the work of McGhee. However, to determine necessary joint torques to support the robot body and achieve desired leg configurations, they use a program called *Corpus*. Their program consists of a dynamic simulator, gait controller, and motor programs. The dynamic simulator forms the base of the simulation and utilizes the gait program to determine desired joint positions, and the motor programs to deliver forces to the joints.[11]

This research utilizes the SimMechanics toolbox for MathWorks’s Simulink is a valuable tool for all kinds of physical system modeling. Systems are modeled using a series of what Simulink calls “blocks” linked to one another via inputs and outputs, like a breadboard of electrical components with wires carrying various kinds of signals. The wires are fairly universal, however each block needs to be set up with various parameters to do what the user desires it to do. Blocks include those that do mathematical computations, signal processing, and with the SimMechanics package, act as physical bodies. After setting various block parameters as joint configurations, physical information about bodies and joints, and environmental information, one can model any physical system imaginable. However, a shortcoming arises when time step dependent input signals are needed. Simulink provides blocks for various mathematical signals, such as sine waves and transfer functions, however for products of polynomial and advanced functions (such as those in this project used to generate joint torques over the course of the step trajectory) one must first write them in Matlab and then interface them with the rest of the system. In robotics, where complex and novel trajectories are frequently used, this can be a challenge. This research also seeks to show methods of integrating such signals into a SimMechanics simulation.

1.4 Accomplishments

Here is a short bulleted list highlighting the goals accomplished in this research.

- Full forward and inverse kinematics for the robotic leg
• Forward and inverse kinematics from the center of the robot body to the end of each leg
• A Matlab script which displays a stick model of leg on the robot body
• Scripts to allow the static stability polygon and the robots center of mass projected on the polygon
• A revised mechanical design for the robot leg
• The full Jacobian for the three degree of freedom leg
• Euler-Lagrange dynamic equations for the robot leg
• Three different Matlab scripts for leg trajectories
• Development of two methods for executing a quintic trajectory in Sim-Mechanics
• Sim-Mechanics model of the robotic leg
• A Simulink block which generates desired joint position, velocity, and acceleration for each leg trajectory
• A Simulink block which uses the Jacobian to translate forces exerted on the robot foot to the joint torques necessary to overcome that force
• A Simulink/Sim-Mechanics model which uses the trajectory as an input and returns the necessary joint torques
• A Simulink/Sim-Mechanics model which takes joint torques as an input and returns the joint motion
• A Matlab function which calculates the Euler-Lagrange equations in matrix form
• A Matlab function which returns a set of torques based on a trajectory loop
• A Matlab script which uses ode45 to determine joint velocity and acceleration based on torques
• Calculation of leg singularities and manipulability throughout the motion of the leg
Chapter 2

Kinematic and Dynamic Modeling of a Robot Leg

The primary focus and scope for this project is to fulfill the need for a dynamic model of a single leg for the multi-legged reconfigurable robot as it steps in such a way to move a full robot body forward. This paper does not investigate the scope of how the entire reconfigurable robot works together or the design of its central processor. This project will outline the development of a trajectory for a general step motion for the leg which could be made adaptive in future work.

2.1 Mechanical Design

The mechanical leg seeks to keep the leg as light and compact as possible while containing leg components (such as motors and sensors) inside of the leg frame. We took the design for the original leg for this project and adapted to achieve these goals. Both the former and newer designs placed motors and potentiometers between rails and terminating each leg link with a gearbox to re-direct the motors force along the joint. Both designs are also influenced by MiniQuad 1 [7] in their construction and reconfigurability.

However, the old design was found lacking in its rigidity. When handing the constructed robot leg one could feel a lot of slack in its joints, and the gearboxes were unlikely to hold during regular motion. It had nearly 10 inch long legs made with one eighth inch aluminum rail on each side. The legs had gear boxes at each end made from blocks aluminum stock that were approximately 2 inch wide by 3 inch long and 1 inch deep with a small one inch by 1 and a half inch area milled out to contain the gears. This resulted in light legs that were heavy and bulky at either end. The gears themselves
were too small (about half inch in diameter) to endure the forces applied to them if the robot were to walk under load. Also, the gear shafts were coupled to the motors with two shafts joined by a coupler that resided in another milled out area of the aluminum block. Holding two leg links, one could move the joint without the motors moving because of the slack in the coupling. Also the coupled gears would often “skip”, that is disengage far enough such that torque could not transfer from one to the other. Figures of the original leg are shown in 2.1 and 2.2.

![Figure 2.1: Former Leg Design](image1)

![Figure 2.2: Former Hip Joint Design](image2)

A revised design for the robotic leg would require the following:
1. The robot leg shall utilize the same motors and potentiometers implemented in the previous design.
2. The robot legs shall be able to support the dynamic forces exerted by the robot whole when at least four legs are used.

3. The robot leg links shall have sufficient length to allow it to step onto or over objects.

4. The robot leg shall be free of objects that stick out to avoid getting caught on objects or limiting the range of motion.

5. The joint motors on the robot leg shall be able to provide torque to joints without loss of motion (no skipping or slipping).

6. The potentiometers shall accurately read the position of the robot joints by some direct factor of the actual joint position in radians.

The new robot design achieves these goals in the following ways:

1. Motors and potentiometers were accurately measured and included in the new model assembly to ensure fit.

2. This has not been tested, but the new design has more supports and less room for flex than the previous model.

3. While the leg links are shorter than the previous model, they are still capable of reaching up to 20 cm above and 40 cm below the robot body at their extremes.

4. Potentiometers have been moved closer to links 2 and 3, and inside of link 1 while they were previously on standoffs up to 2 cm from the legs. The motors for link 2 and 3 still reside inside of link 2, and the motor for link 1 has been moved partially inside the link where it was previously on standoffs above the link.

5. Couplings have been eliminated from the shafts reduce change of slipping, and larger gears for joint 2 and 3 were used to reduce risk of skipping.

6. The potentiometers are now linked to the geartrain with a single coupler rather than two to reduce risk of slipping.

This design started by creating accurate CAD models of the original motors and potentiometers to be used in the final leg assembly. For the second requirement, the legs were made more robust by adding more bracing in the leg frame to avoid collapse. The length of the robot leg was actually shortened to provide the robot with more stability, however they are still sufficiently long to step onto or over small objects. The former design also left the motor and potentiometer for the first link sticking up and out from the leg. The re-designed robot leg frames place the motors inside the leg frame, with the potentiometers mounted flush with the edges of the frame. This keeps the objects from sticking out from the robot leg. The issue of gear slop and skipping can be avoided by either using stronger boxes to contain the gears, or simply using larger gears with deeper teeth. The latter was chosen for simplicity and cost. In order to avoid adding further weight and volume to the size of the boxes with the introduction of larger gears, the
gearboxes were re-designed to be smaller and fit more snugly into the leg rails. To reduce extraneous length to the legs (thus reducing strain on the gears and motors) the motors were placed side by each inside the rails. Also, motors were attached to gears with long shafts and a number of couplings, which introduces slop through potential bending or twisting in the shafts and couplings. The new design brought the motor closer to the gears, thus bringing the motor and potentiometer closer to the gears themselves and reducing shaft length and the number of couplings. By moving the motors and potentiometers closer to the gears and joints, one can keep the legs from having items that stick out, as well as reduce bending, twisting, and skipping that may occur between the legs actual motion and its motors and potentiometers, making the system as a whole more accurate.

The new leg design also allows for the addition of a simple three degree of freedom force sensor to be implemented at the tip of the third link for future work on this project. The force sensor would simply be a piece of square stock of a known material strength, with at least three strain sensors attached. As the foot interacted with the environment, it would place strain on the square stock which could be read by the sensors. After processing signals from the sensors, one could determine the magnitude and direction of forces being exerted on the robot foot. This information would be a highly valuable addition to the overall control design, providing both feedback and knowledge of the environment to the leg controller. The legs are also sturdy and robust enough to support not only their own weight, but also that of the robot body and any other attached peripherals. Such a model has been developed in Pro-Engineer and are shown in Figure 2.3 through 2.4:

Accurate physical models drawn in many CAD programs, such as Pro-Engineer or SolidWorks, can be imported to SimMechanics, making integrating 3D drawings with SimMechanics models simple and convenient, and making such a drawing a valuable tool in system modeling.[17] While this model produces fully defined values for mass, link length, and moments of inertia for the model, this information was not used in the simulation. Rather than importing the model into Sim-mechanics, the parameters were left as variable throughout the simulation. This is done because the new mechanical design was never implemented, and in future work developers may decide to adjust this mechanical model further. The currently implemented model and simulation allow the user to input these values as desired.
Figure 2.3: Rendering of Leg Mechanical Design

Figure 2.4: Leg assembly view from above
Figure 2.5: Leg assembly view from underneath
2.2 Forward Kinematics

To build a numerical and SimMechanics simulation for a robotic leg, one must first define the parameters and set up coordinate systems. For this project, we use them both for creating Euler-Lagrange equations and for setting up the coordinates and orientation for body blocks in the SimMechanics model. The coordinates for a single leg of the robot are defined in Figure 2.6:

![Diagram of Single leg with relative coordinates for each joint](image)

Figure 2.6: Diagram of Single leg with relative coordinates for each joint

In order to define the forward kinematics, we use the Denavit-Hartenberg convention. Using this method each coordinate frame, $n$, is defined by: joint angle $\theta$; link twist $\alpha$; link offset $d$; and link length, $a$. Note that this leg has no linear actuators, only revolute actuators. Applying these variables to the robot leg we have:

<table>
<thead>
<tr>
<th>Link Number</th>
<th>$\theta$</th>
<th>$\alpha$</th>
<th>$d$</th>
<th>$a$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>$\theta_1$</td>
<td>90°</td>
<td>0</td>
<td>$a_1$</td>
</tr>
<tr>
<td>2</td>
<td>$\theta_2$</td>
<td>0</td>
<td>0</td>
<td>$a_2$</td>
</tr>
<tr>
<td>3</td>
<td>$\theta_3$</td>
<td>0</td>
<td>0</td>
<td>$a_3$</td>
</tr>
</tbody>
</table>

Using these values we can define the transformation matrices from one joint to another. To generate a transformation matrix from the origin to joint one, we first need to make a transformation for each rotation and translation from the origin to joint one. All transformation matrices have the following format:

$$
T_{i,j} = \begin{bmatrix} R_{i,j} & a_{i,j} \\ 0 & 1 \end{bmatrix}
$$

(2.2.1)
Where $T_{i,j}$ represents a full three dimensional transformation from an initial coordinate system $i$ to a final coordinate system $j$. Inside of which $R_{i,j}$ is a three by three rotation transformation from coordinate frame $i$ to $j$ and $a_{i,j}$ is the coordinate vector from coordinates $i$ to $j$. For example, the full transformation from the origin coordinate system to the coordinate system of the second joint:

$$
T_{0,1} = \begin{bmatrix}
\cos(\theta_1) & 0 & \sin(\theta_1) & a_1 \cos(\theta_1) \\
\sin(\theta_1) & 0 & -\cos(\theta_1) & a_1 \sin(\theta_1) \\
0 & 1 & 0 & 0 \\
0 & 0 & 0 & 1
\end{bmatrix}
$$

(2.2.2)

We then create a transformation from joint 0 to 1, 1 to 2, 2 to 3. Multiplying the three together and using trigonometric identities produces the full transformation matrix is shown in Equation 2.2.3:

$$
T_{0,3} = \begin{bmatrix}
c_1c_{2+3} & s_1 & a_1c_1 + a_2c_1c_2 + a_3c_1c_{2+3} \\
-c_1s_{3-2} & s_1 & a_1s_1 + a_2s_1c_2 + a_3s_1c_{2+3} \\
s_1s_{3-2} & c_2 + 3 & a_2s_2 + a_3s_{2+3} \\
0 & 0 & 1
\end{bmatrix}
$$

(2.2.3)

Note: where $c_1$ is equivalent to $\cos(\theta_1)$ and $c_{2+3}$ is $\cos(\theta_2 + \theta_3)$, $s_1$ is equivalent to $\sin(\theta_1)$ and $s_{2+3}$ is $\sin(\theta_2 + \theta_3)$, and so forth.

In this matrix format, the rotation from the origin orientation to the tip orientation is given in the first three columns and rows of the matrix, and in x, y, z Cartesian positions from the base of the robot leg to the tip of the third and final link are given in the first three rows of the fourth column. When the transformation matrix is filled in with all three values for $\theta$, it becomes simple to determine the position and orientation of the tip in relation to the base of the robot leg.

The following table provides the dimensions of the robot leg used in the simulation. These parameters can easily be changed as desired if the leg design is changed or for different applications. All measurements given in meters.

<table>
<thead>
<tr>
<th>Link Number</th>
<th>Link Length</th>
<th>Link Width</th>
<th>Link Height</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0.08</td>
<td>0.06</td>
<td>0.06</td>
</tr>
<tr>
<td>2</td>
<td>0.2</td>
<td>0.06</td>
<td>0.06</td>
</tr>
<tr>
<td>3</td>
<td>0.2</td>
<td>0.06</td>
<td>0.06</td>
</tr>
</tbody>
</table>
2.3 SimMechanics Model

Later in this paper we will use Matlab SimMechanics software to verify the mathematical model of the robotic leg. This model will be used to ensure that torques calculated based on desired position, velocity, and acceleration will actually produce the desired positions in simulation. Here, we will introduce how to set up the kinematics of the robotic leg in SimMechanics so that it can later be used to verify the torque calculations.

To implement a model in SimMechanics, we create a file called a model, or .mdl file. Every SimMechanics model must contain an environment block and at least one ground block. "Blocks" are the parts code that make up the simulation. The user can access the block library by typing "simulink" into the command line in Matlab and then pressing enter. Here the user is presented with a collection and Simulink and SimMechanics blocks organized by category. One can also use the search function to find the desired block. When assembling a SimMechanics model is it best to begin by placing an environment block from the library into the model. Parameters of blocks cannot be changed until the block is placed in a model, as the parameters settings will be model specific. The environment block provides the whole model with information about gravity, physical dimension tolerances, and model parameters. The only parameter necessary to change in many cases is the gravity vector. To change a blocks parameters, we double click on the block itself which opens a dialogue box in which we can make changes. For our coordinate system, as seen previously in Figure 2.7, gravity is in the negative Z direction. Since SimMechanics assumes that gravity is always negative, so set the gravity as 9.81 meters per second squared in the Z direction.

![Figure 2.7: SimMechanics Leg Model](image)

The environment block is then connected to a ground block, which is used to connect the remainder of the model to a user defined static position in global coordinates. We connect blocks by clicking on the output port on the environment block (on the right) and drag to the input port of the ground block (on its left). This connection can be seen in the upper left of Figure 2.7. For a single leg, the ground block is where the leg attaches to the robot.
body, and is the origin of the leg coordinate system. In this way we abstract the robotic leg from the walking robot as a whole. Thus, the location of the ground block is set to the origin. Now we can begin to assemble the parts specific to our own model. From this point we use a joint block to provide our first joint. SimMechanics has a number of different joint block types to choose from, but here we select the single rotational axis joint. After placing it in the model, we adjust parameters using the dialogue box as shown in Figure 2.8.

![Figure 2.8: The dialogue box for defining the parameters of joint 1 in SimMechanics](image)

This block allows us to set the joint’s axis of rotation, which, based on our coordinate system, is in the Z direction. The joint block also requires a location, this can be determined by its base or follower, we are going to use the base to define the joints coordinates. We connect it to the ground block by dragging a line from the ground output to the joint input. Now the joint is located at the origin. The next step is to define the first link of the robotic leg. For this we use a body block. Double clicking on SimMechanics blocks will open the dialogue box to define its parameters. We fill in each body block as shown in Figure 2.9 with physical information, such as mass, the inertia matrix, location of the center of mass, and location of the end points where it connects to other blocks. For us we determine the location of the base by its base, the origin. As for the CoM and the endpoint, these are
given from the position the body will take when in its ”initial configuration”. The initial configuration (unless otherwise defined) for SimMechanics models occurs where all joints are 0 °. So, for this joint, the endpoint is at [0, 0.8, 0], where 0.8 is the length of the link in meters. The CoM is halfway down the link (this will be different if one has actually determined the center of mass based on the 3D model, but for simplicity we place it at the center of the link) at [0, 0, 0.4].

