DEVELOPMENT AND APPLICATION OF COMPUTATIONAL DYNAMIC AND KINEMATIC CONSTRAINED MULTI-BODY SYSTEM SIMULATIONS IN MATLAB

A Thesis by

John Dye

Bachelor of Science, University of Kansas, 2008

Submitted to the Department of Mechanical Engineering
and the faculty of the Graduate School of
Wichita State University
in partial fulfillment of
the requirements for the degree of
Master of Science

May 2011
The following faculty members have examined the final copy of this thesis for form and content, and recommend that it be accepted in partial fulfillment of the requirement for degree of Master of Science with a major in Mechanical Engineering.

Hamid M. Lankarani, Committee Chair

Brian Driessen, Committee Member

Gerardo Olivares, Committee Member
ACKNOWLEDGEMENTS

I would like to thank my adviser, Dr. Hamid Lankarani, for his guidance and support throughout graduate school. I would also like to thank the committee members Dr. Brian Driessen and Dr. Gerardo Olivares for reviewing this thesis and for providing comments and suggestions. A special acknowledgement goes to my parents for their dedication and support, and to those who have assisted me through graduate school.
ABSTRACT

Historically machine and mechanism design relied heavily upon analytical and graphical means to evaluate the performance a system. With increasing complexity, these methods have been modified for use with computational tools. General purpose solvers have been created such as Adams, DADS and Dap3d to analyze different machines and mechanisms. Although these tools are available, they allow limited access to source code or utilize a language that is not readily taught in academics.

This thesis will focus on the creation of a general-purpose simulation environment using the currently used programming language Matlab. Four simulation programs have been created allowing simulation of kinematics and dynamics for planar and spatial mechanical systems. Discussed along with the program operation is the mathematics behind normal computational dynamics. A section is dedicated to the solution and its implementation of purely kinematic methods allowing the solution of planar and spatial systems. Constraints are heavily utilized in the formation of multi-body systems and their equations and formulations are detailed. For spatial kinematic simulations, Euler parameters are discussed in detail, and the related equations needed for multibody system simulations have been provided. The mathematics of the dynamic simulations is also discussed, along with addition of non-rigid elements such as springs and dampers.

Example simulations of specific systems have also been included, showing the results of interest utilizing the graphical user interfaces that have been created. Along with these examples is a simulation that includes two dimensional beam elements injected into the dynamic solver, which illustrates how multiple fields of engineering can be included in the simulations.
# TABLE OF CONTENTS

<table>
<thead>
<tr>
<th>Chapter</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. INTRODUCTION</td>
<td>1</td>
</tr>
<tr>
<td>1.1 Background</td>
<td>1</td>
</tr>
<tr>
<td>1.2 Literature Search</td>
<td>4</td>
</tr>
<tr>
<td>1.3 Motivation</td>
<td>6</td>
</tr>
<tr>
<td>1.4 Objective</td>
<td>7</td>
</tr>
<tr>
<td>2. METHODOLOGY</td>
<td>10</td>
</tr>
<tr>
<td>2.1 Mathematics in Matlab</td>
<td>10</td>
</tr>
<tr>
<td>2.1.1 Vectorization</td>
<td>10</td>
</tr>
<tr>
<td>2.1.2 Skew-Symmetric Matrix</td>
<td>11</td>
</tr>
<tr>
<td>2.1.3 Dot Product</td>
<td>12</td>
</tr>
<tr>
<td>2.1.4 Cross Product</td>
<td>12</td>
</tr>
<tr>
<td>2.2 Numerical Methods</td>
<td>13</td>
</tr>
<tr>
<td>2.2.1 Solving Systems of Equations</td>
<td>13</td>
</tr>
<tr>
<td>2.2.2 Newton-Raphson</td>
<td>14</td>
</tr>
<tr>
<td>2.2.3 Runge-Kutta</td>
<td>16</td>
</tr>
<tr>
<td>2.3 Modeling and Program formulation</td>
<td>17</td>
</tr>
<tr>
<td>2.3.1 Code Structure</td>
<td>19</td>
</tr>
<tr>
<td>3. IMPLEMENTATION OF KINEMATICS OF MULTIBODY SYSTEMS</td>
<td>22</td>
</tr>
<tr>
<td>3.1 Kinematic Fundamentals</td>
<td>22</td>
</tr>
<tr>
<td>3.2 Rigid Body Coordinate Systems</td>
<td>22</td>
</tr>
<tr>
<td>3.2.1 Planar Transformations</td>
<td>23</td>
</tr>
<tr>
<td>3.2.2 Spatial Transformations</td>
<td>24</td>
</tr>
<tr>
<td>3.2.3 Euler parameters</td>
<td>24</td>
</tr>
<tr>
<td>3.3 Kinematics of a Body</td>
<td>27</td>
</tr>
<tr>
<td>3.3.1 Velocity Analysis</td>
<td>27</td>
</tr>
<tr>
<td>3.3.2 Acceleration Analysis</td>
<td>28</td>
</tr>
<tr>
<td>3.4 Constrained System Formulations</td>
<td>29</td>
</tr>
<tr>
<td>3.4.1 Driving and Simple Constraints</td>
<td>30</td>
</tr>
<tr>
<td>3.5 Planar Kinematics</td>
<td>31</td>
</tr>
<tr>
<td>3.5.1 Revolute Constraint</td>
<td>32</td>
</tr>
<tr>
<td>3.5.2 Prismatic Constraint</td>
<td>33</td>
</tr>
<tr>
<td>3.6 Spatial Kinematics</td>
<td>35</td>
</tr>
<tr>
<td>3.6.1 Constraints</td>
<td>37</td>
</tr>
<tr>
<td>3.6.1.1 Spherical Constraint</td>
<td>37</td>
</tr>
</tbody>
</table>
TABLE OF CONTENTS (continued)

<table>
<thead>
<tr>
<th>Chapter</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.6.1.2 Revolute Constraint</td>
<td>39</td>
</tr>
<tr>
<td>3.6.1.3 Cylindrical Constraint</td>
<td>42</td>
</tr>
<tr>
<td>3.6.1.4 Prismatic Constraint</td>
<td>45</td>
</tr>
<tr>
<td>3.6.1.5 Spherical-Spherical Constraint</td>
<td>48</td>
</tr>
<tr>
<td>3.6.1.6 Euler Parameter Constraint</td>
<td>50</td>
</tr>
<tr>
<td>3.7 Kinematics Solver Implementation</td>
<td>50</td>
</tr>
</tbody>
</table>

4. IMPLEMENTATION OF DYNAMICS OF MULTIBODY SYSTEMS ........................................ 54

<table>
<thead>
<tr>
<th>4.1 Dynamic Fundamentals</th>
<th>54</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.1.1 Equations of Motion for Unconstrained Systems</td>
<td>54</td>
</tr>
<tr>
<td>4.1.2 Constraint Forces and Augmented formulation</td>
<td>55</td>
</tr>
<tr>
<td>4.1.3 Simulation of Springs/Dampers/Actuators</td>
<td>56</td>
</tr>
<tr>
<td>4.1.3.1 Damping</td>
<td>60</td>
</tr>
<tr>
<td>4.1.3.2 Actuator Forces</td>
<td>62</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>4.2 Dynamic Solver Implementation</th>
<th>63</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.2.1 Baumgarte Constraint Stabilization</td>
<td>66</td>
</tr>
</tbody>
</table>

5. ANALYSIS AND EXAMPLE SYSTEMS ........................................................................ 68

<table>
<thead>
<tr>
<th>5.1 Analysis Tools and Examples</th>
<th>68</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.1.1 Example: Kinematics of a Fourbar Mechanism</td>
<td>71</td>
</tr>
<tr>
<td>5.1.2 Example: Dynamics of a Fourbar Mechanism</td>
<td>76</td>
</tr>
<tr>
<td>5.1.3 Example: Vehicle Suspension - Ackermann analysis</td>
<td>80</td>
</tr>
<tr>
<td>5.1.4 Example: Dynamics of a 3D Crank Slider</td>
<td>84</td>
</tr>
<tr>
<td>5.1.5 Example: Finite Element Method Integration</td>
<td>88</td>
</tr>
</tbody>
</table>

6. CONCLUSIONS AND RECOMMENDATIONS ................................................................ 93

<table>
<thead>
<tr>
<th>6.1 Conclusions</th>
<th>93</th>
</tr>
</thead>
<tbody>
<tr>
<td>6.2 Recommendations and Future Research</td>
<td>94</td>
</tr>
</tbody>
</table>

REFERENCES .................................................................................................................. 96

APPENDIX ....................................................................................................................... 98
# LIST OF TABLES

<table>
<thead>
<tr>
<th>Table</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. <strong>Spatial Constraints Used in Dynamic Simulation</strong></td>
<td>36</td>
</tr>
<tr>
<td>2. <strong>Kinematic Constraints Using Euler Parameters</strong></td>
<td>36</td>
</tr>
<tr>
<td>3. <strong>Kinematic Constraints for Q_D</strong></td>
<td>37</td>
</tr>
<tr>
<td>Figure</td>
<td>Description</td>
</tr>
<tr>
<td>--------</td>
<td>-------------</td>
</tr>
<tr>
<td>2.1</td>
<td>OVERALL PROGRAM OPERATION</td>
</tr>
<tr>
<td>2.2</td>
<td>EXCEL SHEET SETUP</td>
</tr>
<tr>
<td>2.3</td>
<td>MAIN GUI</td>
</tr>
<tr>
<td>2.4</td>
<td>OVERALL PROGRAM STRUCTURE</td>
</tr>
<tr>
<td>3.1</td>
<td>EULER PARAMETER REPRESENTATION (LANKARANI, 2010)</td>
</tr>
<tr>
<td>3.2</td>
<td>REVOLUTE JOINT REPRESENTATION (LANKARANI, 2010)</td>
</tr>
<tr>
<td>3.3</td>
<td>PRISMATIC JOINT REPRESENTATION (LANKARANI, 2010)</td>
</tr>
<tr>
<td>3.4</td>
<td>SPHERICAL JOINT REPRESENTATION (LANKARANI, 2010)</td>
</tr>
<tr>
<td>3.5</td>
<td>REVOLUTE JOINT REPRESENTATION (LANKARANI, 2010)</td>
</tr>
<tr>
<td>3.6</td>
<td>CYLINDRICAL JOINT REPRESENTATION (LANKARANI, 2010)</td>
</tr>
<tr>
<td>3.7</td>
<td>PRISMATIC JOINT REPRESENTATION (LANKARANI, 2010)</td>
</tr>
<tr>
<td>3.8</td>
<td>SPHERICAL-SPHERICAL JOINT REPRESENTATION (LANKARANI, 2010)</td>
</tr>
<tr>
<td>3.9</td>
<td>KINEMATIC SOLVER STRUCTURE</td>
</tr>
<tr>
<td>4.1</td>
<td>VIEW OF NON-RIGID ELEMENTS IN A SYSTEM (LANKARANI, 2010)</td>
</tr>
<tr>
<td>4.2</td>
<td>SPRING DISPLACEMENT OVER 10 SECONDS</td>
</tr>
<tr>
<td>4.3</td>
<td>NET FORCE ON BODY OVER 10 SECONDS</td>
</tr>
<tr>
<td>4.4</td>
<td>DISPLACEMENT OF BODY OVER 10 SECONDS</td>
</tr>
<tr>
<td>4.5</td>
<td>NET FORCES ON BODY DUE TO DAMPING, SPRING AND GRAVITY</td>
</tr>
<tr>
<td>4.6</td>
<td>RUNGA-KUTTA ALGORITHM</td>
</tr>
<tr>
<td>4.7</td>
<td>ODE45 IMPLEMENTATION</td>
</tr>
<tr>
<td>4.8</td>
<td>NUMERICAL ERROR ACCUMULATION (LANKARANI, 2010)</td>
</tr>
<tr>
<td>4.9</td>
<td>BAUMGARTTE CONSTRAINT STABILIZATION SHOWN (LANKARANI, 2010)</td>
</tr>
<tr>
<td>Figure</td>
<td>Description</td>
</tr>
<tr>
<td>--------</td>
<td>-------------</td>
</tr>
<tr>
<td>5.1</td>
<td>The post-processor for planar systems</td>
</tr>
<tr>
<td>5.2</td>
<td>Animation progression of a 5 link system in free fall</td>
</tr>
<tr>
<td>5.3</td>
<td>Excel sheet for extended chain simulation</td>
</tr>
<tr>
<td>5.4</td>
<td>Representation of quick return mechanism</td>
</tr>
<tr>
<td>5.5</td>
<td>Excel sheet for fourbar kinematic system</td>
</tr>
<tr>
<td>5.6</td>
<td>Animation of the quick return mechanism</td>
</tr>
<tr>
<td>5.7</td>
<td>Path of point attached to mechanism</td>
</tr>
<tr>
<td>5.8</td>
<td>Kinematic analysis of a fourbar (Matlab)</td>
</tr>
<tr>
<td>5.9</td>
<td>Kinematic analysis of a fourbar mechanism (Adams)</td>
</tr>
<tr>
<td>5.10</td>
<td>Excel sheet for dynamic forbar simulation</td>
</tr>
<tr>
<td>5.11</td>
<td>Animation of dynamic fourbar mechanism</td>
</tr>
<tr>
<td>5.12</td>
<td>Matlab plot of dynamic fourbar</td>
</tr>
<tr>
<td>5.13</td>
<td>Adams plot of dynamic fourbar</td>
</tr>
<tr>
<td>5.14</td>
<td>Solidworks sketch shown</td>
</tr>
<tr>
<td>5.15</td>
<td>Excel sheet for Ackermann simulation</td>
</tr>
<tr>
<td>5.16</td>
<td>Matlab model shown</td>
</tr>
<tr>
<td>5.17</td>
<td>Progression of Ackermann simulation</td>
</tr>
<tr>
<td>5.18</td>
<td>Ackerman shown, y-axis degrees, x axis translation of the rack</td>
</tr>
<tr>
<td>5.19</td>
<td>Body information for 3D Crank Slider</td>
</tr>
<tr>
<td>5.20</td>
<td>Joint information for 3D Crank Slider</td>
</tr>
<tr>
<td>5.21</td>
<td>Animation of the 3D crank slider</td>
</tr>
<tr>
<td>5.22</td>
<td>Slider displacement for 3D crank slider</td>
</tr>
<tr>
<td>Figure</td>
<td>Description</td>
</tr>
<tr>
<td>--------</td>
<td>-------------</td>
</tr>
<tr>
<td>5.23</td>
<td>EXCEL SHEE FOR FEM INTEGRATION SIMULATION</td>
</tr>
<tr>
<td>5.24</td>
<td>ANIMATION OF PENDULUM ATTACHED TO BEAM</td>
</tr>
<tr>
<td>5.25</td>
<td>VERTICAL DISPLACEMENT OF CENTER BEAM</td>
</tr>
</tbody>
</table>
LIST OF NOMENCLATURE

C  Constraint Matrix
$C_q$  Jacobian Matrix
$Q_d$  Vector of Constraint Forces
$Q_e$  Vector of Body Forces
$q$  Vector of Body Coordinates
$\omega$  Angular Velocities
$\dot{\omega}$  Angular Accelerations
R  Global Body Position
A  3x3 Transformation Matrix
P  4x1 Vector of Euler Parameters
G  3x4 Left Hand Side of Rotation Matrix
L  3x4 Right Hand Side of Rotation Matrix
J  Inertia Tensor
M  Mass Matrix
## LIST OF SYMBOLS

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>′</td>
<td>Denotes local vector</td>
</tr>
<tr>
<td>~</td>
<td>3D Skew-Symmetric matrix</td>
</tr>
<tr>
<td>−</td>
<td>4D Skew-Symmetric matrix</td>
</tr>
<tr>
<td>.</td>
<td>First derivative</td>
</tr>
<tr>
<td>..</td>
<td>Second derivative</td>
</tr>
<tr>
<td>t</td>
<td>Transpose of vector or matrix</td>
</tr>
</tbody>
</table>
CHAPTER 1

INTRODUCTION

1.1 Background

How machines, mechanisms and bodies propagate through time and space has been extremely important to the engineering profession. Understanding motions of multi-body systems apart from the forces producing the motion is referred to as kinematics, which is integral in any mechanical system design. Dynamics is not strictly limited to the kinematics of a system and include the forces and torques that can influence movement. It may also include non-rigid elements that have an effect on the motion of a system. For an engineer these concepts are critical to any designs including machinery, mechanisms or physical bodies (Uicker, 2003).

In kinematics, the details of the motion of objects can be described by its location in space and its derivatives. Most common in design is the analysis of position and the following derivatives, normaly velocity and acceleration. Although higher order derivatives, such as jerk or snap, can be useful in specific designs, but often not needed. This leads to the analysis of mechanisms which consists of multiple bodies that can influence one another’s motions through the mechanical interconnections or kinematic pairs or joints. Links of a mechanism are arranged in a kinematic chain where joints are used to transmit motion between an input link and eventually a follower link of a mechanism (Shabana, 2010). A closed kinematic chain occurs when every link in a mechanism is connected to at least two or more other links. Some examples of these mechanisms are any assortment of fourbar systems or crank slider mechanismss. If the system is not a closed kinematic chain, then it is considered to be an open system, with examples such as robotic manipulators or pendulums. When a link is fixed in a kinematic chain, it is often referred to as ground link, and forms a specific mechanism. This fixed link provides a frame of
reference for the motion of the connected links of the mechanism. Depending on the link that is fixed (or grounded), the motions of the system can greatly change the mechanism’s motion, then resulting in various kinematic inversions of a mechanisms. A good example is the four bar inversions where depending on the link that is fixed, the mechanism would be a crank rocker, double crank etc. (Uicker, 2003).

In dynamics, is the motion of bodies is analyzed in a way that can be influenced by forces. A system where multiple bodies transmit mechanical or is influenced by forces is commonly known as a machine. These analyses can also include non-rigid bodies and elements that influence motion, such as springs, dampers and gravity. Along with the position, velocity and acceleration done in a dynamic analysis, the forces and torques applied to each body is also calculated. These forces generally results from inertia, gravity, kinematic pairs, internal or reaction forces, or any external element that provides a force to the system. An example of a dynamic system is a modern piston engine, where a crank rocker mechanism can be modeled to represent the crank and piston. With a dynamic analysis the torques and inertial forces from the piston changing acceleration can be studied. These types of analysis can be helpful when determining dynamic loads or plotting the path of a body that is influenced by external forces.

Before the widespread of computing, analytical methods analyzing the kinematics and dynamics simple systems were developed centuries ago. These methods usually involved analyzing a system at a certain point in time, often using multiple algebraic equations or graphical methods. For simple systems, things such as motion ratios, power transmission, and forces could be calculated only at instants. The type of methods used where mostly graphical employing acceleration and velocity polygons, or closed vector loops. Equations of motion could be broken down into many algebraic equations for analysis different instants. Multiple
mechanisms such as four bar and its variations where physically hand drawn and printed in large books, in order to show the path of points on different links of a mechanism. While simple systems could be calculated usually resulting in lengthy algebraic equations, stepping a system through time or addition of multiple bodies adds significant complexity to the analysis (Shabana, 2010).

With the rise in computers and their ability to process information, formulation of multi-body codes and methods were devised in order to simulate systems in time. These codes generally used a Cartesian coordinate system defining three translations and three rotational movements. Using the Cartesian coordinate systems has allowed computers to automatically generate the equations of motions from pre computed tables, while also breaking non-linear equations into a system of linear equations and ordinary differential equations (Lankarani, 2010) (Shabana, 2010). Because of the limited computing power required, most engineering programs were done in a low level programming language such as FORTRAN and C. The Dap3d program is an example of a general-purpose dynamic simulation program that is coded in FORTRAN used in academics. With the evolution of computing, academically FORTRAN has been replaced by a mathematical high level language, such as Matlab. Matlab allows the use of powerful in built mathematical functions and graphical capabilities for many simulation purposes. Being a high level interpreted language, code in Matlab can quickly be put together, readable and debugged allowing the user to focus on the problem at hand rather than the computer implementation, which is sometimes a challenge in lower level languages (Chapman, 2006).

Currently commercial programs such as Adams, DADS and MotionView exist and are widely spread in industry to model complex systems. Although academically are however difficult to modify when compared to an open source solution such as the Dap3d, and are written
in a language that is not as well taught as Matlab is. Matlab also can give very good graphical representation of systems and the parameters, something the dap3d, for example cannot do and Adams can struggle with more complex analysis (Chapman, 2006) (Lankarani, 2010).

1.2 Literature Search

The book “Theory of Machines and Mechanisms” (Uicker, 2003) discusses the graphical and analytical methods classically used to analyze machines and mechanisms. Discussed in great detail is the design of machinery and mechanisms. Where the focus is kinematic studies along with the static and dynamic forces commonly found in design. This book is essential for the theory and understanding in the field of mechanics where the next book assumes this knowledge and discusses primarily computational means.

Discussed regularly are the numerical methods used to solve the simulations. The book “Numerical Methods for Engineers” (Chapra, 2006) discusses the numerical methods used in computational codes. This book not only discusses the theory behind methods but also provided the pseudo code and analyzes the computational costs for each method. Many different methods such as solving systems of equations, root finding and solving ordinary differential equations (ODE) are discussed. Commonly used in methods discussed in this thesis are the Newton-Raphson and Runge-Kutta algorithms, which are discussed in great detail in this book.

There have been a few sources that describe the computational methods used in multi-body systems. Shabana’s book on computational dynamics is very thorough on the formulation of both kinematic and dynamic computer codes (Lankarani, 2010). The recent third edition includes SAMS/2000 a general purpose simulation program allowing non-rigid elements. This book described in great detail many different methods and the computer implementation, leaving the user to choose which method to use. Constraints and their formulations are also discussed in
detail along with different approaches to error control. There are special chapters that include formulations for special systems utilizing different approaches that are not as popular. Shabana also has another book on multibody dynamics, where primarily dynamics and non-rigid bodies. This extra book describes the mechanics and finite element formulation for implementation into dynamic studies.

A book on planar multibody dynamics is available by (Nikravesh, 2008) which describes the formulation and applications using Matlab. After describe the fundamentals of planar kinematics and dynamics, Nikravesh shows many examples of systems being solved using personalized functions in Matlab. His older book “Computer-Aided Analysis of Mechanical Systems” (Nikravesh, 1988) forms the basis of this thesis and describes both spatial and planar dynamics. This book also discusses the constraints and implementation of systems, including using the Cartesian coordinate system for generic computational methods. Another book very similar to Nikravesh’s explains the same principles, but with a focus on applications of unique systems (Haug, 1989).

Deformable bodies are commonly implemented in more advanced codes, where multiple books and papers have been written. The source “Computer-Aided Analysis of Rigid and Flexible Mechanical Systems” (Pereira, 1994) takes the majority of the time on non-rigid implementation of multibody dynamics. It is an assortment of documents that explain and discusses problems when using flexible elements along with their implementation.

More recently there have been a number of technical articles describing advances and different methods in computational dynamics (Ambrosio, 2003) (Ambrosio, 1999). These assortments of papers discuss the advances in the implementation of dynamics in different fields
such as robotics, vehicle dynamics, and biomechanics. There are a number of case studies using specific methods to solve problems and explore different methods in modern dynamics.

1.3 Motivation

A need for a general purpose dynamic simulation program is essential for thorough design and for use in an academic setting. It is often times in design where values are estimated for things such as dynamic loads, motion ratios and paths of mechanisms. With a simulation package it is possible to plot these variables and directly analyze how the system behaves either through time or through a range of motion. Often values such as dynamic loads are evaluated for their worst case scenario and an overestimation is used to prevent failure, with a simulation program it is possible to refine these values for optimization. Dynamic simulations can also be useful for analyzing frequencies and inertial forces of complex systems.

Commercial programs such as Adams and DADS are extremely useful for modeling systems but can be rather difficult to use. While powerful the user interface can be difficult to operate and the prices for such programs are costly. Adams also requires additional modeling of a system, which can be time lost due to most designs start with a CAD model and a doubling of efforts resulted. These programs are also closed source allowing modification of code for unique systems and understanding for academics rather difficult. While these packages are very successful, a general purpose program that can be easily modified and tailored to fit a specific problem can sometimes be advantageous.

Dap3d is an alternative to Adams, and is an open source general purpose solver. The features are also very basic allowing the solution of simple systems. There is little user interface and can be tedious to input system data for large simulations. The output is also difficult to parse and
graph reducing its usefulness for common study. Modification of code is rather difficult due to the coding is done in FORTRAN, which today is rarely taught in a normal mechanical engineering curriculum and is somewhat limited compared the now widely used Matlab. While this program is often used in academics, it is limited by these drawbacks and can cause confusion and difficulties.

Use of more commonly used programs such as Matlab and Excel is far more viable today than in the past. Advances in computational power and storage in modern computers allows for more complex and easy to use code. This allows higher level languages such as Matlab to process more data and complex systems quickly. A useful utility in Matlab called GUIDE allows the rapid generation of graphical user interfaces to be used with programs. This allows for extremely easy to use controls and data processing to programs. These programs can be useful giving the user an option to use more modern tools.

1.4 Objective

The goal of this thesis is to utilize current programs and tools such as CAD, Matlab and Excel to create a general purpose simulation program. The strengths of each program should be utilized to lessen the burden of the user and allow for the problem at hand to be solved without distraction into details, such as coding and user interface issues. The design process should be fluid and focused on the system or modifications currently being analyzed.

Since most designs begin with some a CAD program such as Catia or Solidworks, system details such as constraint locations, center of gravity, mass and inertias can be easily obtained for a models with high accuracy. For systems with increasing complexity, the data required for simulation can be numerous and rather difficult to organize. Using Excel’s strength in data entry
and organization, system information can be easily tabulated, modified and organized. Excel can act as an exceptional user interface for data input for simulations. Where system data can be packaged in multiple excel files, allowing each excel file to represent different system.