Figure 2.9: The dialogue box for defining the initial position of a body block in SimMechanics

The next portion is to set up the orientation of the leg as shown in Figure 2.10. This is accessed by clicking on the orientation tab in the body parameters dialogue box. Orientation can be defined a number of ways, however, once the transformation matrices have been derived, it is simplest to use the three by three rotation matrix option in the body block. Here it is important to maintain consistency without numerical model defined above. One will see that along Link 1 we re-oriented the coordinate system, the same must occur in the SimMechanics model. We accomplish this using the links orientation parameters. Since we know that the transformation matrix from joint 1 to joint 2 is a 90 ° rotation about the X axis. Using the first three columns and rows of our transformation matrix, giving us the rotation matrix, we fill these into our body parameters orientation for link 1.

Now we can add a second single axis joint block for link 2, labeled as Joint 2 in Figure 2.7. This joint we are defining based on its base, the first body, link 1. This way the joint will include the transformation that occurred along link 1. So, when we set the parameters, we define joint 2 as rotating about the Z axis. Not the global Z axis, as joint one, but rather the redefined Z axis from it’s base, link 1. Now we can attach another body block, Link 2.
Once again the inertia matrix, mass, position, and orientation for this link need to be defined.

The SimMechanics model also makes use of joint actuators and joint sensors. The actuators allow a user to move a body or joint, and sensors produce desired information about the joint. In our model, for each joint, we need to add a joint actuator from the SimMechanics library. As shown in Figure 2.11, to define a joint actuator, one can specify joint torque or joint movement as an input. The movement option allows the user to input position, velocity, and acceleration and the actuator will drive the joint block move within those parameters. We however use torque as an input because the model is intended to verify the Euler-Lagrange equations that generate the necessary torques to move the leg. Also, be sure to select the correct torque units when setting up this block.

Joint sensors are capable of reading position, velocity, acceleration, extrapolated torque, or all three using check boxes in the sensor block’s dialogue box. Since desired position is what we use to drive our torque producing algorithm, we seek to verify the model by ensuring that the position read by the sensors matches what was put into the algorithm, and thus check those boxes in the dialogue box.

Note that once this is applied, there will be not one but three ports from the sensor block, one for each metric. Parameters for defining a sensor block are shown in Figure 2.12. In order for the user to view these values, one has to use what is called a “sink” in the Simulink block library. For initial development, we used a display or floating scope block, which show real time values while the simulation is running. For the final model, we use “to file”
blocks. Opening the dialogue box for these blocks the user defined the name of the file and variable. After the simulation is run, one can create plots using the timeseries data saved in this file. Such a blocks are not shown in the figure. At this point, the SimMechanics model is complete, everything but the input torque (which will come from our Euler-Lagrange equations and or trajectory generation) has been assembled.
2.4 Inverse Kinematics

With the knowledge of the forward kinematics, we can also determine the inverse kinematics, which aim to give us the three joint positions based on a desired Cartesian position (x, y, z coordinates) of the robot foot. The inverse kinematics can be solved for using \( a_{0,i} \) of equation 2.2.1 in the transformation matrix in equation 2.2.3 and setting them equal to positions x, y, and z. We must then solve for each of the \( \theta_i \), where \( i=1,2 \) and 3, values in terms of x, y, and z. The inverse kinematics for a single leg in our case are as follows:

\[
\theta_1 = \arctan\left(\frac{x}{y}\right)\quad (2.4.1)
\]
\[
\theta_2 = -\arctan\left(\frac{a_3 \sin(\theta_3)}{a_2 + a_3 \cos(\theta_3)}\right) + \arcsin\left(\frac{z}{\sqrt{(a_2 + a_3 \cos(\theta_3))^2 + a_3^2 \sin(\theta_3)^2}}\right)
\]

(2.4.2)

\[
\theta_3 = -\arccos\left(\frac{x \cos(\theta_1) - a_1}{\frac{(x \cos(\theta_1) - a_1)^2 + z^2 - a_2^2 - a_3^2}{2a_2a_3}}\right)
\]

(2.4.3)

These calculations can be used for defining a trajectory for the leg in joint space based on desired leg locations in global Cartesian space.

### 2.5 The Full Robot Body

Section 2.4 covered the forward and inverse kinematics of a single robot leg, which is similar to calculating the kinematics of any other three degree of freedom manipulator. This project diverges from others with the need to calculate the forward and inverse kinematics of a full robot body. Instead of placing the origin of the coordinates at the first joint, we need to translate the origin to the center of the robot body. Our robotic body is a simple rectangle with a number of ports along the perimeter at which legs can be placed, as shown in Figure 2.13. Each port has a unique kinematic transformation from

![Diagram Robot Body with Ports Numbered](image)

Figure 2.13: Diagram Robot Body with Ports Numbered

the coordinate system at the base of the leg (joint 1) to a coordinate system place at the center of the robot body. For example, the inverse translation for port 0 is thus:

\[
y_{\text{robot}} = y_{\text{leg}} - \frac{W}{2}
\]

(2.5.1)
\[ x_{robot} = x_{leg} + \frac{L}{8} - \frac{pL}{4} \quad (2.5.2) \]

\[ z_{robot} = z_{leg} \quad (2.5.3) \]

Where \( W \) is the width of the robot body, \( L \) is its length, and \( p \) is the port number. As one can see, this is a simple two dimensional shift from port/leg coordinate frame to a coordinate frame at the center of the robot.

### 2.6 Chapter 2 Summary

In chapter two we began by defining the coordinate systems used to define the single robot leg. Based on these frames we were able to derive the forward and inverse kinematics for the single leg. We then discussed the mechanical features of re-designed robot leg, and how this new design adds improvements over the previous generation of this robots design. We then introduced use of the simulation software SimMechanics, based on Matlab. We walked through how to construct a simulation model of the single robot leg using their block library, and how to set the block parameters using information derived in the forward kinematics. We then briefly outlined the forward and inverse kinematics for translating from coordinate systems in terms of the leg to those in terms of the whole robot body. With the basic knowledge of the forward and inverse kinematics, and how to construct a SimMechanics simulation model, we can now move forward to define the dynamics and see how they may be verified using the SimMechanics software.
Chapter 3

Robot Leg Dynamics

After understanding the robot in terms of its kinematics, one can begin to investigate the dynamics. It is important in modeling a robot motion to calculate dynamics so that velocity, acceleration, and the forces associated with the robot motion are accounted for. In this research, we calculate the necessary dynamic equations to determine the necessary torques to apply to the joint in order to produce the desired motion. This produces a dynamic model of the leg system which can later be verified.

3.1 Calculating the Jacobian Matrix

Once the kinematic equations have been developed, the next part of creating a mathematical model of the leg system is to calculate the Jacobian matrix. The Jacobian matrix is derived from the transformation matrices and is used for various purposes in robot manipulator control. The Jacobian can be defined as the relationship between joint velocity and manipulator end velocity.

\[
\xi = J(q)\dot{q}
\]  

(3.1.1)

where \( \xi \) is the end of the manipulator velocity and \( \dot{q} \) is a vector of joint velocities, and \( J \) represents the Jacobian matrix.[1] One valuable use of the Jacobian is for calculating joint torques necessary in robot leg to maintain configuration in response to external forces applied to the foot. This is done using using the transpose of the Jacobian matrix, as shown in the equation below:

\[
\tau = J^T(q)F
\]  

(3.1.2)

where \( \tau \) is a vector of joint torques, \( J \) is the Jacobian of the joint position vector \( q \), and \( F \) a three dimensional vector of force on the robot foot, arranged
in the form \( F = [F_x, F_y, F_z, n_z] \). This equation will be used in the final robot model to incorporate the effect of external forces on the end of leg (the foot).

The Jacobian matrix to be used for these purposes is set out by visualizing a set of equations being multiplied by the set of joint velocities to produce end point velocities:

\[
\begin{bmatrix}
\dot{x} \\
\dot{y} \\
\dot{z} \\
\omega_x \\
\omega_y \\
\omega_z
\end{bmatrix} = \begin{bmatrix}
J_{v1} & J_{v2} & J_{v3} \\
J_{w1} & J_{w2} & J_{w3}
\end{bmatrix} \begin{bmatrix}
\dot{\theta}_1 \\
\dot{\theta}_2 \\
\dot{\theta}_3
\end{bmatrix}
\]

(3.1.3)

where each \( J_v \) and \( J_w \) are 3x1 column vectors. One can see in the matrix that each column is associated with a different joint. Each of which is broken into what we call the upper and lower Jacobian. Each is also calculated differently depending if the joint is prismatic or revolute. All the joints in our leg are revolute. Therefore, for each \( J_v \):

\[
J_v_i = z_{i-1} \times (o_n - o_{i-1})
\]

(3.1.4)

In which \( z_i \) the orientation of the z axis after it has been translated from the first or origin coordinate frame to the coordinate frame of joint \( i \). It can be taken from the last column in the rotation matrix from the origin to joint 0, which recall are the first three columns and rows of that same transformation matrix. \( o_i \) is the 3x1 coordinate vector from the base of the leg to joint \( i \). It can be taken from the first three values in the last column of the the transformation matrix from the leg base to joint \( i \). \( o_n \) the three by one orientation vector for the tip of the robot leg, which is gathered from the full transformation matrix from the leg base to the final point of the manipulator. The calculation for \( J_{v2} \) is provided as an example:

\[
J_{v2} = z_1 \times (o_3 - o_1)
\]

(3.1.5)

\[
J_{v2} = \begin{bmatrix}
\sin(\theta_1) \\
-\cos(\theta_1) \\
0
\end{bmatrix} \begin{bmatrix}
o_{3-1x} \\
o_{3-1y} \\
o_{3-1z}
\end{bmatrix} = \begin{bmatrix}
-\cos(\theta_1)o_{3-1z} \\
\sin(\theta_1)o_{3-1z} \\
\sin(\theta_1)o_{3-1y} + \cos(\theta_1)o_{3-1x}
\end{bmatrix}
\]

(3.1.6)

The results of these calculations can be seen in the final Jacobian. As opposed to \( J_{v1} \), the upper Jacobian which calculates the linear velocities of the joint, \( J_{w1} \) calculates angular velocities, and for each:

\[
J_{w1} = z_{i-1}
\]

(3.1.7)
Which is already known from calculating $Jv_i$. In our case, the fully assembled Jacobian for the manipulator is:

$$
J = \begin{bmatrix}
-a_1s_1 - a_2s_1c_2 - a_3s_1c_{2+3} & -c_1(a_2s_2 + a_3s_{2+3}) & -c_1(a_3 + s_{2+3}) \\
a_1c_1 + a_2c_1c_2 + a_3c_1c_{2+3} & -s_1(a_2s_2 + a_3s_{2+3}) & -s_1(a_3 + s_{2+3}) \\
0 & 0 & 0 \\
0 & -c_1 & 0 \\
1 & s_1 & s_1 \\
0 & -c_1 & 0 \\
c_{2+3} & s_2 & 0 \\
s_2 & c_{2+3} & 0 \\
0 & 0 & 0
\end{bmatrix}
$$

(3.1.8)

For this project, the Jacobian matrix provides us with a way to include the weight of the robot in the joint torque calculations without communication from the robot body (provided this weight is known). In future work, this provides a way for force information fed from a manipulator tip sensor to be incorporated to calculations for joint torques. In the simulation, the Jacobian is used to translate external forces on the robot leg to joint torques, which can then be added to the joint torques calculated elsewhere in the simulation. The Jacobian as implemented in the SimMechanics simulation is shown in Figure 3.1.

![Figure 3.1: The Jacobian as a User-Defined block in a SimMechanics Sub-system](image)

### 3.2 Dynamics

The Lagrangian equation used throughout the field of engineering as an energy based model of a physical system. The Euler-Lagrange equation for manipulator control is a variant of the Lagrangian specifically suited for determining joint torques for a multi-link manipulator. This equation provides the dynamic model for the manipulator. The Euler-Lagrange equations for
robotic manipulators represent the overall energy in the robotic leg system. Using this concept, we can create an equation where one side determines energy driven by manipulator configuration (joint position, velocity, and accelerations) and the other side to determine the torques required to achieve a desired configuration.

3.2.1 Components of the Manipulator Lagrangian

The basic concept for the Lagrangian is set out in equation 3.2.1:

$$L = K - P$$

(3.2.1)

where the Lagrangian is the difference between kinetic energy terms $K$ and potential energy $P$ throughout the moving body(s). However, in manipulator control, we use the Lagrangian to calculate the various joint torques necessary in a manipulator to generate desired joint positions, velocities, and accelerations. Those equations which provide desired torque are called the *Euler-Lagrange* equations, and are derived from the calculations collected in the former sections of this paper. We will be using a common method for calculating the Euler-Lagrange equations specifically for a robotic manipulator made up of rigid-body links [1]. We begin with the kinetic energy for a single link which is given as:

$$K = \frac{1}{2}mv^T + \frac{1}{2}\omega^T I \omega$$

(3.2.2)

where $K$ is kinetic energy, $m$ is link mass, $v$ is the linear velocity of the link at center of mass, $\omega$ is rotational velocity of the link, and $I$ is the inertia matrix for the link. One can see the linear and rotational components of kinetic energy, where the energy equals one half mass times a velocity vector, plus one half the angular velocity vector multiplied with matrix $I$, which is our inertia tensor. The inertia calculations are the next part of the puzzle. This is calculated:

$$I = RIR^T$$

(3.2.3)

In which $R$ is the rotation matrix for that particular link, taken, as mentioned before, from the first three columns and rows of its transformation matrix. $I$ however is the inertia matrix, which is calculated based on the physical properties of the link. If one is working with a specific mechanical design that has been fully specified using 3D modeling software, these parameters can be taken from the model. In many cases, however, it is sufficient to
model the link as a uniform rectangular solid, in which case the inertia can be calculated as in 3.2.4:

\[
I = \begin{bmatrix}
(l^2_h + l^2_w) \frac{m}{12} & 0 & 0 \\
0 & (l^2_h + l^2_w) \frac{m}{12} & 0 \\
0 & 0 & (l^2_w) \frac{m}{12}
\end{bmatrix}
\]

(3.2.4)

Where \( l \) is the link length, \( l_h \) is the link height, and \( l_w \) is link width. Since we use the Denavit-Hartenberg convention, the dimension in which the link height is measured parallel to the z axis (parallel to the axis around which the previous joint rotates) and the link width is along the x axis, since the length of the link always runs along the y axis. Now that we have all the necessary information for each link, we calculate the kinetic energy for an n-link manipulator, which is calculated as in 3.2.5:

\[
K = \frac{1}{2} \dot{q}^T \left[ \sum_{i=1}^{n} \left\{ m_i J_{vci}(q)^T J_{vci}(q) + J_{wci}(q)^T R_i(q) IR_i(q)^T J_{wci}(q) \right\} \right] \dot{q}
\]

(3.2.5)

Many notations will also use \( D \) to represent the matrix form of the kinetic energy computed within the brackets above such that:

\[
K = \frac{1}{2} \dot{q}^T D \dot{q}
\]

(3.2.6)

The next variables \( J_{vi} \) and \( J_{wi} \) are the upper and lower halves of a full Jacobian matrix taken from the origin to the center of mass of link \( i \). These are not components of the already calculated Jacobian matrix. Instead, one must calculate each Jacobian matrix for each center of mass \( i \) in the same way as for a full manipulator described in section 3.1. One merely substitutes the \( o_i \) variable with \( o_{ci} \) to represent the center of mass, as shown:

\[
J_{vi} = z_{i-1} \times (o_{ci} - o_{i-1})
\]

(3.2.7)

Another change is that unlike the full Jacobian matrix to the end of the manipulator, the Jacobian matrix for links 1 and 2 will mean that for some columns, \( i - 1 \) will be greater than \( n \), in these cases, \( J_{vi} \) or \( J_{wi} \) will be a three by one column of zeros. As an example, take the center of mass on the second link.