Once the data is organized, Matlab can import the data from the excel file and begin processing it into variable structures automatically. Multiple simulation solvers can be created for both spatial and planar simulations with each including a solver for kinematics and dynamics. Although spatial kinematic system can be rather difficult to use, it has importance academically. The program should be modular and in many separate files in order to keep the code short and easy to read, this allows for easily modifiable code. Matlab’s optimized functions and vectorization is utilized to increase speed and readability. Graphical user interfaces can be constructed to simplify procedures for program operation.

Once the simulation is complete, the data should then be processed and made available to the user. All the data from a simulation should be saved into a standard file that the user can access and process without redoing the simulation. A graphical user interface can be made to plot the data utilizing Matlab’s in built functions, allowing the user to quickly analyze and verify the simulation. A quick animation can be done to replay the data back to the user, in order to visualize the system simulated.

Once all the program and implementation is complete, the user should be able to use a simulation program to supplement any system built in CAD. For unique systems, the program can be easily modified and tailored allowing special elements to be injected into the solver directly in order to create a more realistic simulation. With the code source utilizing matrices and
vectors, much like the equations used in academic studies, this will ultimately allow for more academic understanding.
CHAPTER 2
METHODOLOGY

The theory and concepts are reviewed in the chapter, in order to discuss fluently the procedures involved in the simulation programs. Also some discussion on why some methods in Matlab may be used over others, and the numerical methods that will be necessary to use for the solvers.

2.1 Mathematics in Matlab

Most of the solver code that was created utilizes the core concepts that Matlab provides in order to compute efficiently. Generally very complex math with many operations are simplified to single lines of code, which is easy to read and implement. This will be important for future academic studies as modification of code may be necessary to bridge different fields of engineering into the solvers.

2.1.1 Vectorization

One of the drawbacks to an interpreted language such as Matlab, conventional programming utilizing many loops and operations can be an order of magnitude slower. Often in lower level languages equations would need to be broken down to their simplest form in order to do operations. For example when multiplying or filling values into a matrix it is common to utilize multiple loops, translating these loops into Matlab could cause poor performance (Langtangen, 2009).

Matlab instead heavily emphasizes utilizing matrix math to produces high performance computing. This concept is called vectorization, which means converting typical for and while loops into matrix operations (Matlab, 2010). Matrix operations are highly efficient and will
utilize different methods and techniques automatically, such as parallelization which can be rather difficult to implement in lower level languages. Instead of solving equations or operating on numbers one at a time, it is often much more efficient to compose systems of equations in matrix form and operate all at once. This requires equations and routines to be repurposed into matrix form in order to run well. This is often advantageous as a lot of theory and equations are written in this compact form using matrices and vectors already, needing no further derivation for code implementation. This way lets to be quickly generated and easily read as it follows simplified expressions almost identically.

2.1.2 Skew-Symmetric Matrix

A skew-symmetric matrix converts a vector into a corresponding matrix. The definition of the matrix will change depending on the dimensions of the vector, for this paper the three and four dimension vectors are used. It will be very common to convert three dimension vectors into the skew symmetric matrix. The three dimensional form is shown in (2.1) and can be identified by a tilde for an accent.

\[
\tilde{a} = \begin{bmatrix}
0 & -a_3 & a_2 \\
a_3 & 0 & -a_1 \\
-a_2 & a_1 & 0 \\
\end{bmatrix}
\] (2.1)

The four dimensional skew symmetric matrix will be used when dealing with Euler parameters in spatial systems. The general form is shown in equation (2.2) for the four dimensional skew symmetric matrix, which is identified by using a bar for its accent.

\[
\bar{a} = \begin{bmatrix}
0 & -a_1 & -a_2 & -a_3 \\
a_1 & 0 & a_3 & -a_2 \\
a_2 & -a_3 & 0 & a_1 \\
a_3 & a_2 & -a_1 & 0 \\
\end{bmatrix}
\] (2.2)
Two separate functions were constructed, named Skew and Skew4 that receive a vector and return the 3x3 or 4x4 skew matrix respectively. These functions will prove to be useful in order to simplify the main code and is available to all simulation programs.

2.1.3 Dot Product

The dot product is a vector operation that computes a scalar from two vectors. Unique characteristics of the dot product are if two vectors are zero or orthogonal, the operation will produce a zero result (Chapra, 2006). In Matlab, using the transpose operator will be favored over the inbuilt dot function for its efficiency. The definition of the dot product is described in equation (2.3).

\[
a \cdot b = a^t b = \begin{bmatrix} a_1 & a_2 & a_3 \end{bmatrix} \begin{bmatrix} b_1 \\ b_2 \\ b_3 \end{bmatrix}
\]

(2.3)

In Matlab this operation is easily identified using the back quote operator that transposes a vector or matrix. This can be used inline any equations, and equation (2.4) shows a Matlab example of how this is accomplished.

\[
a^t b
\]

(2.4)

2.1.4 Cross Product

The cross product is a vector operation that computes a vector that is perpendicular to two other vectors. If any of the two vectors of the cross product are zero or parallel, the result will produce a zero vector. It will be important to note this behavior, where there may be some situations this may be numerically unstable. Shown below is the general form of the cross product.
While Matlab does provide functions for cross product specifically, a more matrix friendly method will be employed. Utilizing the skew-symmetric matrix, it is possible to reproduce the same math operations found in the cross product definition.

\[
a \times b = \begin{vmatrix} i & j & k \\ a_1 & a_2 & a_3 \\ b_1 & b_2 & b_3 \end{vmatrix}
\]

(2.5)

\[
a \times b = \tilde{a}b
\]

(2.6)

Favoring this matrix method, the skew can be calculated and immediately multiplied by the second vector and will result into code like in equation (2.7).

\[
Skew(a) \ast b
\]

(2.7)

2.2 Numerical Methods

At the heart of the simulations, many numerical methods will need to be employed in order to solve a system through time. Matlab does offer many inbuilt functions that are highly optimized and used frequently. The three main methods used are described and include;

- Solving systems of equations
- Root finding algorithms
- ODE algorithms

Matlab functions were preferred over hand built functions wherever possible, due to their optimized nature and error control (Shampine, 2011).

2.2.1 Solving Systems of Equations

Almost all the simulations equations will be set up in the form of equation (2.8), where \( \{x\} \) is the unknown vector of variables.
\[ [a][x] = \{b\} \quad (2.8) \]

While Matlab does provide a method for matrix inversion, it is numerically not the best way to implement such methods in Matlab. Instead the backslash operator known as the matrix left division operator, and its use is shown in equation (2.9) to solve the same system for \( \{x\} \). Using this method equations can be quickly solved leaving the code simple to follow.

\[ x = a \backslash b \quad (2.9) \]

This method will solve a system of equations using a method based on the matrix used. According to the help files in Matlab (Matlab, 2010), if the matrix is square Gauss elimination is used which and is more efficient than using the matrix inverse. Partial pivoting is employed in order to avoid the pitfalls in a matrix that is diagonally weak or contains zeros in the diagonal (Chapra, 2006). If the matrix is not square the least squares method is used, although most matrices will be square in this thesis (Matlab, 2010).

### 2.2.2 Newton-Raphson

It is common to have a function dependant on a variable that is equation to zero, solving for this variable can be done with a root finding method.

\[ f(x) = 0 \quad (2.10) \]

One of the most widely used root finding method is known as the Newton-Raphson (NR) method. As long as the derivative of a function can be obtained, the NR method allows for a very efficient root finding, using only a few loops. Using an initial guess \( x_i \), the algorithm then computes a tangent line from the location to the x axis, improving the initial guess. This method will be used extensively for the spatial and planar kinematics to solve the constraint matrix.
Mathematically the Newton-Raphson method can be derived from beginning with the Taylor series expansion shown above (Chapra, 2006). With \( f(x) \) representing our function, \( x_i \) the current value to be improved and finally \( R_n \) representing the higher order pairs from the Taylor expansion. Since it is a numerical approximation, the higher ordered terms in the series are truncated right after the first order. This results in the following equation.

\[
f(x_{i+1}) = f(x_i) + f'(x_i)(x_{i+1} - x_i) + \frac{f''(x_i)}{2!}(x_{i+1} - x_i)^2 + \frac{f'''(x_i)}{3!}(x_{i+1} - x_i)^3 + R_n
\]  

(2.11)

Since the interest lies in the root of the equation the value that \( f(x_{i+1}) \) considered to be zero. After the zero substitution for \( f(x_{i+1}) \) and a rearrangement of terms, the improved value \( x_{i+1} \) can be solved for. This gives the final form of the Newton-Raphson algorithm.

\[
x_{i+1} = x_i - \frac{f(x_i)}{f'(x_i)}
\]  

(2.13)

If the full Taylor series could be utilized the equation would give the exact answer to the solution. Since the higher order terms that were truncated after the second order term, the rate of convergence is said to be approximately proportional to the square of the previous error (Chapra, 2006). This type of convergence is sometimes referred to as quadratic convergence and is the reason the Newton-Raphson only requires little iteration for convergence.

There are a few cases where the algorithm performs poorly and diverge. If the function iterates around an inflection point the algorithm may jump between the two sides quickly diverging from the solution. A system with multiple roots may jump away from the root of interest. Multiple maximums and minimums in a solution can cause the solution to diverge around one of the locations. If the algorithm approaches a local minimum where the slope is
zero, will cause a divide by zero error (Chapra, 2006). Most of these issues can be avoided by picking an initial estimate close to the solution and will be important when setting up initial conditions of a kinematic system.

2.2.3 Runga-Kutta

The initial value problem that will need to be solved is shown in generic form in equation (2.14).

\[ \dot{y} = f(t) \]
\[ y(0) = y_0 \]  

(2.14)

For dynamic simulations it will be necessary to integrate a system of ordinary differential equations. A very popular algorithm is known as the fourth order Runga-Kutta algorithm and will be strictly used for the dynamic simulations. Shown in equation (2.15) is the classical fourth order Runga-Kutta method for functions in the form of \( f(x, y) \), with \( h \) representing the time step (Chapra, 2006).

\[
y_{i+1} = y_i + \frac{h}{6}(k_1 + 2k_2 + 2k_3 + k_4)
\]
\[
k_1 = f(x_i, y_i)
\]
\[
k_2 = f(x_i + \frac{1}{2}h, y_i + \frac{1}{2}k_1h)
\]
\[
k_3 = f(x_i + \frac{1}{2}h, y_i + \frac{1}{2}k_2h)
\]
\[
k_4 = f(x_i + h, y_i + h)
\]

(2.15)

There are many different formulations; Matlab utilizes the Dormand-Prince algorithm (Shampine, 2011), which has higher accuracy and efficiency than most classical methods. This method is a fourth order Runga-Kutta algorithm with fifth-order error correction, and is called
ODE45 in Matlab. Using the ODE45 function will allow the possibility to numerically integrate a system of ODE’s in order to solve dynamic simulations. The order of this method is 5, which means if the stepsize h is halfed; the error is expected to be reduced by a factor of 32. Other methods are also available in Matlab.

2.3 Modeling and Program formulation

It will be important to create code that is organized and separated into many functions in separate program files; the final program contains over 40 .m files. This avoids sections of the program to become lengthy and allows reuse of code. Figure 2.1 illustrates the concept of the different phases and how the code is organized. First is the preprocess section where all the system data is taken from Excel and packaged into a single structure variable that the solvers understand. Next the simulation will use the data that was processed and simulate the system, all the calculated data is then saved at the end of this step. Finally the post process portion of the code is where the calculated data is loaded and is presented to the user in a generic graphing and animation graphical user interface. Although this is the generalized way the code works, different pre and post processors can be written or customized to either automate or analyze specific portions of a systems.

![Figure 2.1: Overall program operation](image)

Before beginning the program it will be required to fill out an excel sheet with all the system information that the program will need for the solution. This includes all the initial conditions, constraints, solution details and other specific options. Most of this data can be found using CAD models or in the case of simple systems calculated directly.
Using excel gives the advantage to easily store and organize data as shown in Figure 2.2. In order to avoid clutter multiple sheets where utilized which each sheet having a specific purpose. The sheets are organized in the following; Solver info, Body data, Constraint Data, Driving Constraints, Springs/Dampers and Post Processor options. When the program processes the data in the Excel files, it creates a single structured variable that contains all the data and saves it in a Matlab .mat file. Once the data has been entered into the program the solvers are ready to be called.
There have been a few graphical user interfaces in order to streamline the process, upon running the main file a graphical user interface shown in Figure 2.3 is brought up. This allows the user to easily run a system giving a host of options such as skipping the solver for already computed systems etc. Files can be either imported from excel and ran automatically, or to save time from an already processed excel file be loaded directly from a .mat file. The two list boxes are auto generated by searching specific folders containing the information.

2.3.1 Code Structure

The overall structure was described in the previous section, but a little more in depth of how the program operates is shown in Figure 2.4. This figure only shows the main program files that are used for the solver systems. What is not shown is the numerous helper functions and
formation of large matrices that were functionalized into separate Matlab files, these extra functions are primarily used in the solver portions of the code.

Figure 2.4: Overall program structure

The figure shows that when the system is ran it imports the data from excel then sends that information into the appropriate preprocessor for either three dimensional or two dimensional systems which will return a system structure variable that the solvers the contains the system information that the solvers can understand. The program then identifies the correct solver based on the system information and passes the system variable to the solver for
processing. Once the solver has finished the program saves the simulation data and opens the graphing and animation GUI and passes along where the simulation data was saved.
CHAPTER 3
IMPLEMENTATION OF KINEMATICS OF MULTIBODY SYSTEMS

3.1 Kinematic Fundamentals

It will be important to discuss some of the terminology and methods used in kinematics. All the details discussed here will be functionalized and shared between all simulation programs that will be used extensively in the solver implementations. These methods are based on the assumption that all bodies are rigid. For the kinematic analysis the variables of interest are almost strictly the position, velocity and accelerations of the bodies, or points attached to a body. For a system to be solved kinematically, the total degree of freedom of a system must be zero (including the driving constraints). To fully constrain a system’s degrees-of-freedom, it will be necessary to utilize joints, simple and driving constraints. Before the code implementation can be discussed thoroughly, the concepts and formulations will be described.

3.2 Rigid Body Coordinate Systems

There are a number of coordinates that could be used such as generalized or Lagrangian coordinates. Most of these coordinate systems utilize few equations overall and can prove difficult to programmatically auto-generate constraints and solve. The Cartesian coordinate system is far more suited for computational methods although it requires significantly more equations (Lankarani, 2010).

Cartesian coordinates in spatial systems can be broken down into three translations and three rotations along the x, y and z axis. This provides six degrees of freedom per body that will need to be constrained. All the bodies in a multi-body analysis can be referenced between each other in the inertial (global) coordinate system. This is the reference frame were the translations and
rotations can be described for each body and the data of interest. Due to the assumption of rigidity it is given that location of any joints or a point of interest in relation to a body remains constant (Shabana, 2010).

With a local coordinate system attached to each body, it is then possible to describe the rotations of each body in space. Using this concept a vector can be attached to the local coordinate system which can then be described in global coordinates utilizing a transformation matrix. The transformation matrices take into account the rotations of the body to project the coordinates into the global reference frame. Using these concepts vectors attached to bodies can be described in the global frame as a body is moving through space. This will be useful in constraint definitions and analyzing points of interest attached to a body.

3.2.1 Planar Transformations

In a planar system the degrees of freedom are limited to three, the x and y translations and the z rotation. It can be noted that rotation about the z axis affects the x and y axis only. The transformation matrix to local coordinates to global is described in equation (3.3) for body i.

\[
A_i = \begin{bmatrix} \cos \theta_i & -\sin \theta_i \\ \sin \theta_i & \cos \theta_i \end{bmatrix}
\]  

(3.1)

To transform a vector between the local coordinate system to global can be done using the following equation (3.2). The single quote in the superscript denotes a vector in local coordinates.

\[
\begin{bmatrix} x'_i \\ y'_i \end{bmatrix} = \begin{bmatrix} \cos \theta_i & -\sin \theta_i \\ \sin \theta_i & \cos \theta_i \end{bmatrix} \begin{bmatrix} x_i \\ y_i \end{bmatrix}
\]  

(3.2)

It is now possible to describe a point on a body in global coordinates using the local vector, body position and the transformation point shown in equation (3.3).
3.2.2 Spatial Transformations

Spatial transformations operate much like planar transformations where instead of a system with three degrees of freedom, they now have six. This adds two other angles that need to be included in the formation of the transformation matrix, which will be a three by three matrices shown in equation (3.4).

\[
\begin{bmatrix}
x_i^p \\
y_i^p
\end{bmatrix} = \begin{bmatrix}
x_i \\
y_i
\end{bmatrix} + \begin{bmatrix}
\cos \theta_i & -\sin \theta_i \\
\sin \theta_i & \cos \theta_i
\end{bmatrix} \begin{bmatrix}
x_i^p \\
y_i^p
\end{bmatrix}
\]

(3.3)

Each column represents the unit vectors of the local coordinate system that this matrix transforms. For example if the unit vector of the x axis of a local coordinate system would be in first column. Using this matrix we can do the same operations as those used in planar transformations but in three dimensions instead of two. Equation (3.5) represents the same methods but in more general form to transform a local vector into global coordinates.

\[
s_i = A_i s_i^l
\]

(3.5)

Because of the use in Euler parameters for spatial systems, the transformation matrix is never calculated directly. Instead conversations from Euler parameters to the transformation matrix was utilized to find the transformation matrix and described in the next section.

3.2.3 Euler parameters

Most of this section is from Lankarani’s class notes (Lankarani, 2010) where the idea of Euler parameters is to reduce the rotations in a spatial system into four parameters, which only three are independent. The rotation of an arbitrary point in space is shown in Figure 3.1, where
point \( P \) located by the vector \( S \) is rotated by a certain amount about an axis used to describe rotation.

Figure 3.1: Euler parameter representation (Lankarani, 2010)

Euler parameters are generally combined into a single vector \( P \) shown in equation (3.6).

\[
P = \begin{bmatrix}
e_0 \\
e_1 \\
e_2 \\
e_3
\end{bmatrix}
\] (3.6)

The equations for Euler parameters can be shown in equation (3.7) and (3.8) which is easily computed with just the angle and axis vector \( u \).

\[
e_0 = \cos \left( \frac{\phi}{2} \right)
\] (3.7)

\[
\begin{bmatrix}
e_1 \\
e_2 \\
e_3
\end{bmatrix} = \begin{bmatrix}
u_1 \\
u_2 \\
u_3
\end{bmatrix} \sin \left( \frac{\phi}{2} \right)
\] (3.8)

While this produces four parameters only three of them are independent, due to their only being three rotations in a spatial system. The equation (3.9) links all the parameters together and
reduces the independence, this will be important in order to limit the degrees of freedom and analyze error for spatial systems.

\[ P^t P = e_0^2 + e_1^2 + e_2^2 + e_3^2 = 1 \]  

(3.9)

There are methods in converting Euler parameters to the transformation matrix and is shown in equation (3.10). A function was created in Matlab in order to streamline the equation, and is done in vector form just as in the equation shown. Where e represents the first, second and third Euler parameters in vector form and I represent the 3x3 identity matrix.

\[ A = (2e_0^2 - 1)I + 2ee^t + 2e_0\ddot{e} \]  

(3.10)

It will be necessary to show some relations the Euler parameters and to start is the G and L matrices will be defined in equation (3.11) and (3.12). These two matrices form a three by four matrix.

\[
G = \begin{bmatrix} e_0, & \dot{e} + e_0I \end{bmatrix} \\
L = \begin{bmatrix} e_0, & -\dot{e} + e_0I \end{bmatrix}
\]  

(3.11)  

(3.12)

\[
e = \begin{bmatrix} e_1 \\ e_2 \\ e_3 \end{bmatrix}
\]  

(3.13)

These two matrices can be related to the transformation matrix by a simple matrix multiplication shown in equation (3.14).

\[ A = GL' \]  

(3.14)

A relation to the angular velocity vector is shown in equation (3.15) and will be used in the dynamic simulation.
\[ \omega' = 2L\dot{\theta} \]  

(3.15)

3.3 Kinematics of a Body

It was shown in the section 3.2 that the transformation matrix can be used to convert a vector that is in local coordinates to global coordinates. This is the basic concepts that will be used to define constraints when the vector loops are performed and the manipulation of vectors. It will be necessary to convert vectors located on a body’s local coordinates to global coordinates in order to compare multiple points between different bodies. To locate the position of a vector attached to the local coordinates of a body is shown in equation (3.16). This is known as the position equation for a point in global coordinates.

\[ r_i = R_i + A_iS'_i \]  

(3.16)

3.3.1 Velocity Analysis

To analyze the velocity of a point it will be necessary to take the first derivative of equation (3.16) and the general form is shown in equation (3.17)

\[ \dot{r}_i = \dot{R}_i + \dot{A}_iS'_i \]  

(3.17)

The derivative of the transformation matrix in planar systems and is the easiest method to use can be equated as

\[ \dot{A}_i = \dot{\theta} A_{i,\theta} = \dot{\theta} \begin{bmatrix} -\sin(\theta_i) & -\cos(\theta_i) \\ \cos(\theta_i) & -\sin(\theta_i) \end{bmatrix} \]  

(3.18)

For spatial systems the cross product is used along with the angular velocities shown in equation (3.19). The \( \omega \) term is represents the three angular velocities about the x, y and z axis.
\[
\dot{\omega}_i = \omega_i \times (A_i S'_i) = \ddot{\omega}_i (A_i S'_i)
\]  
(3.19)

It is assumed that the angular velocities \(\omega\) is referenced with the global frame, although this can be changed from local coordinates to global by the following equation.

\[
\omega_i = A_i \ddot{\omega}_i ^t A_i^t
\]  
(3.20)

With these identities for spatial systems the equation (3.19) can be written as

\[
\dot{r}_i = \ddot{R}_i + A_i \ddot{\omega}_i ^t S'_i
\]  
(3.21)

### 3.3.2 Acceleration Analysis

The acceleration equations found by taking the second derivative of equation (3.17). The left hand term is split into two separate terms by the product rule and for planar systems the equations is shown below.

\[
\ddot{r}_i = \ddot{R}_i - \dddot{\theta}_i A_i S'_i + \dddot{\theta}_i A_i S'_i
\]  
(3.22)

For spatial systems, equations can be formulated using the angular velocity vector \(\omega\) and its derivative angular acceleration vector \(\dot{\omega}\). This is shown in equation (3.23).

\[
\ddot{r}_i = \dddot{R}_i + \dddot{\omega}_i S'_i + A_i \dddot{\omega}_i S'_i
\]  
(3.23)

The equation (3.23) can also be written in a more friendly matrix terms shown in equation (3.24). The \(\dot{\omega}\) term is represents the three angular accelerations about the x, y and z axis.

\[
\ddot{r}_i = \dddot{R}_i + A_i \dddot{\omega}_i ^t S'_i + A_i \dddot{\omega}_i ^t \dddot{\omega}_i ^t S'_i
\]  
(3.24)

While there are some other equations that can be developed through acceleration analysis, these are the ones specifically used for a specific point located on a rigid body. The two extra components that form the derivatives of the rotations are the tangential and normal accelerations.
3.4 Constrained System Formulations

Most of the codes start with a constraint matrix \( [C] \), where each row of this matrix constrains a single degree of freedom from the system. For a kinematic system it is important that there are just as many constraints as there is degree of freedoms. Shown in equation (3.25) is the general form of the constraint matrix, it is important to note all constraints are rearranged to equal zero on the right hand side.

\[
C(q, t) = \begin{bmatrix} c_1 \\ c_2 \\ \vdots \\ c_n \end{bmatrix} = [0] \quad (3.25)
\]

\[
q = \begin{bmatrix} R_1^x & R_1^y & \theta_1 & \cdots & R_n^x & R_n^y & \theta_n \end{bmatrix} \quad (3.26)
\]

For velocity analysis it will be necessary to take the derivative of the constraint matrix defined in equation (3.25). By differentiating with respect to time along with the vector of coordinates the result can be seen in equation (3.27).

\[
C_qq = -C_t \quad (3.27)
\]

Shown in equation (3.27) is the result where due to the derivation \( C_t \) is the matrix differentiated by time only and generally only containing the driving constraints due to the joint constraints are not dependent on time. The jacobian \( C_q \) is the matrix that is differentiated by the vector of coordinates, where each coordinate represents one of the degrees of freedom. For kinematic simulations the jacobian must be a square matrix in order to be solvable. To further illustrate how the jacobian will be interpreted the following equation (3.28) shows that the columns relate to the degrees of freedom of a system and the rows relate to the constraints that are attached to the degree of freedom in a planar system (Shabana, 2010).
The equation shows how a single constraint between body one and two may be interpreted in the jacobian matrix. Each degree of freedom relates to the vector of velocities shown in equation (3.27). For the acceleration analysis the derivative of the equation (3.27) will result in a form that relies on $\ddot{\mathbf{q}}$ and allows the solution of the accelerations. The derivative can be shown in equation (3.29). This would relate the jacobian to the acceleration coordinates instead of velocity coordinates.

$$C_q \ddot{\mathbf{q}} = -(C_q \dot{\mathbf{q}})_q \dot{\mathbf{q}} - 2C_q \dot{\mathbf{q}} \dot{\mathbf{q}} - C_{\mathbf{tt}}$$  (3.29)

While the left hand side of equation (3.29) is rather simple the right hand side is rather complex. The right hand side of equation (3.29) will be lumped in a matrix $Q_d$ and for most constraints be programmed from precompiled tables (Lankarani, 2010).