In the equation for the kinetic energy we also use \( q \) and \( \dot{q} \), which represent three by one vectors of joint positions and velocities, respectively. We take the summation of the linear and angular velocity components for all the links, depending on their configuration. Next we compute the potential energy.
for the n-Link manipulator, which is simply a summation of the potential energies for each link shown in equation 3.2.8.

\[ P = \sum_{i=1}^{n} m_i g^T r_{ci} \]  

(3.2.8)

In which \( m_i \) is mass of link \( i \), \( g \) is the gravity vector in terms of the base coordinate frame, and \( r_{ci} \) provides the coordinates of the center of mass of link \( i \) from the base to the center of mass of link \( i \).

### 3.2.2 Methods of Computing the Euler-Lagrange for a Manipulator

The next step in this process is to assemble the described matrices into the manipulator Lagrangian by forming the kinetic and potential energy parts and then differentiating to calculate the torques for each joint necessary to produce the desired manipulator configuration. Two different methods and two different software packages were used to compute the Lagrangian.

Let us begin by outlining the computational steps for calculating torque of each joint using the manipulator Lagrangian. First, the partial derivative of the Lagrangian with respect to the joint velocity of joint \( k \):

\[ \frac{\partial L}{\partial \dot{q}_k} = \sum_j d_{kj} \ddot{q}_j \]  

(3.2.9)

Here the \( j \) subscript notes columns and rows of the corresponding matrices. \( d_{kj} \) represents the \( j \)th row of the \( D \) matrix for the \( k \)th joint. We then take the full derivative of the same by doing the following:

\[ \frac{d}{dt} \frac{\partial L}{\partial \dot{q}_k} = \sum_j d_{kj} \ddot{q}_j + \sum_{i,j} \frac{\partial d_{kj}}{\partial q_i} \dot{q}_i \dot{q}_j \]  

(3.2.10)

We also take the partial derivative of the Lagrangian with respect to the \( k \)th joint position:

\[ \frac{\partial L}{\partial q_k} = \frac{1}{2} \sum_{i,j} \frac{\partial d_{ij}}{\partial q_k} \dot{q}_i \dot{q}_j - \frac{\partial P}{\partial q_k} \]  

(3.2.11)

These four components can then be laid out in the following way:

\[ \sum_j d_{kj} \ddot{q}_j + \sum_{i,j} \left\{ \frac{\partial d_{kj}}{\partial q_i} - \frac{1}{2} \frac{\partial d_{ij}}{\partial q_k} \right\} \dot{q}_i \dot{q}_j + \frac{\partial P}{\partial q_k} = \tau_k \]  

(3.2.12)
If we take the first summation to equal $D$, the two expressions in the brackets as $C$, and the last part as $G$, and adding the Jacobian and external forces, we have:

$$\tau = D\ddot{q} + C(q, \dot{q})\dot{q} + G(q) + J^T F_{ext} \quad (3.2.13)$$

The result is the the Euler-Lagrange equations in matrix form.

The $D$ matrix value comes from taking the second derivative the kinetic energy of the manipulator, and uses the input of joint acceleration, thus this factor represents torque based on angular force. The next factor $C$ represents what are called the Coriolis and centrifugal or coupling effects of the manipulator system on joint torques. Note it includes both angular position and velocity terms. The $G$ matrix includes forces based on the influence of gravity. The final factor takes account for external forces on the manipulator tip. Multiplying a vector for external forces by the transverse Jacobian results returns the extra torque on each joint due to external forces.

In this project, we used Matlab code to define the various matrices used in the Euler-Lagrange equations. However, when it came to differentiating these matrices, there appeared to be no elegant way to accomplish this in Matlab. It was attempted with Matlab’s symbolic toolbox, however it was difficult, if not impossible, to properly define joint accelerations and velocities as derivatives of joint position and maintain this definition while Matlab computed the time derivative. Eventually, the entire code for calculating the Lagrangian was copied to Wolfram Mathematica for the derivations, and then copied back to Matlab to provide a function that took joint positions, velocities, and accelerations and calculated joint torques that would provide these configurations.

Using Wolfram’s Mathematica software, the Lagrangian was computed by taking a series of derivatives and partial derivatives to the equation. After setting out the various parts of the Lagrangian, the Mathematica script assembled the kinetic energy for the Lagrangian as such:

*The Kinetic energies for each link*

$D_{k1} = m_1 \cdot (Jv1^T \cdot Jv1) + Jw1^T \cdot R1 \cdot I1 \cdot Jw1$;  
$D_{k2} = m_2 \cdot (Jv2^T \cdot Jv2) + Jw2^T \cdot R2 \cdot I2 \cdot Jw2$;  
$D_{k3} = m_3 \cdot (Jv3^T \cdot Jv3) + Jw3^T \cdot R3 \cdot I3 \cdot Jw3$;  
$D_{tot} = D_{k1} + D_{k2} + D_{k3}$;

This $D$ matrix was multiplied by a vector of joint velocities, and the Potential Energy for each leg was added producing:

$L = \text{FullSimplify}[(1/2) \cdot Jv^T \cdot D_{tot} \cdot Jv - (P1 + P2 + P3)]; \quad *7.53*$
Where the “full simplify” function is used to simplify the result. Then for each joint \( i \), the script takes the partial differential of the Lagrangian \( L \) with respect to velocity \( \dot{q}_i \). This is equivalent to equation 3.2.9 above:

\[
\text{*Partial differentials of Lagrangian with respect to qdot*} \\
L_{\dot{q}_i} = D[L, Dt[q_i[t], t]]; 
\]

*7.55*

then the time derivative of the result, equivalent to equation 3.2.10.

\[
\text{*Derivatives (d/dt*dL/dqdot)*} \\
L_{\ddot{q}_i} = Dt[L_{\dot{q}_i}, t]; 
\]

*7.56*

Then the partial derivative of \( L \) with respect to \( q_i \) just as in equation 3.2.11

\[
\text{*Partial Derivatives of Lagrangian with respect to q*} \\
L_{q_i} = D[L, q_i[t]]; 
\]

and finally the torque \( \tau_i \) by taking the difference between time derivative and the partial with respect to \( q_i \).

\[
\text{*Calculate Torques t=d/dt* L/dqdot - L/q* *7.42, 7.62*} \\
\tau_i = \text{Chop[FullSimplify[L_{\ddot{q}_i} - L_{q_i}]}} 
\]

The symbolic computations for deriving the dynamic model has been done in Mathematica and MATLAB is used for the numerical implementation. Appendix A presents the Mathematica notebook and Appendix B shows the Matlab script.

The Euler-Lagrange equations were later calculated in Matlab when it was realized that there existed a method which only used partial derivatives with respect to \( q_i \) and not time. With this method, rather than taking time derivatives it uses the matrix form set out by Spong [1]. To this end, we created the files NewLagrangian.m and Christoffel.m. NewLagrangian.m takes all the parts from the previous section and uses them to create the \( D \), \( C \), and \( G \) matrices. The \( D \) matrix is simply:

\[
%\text{Compute portions of D matrix} \\
D_{\text{tran1}}=(m1/2)*Jcv1'*Jcv1; \\
D_{\text{tran2}}=(m2/2)*Jcv2'*Jcv2; \\
D_{\text{tran3}}=(m3/2)*Jcv3'*Jcv3; \\
D_{\text{rot1}}=Jcw1'*R1*I1*R1'*Jcw1; \\
D_{\text{rot2}}=Jcw2'*R2*I2*R2'*Jcw2; \\
D_{\text{rot3}}=Jcw3'*R3*I3*R3'*Jcw3; \\
D=D_{\text{tran1}}+D_{\text{tran2}}+D_{\text{tran3}}+D_{\text{rot1}}+D_{\text{rot2}}+D_{\text{rot3}}; 
\]
where each component of the $D$ matrix, $D_{k,j}$, are used in the torque calculation as shown in 3.2.12. The components of the $C$ matrix are generated using the function Christoffel.m which is:

```matlab
function c=Christoffel(D,q,i,j,k)
c=(1/2)*(diff(D(k,j),q(i))+diff(D(k,i),q(j))-diff(D(i,j),q(k)));
```

Which takes the $D$ matrix and a vector of three $q$ values, and calculates the Christoffel symbols for any $i$, $j$, or $k$. The symbols are then used as following:

```matlab
%Compute Christoffel Symbols:
c11=Christoffel(D,q,1,1,k);
c12=Christoffel(D,q,1,2,k); '%'=c21
%c22=Christoffel(D,q,2,2,k);
c13=Christoffel(D,q,1,3,k); '%'=c31
%c23=Christoffel(D,q,3,2,k); '%'=c32
%c33=Christoffel(D,q,3,3,k);
```

Here the Christoffel symbols are calculated for a single $k$ value, since in this code $k$ denotes the joint number that the function is calculating torque for. The code is modeled after the equations shown in in 3.2.14 and 3.2.15.

\[
c_{ijk} = \frac{1}{2} \left\{ \frac{\partial d_{kj}}{\partial q_i} + \frac{\partial d_{ki}}{\partial q_j} - \frac{\partial d_{ij}}{\partial q_k} \right\}
\]  

(3.2.14)

Where the $(k,j)$th element of the $C$ matrix is:

\[
c_{kj} = \sum_{i=1}^{n} c_{ijk}(q) \dot{q}_{i}
\]

(3.2.15)

The $G$ matrix components, which is actually a column vector of the potential energies for each link, is computed:

```matlab
g=diff(P,q(k));
```

Which is computed for each joint $k$. As is the torque, as shown below:

```
tau=D(k,1)*q1ddot+D(k,2)*q2ddot+D(k,3)*q3ddot ... 
  + c11*q1dot^2+2*c12*q1dot*q2dot+c22*q2dot^2 ... 
  + 2*c13*q1dot*q3dot+2*c23*q2dot*q3dot+c33*q3dot^2+g;
```

This result of this calculation was also used in the Matlab script placed in the SimMechanics simulation in the same way that the torque calculation from Mathematica was, however, once again, it was to no avail.
3.3 Using User Defined Blocks

In SimMechanics, one can create mathematical functions using series of math function blocks, however, with lengthy calculations it seemed best to create user defined blocks. This is the case with the Euler-Lagrange equations. In Matlab, each can be set out as a Matlab function, in which joint position, velocity, and acceleration are input and the torque the output. To place such a function, one places a “user defined block” from the Simulink function library into the model. After double clicking the block, the user is presented with a box that looks identical to that of the standard Matlab function. After coding out the Lagrangian from either method described above, click okay. The result is then a user defined block with a number of inputs equivalent to the number of inputs defined by the function, and a single output.

3.4 Chapter 3 Summary

In this chapter we discussed how to assemble the matrix components for the Euler-Lagrange equations. These included: Jacobians (revolute and linear) for each links center of mass, rotation matrices, the initial matrices, and potential energy. These parts were then assembled into a kinetic and potential energy matrix. This chapter then outlined two different methods for taking the necessary derivatives to calculate torque. Finally, we described how to incorporate the equation for torque into the SimMechanics model.
Chapter 4
Simulating the Leg

The overall simulation for the robot leg dynamics is broken down into three basic subsystems: trajectory generation, the dynamic model, and the simulation of the robot leg; shown in Figure 4.1. The trajectory generation function for this model is for a single step, but using a control loop, the model could easily be altered to make it adaptive on each iteration. It is currently constant in the sense that it moves through the same set points for each step, as though assuming flat terrain. An adaptive step would mean that the set points which define the step’s trajectory could be changed for each step based...
on what the robot may know about its environment through sensors.

Currently it takes constant user inputs for desired step location and step length, the trajectory generation function generates the necessary joint positions, velocities, and accelerations needed to achieve a smooth motion from point to point. This “subsystem” will be described in detail later in this chapter.

The dynamic model portion of the simulation is made up of several user-defined blocks. These blocks behave in the same manner as any other block in Simulink, but instead of setting various parameters, the blocks inputs and outputs are determined based on Matlab code placed inside the block. User defined blocks are the only option for integrating a mathematical model as high level as the dynamics for this project into a Simulink simulation. The mathematical model for calculating joint torques has already been described throughout this paper, thus, it is convenient to build SimMechanics model while determining the environment and kinematics.

![SimMechanics Visualization of the Leg](image)

**Figure 4.2: SimMechanics Visualization of the Leg**

A quick, simple way to verify if the body and joint blocks described earlier in this paper is to generate the visualization of the model in SimMechanics as shown in 4.2. By going into the machine environment and configuration
parameters, one can select an option to run a visualization of the model while it is running. This will produce a pop-up window of the model moving in real time. This will help to verify that the coordinate frames for the links. Also, we use it to determine if the movement of the leg over time follows the trajectory we expect to see. Meaning that we will see the foot lifting up, setting back down, and moving across the ground rather than moving irrationally; for example lifting the leg further up rather than stepping down.

4.1 Leg Trajectory

One challenge in robot control is setting trajectories for robotic manipulators. This effort will be entirely configuration dependent, meaning the design of a trajectory depends both on the structure of the manipulator (our robotic leg) and the intent of the movement. The movement must not go outside of the range of the manipulator, velocities and accelerations must be smooth to avoid sudden jerks or physically impossible jumps, and for a leg the step must not upset the balance of the robot.

For the sake of this simulation, a trajectory had to be chosen to make a single robotic step. The trajectory generated for this simulation is one that would maintain stable walking on a flat, level surface. The simulation is left open to be able to accept inputs from another source and thus be able to adjust changing step lengths and heights, so as to be able to step on or around obstacles. This other source would need to evaluate the surface that the robot needed to navigate, and then provide to the simulation a place on which to set the foot which was within the workspace of the leg and allowed the leg to provide support for the body. The point chosen for the simulation have been verified to fulfill these criteria.

4.1.1 Workspace

To design a trajectory for the robotic leg, first we need to ensure that desired locations for the foot or tip of the robotic leg, as well as a path between these locations, are within the workspace of the manipulator. The workspace is the area that the foot can reach given its geometry, joint limits, and degrees of freedom. We can visualize the workspace by using the forward kinematics for the leg and plotting in space the location of the leg tip at every possible joint configuration. Such a set of points is graphed using Matlab to show potential workspace for the tip of our robot leg (the foot) in four views shown in Figure 4.3.

Here the “top” view is parallel to the robot body, or perpendicular to the
Figure 4.3: Workspace for Robot Leg

plane on which the leg is mounted. The front view is also perpendicular to the frame on which the leg is mounted, and parallel to the first link when it is in the zero position. The fourth view, “right” is position in parallel with the plane on which the robot leg is mounted.

4.1.2 Set Points

The trajectory for this leg was generated by determining x, y, and z positions within Cartesian space in the same coordinate frame as the base of the leg as set points for the leg to reach during the step. An example is shown in Figure 4.4.

For a step, there are three points of particular interest. First, the anterior extreme position, or the farthest position in front of the robot. This is where the foot should be just before placing weight on the leg. Next is the posterior extreme position, the farthest position behind the robot, and where the foot should be when the robot takes weight off the leg. Third we will call the max height position, which represents the half-way point of when the robot is swinging the leg forward for the next step. These three points define three
Figure 4.4: A single set point for the robot step shown with Cartesian coordinates.

phases of leg movement: the stance, when the robot’s weight is on the leg; the lift, when the robot is moving the leg forward to the max height position, and the set; where the leg is approaching the anterior extreme position. These three phases were adapted from a step used by Maufroy in his study of posture in dynamic walking [9]. A diagram of the step is shown in Figure 4.5:

Figure 4.5: Diagram of Swing and Stance Phases in Each Step

In the diagram, each dot represents a joint in the robotic leg. The links and joints in green are the leg in an extreme anterior position, and those in blue illustrate the extreme posterior position. The dotted lines represent relatively how the foot would travel, where the purple line represents the “stance” in which the foot rests on the ground and pushes the robot forward,
and the “swing” phase in red in which the foot lifts up from the extreme posterior position and sets down again in the extreme anterior position.

4.1.3 Generating Smooth Trajectories

Now that these points are defined, we need to outline the leg’s position between these points. Note also that the three points we outline all represent a change in direction for the leg. Since inertia plays a significant role in robot dynamics, we wish to limit sudden changes in robot velocity. For this reason, our simulation then solves a quintic trajectory equation to get a set of joint positions, velocities, and accelerations for a number of time steps in between the points. It is called a quintic trajectory because in order to allow all three orders of motion (position, velocity, and acceleration) to begin and end at zero, one must create a path for position that is a fifth order polynomial.

This matrix is used to generate a series of positions that, rather than being linear over time allow for the velocity and acceleration to be zero at both the start and stopping points of the motion and reach a maximum in the middle of the motion, thus creating a smooth path and reducing error and strain on the manipulator assembly from sudden jerky movements. An example of a single dimensional quintic path over time is shown in Figure 4.6, where position is red, velocity in blue, and acceleration in purple.

![Figure 4.6: A Generic Quintic Trajectory showing Position, Velocity, and Acceleration over Time](image)

The algorithm for determining a quintic path uses the following equations [1]:

43
where $t_0$ and $t_f$ are the initial and final times over which the quintic trajectory will run, $p_0$ and $p_f$ are the initial and final positions, $v_0$ and $v_f$ the initial and final velocities, and $a_0$ and $a_f$ the initial and final accelerations. The $A$ column vector is made up of the quintic coefficients needed to achieve a solution of desired initial and final positions, velocities and accelerations over the given time span. An algorithm that solves for the $A$ coefficients based on the quintic matrix and desired initial and final positions, velocities, and accelerations was coded in Matlab for use in the simulation. Throughout the simulation, initial and final positions and time come from user defined constants, and the initial and final velocities and accelerations are set as constants. The solution for our position trajectory is shown in equation

$$p(t) = a_0 + a_1 t + a_2 t^2 + a_3 t^3 + a_4 t^4 + a_5 t^5$$

(4.1.2)

Where $a_i$ are the values of the column vector $A$. The equations for velocity and acceleration are simply derivatives of this equation. The quintic path can be used in one of two ways. In the first method generates the desired set points and finds the inverse kinematics of each. The quintic path is then used to determine the range of joint positions as the robot leg travels from one configuration to the next. This generates a smooth curve for position, velocity, and acceleration of each joint. This continuous curve is convenient to use in modeling software such as Simulink because a continuous signal can easily be manipulated. The drawback of this approach however is that the leg will move in a manner that is most convenient for the joints, and the manipulator configuration in between the set points is left uncontrolled. Using the second method differs only in one determines primary set points, and then a full curve of points in each global $x$, $y$, and $z$ dimensions between the primary ones. This differs in the sense that the curves of points are generated before the inverse kinematics are calculated. Using this method, one can deterministically decide the location of every point the manipulator tip reaches throughout its motion. We use the quintic trajectory to generate the curve of points between the primary set points. After these points are determined, the inverse kinematics are calculated for each. The major drawback here is that the final trajectory of joint position will not be a smooth
quintic one, but rather a step function made up of points, which is difficult to process in Simulink. Both methods were used in this research to show how each would work.

4.1.4 Trajectory Generation Methods in the Simulation

![Figure 4.7: The Trajectory Subsystem](image)

On the far left of Figure 4.7 we see some “constant” blocks, these provide the user defined parameters for step size and time. Both trajectory generation systems use these. Since the gait in our simulation is not adaptive, this data is set as constant. However, to develop an adaptive gait, one could create a simulation to provide different locations and times to begin and end the step, and the simulation would be able to generate the quintic trajectory between the user desired points provided said points were still within the leg.
workspace. The next step for this method is to generate the set of points between these constant primary set points. This is done by four quintic path blocks, one for each phase and stage over which the step moves. The path in the global x direction (along the body of the robot) is broken up into two stages, the swing and stance.

The swing is for stepping forward, and the stance for when the robots weight is supported by the leg. The z direction is also broken up into two stages, these are both for the robot’s step, one for lifting the leg and another for setting it down. During the stance phase z remains constant. Movement in the y direction for the entire step is constant. When any of the variables remain constant, they do not need to be put through the quintic trajectory block, but rather are routed directly to the inverse kinematics for each step as shown in Figure 4.7.

These are user defined blocks coded to solve for the quintic trajectory solution; the $A$ matrix of polynomials which define the quintic path. The $A$ matrix is then passed on to another user defined block which generates the quintic position. This block is unique in that it also contains an “if-then” condition that decides which $A$ matrix to use to generate the position depending on what stage in the the step the simulation is in.

Finally, the positions go through the inverse kinematics to generate the joint positions necessary to reach the Cartesian set points given in terms of the coordinate frame on the base of the leg. The set of joint positions and pass on through two more blocks to generate its two derivatives, velocity, and acceleration. These three, for each joint, can then be passed on to the rest of the simulation. The process of taking the derivatives for this function is discussed later in this section. The resulting trajectory for a single step in joint space is shown in the graphs with position in red, velocity in blue, and acceleration in green in Figure 4.8.

The second method works very differently in the simulation, starting with the given constant primary set points they are each passed directly onto the inverse kinematics. Recall that for each stage of the step, the set points will either serve as the start or end point. For this reason, the three set points, converted into joint space, are fed into Simulink switch blocks. This can be seen in Figure 4.9.

A user defined function based on system time in the trajectory subsystem controls the switch, telling it to select whichever set of joint positions is appropriate depending on the stage in the simulation. The selected joint positions are sent to three quintic trajectory blocks, which solve for the A matrix for each joint. The subsystem then returns the joint position, velocity, and acceleration as determined by the A matrix by the method outlined earlier in this section. In this manner, the position, velocity, and acceleration
are all smooth polynomial functions. As seen in the graphs in Figure 4.10.

4.1.5 Stability Polygon

This set of chosen joint positions used for the simulation were tested using a Matlab script which generates a stick model, as shown in Figure 4.11: This stick model was generated with a Matlab script that draws four legs attached at the four corner ports. This script can also be used for any number of legs located at any of the specified ports on the robot body. Note the triangle on the “floor“ (a flat plane drawn between the three ”feet” on the ground) which represents the stability polygon. This merely draws a shape on the floor between the remaining feet that are not actively taking a step. There is also a blue circle in this diagram.

This circle represents the whole robot’s center of mass projected on the floor. This stick model is a convenient way to determine if a joint configuration is possible and what it will look like. More importantly, it determines if a set of joint positions for each leg combined does not throw the robots center of mass outside the stability polygon, thus causing the robot to fall over.
4.1.6 Checking for Singularities and Manipulability

Another potential issue to check for is areas in the trajectory where the legs motion may be limited by its configuration. This can be caused by a singularity or particularly small manipulability in the configuration that the leg was desired to be in at the time the error occurred. A simple script was written using the Jacobian described in Section 3.1 to determine the rank of the Jacobian and the manipulability for each configuration in the desired trajectory. This is done to check if there is a singularity at the time when the data shows an asymptote or when the solver returns an error.

Jacobian matrix will help to identify singularities and calculate manipulability of the leg in certain configurations. Singularities occur when the rank of the Jacobian matrix is less than six, meaning when it has less than six linearly independent columns. Physically, this means that the leg is not able to reach certain velocities in the configuration defined as a singularity. Singularities can be found by calculating:

\[
\text{rank} J(q) < 6 \quad (4.1.3)
\]

for every joint configuration. In the same way, the manipulability (ability for the leg to move within global space) for each configuration is calculated:

\[
\mu = \sqrt{\det(J^T(q)J(q))} \quad (4.1.4)
\]

A singularity is indicated by a rank of less than three, where the leg may not be able to attain certain velocities. Low manipulability would indicate that the movement of the leg in that configuration is limited. The data can be seen graphically in Figures 4.12 and 4.13.
4.1.7 Derivatives of Time Step Functions in Simulink

As described earlier in this section, a major drawback for the first quintic trajectory approach, is that it does not generate a smooth path of joint positions. Although joint positions, velocities, and accelerations follow a curve, the joint positions must be generated separately for each time step. This presents an issue in Simulink, where it is difficult to manipulate such curves, specifically, to take their derivatives.

This issue emerged in the research when the joint positions for the model were generated once per time step, generating a step-curve, as shown in Figure 4.14. This means that the derivative (velocity) only appears at the end of each step cycle, rather than being a continuous curve. This presents itself graphically in Figure 4.15.

This cannot be accurately handled by Simulink’s derivative block, or by Simulink’s smoothing and transfer function blocks. When this is attempted, SimMechanics generates some kind of error and the solver crashes. Instead, one has to make a customized derivative block. First, use a memory block to hold back the signal by one time step, one can then subtract the value from the prior time step from the current value, and divide by the time step (this...
will be the same time step used to produce the position function in the first place). This will produce a derivative that looks like that in Figure 4.15.

Adding a hold block after the division will prevent the signal from dropping to zero between time steps and produce accurate step graphs. Use the same derivation block a second time to get acceleration. The derivative subsystem for calculating the joint velocity and acceleration is shown in Figure 4.16.

Joint position, velocity, and acceleration data produced in this manner for Joint 1 is shown in Figure 4.17.
Figure 4.12: Rank of Jacobian over Time based on Configuration

Figure 4.13: Manipulability ($\mu$) over Time based on Configuration

Figure 4.14: Section of joint position over time graph, zoomed in to show step detail

Figure 4.15: Section of joint velocity over time graph, zoomed in to show spikes

Figure 4.16: SimMechanics subsystem used to take derivative of non-continuous trajectory
Figure 4.17: Joint 1 Position, Velocity, and Acceleration Data, with Derivatives Calculated using Described Subsystem
4.2 User Defined Blocks

The math for the dynamic model was implemented in the simulation as three user-defined blocks, one to calculate torque for each joint. Each block requires inputs of joint position, velocity, and acceleration for all three joints. The blocks contain the dynamic equations produced by the process outlined in the subsection on dynamics earlier in the paper. They are used to send desired torque to the joint actuator block in the SimMechanics model as described in Section 2.3 and shown in Figure 2.8.

The Jacobian is also incorporated into our SimMechanics model using a user-defined block. The user can input a force vector using global X, Y, and Z coordinates to be applied on the manipulator end. The Jacobian block can then calculate a set of three torques that would be needed to counteract that force. These torques are then added to the torques calculated by the dynamic model before being applied to the joint actuators.

For both user defined blocks, there are variables within the Matlab scripts that can be changed. As mentioned before, this model was designed such that the user could change the parameters to suit a similar robot with slightly different physical features. For example, one can change the dimensions and mass of the links without having to derive the full dynamic model all over again. This must be done by changing the values in two places. First, that parameter must be changed in the SimMechanics block which defines that link, and recalculate the moment of inertia in the body block to reflect the change in physical properties. Then we need to change the value in the dynamic model to match. This must be done by opening the user defined blocks, and changing that variable in the beginning of the Matlab code. These variables should be sufficiently labeled and commented to enable the user to find the desired variable to change and change it. The value of the user defined blocks is in part that anyone who understands Matlab can easily understand the blocks and alter them as desired.

4.3 Verifying the Simulink Model

The full Simulink model is comprised of the following sub-systems: trajectory generation, Euler-Lagrange dynamic equations, the Jacobian, and the SimMechanics physical model. This can be seen in Figure 4.18.

Here we can see all the subsystems together in the same model. First the trajectory generation on the far left, which sends out all three joint positions, velocities, and accelerations to each of the three user-defined Lagrangian blocks. The three Lagrangian blocks together make up the mathematical
dynamic model. It is broken up such that each Lagrangian block contains the equation to the torque for one joint. The blocks then send the desired torque signal into the joint actuator blocks in the SimMechanics physical model. These feed into the three joints that drive the simulated motion of the links. The physical model terminates with three joint sensors sending position data out for feedback. While these blocks have many options, we choose to read joint position since it can be checked against the desired positions we began the model with.

Figure 4.19: Desired vs. Real Joint Positions

This feedback is the information taken from the model to provide information to the user. During development, we made use of the Simulink scope block to produce real time graphs of various signals. For the final model,
and to see trends in the graph over the full run-time, it is best to save data to file. This sends the data to a timeseries that can be displayed later using Matlab plot commands.

The simulation seeks to verify the mathematical model by testing if the torques sent into the joint actuators actually move the joints into the desired configuration. The goal of the Euler-Lagrange equations is to calculate the torques necessary to achieve the desired configuration. If the math for the torque generation is correct, the torques generated should move the leg along the desired trajectory over time. If the dynamic model is correct, and the simulation parameters are set up correctly the joint position from the joint sensors and desired joint positions should match. In Figure 4.19 and 4.20, the two can be seen with desired and blue and actual position in red for each joint, using both the non-continuous and continuous trajectories.

As the reader can see, these two graphs are entirely different. The desired position in blue is a continuous curve which one would expect, however, the actual results in red are not remotely close to the desired trajectory, in fact, they almost appear to be noise. This is not actually noise, but rather numerical error. The graphs demonstrate that instead of smoothly moving from one configuration to the next, the joints on the leg rotate around and around erratically. Troubleshooting performed and possible explanations for this error will be discussed in the following section.
Chapter 5

Conclusions and Future Work

5.1 Results

5.1.1 Simulation Result Analysis

In Section 4.3 in figures 4.19 and 4.20, desired position is compared to model position. The desired position is correct, however the actual position looks more like noise. This is the case when specified torque is incorrect. The combined effects of the torque, gravity, and inertia leave us with a set of joints that move wildly. We did not limit the range of movement for the model since the expected positions were well within this range, as verified by the stick model. However, when torque is incorrect this allows the model to swing the links in full rotations about their joints, leading to the spikes seen in the resulting position graph.

5.2 Inverting the Model

As mentioned earlier in this paper, blocks in SimMechanics can be configured a number of different ways. Using the same series of body and joint blocks in our system model, one can invert its purpose. Rather than generating joint positions as a result of torque inputs, we can generate torques as a result of desired position, velocity, and acceleration.

Our first simulation models how the robotic leg is actually constructed. It utilizes actuator blocks that act as simulated motors that move the connected joints in response to a specified torque input. The model then reads position feedback from sensors attached to these same joints. However, in SimMechanics, one can also actuate joints using desired position, velocity, and acceleration. The joint sensor block can also be configured to provide
the user with the computed torque values to achieve the position that the attached joint is in. This model, shown in Figure 5.1, a model does exactly the inverse of our first model. We set up the inverted model to show the necessary torques to achieve the desired position, and compared them to the torques produced by the Euler-Lagrange calculations to verify. With this inverted model, we can use the same trajectory blocks used in the forward model to ensure that the two are working with identical driving data. The results are shown in Figures 5.2 and 5.3 using both the trajectory generation subsystems described earlier in this paper.

Looking at the two, they have similar curvature however the graphical peaks differ in amplitude and there are some asymptotes in the calculated torque that do not appear in the calculated torque. The differences in input torque vs the torque that the model suggests should be used is enough to offset the results. So much so that using our input torque, the desired position
Figure 5.3: Continuous Calculated Torque (green) vs. Simulation Torque (blue)

is a line and the actual position looks like random noise, as seen in Figures 4.19 and 4.20. In contrast, when the torques suggested by SimMechanics were fed back into our original model in place of the mathematical model we developed, the simulation produces graphs without noise that are identical to the original desired results.