### 3.4.1 Driving and Simple Constraints

Simple constraints are a constraint where a single degree of freedom is fixed to a constant value (Lankarani, 2010). These constraints do not rely on time or any other value and can be used to restrict rotations or locations of a body in the global reference frame. Equation (3.30) shows the form for the X coordinate and its derivatives, and because the value is a numeric scalar with no relations to time or coordinates, the corresponding first and second derivatives are zero. The equation (3.30) also can be used for the other degrees of freedom by simply replacing the X coordinate with the desired coordinate.

$$C = X_i - \text{constant} = 0$$
$$\dot{C} = \dot{X}_i = 0$$
$$\ddot{C} = \ddot{X}_i = 0$$  (3.30)
Driving constraints work off the same principle but instead of using a constant the constraint is usually referenced by time (Lankarani, 2010). Due to the driving constraints relying on time the first and second derivatives may not be zero and must be taken into consideration. Shown in equation (3.31) is the initial formulation with the common function that is used to reference the translational movement about the X axis with respect to time. Using the following function allows the user to specify a starting position, velocity and acceleration. While this example is shown for the X coordinate any coordinate can used by replacing the X variable.

\[
\begin{align*}
C &= X - (d_0 + vt + \frac{1}{2}at^2) = 0 \\
\dot{C} &= \ddot{X} - (v + \frac{1}{2}at) = 0 \\
\ddot{C} &= \dddot{X} - \frac{1}{2}a = 0
\end{align*}
\] (3.31)

For the oscillatory movement the coordinates and function can be replaced with the ones shown in equation (3.32) (Lankarani, 2010). This is most common as it lets the user define the amplitude, frequency and the phase of the oscillations. These can be applied to any coordinates, although X is the coordinate represented in equation (3.32).

\[
\begin{align*}
C &= X - (A\sin(\Omega t + \gamma)) = 0 \\
\dot{C} &= \ddot{X} - (A\Omega\cos(\Omega t + \gamma)) = 0 \\
\ddot{C} &= \dddot{X} + (A\Omega^2\sin(\Omega t + \gamma)) = 0
\end{align*}
\] (3.32)

The driving constraints and their functions shown can be altered utilizing any equations can in any form that involves time, as long as the following derivatives are done appropriately.

### 3.5 Planar Kinematics

Many systems can be simplified or represented into a planar form, where all bodies and their motions lie on a single plane. The global coordinate system will use the x and y axis for the horizontal and vertical axis respectively, all rotations will be considered to be about the z axis.

For the planar systems only two of the most common constraints where implemented the
revolute and the translational joints and were formulated from tables found in Lankarani’s class notes (Lankarani, 2010). There are other specialized joints that are not implemented where their use is strictly limited to specific systems. Lastly the driving and simple constraints are utilized to drive or constrain the remaining degree of freedoms for a kinematic analysis.

3.5.1 Revolute Constraint

This is one of the most common joint that is used in the four-bar, crank-slider and a majority of other mechanisms. The revolute joint is much like a hinge on a cabinet, where two degrees of freedom between two bodies are restricted, only allowing relative rotation between the two bodies. To formulate the revolute joint constraint a vector loop is calculated by subtracting the global position of the point connection from one body to the other.

![Revolute joint representation](image)

Figure 3.2: Revolute joint representation (Lankarani, 2010)

One condition is that the points from both bodies share the same global coordinates as shown in Figure 3.2. Due to the point occupying the same space the vector loop difference will
result a zero vector if the system is without error. Shown in equation (3.33) is the constraint equation used in the program and is used to populate the constraint matrix.

\[ C = R_i + A_i s_i^p - R_j - A_j s_j^p = 0 \] (3.33)

Taking the time derivative of equation (3.33) can define the jacobian portion of the code and is described in equation (3.28). Since this is a planar system, the equations form a three by three matrix for each body that is simply inserted into the correct position of the jacobian.

\[
C_q \dot{q} = \begin{bmatrix}
I & A_i s_i^l & -I & -A_j s_j^l \\
\dot{R}_i & -\dot{R}_j & \dot{\theta}_i & -\dot{\theta}_j
\end{bmatrix}
\begin{bmatrix}
\dot{R}_i \\
\dot{\theta}_i \\
\dot{R}_j \\
\dot{\theta}_j
\end{bmatrix} = \begin{bmatrix}
0 \\
0 \\
0 \\
0
\end{bmatrix}
\] (3.34)

The final portion of the constraint needed will be what develops on the right hand side of the equation (3.29). The following equation (3.35) was formulated by taking the second time.

\[ c_q \ddot{q} = A_i s_i^l \ddot{\theta}_i - A_j s_j^l \ddot{\theta}_j \] (3.35)

### 3.5.2 Prismatic Constraint

A prismatic constraint is commonly used to define a slider element in a mechanism. The constraint restricts movement along a single line of translation and is therefore defines a linear slider. Two degrees of freedom are constraint per prismatic constraint, this is due to the locked rotation and relationship between the x and y axis along the line of translation. To define a translation joint three points are required along the line of translation shown in Figure 3.3.
The points defined are arbitrary as long as each point is located on the line of translation, also two points must be defined on a single body and the third on the other. Shown in equation (3.36) is the constraint definition used to implement a prismatic joint.

\[
C = \begin{bmatrix}
(N_i)^t \left( R_i + A_i s_i^p - R_j - A_j s_j^p \right) \\
(\theta_i - \theta_j) - (\theta_i - \theta_j)_{initial}
\end{bmatrix} = 0
\] (3.36)

\[
N_i = \begin{bmatrix}
\cos(90) & -\sin(90) \\
\sin(90) & \cos(90)
\end{bmatrix} \left( A_i s_i^{\prime q} - A_i s_i^{\prime p} \right)
\] (3.37)

The reasoning behind this equation is evident when looking at the first part where the normal vector is transposed and multiplied by the d vector. This represents a dot product and recall from previous sections that the dot product is zero when the two vectors are perpendicular. To obtain the perpendicular vector it was necessary to calculate the global vector \( S_i \) and then rotate it by 90 degrees in order to find a normal vector. Because it is a prismatic joint the rotation needs to be constrained and is done so in equation (3.37)

The jacobian portion of the matrix is shown in equation (3.38) and is obtained by taking the first derivative of equation (3.36)
Finally the last step is to take the second derivative, where following the form of equation (3.36) and taken from tables and rearranged and shown in matrix form. The equations are sourced from Lankarani’s lecture notes.

\[
\dot{C}_d\ddot{q} = \begin{bmatrix}
N_t^r & (R_i + A_i S_i^p - R_j - A_j S_j^p) & A_{i,\theta} n_i + N_t^r A_{i,\theta} S_i^p & -N_t^r & -N_t^r A_{i,\theta} S_j^p
\end{bmatrix}
\begin{bmatrix}
\dot{R}_i \\
\dot{R}_j \\
\theta_i \\
\theta_j
\end{bmatrix} = 0
\]  

(3.38)

\[
C_d\ddot{q} = \begin{bmatrix}
-2\dot{\theta}_i(A_{i,\theta} n_i)^t(\dot{R}_i - \dot{R}_j) - \dot{\theta}_i^2 (s_i^p)^t n_i - (s_i^p - s_j^p)^t n_j + 2\dot{\theta}_i\dot{\theta}_j(A_{i,\theta} n_i)^t(A_{j,\theta} S_j^p) - \dot{\theta}_i^2 N_t^r A_{j,\theta} S_j^p
\end{bmatrix}
\]  

(3.39)

### 3.6 Spatial Kinematics

The kinematics of a spatial system follows the same principles and equations of planar systems but with different formulations of the constraints. The main difference is the change in degrees of freedoms and the utilization of Euler parameters that were discussed previously. The constraints themselves are more numerous and complex due to the extra degrees of freedoms a body can have. Most of the constraints are formed from simpler constraints such as spherical and perpendicular constraints. For example a revolute joint is a spherical joint with two perpendicular joints to limit two of the three rotational degrees of freedom. It should be noted that two perpendicular joints where utilized in favor of the parallel joint, due to the parallel’s joint instability from the use of the cross product. The constraints were used from tables provided by Lankarani’s lecture notes on spatial kinematics (Lankarani, 2010). Shown in Table 3.1 is the dynamic table that was used to construct the constraint equations for the jacobian and the right hand side vectors. The different nomenclature used between the lecture notes and this paper is due to the symbols are difficult to use as variables in programing, and a more standard form was used. In this table Φ is used in place of C for the constraint matrix, and λ is used in place of Q_d for the right hand side of the equations. The
first two constraints represent are the type one and two perpendicular constraints, the next

two are parallel constraints that will be ignored, and finally the last two are the spherical

constraint and spherical-spherical constraints.

Table 3.1: Spatial Constraints used in Dynamic Simulation (Lankarani, 2010)

<table>
<thead>
<tr>
<th>( \Phi )</th>
<th>( \Phi_{e}^{(0)} )</th>
<th>( \frac{1}{2} \Phi_{e}^{(0)} L^{T} )</th>
<th>( \Phi_{e}^{(1)} )</th>
<th>( \frac{1}{2} \Phi_{e}^{(1)} L^{T} )</th>
<th>( \gamma^{(m)} )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \Phi^{(0,1)} )</td>
<td>0 ( \bar{I} )</td>
<td>(- \bar{I}^{T} \bar{S} \bar{I} A_{d} )</td>
<td>0 ( \bar{I} )</td>
<td>(- \bar{I}^{T} \bar{S} \bar{I} A_{d} )</td>
<td>(- 2 \bar{I}^{T} \bar{S} \bar{I} \bar{A} \bar{R} \bar{S} + \bar{I}^{T} \bar{A} \bar{R} \bar{S} )</td>
</tr>
<tr>
<td>( \Phi^{(0,2)} )</td>
<td>(- \bar{I} )</td>
<td>((- d + \bar{S} \bar{A})^{T} \bar{S} \bar{I} A_{d} )</td>
<td>( \bar{S} \bar{I} )</td>
<td>(- \bar{S}^{T} \bar{S} A_{d} )</td>
<td>(- 2d^{T} \bar{S} A_{d} + \bar{S}^{T} \bar{A} \bar{R} \bar{S} - \bar{S}^{T} \bar{A} \bar{R} \bar{S} )</td>
</tr>
<tr>
<td>( \Phi^{(p,1)} )</td>
<td>( \bar{S} \bar{S} \bar{I} A_{d} )</td>
<td>( 0 )</td>
<td>( \bar{S} \bar{S} \bar{I} A_{d} )</td>
<td>( 0 )</td>
<td>(- 2 \bar{S} \bar{S} \bar{I} \bar{A} \bar{R} \bar{S} )</td>
</tr>
<tr>
<td>( \Phi^{(p,2)} )</td>
<td>( \bar{S} \bar{S} \bar{I} A_{d} )</td>
<td>((\bar{S} \bar{S} \bar{A} + \bar{S} \bar{S} \bar{A})^{T} \bar{S} \bar{I} A_{d} )</td>
<td>( \bar{S} \bar{S} \bar{I} A_{d} )</td>
<td>(- \bar{S} \bar{S} \bar{I} \bar{A} \bar{R} \bar{S} )</td>
<td>(- 2 \bar{S} \bar{S} \bar{I} \bar{A} \bar{R} \bar{S} )</td>
</tr>
<tr>
<td>( \Phi^{(r,1)} )</td>
<td>( \bar{S} \bar{S} \bar{I} A_{d} )</td>
<td>( \frac{3}{4} )</td>
<td>( \bar{S} \bar{S} \bar{I} A_{d} )</td>
<td>( \frac{3}{4} )</td>
<td>(- \bar{S} \bar{S} \bar{I} \bar{A} \bar{R} \bar{S} )</td>
</tr>
<tr>
<td>( \Phi^{(r,2)} )</td>
<td>( \bar{S} \bar{S} \bar{I} A_{d} )</td>
<td>( \bar{S} \bar{S} \bar{I} A_{d} )</td>
<td>( \bar{S} \bar{S} \bar{I} A_{d} )</td>
<td>( \bar{S} \bar{S} \bar{I} A_{d} )</td>
<td>(- \bar{S} \bar{S} \bar{I} \bar{A} \bar{R} \bar{S} )</td>
</tr>
</tbody>
</table>

For the kinematic constraints where used from different set of tables involving Euler

parameters. Two tables were used to define the constraints with Euler parameters; while the

representation is different the formulations are the same. Shown in Table 3-2 is the list of

kinematic constraints using the same nomenclature as described previously. These are the

constraints only, for acceleration analysis another table was used to obtain \( Q_{a} \), the right hand side

of the derivations.