5.2.1 Matlab

When running the SimMechanics simulation for the leg, one could often see that the solver was running into a common error which read something to the effect of “the model is unable to meet tolerances without reducing step size below the smallest value allowed at time t”. Reading Matlab documentation reveals that this may be because the model is too “stiff”[17].

While SimMechanics does allow the user to specify a particular solver, we had left the program to pick it the solver best suited the mathematical model at hand. This was the differential equation solver ode45. The great downside of SimMechanics based simulation is that too often errors came up outside of the user designed model and were impossible to see in the high level block format laid out by this particular tool. Therefore, a simplified mathematical model using ode45 to solve for actual joint positions and velocities based on input torque was developed. This simplified model still contained all the same parts as the SimMechanics model used, including the same step path and trajectories and the same equations for generating torque. What was added was an ode45 solver that returned position and velocity for each joint based on setting position and velocity initial conditions. Ode45 was used to solve the function:

\[ A = D^{-1} \times (\tau - C(q)\dot{q} - G) \]  

(5.2.1)

Where \( A \) is the column vector of joint accelerations, \( D \) is the kinetic energy matrix described, \( C \) is the Coriolis-Coupling matrix described above, \( \dot{q} \) is
a column vector of joint velocities and $G$ is a column vector of potential energy on each link. Each matrix is configuration dependent. The solver was run in a loop in which initial position and velocity was set as the desired values based on the desired trajectory, and initial torque set as desired torque generated by the Euler Lagrange equations. The full set of Matlab scripts and functions used for this work is contained in Appendix C.

Running this mathematical model proved to have very different results from the SimMechanics simulation. Here the model returned a curve that was at least recognizable as the desired position and velocity. The exception of course are the “spikes” of simulation noise that occurred. Figure 5.4 shows the original desired position, velocity, and acceleration which has been kept consistent since its development for the SimMechanics model. Next, Figure 5.5 shows the results returned by Matlab’s ode45 solver.

The spikes seen in the graphs represent some inaccuracies and system noise in this area. Note that the time scaling on these graphs goes to 5000, this is because ode45 solved for 50 time steps each time it was run, and the loop was run 100 times. If one looks back to the manipulability graph over time (Figure 4.12), one can see that the spikes occur mostly in the first area.
where manipulability was low, and then are propagated further with each joint. This is an accurate representation of what might happen in the real world, thus demonstrating the need to either change the trajectory to avoid this area, or add some control functions (such as PID or PD control) in future work.

5.3 Future Work

5.3.1 PID or PD Control

In the simulation for this manipulator system, the Euler-Lagrange equations determine the torque for each joint based on desired joint position, velocity, and acceleration. The model also makes use of sensors that keep track of actual joint positions. A standard PID loop could be used to minimize the difference between the desired and actual position of the actuator by feeding back into the block providing desired positions to calculate the necessary torque to move the joint in the next time step. A flow chart of this loop is
shown in Figure 5.6:

![Flow Chart of Control Loop Abstracted from Simulation](image)

Since both joint velocity and acceleration are derived from the set of desired positions, those too will reflect the change in desired position, though one and two time steps later, respectively. A full simulation of the physical system, the equations of motion, and the PID loop could be implemented in a single SimMechanics simulation.

5.3.2 Using the Leg as a Manipulator

The calculations developed in this project, specifically the joint calculations and control, can be used for any three joint manipulator in the future simply by changing the physical parameters, such as mass, dimensions, and inertia matrices. In fact, the leg itself developed in this project could be used as a manipulators with little adjustment. Previously in this paper, we discussed the advantages to placing a three degree of freedom force sensor on the end of the foot. With such a sensor, the robot could use two of its manipulators, in parallel, to pick up a simple object. Chen’s MiniQuad can do a similar task with its two front legs with the remaining four on the ground[7] as shown in Figure 5.7.

The force sensor on the foot/hand would provide feedback to inform the
robot control system when it was in contact with the object and whether or not it had a firm grasp on the object.

5.3.3 Step Planning

If the robot were to have the ability to sense its environment, as most robots do, it could have an adaptive gait. The adaptive gait could work one of two ways. One would be if it could "see" the environment around it, using a camera or range finder, and plan steps accordingly. The other would be to use a reactive system, such as that outlined by Espenschied inspired by insects that sense the ground with their foot [15]. This strategy would utilize force sensors on the end of the robot's foot, as suggested earlier in this paper. In the current simulation, we use a constant position and time to determine where the step begins and ends. The simulation developed for this research could be adapted to suit an adaptive gait. Instead of running the trajectory planning algorithm on constant set points, it could be run on set point generated by another algorithm. The design of this algorithm is outside the scope of this research, and would depend on the sensing method used.

5.3.4 Central Processing

As it stands, this paper addresses a single leg operating alone on a robot that is statically stable. In the future, a central processor for the robot should be developed. Such a processor would be used to distribute necessary data.
to the legs would be necessary to set gait patterns and predict where the legs should be placed. The amount of force that would be resting on each leg based on the total number of legs attached to the robot body and the full robot configuration. In addition, a central processor would be able to handle and utilize inputs from sensors to better control the robot by providing more accurate position data, predict oncoming external forces, and alter step trajectories to avoid objects or adapt to uneven terrain.

5.3.5 Universal Modules

The intended goal for this project is to create a fully reconfigurable robot. While this paper only covers control of legs for the robot, in the future other attachments such as sensors and manipulators can be developed. We have talked much in this paper on the idea of modules that should be developed in the future to suit this project. Specifically sensors. It is important to mention that any attachment for this robot must fulfill the following criteria: One, its processing must be handled on the module to the furthest extent possible. This robot is designed so that the central processor carries as little load as possible. The modules should contain more processing than the central body processor itself. This is done to ensure that when that module is removed, the remaining robot is as free of that load (physically and computationally) as possible. Two, the modules must attach to the robot body the same way. This means that physically, the bolt patterns on the robot body and the module must match. Computationally, this means that all communication between body and modules happens over the same port and with the same communication protocol.


Appendix A

Appendix A: Mathematica Script
In[1]:= (CALCULATING THE LAGRANGIAN) (*Set masses and lengths as constants*)

SetAttributes[m1, Constant];
SetAttributes[m2, Constant];
SetAttributes[m3, Constant];
SetAttributes[a1, Constant];
SetAttributes[a2, Constant];
SetAttributes[a3, Constant];
SetAttributes[com1, Constant];
SetAttributes[com2, Constant];
SetAttributes[com3, Constant];
SetAttributes[lh, Constant];
SetAttributes[lw, Constant];

(*Jv1, Jv2, and Jv3 are the three upper Jacobians for centers of links. Jw1, Jv2, and Jv3 are the three lower Jacobian.*)

Jv1 = 
\[
\begin{pmatrix}
-\text{com1} \cdot \text{Sin}[q1[t]] & 0 & 0 \\
\text{com1} \cdot \text{Cos}[q1[t]] & 0 & 0 \\
0 & 0 & 0 \\
\end{pmatrix}
\]

Jv2 = 
\[
\begin{pmatrix}
-\text{a1} \cdot \text{Sin}[q1[t]] - \text{com2} \cdot \text{Sin}[q1[t]] \cdot \text{Cos}[q2[t]] - \text{Cos}[q1[t]] \cdot \text{com2} \cdot \text{Sin}[q2[t]] & 0 \\
\text{a1} \cdot \text{Cos}[q1[t]] + \text{com2} \cdot \text{Cos}[q1[t]] \cdot \text{Cos}[q2[t]] - \text{Sin}[q1[t]] \cdot \text{com2} \cdot \text{Sin}[q2[t]] & 0 \\
0 & \text{com2} \cdot \text{Cos}[q2[t]] & 0 \\
\end{pmatrix}
\]

Jv3 = 
\[
\begin{pmatrix}
-\text{Sin}[q1[t]] \cdot (\text{a1} + \text{a2} \cdot \text{Cos}[q2[t]] - \text{com3} \cdot \text{Cos}[q2[t] + q3[t]]) - \text{Cos}[q1[t]] \cdot (\text{a2} \cdot \text{Sin}[q2[t]] + \\
\text{Cos}[q1[t]] \cdot (\text{a1} + \text{a2} \cdot \text{Cos}[q2[t]] + \text{com3} \cdot \text{Cos}[q2[t] + q3[t]]) - \text{Sin}[q1[t]] \cdot (\text{a2} \cdot \text{Sin}[q2[t]] + \\
0 & \text{a2} \cdot \text{Cos}[q2[t]] + \text{com3} \cdot \text{C}
\end{pmatrix}
\]

Jw1 = 
\[
\begin{pmatrix}
0 & 0 & 0 \\
0 & 0 & 0 \\
1 & 0 & 0 \\
\end{pmatrix}
\]

Jw2 = 
\[
\begin{pmatrix}
0 & \text{Sin}[q1[t]] & 0 \\
0 & -\text{Cos}[q1[t]] & 0 \\
1 & 0 & 0 \\
\end{pmatrix}
\]

Jw3 = 
\[
\begin{pmatrix}
0 & \text{Sin}[q1[t]] & \text{Sin}[q1[t]] \\
0 & -\text{Cos}[q1[t]] & -\text{Cos}[q1[t]] \\
1 & 0 & 0 \\
\end{pmatrix}
\]

(*R1, R2, and R3 are the rotation matrices (first 3 rows and columns of the transformation matrix) From 0 to 1, 0 to 2, and 0 to 3 respectively.*)
$$R_1 = \begin{pmatrix} 
\cos[q_1[t]] & 0 & \sin[q_1[t]] \\
\sin[q_1[t]] & 0 & -\cos[q_1[t]] \\
0 & 1 & 0 
\end{pmatrix};$$

$$R_2 = \begin{pmatrix} 
\cos[q_1[t]] \cos[q_2[t]] - \cos[q_1[t]] \sin[q_2[t]] & \sin[q_1[t]] \\
\sin[q_1[t]] \cos[q_2[t]] - \sin[q_1[t]] \sin[q_2[t]] & -\cos[q_1[t]] \\
\sin[q_2[t]] & \cos[q_2[t]] & 0 
\end{pmatrix};$$

$$R_3 = \begin{pmatrix} 
\cos[q_1[t]] \cos[q_2[t] + q_3[t]] - \cos[q_1[t]] \sin[q_3[t] + q_2[t]] & \sin[q_1[t]] \\
\sin[q_1[t]] \cos[q_2[t] + q_3[t]] - \sin[q_1[t]] \sin[q_3[t] + q_2[t]] & -\cos[q_1[t]] \\
\sin[q_2[t] + q_3[t]] & \cos[q_2[t] + q_3[t]] & 0 
\end{pmatrix};$$

(*I1, I2, and I3 are the inertia matrices of each of the links respectively. These are all very rough estimates assuming each link is a rectangle with the origin at the pin. lw is the link width and lh is the link height.*)

$$I_1 = \begin{pmatrix} 
(lw^2 + lh^2) * m_1 / 12 & 0 & 0 \\
0 & (a_1^2 + lw^2) * m_1 / 12 & 0 \\
0 & 0 & (a_1^2 + lw^2) * m_1 / 12 
\end{pmatrix};$$

$$I_2 = \begin{pmatrix} 
(lw^2 + lh^2) * m_2 / 12 & 0 & 0 \\
0 & (lh^2 + a_2^2) * m_2 / 12 & 0 \\
0 & 0 & (a_2^2 + lh^2) * m_2 / 12 
\end{pmatrix};$$

$$I_3 = \begin{pmatrix} 
(lw^2 + lh^2) * m_3 / 12 & 0 & 0 \\
0 & (lh^2 + a_3^2) * m_3 / 12 & 0 \\
0 & 0 & (a_3^2 + lh^2) * m_3 / 12 
\end{pmatrix};$$

(*The kinetic energies for each link*)

$$D_{k1} = m_1 * (Transpose[Jv1].Jv1) + Transpose[Jw1].R_1.I_1.Transpose[R1].Jw1;$$

$$D_{k2} = m_2 * (Transpose[Jv2].Jv2) + Transpose[Jw2].R_2.I_2.Transpose[R2].Jw2;$$

$$D_{k3} = m_3 * (Transpose[Jv3].Jv3) + Transpose[Jw3].R_3.I_3.Transpose[R3].Jw3;$$

$$D_{tot} = D_{k1} + D_{k2} + D_{k3};$$

(*The potential energies for each link and g is gravity vector, multiplied by the coordinates of the COM of each link*)

$$g = \begin{pmatrix} 
0 \\
0 \\
-9.81 
\end{pmatrix};$$

$$P_1 = Transpose[g]. \begin{pmatrix} 
\cos_1 \cos[q_1[t]] \\
\cos_1 \sin[q_1[t]] \\
0 
\end{pmatrix} * m_1;$$

$$P_2 = Transpose[g]. \begin{pmatrix} 
\cos[q_1[t]] * (a_1 + a_2 * \cos[q_2[t]] + \cos_3 * \cos[q_2[t] + q_3[t]]) \\
\sin[q_1[t]] * (a_1 + a_2 * \cos[q_2[t]] + \cos_3 * \cos[q_2[t] + q_3[t]]) \\
\cos_2 * \sin[q_2[t]] 
\end{pmatrix} * m_2;$$

$$P_3 = Transpose[g]. \begin{pmatrix} 
\cos[q_1[t]] * (a_1 + a_2 * \cos[q_2[t]] + \cos_3 * \cos[q_2[t] + q_3[t]]) \\
\sin[q_1[t]] * (a_1 + a_2 * \cos[q_2[t]] + \cos_3 * \cos[q_2[t] + q_3[t]]) \\
a_2 * \sin[q_2[t]] + \cos_3 * \sin[q_2[t] + q_3[t]] 
\end{pmatrix} * m_3;$$
(*Lagrangian parts, LK Kinetic, LP Potential*)

qdot = 

\[
\begin{align*}
    \frac{\partial q_1}{\partial t} \\
    \frac{\partial q_2}{\partial t} \\
    \frac{\partial q_3}{\partial t}
\end{align*}
\]

\[
L = \text{FullSimplify}[\frac{1}{2} \text{Transpose} \{q\dot{}}.\dot{q} - (P_1 + P_2 + P_3)]; (*7.53*)
\]

(*Partial differentials of Lagrangian with respect to qdot*)

Lqdot1 = D[L, Dt[q1[t], t]]; (*7.55*)
Lqdot2 = D[L, Dt[q2[t], t]];
Lqdot3 = D[L, Dt[q3[t], t]];

(*Derivatives (d/dt*df/dqdot)*)

Lqddot1 = Dt[Lqdot1, t]; (*7.56*)
Lqddot2 = Dt[Lqdot2, t];
Lqddot3 = Dt[Lqdot3, t];

(*Partial Derivatives of Lagrangian with respect to q*)

Lq1 = D[L, q1[t]];
Lq2 = D[L, q2[t]];
Lq3 = D[L, q3[t]];

(*Calculate Torques \( t = d/dt* L/dqdot - L/q \) (*7.42, 7.62*)

tau1 = Chop[FullSimplify[Lqddot1 - Lq1]]
tau2 = Chop[FullSimplify[Lqddot2 - Lq2]]
tau3 = Chop[FullSimplify[Lqddot3 - Lq3]]

\[
\text{Out}[43]= \{ -0.5 \text{a2 com3 m3 Sin}[2\ q1[t] -q3[t]] - 0.5 \text{a2 com3 m3 Sin}[2\ q1[t] -q2[t] -q3[t]] - 1. \text{a1 com3 m3 Sin}[2\ q1[t] -q3[t]] - 0.5 \text{a2 com3 m3 Sin}[2\ q1[t] +q2[t] +q3[t]] + q1'[t] - \text{q1}'[t] \}\left\{ -0.5 \text{a2 com3 m3 Sin}[2\ q1[t] -q3[t]] + 0.5 \text{a2 com3 m3 Sin}[2\ q1[t] -2\ q2[t] -q3[t]] + 0.5 \text{com3}^2 \text{m3 Sin}[2\ (q1[t] +q2[t] +q3[t])] + 0.5 \text{a2 com3 m3 Sin}[2\ q1[t] +q3[t]] - 0.5 \text{com3}^2 \text{m3 Sin}[2\ (q1[t] +q2[t] +q3[t])] - 0.5 \text{a2 com3 m3 Sin}[2\ (q1[t] +q2[t] +q3[t])] + q2'[t] + 0.5 \text{com3}^2 \text{m3 Sin}[2\ (q1[t] -q2[t] -q3[t])] - 0.5 \text{Sin}[2\ (q1[t] +q2[t] +q3[t])] + q3'[t] + 0.5 \text{com3}^2 \text{m3 Sin}[2\ (q1[t] -q2[t] -q3[t])]\right\} + q1'[t] - 1. \text{q2}'[t] - 1. \text{q3}'[t] - 0.5 \text{com3}^2 \text{m3 Sin}[2\ (q1[t] +q2[t] +q3[t])]\right\}  
\text{q1}'[t] + q2'[t] + q3'[t]\)} + 
{0.5 \text{com3}^2 \text{m3 Sin}[2\ (q1[t] -q2[t] -q3[t])]\right\} (q1'[t] - 1. \text{q2}'[t] - 1. \text{q3}'[t]) + 
\text{a2 com3 m3 Sin}[2\ q1[t] -2\ q2[t] -q3[t]] (0.5 q1'[t] - 0.5 q2'[t] - 0.25 q3'[t]) + 
\text{a2 com3 m3 Sin}[2\ q1[t] - q3[t]] (0.5 q1'[t] + 0.25 q3'[t]) + 
0.25 \text{a2 com3 m3 Sin}[2\ q1[t] + q3[t]] (2 q1'[t] + q3'[t]) - 
0.5 \text{com3}^2 \text{m3 Sin}[2\ (q1[t] +q2[t] +q3[t])]\right\} (q1'[t] + q2'[t] + q3'[t]) - 
0.25 \text{a2 com3 m3 Sin}[2\ (q1[t] +q2[t] +q3[t])] (2 q1'[t] + q2'[t] + q3'[t]) + 
2\ q1'[t] a1 (-1. \text{com2} m2 - 1. \text{a2 m3 Sin}[q2[t]] q2'[t] + 
\text{a2 com3 m3 Sin}[2\ q1[t] - q3[t]] (-0.5 q1'[t] + 0.25 q3'[t]) + 
\text{a2 com3 m3 Sin}[2\ q1[t] - 2\ q2[t] - q3[t]] (-0.5 q1'[t] + 0.5 q2'[t] + 0.25 q3'[t]) + 
\text{a1 com3 m3 Sin}[2\ q1[t] - q2[t] - q3[t]] (-1. q1'[t] + 0.5 q2'[t] + 0.5 q3'[t]) - 
0.25 \text{a2 com3 m3 Sin}[2\ q1[t] + q3[t]] (2 q1'[t] + q3'[t]) - 
0.0416667 a3^2 \text{m3 Sin}[2\ (q2[t] +q3[t])] (q2'[t] + q3'[t]) - 
0.5 \text{com3}^3 \text{m3 Sin}[2\ (q2[t] + q3[t])]\right\} (q2'[t] + q3'[t]) + 
\text{a1 com3 m3 Sin}[2\ q1[t] + q2[t] + q3[t]] (2 q1'[t] + q2'[t] + q3'[t]) - 
0.25 \text{a2 com3 m3 Sin}[2\ (q1[t] +q2[t] + q3[t])] (2 q1'[t] + q2'[t] + q3'[t]) + 
2 \left( 0.5 \text{com1}^3 m1^4 + 0.25 \text{com2}^2 m2^2 + 0.0416667 10^6 m^2 + a2^2 (-0.0416667 m^2 - 0.5 m) \right) \right\} \text{Sin}[2\ q2[t]] q2'[t] + 
10^6 \left( 0.0416667 m^3 + 0.0208333 m^2 + 0.0208333 m^3 \right) a2^2 + 0.0208333 m^2 + 0.25 m3 + 
a1^2 \left( 0.0416667 m^3 + 0.5 m^2 + 0.5 m^3 \right) + 0.0208333 a3^2 m^3 + 0.25 \text{com3}^2 m3 + 
0.0416667 10^6 m^3 + a1 \left( 1. \text{com2} m2 + 1. a2 m3 \text{Cos}[q2[t]] + 
0.25 a2 \text{com3} m3 \text{Cos}[2\ q1[t] - q3[t]] + 0.25 a2 \text{a2 com3 m3 Cos}[2\ q1[t] - 2\ q2[t] - q3[t]] + 
0.5 a1 \text{a1 com3 m3 Cos}[2\ q1[t] - q2[t] - q3[t]] + 0.25 a2 \text{a2 com3 m3 Cos}[2\ q1[t] + q3[t]] + 
0.0208333 a3^2 m3 \text{Cos}[2\ q2[t] + q3[t]] + 0.25 \text{com3}^2 m3 \text{Cos}[2\ (q2[t] + q3[t])] - 
0.0208333 10^6 m^3 \text{Cos}[2\ (q2[t] + q3[t])] + 0.5 a1 \text{a1 com3 m3 Cos}[2\ (q1[t] + q2[t] + q3[t])] + 
0.25 a2 \text{a2 com3 m3 Cos}[2\ (q1[t] + q2[t] + q3[t])]\right\} q1''[t] + 
0.25 a2 \text{a2 com3 m3 Cos}[2\ (q1[t] - q2[t] - q3[t])] - 0.25 a2 \text{a2 com3 m3 Cos}[2\ (q1[t] - 2\ q2[t] - q3[t])] - 
0.25 \text{com3}^2 m3 \text{Cos}[2\ (q1[t] - q2[t] - q3[t])] - 
0.25 a2 \text{a2 com3 m3 Cos}[2\ (q1[t] + q3[t])] + 0.25 \text{com3}^2 m3 \text{Cos}[2\ (q1[t] + q2[t] + q3[t])] + 
0.25 a2 \text{a2 com3 m3 Cos}[2\ (q1[t] + q2[t] + q3[t])] q2''[t] + 
\text{q3}''[t] + \text{q3}'''[t]\}}
\[ -9.81 \text{com2 m2} - 9.81 \text{a2 m3} \cos(q2[t]) - 9.81 \text{com3 m3} \cos(q2[t] + q3[t]) + \\
1. \text{a1 com2 m2 sin(q2[t])} + 1. \text{a1 com2 m3 sin(q2[t])} + 0.0416667 \text{a2 m2 sin(2 q2[t])} + \\
0.5 \text{com2^2 m2 sin(2 q2[t])} - 0.0416667 \text{w^2 m2 sin(2 q2[t])} + 0.5 \text{a2 m3 sin(2 q2[t])} - \\
0.5 a2 \text{com3 m3 sin(2 q1[t] - q3[t])} + 0.5 \text{com3^2 m3 sin(2 (q1[t] - q2[t] - q3[t]))} - \\
0.5 a1 \text{com3 m3 sin(2 q1[t] - q2[t] - q3[t])} + 0.5 \text{a2 com3 m3 sin(2 q1[t] + q3[t])} + \\
0.0416667 \text{a3^2 m3 sin(2 (q2[t] + q3[t]) + 0.5 \text{com3^2 m3 sin(2 (q2[t] + q3[t])))} - \\
0.0416667 \text{w^2 m3 sin(2 (q2[t] + q3[t]) - 0.5 \text{com3^2 m3 sin(2 (q1[t] + q2[t] + q3[t])} + \\
0.5 \text{a1 com3 m3 sin(2 q1[t] + q2[t] + q3[t])) q1'[t]^2} + \\
0.25 a2 \text{com3 m3 sin(2 q1[t] - q3[t])} - 0.25 a2 \text{com3 m3 sin(2 q1[t] - 2 q2[t] - q3[t])} + \\
0.25 a2 \text{com3 m3 sin(2 q1[t] + q3[t])} - 0.25 a2 \text{com3 m3 sin(2 (q1[t] + q2[t]) + q3[t])} \\
q1'[t] q3'[t] - 2. a2 \text{com3 m3 sin(q3[t]) q2'[t] q3'[t] - \\
1. a2 \text{com3 m3 sin(q3[t]) q3'[t]^2} + 0.25 a2 \text{com3 m3 cos(2 q1[t] - q3[t]) q1''[t]} - \\
0.25 a2 \text{com3 m3 cos(2 q1[t] - 2 q2[t] - q3[t]) q1''[t]} - \\
0.25 \text{com3^2 m3 cos(2 (q1[t] - q2[t] - q3[t]) + q3[t]) q1''[t]} - \\
0.25 a2 \text{com3 m3 cos(2 (q1[t] + q2[t]) + q3[t]) q1''[t]} + \\
0.25 a2 \text{com3 m3 cos(2 (q1[t] + q2[t]) + q3[t]) q1''[t]} + \\
0.0833333 \text{a2^2 m2 q2''[t]} + 1. \text{com2^2 m2 q2''[t]} + 0.0833333 \text{1h^2 m2 q2''[t]} + \\
1. a2^2 m2 q2''[t] + 0.0833333 \text{a3^3 m3 q3''[t]} + 1. \text{com3^3 m3 q2''[t]} + \\
0.0833333 \text{1h^3 m3 q2''[t]} + 2. a2 \text{com3 m3 cos(q3[t]) q2''[t]} + \\
0.0833333 \text{a3^3 m3 q3''[t]} + 1. \text{com3^3 m3 q3''[t]} + \\
0.0833333 \text{1h^3 m3 q3''[t]} + 1. a2 \text{com3 m3 cos(q3[t]) q3''[t]} \]
function tau=NewLagrangian(k)
%Script to calculate Lagrangian
%Define symbolic variables:
syms q1 q2 q3 q1dot q2dot q3dot q1ddot q2ddot q3ddot real
syms a1 a2 a3 m1 m2 m3 lw lh real
q = [q1; q2; q3];
%
%Define trigonometric functions
s1=sin(q1);
s2=sin(q2);
s3=sin(q3);
c1=cos(q1);
c2=cos(q2);
c3=cos(q3);
s23=sin(q2+q3);
c23=cos(q2+q3);
%
%Define constant
g=9.81;
%
%Define Jacobians for centers of Mass
Jcv1=[-(a1/2)*s1 0 0; (a1/2) 0 0; 0 0 0];
Jcv2=[-a1*s1-(a2/2)*s1*c2 -c1*(a2/2)*s2 0; a1*c1+(a2/2)*c1*c2 -s1*(a2/2)*s2 0; 0 (a2/2)*c2 0];
Jcv3=[-a1*s1-a2*s1*c2-(a3/2)*s1*c23 -c1*(a2*s2+(a3/2)*s23) -c1*(a3/2)*s23; a1*c1+a2*c1*c2+(a3/2)*c1*c23 -s1*(a2*s2*(a3/2)*s23) -s1*(a2/2)*s23;
\[
\begin{align*}
J_{cw1} &= \begin{bmatrix} 0 & 0 & 0; 0 & 0 & 0; 1 & 0 & 0 \end{bmatrix}; \\
J_{cw2} &= \begin{bmatrix} 0 & s1 & 0; 0 & -c1 & 0; 1 & 0 & 0 \end{bmatrix}; \\
J_{cw3} &= \begin{bmatrix} 0 & s1 & s1; 0 & -c1 & -c1; 1 & 0 & 0 \end{bmatrix}; \\
\%
\text{Rotation Matrices:} \\
R_1 &= \begin{bmatrix} c1 & 0 & s1; s1 & 0 & -c1; 0 & 1 & 0 \end{bmatrix}; \\
R_2 &= \begin{bmatrix} c2 & -s2 & 0; s2 & c2 & 0; 0 & 0 & 1 \end{bmatrix}; \\
R_3 &= \begin{bmatrix} c3 & -s3 & 0; s3 & c3 & 0; 0 & 0 & 1 \end{bmatrix}; \\
\%
\text{Inertia Tensors:} \\
I_1 &= \begin{bmatrix} (m_1/12)*(l_{w}^2+l_{h}^2) & 0 & 0; 0 & (m_1/12)*(a_1^2+l_{w}^2) & 0; 0 & 0 & (m_1/12)*(l_{h}^2+a_1^2) \end{bmatrix}; \\
I_2 &= \begin{bmatrix} (m_2/12)*(l_{w}^2+l_{h}^2) & 0 & 0; 0 & (m_2/12)*(a_2^2+l_{w}^2) & 0; 0 & 0 & (m_2/12)*(l_{h}^2+a_2^2) \end{bmatrix}; \\
I_3 &= \begin{bmatrix} (m_3/12)*(l_{w}^2+l_{h}^2) & 0 & 0; 0 & (m_3/12)*(a_3^2+l_{w}^2) & 0; 0 & 0 & (m_1/12)*(l_{w}^2+a_3^2) \end{bmatrix}; \\
\%
\text{Compute portions of D matrix} \\
D_{tran1} &= (m_1/2)*J_{cv1}'*J_{cv1}; \\
D_{tran2} &= (m_2/2)*J_{cv2}'*J_{cv2}; \\
D_{tran3} &= (m_3/2)*J_{cv3}'*J_{cv3}; \\
D_{rot1} &= J_{cw1}'*R_1*I_1*R_1'*J_{cw1}; \\
D_{rot2} &= J_{cw2}'*R_2*I_2*R_2'*J_{cw2}; \\
D_{rot3} &= J_{cw3}'*R_3*I_3*R_3'*J_{cw3}; \\
D &= D_{tran1}+D_{tran2}+D_{tran3}+D_{rot1}+D_{rot2}+D_{rot3}; \\
\%
\text{Compute Christoffel Symbols:} \\
c_{11} &= \text{Christoffel}(D,q,1,1,k); \\
c_{12} &= \text{Christoffel}(D,q,1,2,k); \ %=c_{21} \\
c_{22} &= \text{Christoffel}(D,q,2,2,k); \\
c_{13} &= \text{Christoffel}(D,q,1,3,k); \ %=c_{31} \\
c_{23} &= \text{Christoffel}(D,q,3,2,k); \ %=c_{32} \\
c_{33} &= \text{Christoffel}(D,q,3,3,k); \\
\%
\text{Compute Potential Energy} \\
P_1 &= 0; \\
P_2 &= m_2*g*(a_2/2)*s_2;
\end{align*}
\]
\[ P3 = m_3 g a_2 s_2 + (a_3/2) s_23; \]
\[ P = P1 + P2 + P3; \]

```matlab
function c = Christoffel(D, q, i, j, k)
c = (1/2) * (diff(D(k, j), q(i)) + diff(D(k, i), q(j)) - diff(D(i, j), q(k)));```

\[ g = \text{diff}(P, q(k)); \]

\[ \% Finally, compute torque:
\]
\[ \tau = D(k,1) q1ddot + D(k,2) q2ddot + D(k,3) q3ddot \ldots 
\]
\[ \quad + c11 q1dot^2 + 2 * c12 q1dot q2dot + c22 q2dot^2 \ldots 
\]
\[ \quad + 2 * c13 q1dot q3dot + 2 * c23 q2dot q3dot + c33 q3dot^2 + g; \]
Appendix C

Appendix C: Matlab Files for ode45 Solver and Singularity/Manipulability Check

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% domodel.m
% By KG Youngsma
% Date: May 14, 2012
% Rev 1 Working wo Control
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%Description: This script is the central "do" file for all the other functions that make up the model for the three DoF robotic leg. It requires:
% -threedoflegrev2.m (the file used for ode45, this calculates real joint positions, velocities, and accelerations based on desired torques)
% -EulerLagrange.m (The file which calculates torques based on desired joint positions, velocities, and accelerations)
% -trajectory.m (The file which creates a trajectory of joint positions based on global coordinate set points)
% -invkin.m (Calculates inverse kinematics)
% -quintic.m (Generates the quintic path for desired joint positions, velocities, and accelerations)

%This do file also hosts the following variables, which are also pasted into every function called by this script:

%Physical Constants:
%Link Lengths in meters
a1=0.08;
\[ a_2 = 0.2; \]
\[ a_3 = 0.2; \]
\%Link Masses in kilograms
\[ m_1 = 1; \]
\[ m_2 = 1; \]
\[ m_3 = 1; \]
\%Link width and height (for inertia matrices) in meters
\[ l_w = 0.06; \]
\[ l_h = 0.06; \]
\%Gravity:
\[ g = 9.81; \]