Table 3-2: Kinematic Constraints using Euler Parameters (Lankarani, 2010)

<table>
<thead>
<tr>
<th>( \Phi )</th>
<th>( \Phi_{e} )</th>
<th>( \Phi_{e}^{(1)} )</th>
<th>( \Phi_{e}^{(0)} )</th>
<th>( \Phi_{e}^{(1)} )</th>
<th>( \Phi_{e}^{(0)} )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \Phi_{e}^{(0,1)} )</td>
<td>0 ( \bar{I} )</td>
<td>( \bar{S} \bar{I} C_{l} )</td>
<td>0 ( \bar{I} )</td>
<td>( \bar{S} \bar{I} C_{l} )</td>
<td>( \bar{S} \bar{I} C_{l} )</td>
</tr>
<tr>
<td>( \Phi_{e}^{(0,2)} )</td>
<td>(- \bar{I} )</td>
<td>( \bar{S} \bar{I} B_{l} + d^{T} C_{l} )</td>
<td>( \bar{S} \bar{I} )</td>
<td>( \bar{S} \bar{I} B_{l} )</td>
<td>( \bar{S} \bar{I} B_{l} )</td>
</tr>
<tr>
<td>( \Phi_{e}^{(p,1)} )</td>
<td>( \bar{S} \bar{S} \bar{I} A_{d} )</td>
<td>( \bar{S} \bar{S} \bar{I} A_{d} )</td>
<td>( \bar{S} \bar{S} \bar{I} A_{d} )</td>
<td>( \bar{S} \bar{S} \bar{I} A_{d} )</td>
<td>( \bar{S} \bar{S} \bar{I} A_{d} )</td>
</tr>
<tr>
<td>( \Phi_{e}^{(p,2)} )</td>
<td>( \bar{S} \bar{S} \bar{I} A_{d} )</td>
<td>( \bar{S} \bar{S} \bar{I} A_{d} )</td>
<td>( \bar{S} \bar{S} \bar{I} A_{d} )</td>
<td>( \bar{S} \bar{S} \bar{I} A_{d} )</td>
<td>( \bar{S} \bar{S} \bar{I} A_{d} )</td>
</tr>
<tr>
<td>( \Phi_{e}^{(r,1)} )</td>
<td>( \bar{S} \bar{S} \bar{I} A_{d} )</td>
<td>( \bar{S} \bar{S} \bar{I} A_{d} )</td>
<td>( \bar{S} \bar{S} \bar{I} A_{d} )</td>
<td>( \bar{S} \bar{S} \bar{I} A_{d} )</td>
<td>( \bar{S} \bar{S} \bar{I} A_{d} )</td>
</tr>
<tr>
<td>( \Phi_{e}^{(r,2)} )</td>
<td>( \bar{S} \bar{S} \bar{I} A_{d} )</td>
<td>( \bar{S} \bar{S} \bar{I} A_{d} )</td>
<td>( \bar{S} \bar{S} \bar{I} A_{d} )</td>
<td>( \bar{S} \bar{S} \bar{I} A_{d} )</td>
<td>( \bar{S} \bar{S} \bar{I} A_{d} )</td>
</tr>
</tbody>
</table>

where:

\[ B_{k} = 2(G_{k} x_{k}^{B} + s_{k}^{B} p_{k}^{B}) \quad ; \quad C_{k} = 2(G_{k} x_{k}^{C} + s_{k}^{C} p_{k}^{C}) \quad ; \quad k = i, j \]
These constraints will be discussed thoroughly in the following sections and assembled in a way to represent common joints in typical systems. The derivations and equations follow the Matlab code almost identically, and is discussed where they do not. Most of the equations from the tables are in global coordinates, where the vectors are converted in the code and may not be represented but explained in the methodology section of this paper.

### 3.6.1 Constraints

For the kinematic studies formulations of the constraints utilizing Euler parameters were strictly used, although for the dynamic systems the solver converts from Euler parameters to normal rotations about the x, y and z axis avoiding Euler parameters for everything but the positional coordinates. Because of this the constraints equations will be provided using both systems of coordinates.

#### 3.6.1.1 Spherical Constraint

The spherical constraint is a kinematic joint that limits the motions between two bodies at a single point. This constraint requires that the location of the point on both the bodies occupies the same location in space, but allows relative rotations about all three axes. Because of this, the
spherical constraint restricts only three degrees of freedom between two bodies. This is a common joint that is often found when connecting multiple linkages.

Figure 3.4: Spherical joint representation (Lankarani, 2010)

Figure 3.4 shows how the definition of a spherical joint, each body contains a local vector that will locate the same point in space. The definition of the constraint equation can be found by simply taking the vector loop relative to the global coordinates. The constraint equation can be found by calculating the global coordinates for the point on each body and setting them equal to each other, due to the requirement that both points on each body occupy the same point in space. With a little rearranging the constraint equation can be formulated as the one shown in equation (3.40).

\[ C = R_i + A_i s_i^p - R_j - A_j s_j^p = 0 \]  (3.40)
By taking the derivative as shown in previous sections, and utilizing tables to construct the Jacobian matrix it is possible to construct the velocity equations for a spherical joint shown in equation (3.41).

\[
C_q \dot{q} = \begin{bmatrix}
I & -\bar{S}_i A_i & -I & \bar{S}_j A_j
\end{bmatrix}
\begin{bmatrix}
\dot{R}_i \\
\omega_i \\
\dot{R}_j \\
\omega_j
\end{bmatrix} = [0] 
\]  

(3.41)

Using Euler parameters the equation can be written as shown in equation (3.42). The \( C_k \) matrix is also described in equation (3.43) and is different from the constraint matrix; this matrix is used to complete the equations for velocity analysis.

\[
C_q \dot{q} = \begin{bmatrix}
I & C_i & -I & -C_j
\end{bmatrix}
\begin{bmatrix}
\dot{R}_i \\
\dot{P}_i \\
\dot{R}_j \\
\dot{P}_j
\end{bmatrix} = [0] 
\]  

(3.42)

\[
C_k = 2(\bar{G}\bar{s}_k^p + \bar{s}_k^p p^t) 
\]  

(3.43)

For the acceleration analysis the following equations for the constraints were used are shown in equation (3.44). These equations were found by taking the second derivative of equation (3.40).

\[
C_q \ddot{q} = -\bar{\omega}_i \dot{s}_i^p + \bar{\omega}_j \dot{s}_j^p 
\]  

(3.44)

The equation (3.44) can also be written in Euler parameters shown in the equations (3.45) and (3.46).

\[
C_q \ddot{q} = h_i^p - h_j^p 
\]  

(3.45)

\[
h_k = -2G_k L_k s_k^p 
\]  

(3.46)

### 3.6.1.2 Revolute Constraint
The revolute constraint is a kinematic joint that allows only a single rotation between two bodies. This is analogous to a hinge, where five degrees of freedom are restricted allowing only rotation about a specified axis. This constraint is much like the spherical constraint, but includes an additional two constraints in order to restrict two of the rotations.

![Figure 3.5: Revolute joint representation (Lankarani, 2010)](image)

**Figure 3.5: Revolute joint representation (Lankarani, 2010)**

Shown in Figure 3.5 illustrates the construction of a revolute joint. Like the spherical joint a common point is located between the two bodies and the vector loop is calculated in order to restrict the three translational motions between the two bodies. Two other points, one point for each body, are described along the axis of rotation and are said to be parallel. Because the parallel constraint utilized the cross product, this method in certain situations can produce a zero row in the jacobian matrix causing it to become singular. Instead the orthogonal triad is formed off the vector $S_i$ which produces two more vectors that are both orthogonal to each other and $S_i$. Because there are infinite amount of vectors that could be found, a single vector was arbitrarily
chosen by using the following equation (3.47) as long as the x coordinate of $S_i$ is not zero. Then to compute the second normal vector the cross product between $W_i$ and $S_i$ will result in a vector normal to both of those shown in equation (3.48).

$$W_i = \left[ (S_i^y + S_i^z)/S_i^x \ -1 \ -1 \right]$$  \hspace{1cm} (3.47)

$$V = \vec{W}_iS_i$$  \hspace{1cm} (3.48)

If x coordinate was zero or close to, then a different formulation for equation (3.49) is used shown in equation (3.49).

$$W_i = \left[ -1 \ (S_i^x + S_i^z)/S_i^y \ -1 \right]$$  \hspace{1cm} (3.49)

If for some reason the x and y coordinates where zero or close to the final equation shown below will replace equation (3.48) and equation (3.49) in order to find the $W_i$ vector.

$$W_i = \left[ -1 \ -1 \ (S_i^y + S_i^x)/S_i^z \right]$$  \hspace{1cm} (3.50)

Finally the equations for the constraint matrix can be shown in equation (3.51), note that the first constraint equation is identical to the spherical constraints.

$$\begin{bmatrix}
C_1 \\
C_2 \\
C_3
\end{bmatrix} = \begin{bmatrix}
R_i + A_iS_i^p - R_j - A_jS_j^p \\
W_i^tS_j \\
V_i^tS_j
\end{bmatrix} = [0]$$  \hspace{1cm} (3.51)

For the velocity equations can be found by taking the derivative and is shown in equation (3.52).

$$C_q \dot{q} = \begin{bmatrix}
I & -\vec{S}_iA_i & -I & \vec{S}_jA_j \\
0 & -S_i^t\vec{W}_iA_i & 0 & W_i^t\vec{S}_jA_j \\
0 & -S_j^t\vec{V}_iA_i & 0 & V_i^t\vec{S}_jA_j
\end{bmatrix} \begin{bmatrix}
\dot{R}_i \\
\omega_i \\
\dot{R}_j \\
\omega_j
\end{bmatrix} = [0]$$

$$D = R_i + A_ip_i - R_j - A_jp_j$$  \hspace{1cm} (3.52)
Using Euler parameters the velocity equations can be written as shown in equation (3.53). The \( C_k \) matrix is also described in equation (3.43) in order to complete the equations for velocity analysis.

\[
\begin{bmatrix}
I & C_i & -I & -C_j \\
0 & -S_jC_i & 0 & W_iC_j \\
0 & -S_jC_i & 0 & V_iC_j
\end{bmatrix}
\begin{bmatrix}
\dot{R}_i \\
\dot{P}_i \\
\dot{R}_j \\
\dot{P}_j
\end{bmatrix} = [0]
\] (3.53)

For the acceleration analysis, the following equations were used and are from taking the second derivative of the constraints.

\[
C_a\ddot{q} = \begin{bmatrix}
-\ddot{\alpha}_i \ddot{S}_i^p + \ddot{\alpha}_j \ddot{S}_j^p \\
-2\dot{W}_i^t \dot{S}_j + \dot{W}_i^t \ddot{\alpha}_j S_j + \dot{S}_j^t \ddot{\alpha}_j W_i \\
-2\dot{V}_i^t \dot{S}_j + \dot{V}_i^t \ddot{\alpha}_j S_j + \dot{S}_j^t \ddot{\alpha}_j V_i
\end{bmatrix}
\] (3.54)

This can also be written into Euler parameters as shown below, where \( h_k \) is describe in equation (3.46).

\[
C_a\ddot{q} = \begin{bmatrix}
h_i^p - h_j^p \\
W_i^t h_j + S_j^t h_i - 2\dot{W}_i^t \dot{S}_j \\
V_i^t h_j + S_j^t h_i - 2\dot{V}_i^t \dot{S}_j
\end{bmatrix}
\] (3.55)

### 3.6.1.3 Cylindrical Constraint

The cylindrical constraint restricts two bodies motion along a single line of translation. Although in this case the typical cylindrical constraint allows rotation about the line of
translation. Because of this the cylindrical constraint only constrains four degrees of freedom between two bodies.

Figure 3.6: Cylindrical joint representation (Lankarani, 2010)

The Figure 3.6 illustrates how a cylindrical joint may be formed. In this definition it is required that four points be specified along the line of translation, two points referenced from each body. The d vector shown in the figure is allowed to change length allowing translation along the axis and can be calculated by taking calculating the vector loop between the two bodies and solving for the d vector is shown in equation (3.56).

\[ D = R_i + A_i p_i - R_j - A_j p_j \] (3.56)

To define a cylindrical joint the vector \( S_j \) can be set parallel to vector \( S_i \) and the vector \( S_i \) can be set parallel to vector d. Defining the system this way will cause the system to remain inline and aligned but due to the instability of the parallel constraint, like in the revolute
constraints, two perpendicular constraints were used in place of a parallel constraint. The vector \( W_i \) and \( V_i \) were constructed to form an orthogonal triad with the vector \( S_i \), the equations for this can be found at equations (3.47) through (3.50) and is to be used for the perpendicular constraints.