\%Trajectory Constants: Define lift, swing, and set times and positions, in
\%which the step begins and ends with set, lift is end of the "lift" phase,
\%and swing is at the end of the swing phase.
\[ x_{set} = -0.15; \]
\[ y_{set} = 0.1; \]
\[ z_{set} = -0.3; \]
\[ t_{set} = 0; \]
\[ x_{lift} = 0; \]
\[ y_{lift} = 0.16; \]
\[ z_{lift} = -0.1; \]
\[ t_{lift} = 25; \]
\[ x_{swing} = 0.15; \]
\[ y_{swing} = 0.1; \]
\[ z_{swing} = -0.3; \]
\[ t_{swing} = 50; \]
\[ t_{end} = 100; \]

\%Prior to runtime for the ode45 solver, all desired torques are calculated for the

\%Aquire all joint positions, velocities, and accelerations:
\[ q_{lift} = \text{trajectory}(t_{set}, x_{set}, y_{set}, z_{set}, t_{lift}, x_{lift}, y_{lift}, z_{lift}); \]
\[ q_{swing} = \text{trajectory}(t_{lift}, x_{lift}, y_{lift}, z_{lift}, t_{swing}, x_{swing}, y_{swing}, z_{swing}); \]
\[ q_{set} = \text{trajectory}(t_{swing}, x_{swing}, y_{swing}, z_{swing}, t_{end}, x_{set}, y_{set}, z_{set}); \]

\%Assemble matrix of columns of all desired joint positions, velocities, and accelerations:
\[ q_{pva} = \text{zeros}(9, 100); \]

\%for t = 1:100
if t <= 25 & & t > 0,
qpva(:,t)=qlift(:,t);
elseif t<=50 && t>25
qpva(:,t)=qswing(:,(t-25));
else
qpva(:,t)=qset(:,(t-50));
end

%Plot Desired Results:
subplot(3,3,1); plot(qpva(1,:)); title('Joint 1 Desired'); xlabel('Time'); ylabel('Position in Radians');
subplot(3,3,2); plot(qpva(2,:)); title('Joint 2 Desired'); xlabel('Time'); ylabel('Position in Radians');
subplot(3,3,3); plot(qpva(3,:)); title('Joint 3 Desired'); xlabel('Time'); ylabel('Position in Radians');
subplot(3,3,4); plot(qpva(4,:)); title('Joint 1 Desired'); xlabel('Time'); ylabel('Velocity in Radians per Second');
subplot(3,3,5); plot(qpva(5,:)); title('Joint 2 Desired'); xlabel('Time'); ylabel('Velocity in Radians per Second');
subplot(3,3,6); plot(qpva(6,:)); title('Joint 3 Desired'); xlabel('Time'); ylabel('Velocity in Radians per Second');
subplot(3,3,7); plot(qpva(7,:)); title('Joint 1 Desired'); xlabel('Time'); ylabel('Acceleration in Radians per Second Squared');
subplot(3,3,8); plot(qpva(8,:)); title('Joint 2 Desired'); xlabel('Time'); ylabel('Acceleration in Radians per Second Squared');
subplot(3,3,9); plot(qpva(9,:)); title('Joint 3 Desired'); xlabel('Time'); ylabel('Acceleration in Radians per Second Squared');
hold on

%Define tau matrix and results:
tau=zeros(100,3);
qresult=[];
time=[];

%Begin ode45 loop, where torque is input every second
for t=1:100
    %Set initial joint positions and velocities, where
    %q=[q1; q2; q3; q1dot; q2dot; q3dot]
    qin=qpva(1:6,t);
    %Use Euler-Lagrange to calculate torque
    tau=EulerLagrange(qpva(:,t));
    %Run ode45 using the threedofleg m file
    [t,qout] = ode45(@(t,q) threedoflegrev2(t,q,tau),[t t+1],qin);
    %Collect Results:
    %time=vertcat(time,t);
    qresult=vertcat(qresult,qout);
end
%Plot results:
subplot(3,3,1); plot(qresult(:,1),'-r'); title('Joint 1 Results'); xlabel('Time');
subplot(3,3,2); plot(qresult(:,2),'-r'); title('Joint 2 Results'); xlabel('Time');
subplot(3,3,3); plot(qresult(:,3),'-r'); title('Joint 3 Results'); xlabel('Time');

77
function qdot=threedoflegrev2(t,q,tau)

%Description: This function sets out the differential equation to be solved by the do file. 
%It returns actual joint position, velocity, and acceleration based on initial joint position and velocity and desired torques.

%Inputs: tau1, tau2, tau3, three tau values.

%Input initial condition variables
%Vector of q’s
q1=q(1);
q2=q(2);
q3=q(3);
q1dot=q(4);
q2dot=q(5);
q3dot=q(6);

%Define trigonometric functions
s1=sin(q1);
s2=sin(q2);
s3=sin(q3);
c1=cos(q1);
c2=cos(q2);
c3=cos(q3);
s23=sin(q2+q3);
c23=cos(q2+q3);
Physical Constants:

Link Lengths in meters
\( a_1 = 0.08; \)
\( a_2 = 0.2; \)
\( a_3 = 0.2; \)

Link Masses in kilograms
\( m_1 = 1; \)
\( m_2 = 1; \)
\( m_3 = 1; \)

Link width and height (for inertia matrices) in meters
\( l_w = 0.06; \)
\( l_h = 0.06; \)

Gravity:
\( g = 9.81; \)

Dynamic Model

Define Jacobians for centers of Mass
\[
J_{c1v} = \begin{bmatrix}
-(a_1/2)*s_1 & 0 & 0 \\
(a_1/2)*c_1 & 0 & 0 \\
0 & 0 & 0
\end{bmatrix};
\]

\[
J_{c2v} = \begin{bmatrix}
-a_1*s_1-(a_2/2)*s_1*c_2 & -c_1*(a_2/2)*s_2 & 0 \\
a_1*c_1+(a_2/2)*c_1*c_2 & -s_1*(a_2/2)*s_2 & 0 \\
0 & (a_2/2)*c_2 & 0
\end{bmatrix};
\]

\[
J_{c3v} = \begin{bmatrix}
-a_1*s_1-a_2*s_1*c_2-(a_3/2)*s_1*c_23 & -c_1*(a_2*s_2+(a_3/2)*s_23) & -s_1*(a_2*s_2*(a_3/2)*s_23) & -s_1*(a_2/2)*s_23 \\
a_1*c_1+a_2*c_1*c_2+(a_3/2)*c_1*c_23 & -s_1*(a_2*s_2*(a_3/2)*s_23) & -s_1*(a_2/2)*s_23 & 0 \\
0 & a_2*c_2+(a_3/2)*c_23 & (a_3/2)*c_23 & 0
\end{bmatrix};
\]

\[
J_{cw1} = \begin{bmatrix}
0 & 0 & 0 \\
0 & 0 & 0 \\
1 & 0 & 0
\end{bmatrix};
\]

\[
J_{cw2} = \begin{bmatrix}
0 & s_1 & 0 \\
0 & -c_1 & 0 \\
1 & 0 & 0
\end{bmatrix};
\]

\[
J_{cw3} = \begin{bmatrix}
0 & s_1 & s_1 \\
0 & -c_1 & -c_1 \\
1 & 0 & 0
\end{bmatrix};
\]

Rotation Matricies: orientation transformation from "the body attached frame and the inertial frame"
\[
R_1 = \begin{bmatrix}
c_1 & 0 & s_1 \\
s_1 & 0 & -c_1 \\
0 & 1 & 0
\end{bmatrix};
\]

\[
R_2 = \begin{bmatrix}
c_1*c_2 & -c_1*s_2 & s_1 \\
c_1*c_2 & -s_1*c_2 & -c_1 \\
s_1 & s_2 & c_2
\end{bmatrix};
\]

\[
R_3 = \begin{bmatrix}
c_1*c_23 & -c_1*s_23 & s_1 \\
c_1*c_23 & -s_1*c_23 & -c_1 \\
s_1 & s_23 & c_23
\end{bmatrix};
\]

Inertia Tensors:
\[
I_1 = \begin{bmatrix}
(m_1/12)*(l_w^2+l_h^2) & 0 & 0 \\
0 & (m_1/12)*(a_1^2+l_w^2) & 0 \\
0 & 0 & (m_1/12)*(l_h^2+a_1^2)
\end{bmatrix};
\]

\[
I_2 = \begin{bmatrix}
(m_2/12)*(l_w^2+l_h^2) & 0 & 0 \\
0 & (m_2/12)*(a_2^2+l_w^2) & 0 \\
0 & 0 & (m_2/12)*(l_h^2+a_2^2)
\end{bmatrix};
\]

\[
I_3 = \begin{bmatrix}
(m_3/12)*(l_w^2+l_h^2) & 0 & 0 \\
0 & 0 & 0 \\
0 & 0 & 0
\end{bmatrix};
\]
\[
\begin{align*}
0 \ (m3/12) \ast (a3^2 + l1^2) \ 0; \\
0 \ 0 \ (m3/12) \ast (l2^2 + a3^2) \\
\end{align*}
\]

% Compute portions of D matrix
\[
\begin{align*}
D_{\text{tran1}} &= (m1/2) \ast J_{\text{cv1}}' \ast J_{\text{cv1}}; \\
D_{\text{tran2}} &= (m2/2) \ast J_{\text{cv2}}' \ast J_{\text{cv2}}; \\
D_{\text{tran3}} &= (m3/2) \ast J_{\text{cv3}}' \ast J_{\text{cv3}}; \\
D_{\text{rot1}} &= J_{\text{cw1}}' \ast R_1 \ast I_1 \ast R_1' \ast J_{\text{cw1}}; \\
D_{\text{rot2}} &= J_{\text{cw2}}' \ast R_2 \ast I_2 \ast R_2' \ast J_{\text{cw2}}; \\
D_{\text{rot3}} &= J_{\text{cw3}}' \ast R_3 \ast I_3 \ast R_3' \ast J_{\text{cw3}}; \\
D &= D_{\text{tran1}} + D_{\text{tran2}} + D_{\text{tran3}} + D_{\text{rot1}} + D_{\text{rot2}} + D_{\text{rot3}};
\end{align*}
\]

% Compute C Matrix (from Mathematica);
\[
\begin{align*}
c_{11} &= c1 \ast ((-0.032 \ast c23 \ast s1 - 0.08 \ast c2 \ast c23 \ast s1) \ast q1dot + (-0.00151667 \ast \sin(2 \ast q2) - 0.00303333 \ast \sin(2 \ast (q2 + q3))) + c1 \ast (0.024 \ast s2 + 0.025 \ast \sin(2 \ast q2 + q3)) - 0.03 \ast \sin(2 \ast q2 + q3)) \ast q2dot; \\
c_{12} &= (-0.00151667 \ast \sin(2 \ast q2) - 0.00303333 \ast \sin(2 \ast (q2 + q3))) + c1 \ast (-0.04 \ast c23 \ast s1 - 0.04 \ast c23^2 \ast s1) \ast q2dot; \\
c_{13} &= ((c1 \ast (0.008 - 0.02 \ast c2 - 0.01 \ast c23)) + (0.008 + 0.02 \ast c2) \ast s1 \ast q2) + (-0.00151667 - 0.00303333 \ast \sin(2 \ast (q2 + q3)) + 0.02 \ast \sin(2 \ast q2 + q3)) \ast q2dot; \\
c_{21} &= (0.00151667 \ast \sin(2 \ast q2) + c1 \ast (0.024 \ast s2 + 0.025 \ast \sin(2 \ast q2) + 0.02 \ast s3 + 0.008 \ast s23 - 0.005 \ast \sin(2 \ast q2 + q3)) \ast q1dot - 0.02 \ast s3 \ast q2dot; \\
c_{22} &= 0.01 \ast \sin(2 \ast q1) \ast s2 \ast s23 \ast q1dot + (-0.04 \ast c2 \ast s23 + c23 \ast (0.04 \ast s2 + 0.02 \ast c1 \ast s23) + (0.01 \ast c23) \ast s23) + (-0.00151667 + 0.00303333 \ast \sin(2 \ast (q2 + q3))) \ast q2dot; \\
c_{23} &= (-0.01 \ast \sin(2 \ast q1) \ast s2 \ast s23 \ast q1dot + (0.02 \ast c2 \ast s23 + c23 \ast (-0.02 \ast s2 + 0.02 \ast c1 \ast s23) + (0.01 \ast c23) \ast s23) + (-0.00151667 + 0.00303333 \ast \sin(2 \ast (q2 + q3))) \ast q2dot; \\
c_{31} &= ((c1 \ast (0.008 + 0.02 \ast c2 - 0.01 \ast c23)) + (0.008 - 0.02 \ast c2) \ast s1 \ast q2) + (-0.00151667 - 0.00303333 \ast \sin(2 \ast (q2 + q3)) + 0.02 \ast \sin(2 \ast q2 + q3)) \ast q2dot; \\
c_{32} &= 0.01 \ast \sin(2 \ast q1) \ast s2 \ast s23 \ast q1dot + (0.02 \ast c2 \ast s23 + c23 \ast (-0.02 \ast s2 + 0.02 \ast c1 \ast s23) + (0.01 \ast c23) \ast s23) + (-0.00151667 + 0.00303333 \ast \sin(2 \ast (q2 + q3))) \ast q2dot; \\
c_{33} &= (0.01 \ast c1 \ast \sin(2 \ast (q2 + q3))) + (-0.01 + 0.01 \ast s1 \ast \sin(2 \ast (q2 + q3))) \ast q2dot;
\end{align*}
\]

\[
C = \begin{bmatrix}
c_{11} & c_{12} & c_{13} \\
c_{21} & c_{22} & c_{23} \\
c_{31} & c_{32} & c_{33}
\end{bmatrix}
\]

% Compute gravity matrix:
\[
\begin{align*}
g_1 &= 0; \\
g_2 &= 0.981 \ast c2 + 9.81 \ast (0.2 \ast c2 + 0.1 \ast c23); \\
g_3 &= 0.981 \ast c23;
\end{align*}
\]

\[
G = [g_1; g_2; g_3];
\]

% Calculate Velocities and Accelerations
\[
q_{\text{dot}} = [0; 0; 0; 0; 0; 0];
\]

% Calculating Accelerations Here, using input tau:
\[
A = D \ast (\text{tau} - C \ast [q_{1\text{dot}}; q_{2\text{dot}}; q_{3\text{dot}}] - G);
\]

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% EulerLagrange.m %
% By KG Youngsma %

80
function tau=EulerLagrange(qall)
%Description: Function to calculate torque based on joint position, velocity, and accleration for a single joint

%Break up inputs:
q1=qall(1);
q2=qall(2);
q3=qall(3);
q1dot=qall(4);
q2dot=qall(5);
q3dot=qall(6);
q1ddot=qall(7);
q2ddot=qall(8);
q3ddot=qall(9);

%Physical Constants:
%Link Lengths in meters
a1=0.08;
a2=0.2;
a3=0.2;
%Link Masses in kilograms
m1=1;
m2=1;
m3=1;
%Link width and height (for inertia matrices) in meters
lw=0.06;
lh=0.06;
%Gravity:
g=9.81;

%Define trigonometric functions
s1=sin(q1);
s2=sin(q2);
s3=sin(q3);
c1=cos(q1);
c2=cos(q2);
c3=cos(q3);
\[ s_{23} = \sin(q_2 + q_3); \]
\[ c_{23} = \cos(q_2 + q_3); \]

\%Define Jacobians for centers of Mass

\[
J_{cv1} = \begin{bmatrix}
-(a_1/2)*s_1 & 0 & 0 \\
(a_1/2)*c_1 & 0 & 0 \\