The constraint equations follows utilizes four perpendicular constraints in order to formulate the cylindrical constraint. Two of these perpendicular joints are of type two, that allows the length of \( D \) to be variable. The equations for the constraint matrix are shown in equation (3.57).

\[
\begin{bmatrix}
C_1 \\
C_2 \\
C_3 \\
C_4
\end{bmatrix}
= \begin{bmatrix}
W_i^t D \\
V_i^t D \\
W_i^t S_j \\
V_i^t S_j
\end{bmatrix} = [0] 
\]  
(3.57)

For the velocity equations the first derivative of the constraint equation is taken and is shown in equation (3.58).

\[
C_q \dot{q} = \begin{bmatrix}
-W_i^t & -(D + S_i^p)^t \hat{W}_i A_i & W_i^t & W_i^t S_j^p A_j \\
-V_i^t & -(D + S_i^p)^t \hat{V}_i A_i & V_i^t & V_i^t S_j^p A_j \\
0 & -S_i^j \hat{W}_i A_i & 0 & W_i^t S_j A_j \\
0 & -S_i^j \hat{V}_i A_i & 0 & V_i^t S_j A_j
\end{bmatrix}
\begin{bmatrix}
\dot{R}_i \\
\omega_i \\
\dot{R}_j \\
\omega_j
\end{bmatrix} = [0] 
\]  
(3.58)

The velocity equations can also be shown using Euler parameters as shown in equations (3.59), where \( G_k \) was described in equation (3.43). A new equation \( B_k \) will also be needed for the type two constraints and is described in equation (3.60) where \( P \) is the Euler parameters, \( S_k^p \) is the location of the point on the body and \( G_k \) is the matrix as described in chapter 2.
The accelerations equations can be found by taking the second derivative and is taken from tables (Lankarani, 2010).

\[
C_q \ddot{q} = \begin{bmatrix}
-\dot{W}_i^t & -\dot{W}_i^t B_i + D_i^t C_i & \dot{W}_i^t & \dot{W}_i^t B_j \\
-\dot{V}_i^t & -\dot{V}_i^t B_i + D_i^t C_i & \dot{V}_i^t & \dot{V}_i^t C_j \\
0 & -S_j^t C_i & 0 & \dot{V}_i^t C_j \\
0 & -S_j^t C_i & 0 & \dot{V}_i^t C_j
\end{bmatrix} \begin{bmatrix}
\dot{R}_i \\
\dot{R}_i \\
\dot{P}_i \\
\dot{P}_j
\end{bmatrix} = [0]
\] (3.59)

\[
B_k = 2(G_k \ddot{s}_k^p + \dot{s}_k^p \dot{p}_k^t)
\]

The acceleration equations can also be written in Euler parameters as shown in equation (3.62).

\[
C_q \ddot{q} = \begin{bmatrix}
-2\dot{D}_i^t \dot{W}_i^t - D_i^t \ddot{a}_i \dot{W}_i^t + \dot{W}_i^t (\ddot{a}_i \dot{s}_i^p - \ddot{a}_j \dot{s}_j^p) \\
-2\dot{D}_i^t \dot{V}_i^t - D_i^t \ddot{a}_i \dot{V}_i^t + \dot{V}_i^t (\ddot{a}_i \dot{s}_i^p - \ddot{a}_j \dot{s}_j^p) \\
-2\dot{W}_i^t \ddot{s}_j + \dot{W}_i^t \ddot{a}_j s_j + \dot{s}_j^t \ddot{a}_j W_i \\
-2\dot{V}_i^t \ddot{s}_j + \dot{V}_i^t \ddot{a}_j s_j + \dot{s}_j^t \ddot{a}_j V_i
\end{bmatrix}
\] (3.61)

The acceleration equations can also be written in Euler parameters as shown in equation (3.62).

\[
C_q \ddot{q} = \begin{bmatrix}
\dot{W}_i^t (h_j^p - h_i^p) + D_i^t h_i - 2\dot{W}_i^t \dot{D} \\
\dot{V}_i^t (h_j^p - h_i^p) + D_i^t h_i - 2\dot{V}_i^t \dot{D} \\
\dot{W}_i^t h_j + S_j^i h_i - 2\dot{W}_i^t \ddot{s}_j \\
\dot{V}_i^t h_j + S_j^i h_i - 2\dot{V}_i^t \ddot{s}_j
\end{bmatrix}
\] (3.62)

### 3.6.1.4 Prismatic Constraint

The prismatic constraint is the same as the cylindrical constraint, but with an extra constraint to restrict the rotation along the line of translation. This is common in slider systems where the two surfaces are irregular shape preventing rotation and is shown in Figure 3.7. Due to the extra constraint the degree of freedom that a prismatic constraint restricts is five, only leaving the relative motion along the line of translation.
Figure 3.7 illustrates the formation of a prismatic joint, again like the cylindrical joint four perpendicular joints are utilized to restrict motion along the line of translation. An extra perpendicular constraint is added limiting the rotation. Instead of just calculating perpendicular vectors to the $S_i$ vector, a perpendicular vector to the $S_j$ vector is also calculated. Utilizing the same notations as in the cylindrical constraint, the extra constraint is created by setting $W_j$ perpendicular to $V_i$. Again vector D is calculated as shown in equation (3.56). The constraints can then be formulated and is shown in equation (3.63).

$$\begin{bmatrix} C_1 \\ C_2 \\ C_3 \\ C_4 \\ C_5 \end{bmatrix} = \begin{bmatrix} W_i^t D \\ V_i^t D \\ W_j^t S_j \\ V_i^t S_j \\ V_i^t W_j \end{bmatrix} = [0]$$

(3.63)
The velocity equations are found by taking the first derivative of the constraint matrix and are shown in equation (3.64).

\[
C_q \dot{q} = \begin{bmatrix}
-W_i^t & -(D + S_i^p)^t \dot{W}_i A_i & W_i^t & W_i^t \dot{S}_j A_j \\
-V_i^t & -(D + S_i^p)^t \dot{V}_i A_i & V_i^t & V_i^t \dot{S}_j A_j \\
0 & -S_i^t \dot{W}_i A_i & 0 & W_i^t \dot{S}_j A_j \\
0 & -S_i^t \dot{V}_i A_i & 0 & V_i^t \dot{S}_j A_j \\
0 & -W_i^t \dot{V}_i A_i & 0 & V_i^t \dot{W}_j A_j \\
\end{bmatrix}
\begin{bmatrix}
\dot{R}_i \\
\dot{\omega}_i \\
\dot{R}_j \\
\dot{\omega}_j \\
\end{bmatrix} = [0]
\] (3.64)

The velocity equations can also be described using Euler parameters shown below, and just like the cylindrical constraints the matrix \(B_k\) is shown in equation (3.60) and the matrix \(C_k\) is shown in equation (3.43).

\[
C_q \dot{q} = \begin{bmatrix}
-W_i^t & -W_i^t B_i + D_i C_i & W_i^t & W_i^t B_j \\
-V_i^t & -V_i^t B_i + D_i C_i & V_i^t & V_i^t C_j \\
0 & -S_i^t C_i & 0 & W_i^t C_j \\
0 & -S_i^t C_i & 0 & V_i^t C_j \\
0 & -W_i^t C_i & 0 & V_i^t C_j \\
\end{bmatrix}
\begin{bmatrix}
\dot{R}_i \\
\dot{P}_i \\
\dot{R}_j \\
\dot{P}_j \\
\end{bmatrix} = [0]
\] (3.65)

The acceleration equations can also be found by taking the second derivative of the constraint matrix and is shown in equation (3.66).

\[
C_q \ddot{q} = \begin{bmatrix}
-2D_i \ddot{W}_i - D_i \ddot{\omega}_i \dot{W}_i + W_i^t (\ddot{\omega}_i S_i^p - \ddot{\omega}_j \dot{S}_j^p) \\
-2D_i \ddot{V}_i - D_i \ddot{\omega}_i \dot{V}_i + V_i^t (\ddot{\omega}_i S_i^p - \ddot{\omega}_j \dot{S}_j^p) \\
-2W_i^t \ddot{S}_j + W_i^t \ddot{\omega}_j S_j + S_i^t \ddot{\omega}_j W_i \\
-2V_i^t \ddot{S}_j + V_i^t \ddot{\omega}_j S_j + S_i^t \ddot{\omega}_j V_i \\
-2W_i^t \ddot{W}_j + V_i^t \ddot{\omega}_j W_j + W_i^t \ddot{\omega}_j V_i \\
\end{bmatrix}
\] (3.66)
These equations can also be written in Euler parameters as shown in equation (3.67).

\[
C_q \ddot{\mathbf{q}} = \begin{bmatrix}
W_i^t (h_j^p - h_i^p) + D^t h_i - 2\dot{W}_i^t \dot{D} \\
V_i^t (\dot{h}_j^p - \dot{h}_i^p) + D^t h_i - 2\dot{V}_i^t \dot{D} \\
W_i^t h_j + S_i^t h_i - 2\dot{W}_i^t \dot{S}_i \\
V_i^t h_j + S_i^t h_i - 2\dot{V}_i^t \dot{S}_i \\
V_i^t h_j + W_i^t h_i - 2\dot{V}_i^t \dot{W}_j
\end{bmatrix}
\]  

(3.67)

3.6.1.5 Spherical-Spherical Constraint

The spherical-Spherical constraint is a composite constraint that acts as a double spherical joint. Commonly used when the mass and inertias of a link are negligible to the rest of the system and will not affect the results significantly. This joint effectively simulates a massless link with the length \(d\) between two bodies. Because it is a double spherical joint it only limits a single degree of freedom and thus only takes up one row of the constraint and jacobian matrices.

![Figure 3.8: Spherical-Spherical joint representation (Lankarani, 2010)](image)

Figure 3.8 shows the formulation of the spherical-spherical constraint where the length \(d\) is calculated by taking the vector loop of the two bodies as shown below in equation (3.68).
The constraint formulation is shown below where $L$ is the described length of the massless link.

$$C = d^{t}d - L^{t} = 0$$  \hfill (3.69)

The velocity equations can be calculated by taking the derivative of the constraint equation and is shown in equation (3.71).

$$C_{q} \dot{q} = \begin{bmatrix} -2d^{t} & 2d^{t} \hat{S}_{i}^{p} A_{i} & 2d^{t} & -2d^{t} \hat{S}_{j}^{p} A_{j} \end{bmatrix} \begin{bmatrix} \dot{R}_{i} \\ \omega_{i} \\ \dot{R}_{j} \\ \omega_{j} \end{bmatrix} = 0 \hfill (3.70)$$

Using Euler parameters the equation can be written as shown in equation (3.71). The $C_{k}$ matrix is also described in equation (3.43) in order to complete the equations for velocity analysis. The $B_{k}$ matrix was also previously described in equation (3.60).

$$C_{q} \ddot{q} = \begin{bmatrix} -2d^{t} & -2d^{t} B_{i}^{p} & 2d^{t} & 2d^{t} B_{j}^{p} \end{bmatrix} \begin{bmatrix} \ddot{R}_{i} \\ \dot{\omega}_{i} \\ \ddot{R}_{j} \\ \dot{\omega}_{j} \end{bmatrix} = 0 \hfill (3.71)$$

The acceleration equations are shown in equation (3.72) and are obtained by taking the second derivative of the constraint equations.

$$C_{q} \ddot{q} = -2d^{t} \ddot{d} + 2d^{t} (\ddot{\omega}_{i} \hat{S}_{i}^{p} - \ddot{\omega}_{j} \hat{S}_{j}^{p}) \hfill (3.72)$$

The equation (3.72) can also be written in Euler parameters shown in the equation (3.73).

$$C_{q} \ddot{q} = 2d^{t} \left( h_{j}^{p} - h_{i}^{p} \right) - 2d^{t} \ddot{d} \hfill (3.73)$$
3.6.1.6 Euler Parameter Constraint

For the spatial kinematic analysis using strictly Euler parameters it will be necessary to automatically generate a constraint for each body. This is because when using Euler parameters there are seven degrees of freedom where only six of these are independent and one dependent. The dependent constraint comes from equation (3.9) that relates all four Euler parameters. Using this equation it is possible to write a constraint that limits the seventh degree of freedom. The constraint equation is shown in equation (3.74) utilizing Euler parameters.

\[ C = P_i^T P_i - 1 = 0 \]  \hspace{1cm} (3.74)

For the jacobian it is just a matter of taking the derivative of the constraint equations and is shown in equation (3.75).

\[ C_q \dot{q} = \begin{bmatrix} 0 & 2P_i^T \end{bmatrix} \begin{bmatrix} \dot{R}_i \\ \dot{P}_i \end{bmatrix} = [0] \]  \hspace{1cm} (3.75)

Lastly to find the acceleration equation, this is found by taking the second derivative of the constraint equation (3.74).

\[ C_q \ddot{q} = - \dot{P}_i^T \dot{P}_i \]  \hspace{1cm} (3.76)

3.7 Kinematics Solver Implementation

Discussed in this section is the procedure that was used to solve for the kinematics in a planar system. It will be necessary to functionalize all the matrices such as the constraint, jacobian, constraint forces and their time derivatives. This ensures that the methods used in the solution of the system remain short and compact. In order to solve the system the matrices will need to be
created for the current instant in time. Once the matrices have been populated the solution will then solve for the position followed by velocity and accelerations.

The equations are the same for planar systems and spatial systems. The main difference is the choice of coordinates where the planar systems use the constraints described in the planar constraints section, where the spatial systems use strictly Euler parameters and were described alongside with the normal definition of the spatial constraints. This does not change the procedure at all between the two systems.

![Kinematic Solver Structure](image)

Figure 3.9: Kinematic Solver Structure

The general form of how the code is structured is illustrated in Figure 3.9. The start of the simulation utilizes a for loop to step through time, with each loop the program will calculate and store the positions, velocities and accelerations at that instant in time. In this main loop, another loop will be required in order to perform the Newton-Raphson method on the constraint matrix at
that instant. Once the iterations are complete the program then continues to solve for the velocities and accelerations before looping to the next instant in time.