0 & 0 & 0 \\
\end{bmatrix};
\]

\[
J_{cv2} = \begin{bmatrix}
-a_1*s_1-(a_2/2)*s_1*c_2 & -c_1*(a_2/2)*s_2 & 0 \\
a_1*c_1+(a_2/2)*c_1*c_2 & -s_1*(a_2/2)*s_2 & 0 \\
0 & 0 & 0 \\
\end{bmatrix};
\]

\[
J_{cv3} = \begin{bmatrix}
-a_1*s_1-a_2*s_1*c_2-(a_3/2)*s_1*c_23 & -c_1*(a_2*s_2+(a_3/2)*s_23) & -c_1*(a_3/2)*s_23 \\
a_1*c_1+a_2*c_1*c_2+(a_3/2)*c_1*c_23 & -s_1*(a_2*s_2+(a_3/2)*s_23) & -s_1*(a_2/2)*s_23 \\
0 & a_2*c_2+(a_3/2)*c_23 & (a_3/2)*c_23 \\
\end{bmatrix};
\]

\[
J_{cw1} = \begin{bmatrix}
0 & 0 & 0 \\
0 & 0 & 0 \\
1 & 0 & 0 \\
\end{bmatrix};
\]

\[
J_{cw2} = \begin{bmatrix}
0 & s_1 & 0 \\
0 & -c_1 & 0 \\
1 & 0 & 0 \\
\end{bmatrix};
\]

\[
J_{cw3} = \begin{bmatrix}
0 & s_1 & s_1 \\
0 & -c_1 & -c_1 \\
1 & 0 & 0 \\
\end{bmatrix};
\]

\%Rotation Matricies: orientation transformation from "the body attached frame and the inertial frame"

\[
R_1 = \begin{bmatrix}
c_1 & 0 & s_1 \\
s_1 & 0 & -c_1 \\
0 & 1 & 0 \\
\end{bmatrix};
\]

\[
R_2 = \begin{bmatrix}
c_1*c_2 -c_1*s_2 & s_1 \\
s_1*c_2 -s_1*s_2 & -c_1 \\
0 & c_2 & 0 \\
\end{bmatrix};
\]

\[
R_3 = \begin{bmatrix}
c_1*c_23 -c_1*s_23 & s_1 \\
s_1*c_23 -s_1*s_23 & -c_1 \\
0 & c_23 & 0 \\
\end{bmatrix};
\]

\%Inertia Tensors:

\[
I_1 = \begin{bmatrix}
(m_1/12)*(l_1^2+lh^2) & 0 & 0 \\
0 & (m_1/12)*(a_1^2+1w^2) & 0 \\
0 & 0 & (m_1/12)*(lh^2+a_1^2) \\
\end{bmatrix};
\]

\[
I_2 = \begin{bmatrix}
(m_2/12)*(l_2^2+1w^2) & 0 & 0 \\
0 & (m_2/12)*(a_2^2+1w^2) & 0 \\
0 & 0 & (m_2/12)*(lh^2+a_2^2) \\
\end{bmatrix};
\]

\[
I_3 = \begin{bmatrix}
(m_3/12)*(l_3^2+1w^2) & 0 & 0 \\
0 & (m_3/12)*(a_3^2+1w^2) & 0 \\
0 & 0 & (m_3/12)*(lh^2+a_3^2) \\
\end{bmatrix};
\]

\%Compute portions of D matrix (Equation 7.50):

\[
D_{tran1} = (m_1/2)*J_{cv1}'*J_{cv1};
\]

\[
D_{tran2} = (m_2/2)*J_{cv2}'*J_{cv2};
\]

\[
D_{tran3} = (m_3/2)*J_{cv3}'*J_{cv3};
\]

\[
D_{rot1} = J_{cw1}'*R_1*I_1*R_1'*J_{cw1};
\]

\[
D_{rot2} = J_{cw2}'*R_2*I_2*R_2'*J_{cw2};
\]

\[
D_{rot3} = J_{cw3}'*R_3*I_3*R_3'*J_{cw3};
\]

\[ D = D_{tran1}+D_{tran2}+D_{tran3}+D_{rot1}+D_{rot2}+D_{rot3}; \]

\%Compute C Matrix (from Mathematica);

82
c11=c1*(-0.032*c23*s1-0.08*c2*c23*s1)*q1dot+(-0.00151667*sin(2*q2)-0.00303333*sin(2*(q2+q3))+c1^2*(-0.024*s2-0.025*sin(2*q2+q3))-0.03*sin(2*q2+q3))+s1^2*(-0.024*s2-0.025*sin(2*q2)+0.01*s3+0.016*s23-0.01*sin(2*(q2+q3))+0.03*sin(2*q2+q3)))*q2dot;

c12=(-0.00151667*sin(2*q2)-0.00151667*sin(2*(q2+q3))+c1^2*(-0.024*s2-0.025*sin(2*q2)-0.008*sin(q2+q3)-0.005*sin(2*(q2+q3))+0.02*sin(2*q2+q3)))*q1dot+(c1*(-0.04*c2*c23*s1-0.04*c23^2*s1)+sin(2*q1)*(0.03*s2*s23+0.02*s23^2))*q2dot;

c13=((c1^2*(-0.008-0.02*c2-0.01*c23)+(0.008+0.02*c2)*s1^2)*s23+(-0.00151667-0.005*s1^2)*sin(2*(q2+q3)))*q1dot+(-0.04*c1*c23^2*s1+sin(2*q1)*(0.01*s2+0.02*s23)*s23)*q2dot;

c21=(0.00151667*sin(2*q2)+c1^2*(0.024*s2+0.025*sin(2*q2)+0.02*s3+0.008*s23-0.005*sin(2*(q2+q3)))+s1^2*(0.024*s2+0.025*sin(2*q2)-0.02*s3-0.008*s23+0.015*sin(2*(q2+q3)))+0.00151667*sin(2*(q2+q3)))*q1dot+0.01*sin(2*q1)*s2*s23*q2dot;

c22=-0.02*s3*q2dot;

c23=0.01*sin(2*q1)*s2*s23*q1dot+(-0.04*c2*s23+c23*(-0.02*s2+0.02*c1^2*s23)+(-0.01+0.01*s1^2)*sin(2*(q2+q3)))*q2dot;

c31=((c1^2*(0.008+0.02*c2-0.01*c23)+(-0.008-0.02*c2)*s1^2)*s23+(0.00151667+0.015*s1^2)*sin(2*(q2+q3)))*q1dot-0.01* sin(2*q1)*s2*s23*q2dot;

c32=-0.01*sin(2*q1)*s2*s23*q1dot+(0.02*c2*s23+c23*(-0.02*s2+0.02*c1^2*s23)+(-0.01+0.01*s1^2)*sin(2*(q2+q3)))*q2dot;

c33=(0.01*c1^2*sin(2*(q2+q3))+(-0.01+0.01*s1^2)*sin(2*(q2+q3)))*q2dot;

C=[c11 c12 c13; c21 c22 c23; c31 c32 c33];

%Compute gravity matrix:
 g1=0;
 g2=0.981*c2+9.81*(0.2*c2+0.1*c23);
 g3=0.981*c23;

G=[g1; g2; g3];

%Finally, compute torques (Equation 7.63):
 tau=D*[q1ddot; q2ddot; q3ddot]+C*[q1dot; q2dot; q3dot]+G;

function qall=trajectory(t0,x0,y0,z0,tf,xf,yf,zf)
 %Description: This function returns a quintic path of joint positions, velocities, and accelerations.
 %Find the inverse kinematics of each beginning and end point:
 q0s=invkin(x0,y0,z0);
 qfs=invkin(xf,yf,zf);

 %Get the quintic path of joint positions, velocities, and accelerations between each configuration:
 q1pva=quintic(t0,q0s(1),tf,qfs(1));
 q2pva=quintic(t0,q0s(2),tf,qfs(2));
 q3pva=quintic(t0,q0s(3),tf,qfs(3));
% Assemble the qall array, placing each value in a different row and each time in a different column.

qall(1,:) = q1pva(1,:);               % Row 1: Joint 1 series of positions
qall(2,:) = q2pva(1,:);               % Row 2: Joint 2 series of positions
qall(3,:) = q3pva(1,:);               % Joint 3 series of positions
qall(4,:) = q1pva(2,:);               % Joint 1 series of velocities
qall(5,:) = q2pva(2,:);               % Joint 2 series of velocities
qall(6,:) = q3pva(2,:);               % Joint 3 series of velocities
qall(7,:) = q1pva(3,:);               % Joint 1 series of accelerations
qall(8,:) = q2pva(3,:);               % Row 8: Joint 2 series of accelerations
qall(9,:) = q3pva(3,:);               % Row 9: Joint 3 series of accelerations

%~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
% quintic.m
% By KG Youngsma
% Date: April 16, 2012
% Rev 0  Draft
%~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

function qpva=quintic(t0,q0,tf,qf)
% Description: Returns the quintic polynomial path of joint positions, velocities, and accelerations between the two joint positions:

v0=0;
a0=0;
vf=0;
af=0;

% Set up the matrix:
Matrix = [1 t0 t0^2 t0^3 t0^4 t0^5;
          0 1 2*t0 3*t0^2 4*t0^3 5*t0^4;
          0 0 2 6*t0 12*t0^2 20*t0^3;
          1 tf tf^2 tf^3 tf^4 tf^5;
          0 1 2*tf 3*tf^2 4*tf^3 5*tf^4;
          0 0 2 6*tf 12*tf^2 20*tf^3];

% The column vector of givens:
Given = [q0; v0; a0; qf; vf; af];
% Solve for a set of constants in matrix a to feed back into position, velocity and accelerations over time.
A = inv(Matrix)*Given;
% Return position, velocity, and acceleration trajectory over time.
% Set up loop and matrices for return:
i = 1;
length=tf-t0;
qpva = zeros(3,length);
for t = t0:tf
    qpva(1,i) = [1 t t^2 t^3 t^4 t^5]*A;
    qpva(2,i) = [0 1 2*t 3*t^2 4*t^3 5*t^4]*A;
    qpva(3,i) = [0 0 2 6*t 12*t^2 20*t^3]*A;
    i = i + 1;
end

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% invkin.m %
% By KG Youngsma %
% Date: May 15, 2012 %
% Rev 1 Taken from SimMechanics model %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
function qs = invkin(x,y,z)
%Link length constants
a1=0.08;
a2=0.2;
a3=0.2;
%Inverse kinematics equations (tip position to joint position)
if x>0
    t1 = atan(y/x);
elseif x==0
    t1=1.57;
else
    t1 = pi - atan(y/abs(x));
end

t3a = (x/cos(t1))-a1;
t3b = t3a^2 + z^2 - a2^2 - a3^2;
t3c = 2*a2*a3;
t3 = -acos(t3b/t3c);
t2 = -atan(a3*sin(t3)/(a2+a3*cos(t3))) ...
    + asin(z/sqrt((a2+a3*cos(t3))^2+a3^2*sin(t3)^2));
%Call for an error message if any of the results are non-real.
if ((-pi/2)<t1<(pi/2) && (isreal(t2)) && (isreal(t3))),(qs = [t1; t2; t3];
end
%Description: This script is the central "do" file to check a) the rank of the
% Jacobian and b) the manipulability of the leg along the trajectory. It requires:
% -threedoflegrev2.m (the file used for ode45, this calculates real joint positions)
% -trajectory.m (The file which creates a trajectory of joint positions based
% -quintic.m (Generates the quintic path for desired joint
% positions, velocities, and accelerations)
% -invkinematics.m (Calculates inverse kinematics)

%This do file also hosts the following variables, which are also pasted
%into every function called by this script:

%Physical Constants:
%Link Lengths in meters
a1=0.08;
a2=0.2;
a3=0.2;
%Link Masses in kilograms
m1=1;
m2=1;
m3=1;
%Link width and height (for inertia matricies) in meters
lw=0.06;
lh=0.06;
%Gravity:
g=9.81;

%Initialize arrays of results:
mus=[];
ranks=[];

%Trajectory Constants: Define lift, swing, and set times and positions, in
%which the step begins and ends with set, lift is end of the "lift" phase,
and swing is at the end of the swing phase.
xset=-0.15;
yset=0.1;
zset=-0.3;
tset=0;
xlift=0;
ylift=0.1;
zlift=-0.1;
tlift=25;
xswing=0.15;
yswing=0.1;
zswing=-0.3;
tswing=50;
tend=100;

%Aquire all joint positions, velocities, and accelerations:
qlift=trajectory(tset,xset,yset,zset,tlift,xlift,ylift,zlift);
qswing=trajectory(tlift,xlift,ylift,zlift,tswing,xswing,yswing,zswing);
qset=trajectory(tswing,xswing,yswing,zswing,tend,xset,yset,zset);

%Assemble matrix of columns of all desired joint positions, velocities, and accelerations in rows of t:
qpva=zeros(9,100);
for t = 1:100
if t<=25 && t>0,
qpva(:,t)=qlift(:,t);
elseif t<=50 && t>25
qpva(:,t)=qswing(:,(t-25));
else
qpva(:,t)=qset(:,(t-50));
end
end

%Start a loop to perform the checks for each point
for t=1:100
%Calculate the Jacobian
J=Jacobian(qpva(1,t),qpva(2,t),qpva(3,t));
%Calculate rank (for singularities)
%ra=det(J);
ra=rank(J,0);
Calculate manipulability
mu = sqrt(det(J' * J));

Collect Results:
ranks(t) = ra;
mus(t) = mu;

end

Plot results:
plot(ranks); title('Rank of the Jacobian over Trajectory'); xlabel('Time in Seconds');
plot(mus); title('Manipulability over Trajectory'); xlabel('Time in Seconds');

Jacobian: This function uses the Jacobian of the kinematics to transform
the joint velocities into torques. It takes the three joint positions that
define the leg's configuration returns the Jacobian matrix
function J = Jacobian(t1, t2, t3)

% Calculate trig functions of the given joint configuration
s1 = sin(t1);
s2 = sin(t2);
s3 = sin(t3);
s23 = sin(t2 + t3);
c23 = cos(t2 + t3);
c1 = cos(t1);
c2 = cos(t2);
c3 = cos(t3);

% Physical Constants
% Link Lengths in meters
a1 = 0.08;
a2 = 0.2;
a3 = 0.2;

% Assemble the Jacobian:
J = [-a1*s1-a2*s1*s2-a3*s1*c23 -c1*(a2*s2+a3*s23) -c1*(a3+s23);
a1*c1+a2*c1*c2+a3*c1*c23 -s1*(a2*s2+a3*s23) -s1*(a3+s23);
0 s1*(a2*s1*c2+a3*s1*c23)+c1*(a2*c1*c2+a3*c1*c23) s1*(a3*s1*c23)+c1*(
% 0 1 0] 
% 0 -c1 -c1;
% 1 0 0] 

% Note: the last three rows were removed so this match the model for finding
% singularities found in example 4.9 on page 144.