The Newton-Raphson loop iterates to find the position data represented as the vector $q$. With each iteration the root finding algorithm is applied to the constraint matrix which is filled out based on the current instant in time. Using the definition of the Newton-Raphson algorithm, the same method can be applied to produce an updated value of the positions. The equation (3.77) is the algorithm simplified and equation (3.78) is the Matlab code utilizing the backslash command, which solves using gauss Jordan with pivoting. It will be important to use a small enough time step in order to avoid the pitfall of the Newton-Raphson algorithm discussed in Chapter 2 that could lead to an unwanted result.

\[
q_{i+1} = C_q^{-1}C + q_i \tag{3.77}
\]
\[
q_{i+1} = C_q \backslash C + q_i \tag{3.78}
\]

With each loop the estimates are improved until the difference meets a tolerance value and the loop is then exited. After this point the position and rotations of the system has been calculated. A maximum loop limit is put in place to avoid infinite loops which can happen due to poor system definition. Once the position is calculated the matrices are updated and then ready to solve the velocities and accelerations. Equation (3.79) shows the matrix algebra that was used to solve for the velocities and equation (3.80) is the Matlab code utilizing the backlash operator to invoke a gauss Jordan solution to the system. These equations were taken from the equation (3.27).

\[
\dot{q} = C_q^{-1}(-C_t) \tag{3.79}
\]
\[
\ddot{q} = C_q \backslash (-C_t) \tag{3.80}
\]
The last component that needs to be solved is the acceleration vector which is shown in equation (3.81) where $Q_d$ represents the right hand side of equation (3.29). The code to implement the solution in Matlab is shown in equation (3.82).

$$\ddot{q} = C_q^{-1} Q_d \quad (3.81)$$

$$\ddot{q} = C_q \backslash (Q_d) \quad (3.82)$$

Once all the values are calculated they are then saved in a standard format and the for loop then iterates to the next timestep starting the process over. The system will then proceed until the simulation is complete, and because it is technically a function will return to the main motion simulation file.
CHAPTER 4
IMPLEMENTATION OF DYNAMICS OF MULTIBODY SYSTEMS

4.1 Dynamic Fundamentals

For the dynamic solvers the concentration is not how the motions of one body directly affect the other, but in the forces that cause these motions and how they transfer from one body to the other. First there will be a discussion about unconstrained systems in general, which is a system where the bodies are not connected by any kinematic constraints. In constrained system the augmented formulation combines kinematic constraints, with the dynamics of motion in order to solver for both simultaneously.

In dynamic analysis bodies are treated differently, where the location of each body corresponds to the center of gravity and the mass will need to be given in order for inertial forces to be calculated. Setting the body positions this way greatly simplifies the system from using any parallel axis theorems and is very easily calculated from CAD models. Rotational inertias will be needed and can also be easily obtained from a CAD model. To simply the system it will be assumed the products of inertia are zero and only the principle inertias are needed to for computation (Lankarani, 2010). This will require the axis of each body be rotated in order for the products to be zero, this information is also easily obtained from a CAD model.

4.1.1 Equations of Motion for Unconstrained Systems

With the locations of the bodies at the center of gravity and inertias about the principle axis, it is possible to relate the accelerations with the forces. For a planar system the equations (4.1) through (4.3) show that each of the translational degrees of freedom accelerations can be related to the forces and torques of a system.
\[ f = ma \]  
\[ f = ma \]  
\[ n = J \dot{\theta} \]  

For spatial systems the equations for the translational movements stay nearly the same. The inertial equations an extra term appears and is the resulting gyroscopic moments that are able to form in three dimensional systems. The \( M \) is the mass matrix, representing the mass in diagonal form with each row representing the \( x \), \( y \) and \( z \) translations. The other parameters for the rotational parts are \( J \) which is the inertia tensor, and \( n_i \) are the torques acting on the system. It will be important to note the all of the angular velocities used in the equations are in local coordinates, which is the general form that is calculated later (Lankarani, 2010).

\[ f = M \ddot{R} \]  
\[ n = J\dot{\omega} + \ddot{\omega}J\omega \]  

The right hand side of all these equations generalizes the inertial forces based on movement of the bodies. Based on D’Alembert’s principle the inertial forces and moments are equal to the external forces applied (Shabana, 2010). This allows the forces and moments on the left hand side of the equations be directly applied to things such as gravity, or torques applied to systems.

### 4.1.2 Constraint Forces and Augmented formulation

It is possible to the use the same equations found in section 2.2.7 and utilize them in a dynamic simulation. The same matrices such as the jacobian \( C_q \) are used and can be shared between both kinematic and dynamic analysis. From the previous analysis it is possible to
generalize the constraint forces in the form in equation (4.6) by decomposing the system using Eigen values (Shabana, 2010).

\[ Q_d = -C^r_q \lambda \]  \hspace{1cm} (4.6)

By adding the constraints to the equations of motion the system is now constrained. The equation (4.7) is the re arranged version with the vector of forces and moments \( Q_e \) on the right hand side.

\[ M\ddot{q} + C^e_q \lambda = Q_e \]  \hspace{1cm} (4.7)

Finally the equation (4.6) and equation (4.7) can be written in augmented matrix form where the constrained system of equations can be written in the \( Ax = b \) form and both the body and constraint forces are calculated simultaneously (Lankarani, 2010). Because the equation (4.8) is in matrix form it can be easily generated and solved in Matlab in just a few steps.

\[
\begin{bmatrix}
M & C^r_q \\
C_q & 0
\end{bmatrix}
\begin{bmatrix}
\dot{q} \\
\lambda
\end{bmatrix} =
\begin{bmatrix}
Q_e \\
Q_d
\end{bmatrix}
\]  \hspace{1cm} (4.8)

### 4.1.3 Simulation of Springs/Dampers/Actuators

With the source code being modifiable having full access to equation (4.8) allows integration of different elements and fields of engineering. Within each iteration, forces on the bodies can be updated to reflect more than just kinematic constraint forces. For this research a generic spring, damper and actuator elements were included into the simulation programs. The way to include these elements is during each iteration of the Runga-Kutta algorithm the spring forces are added to the body forces \( Q_e \) shown in equation (4.8). This will directly influence the motion of the bodies as if it were attached to a spring. The methods used to calculate spring forces along with an example system showing the effects of a simple spring mass system.
Figure 4.1: View of non-rigid elements in a system (Lankarani, 2010)

The first element to be implemented is the spring element, which can give a wide range of dynamic systems, such as in fields like vehicle dynamics. The equations that will tie body positions to body forces are the Hooke’s law equations shown in equation (4.9).

\[ F = -kx \]  
\[ (4.9) \]

Where \( F \) is the spring force, \( k \) is the spring constant and \( x \) is the spring displacement. To start the user will need to enter in the spring information into the Excel sheet. The data needed will be the spring constant, initial spring length and locations on the bodies these springs are attached too. With this information it will be possible to calculate the spring forces during simulation. The first thing that will be needed is to calculate the spring length at the current instant in time. To calculate the current spring length is shown in equation (4.10) and is simply done by taking the vector loop.

\[ \text{DisplacementVector} = R_i + A_is_i^p - R_j - A_js_j^p \]  
\[ (4.10) \]

This equation will return a vector, it will be necessary to compose this vector into a scalar length and find its unit vector which is shown in equations (4.11) and (4.12).
\[
SpringLength = \sqrt{\text{DisplacementVector}^t \ast \text{DistanceVector}} \quad (4.11)
\]

\[
SpringDirection = \frac{\text{DisplacementVector}}{SpringLength} \quad (4.12)
\]

To find the displacement all that has to be done is subtract the current spring length from the spring initial length that was given from the user and is shown in equation (4.13).

\[
x = SpringLength - SpringInitial \quad (4.13)
\]

It is then possible to calculate the spring forces utilizing the equation (4.9). Once the spring force is known it is then multiplied to the spring direction vector, which due to it being a unit vector will now have the correct force magnitude applied. Shown in equation (4.14) is the application of force to the system body location, this is allowed due to the forces applied to a point away from a body can be directly transcribed on the body center.

\[
\text{ForceVector} = SpringDirection \ast k \ast (SpringLength - SpringInitial) \quad (4.14)
\]

Because the spring can be attached away from the center of gravity of a body, it will also be necessary to calculate the resulting moments. This is done easily by taking the cross product of the spring location vector with the force vector that was found in equation (4.14). The moment vector result is shown below in equation (4.15).

\[
\text{MomentVector} = SpringLocation \times ForceVector \quad (4.15)
\]

Now that all the forces and moments are calculated, it is simply a matter of assembling the body force and moment vector that will be added to \( Q_e \) from equation (4.8). Shown in equation (4.16) is the way the spring forces are added to the body forces, note that some forces are multiplied by negative one due to the spring acting in equal and opposite directions on the two bodies.
TotalBody Forces = \( Q_e + \text{spring forces} = \begin{bmatrix} F_i \\ M_i \\ F_j \\ M_j \end{bmatrix} + \begin{bmatrix} -F_i^{\text{spring}} \\ -M_i^{\text{spring}} \\ F_j^{\text{spring}} \\ M_j^{\text{spring}} \end{bmatrix} \) (4.16)

With this it is possible to setup a spring mass system, where the body hanging by a spring and is set exactly at the initial length of the spring. What will happen is the weight of the mass will pull the spring down causing it to oscillate and is shown in Figure 4.2. Due to the body have a mass of five pounds and a spring constant of 50 pounds per foot, the resulting net forces from the spring and gravity is shown in Figure 4.3.

![Figure 4.2: Spring displacement over 10 seconds.](image)
4.1.3.1 Damping

Damping is also very often used so it will be beneficial to add these elements in with the simulation. Fortunately the process is very similar to the spring elements shown in the previous section. Therefore the steps that are different will be shown and explained how it is added to the vector of forces. The main difference between a spring element and a damping element is instead of a proportional gain attached to the displacement, a gain is attached to the velocities. So to do the velocity vector is calculated between the two points on the body by simply taking the derivative of equation (4.10) and is shown below.

\[
VelocityVector = \dot{R}_i + A_i \tilde{\omega}_i' s_i' p - \dot{R}_j - A_j \tilde{\omega}_j' s_j' p (4.17)
\]

With the velocity vector in place to calculate the damping forces, it is simply a matter of multiplying by the damping constant c.

\[
DampingForces = c \times VelocityVector (4.18)
\]

To include this with the spring forces equation (4.14) is modified as shown below.
With the addition with the damping forces to the spring forces, the same procedure that was described after equation (4.14) was carried out. By combining the spring forces with the damping forces this will automatically add the damping forces and torques to the body.

Shown in Figure 4.4 is the displacement of the body, which shows the normal spring-damper system where the oscillation is dampened until equilibrium. Figure 4.5 shows the forces acting on the body as it too comes to an equilibrium value due to damping. Note the equilibrium value is zero is because the net forces are shown; where in this case gravity is being canceled by the spring force.

\[ \text{ForceVector} = \text{SpringForces} + \text{DampingForces} \]  
(4.19)
4.1.3.2 Actuator Forces

An actuator force can be modeled as a force that is constantly applied to the body. Because the forces act between the two vectors the same calculations found in the spring element can be used to inject the actuating forces. Shown below is the addition of actuator force to the equation (4.19).

\[ \text{ForceVector} = \text{SpringForces} + \text{DampingForces} + \text{ActuatorForce} \times \text{SpringDirection} \]  \hspace{1cm} (4.20)

With equation (4.20) all the spring, damping and actuator forces are included allowing the same procedure used after equation (4.19) to be used to calculate the torques and add to the vector of body forces and torques. It should be noted that if the spring constant, damping constant or actuator forces are zero there will be no effect on the forces and can be considered not in play.
4.2 Dynamic Solver Implementation

So far the building blocks to solve dynamic systems have been discussed. In this section the code implementation and methods for the solver used will be discussed. The kinematic solver used the Newton-Raphson root finding method on the constraint matrices in order to solve for the position data. This required a system that has as many constraints as degrees of freedom producing a square jacobian matrix. In dynamic simulations the total degrees of freedoms of a system can be more than zero unlike in a kinematic simulation. Because of this the square jacobian matrix requirement is no longer required and the Newton-Raphson method is no longer viable. A different method will be required for the dynamic solver and the method chosen is the Runga-Kutta algorithm.

This algorithm works differently from the kinematic solver using the Newton-Raphson. Where the kinematic system calculated the position, velocity and acceleration in that order, the dynamic solver calculates the acceleration, velocity and final position in that order. Although Matlab allows a system of equations to be used in the ODE solver a one-step solution. Before the Matlab ODE function is discussed it is important to visualize how the ODE solver works. Technically this simulation the Runga-Kutta algorithm solves for the next step in time and not directly numerically integrating the equations, although this is how it can be represented using the methods that will be shown.
Figure 4.6: Runga-Kutta Algorithm

Figure 4.6 shows how the algorithm will be used in order to solve forward through time. To solve for the velocities at the next time step the accelerations and velocities will be required, using the Runga-Kutta algorithm the velocity at the next time step can be calculated. For position the derivative is the velocities which are already given and can be used directly to calculate the position at the next time (Lankarani, 2010).

Matlab provides the popular fourth order Runga-Kutta algorithm with fifth order error correction, and is the ODE45 function that was used. The error correction is done by analyzing the error at multiple time steps and using that to provide an improved solution. The algorithm also uses an adaptive time step to improve performance and error. If the time step is large for stiff systems error can be quite large, and the ODE solver will reduce the time step. Where if the system is not stiff the time step will be increased which will improve performance without impacting error (Shampine, 2011).

The ODE45 function was used by inputting the initial velocities and positions and running the function. A separate function that takes in the current velocities and positions and
outputs the velocities and accelerations will be needed for the Matlab function. The Matlab ODE45 will call this function expecting the derivative of the inputted variables.

Figure 4.7: ODE45 Implementation

Figure 4.7 shows how the ODE45 solver was implemented in Matlab. What the figure shows is the current velocities and positions are inputted to the function that user must create. The algorithm will expect the derivatives of these variables as an output; in this case it will need the velocities and accelerations. The velocities as simply passed through as the current velocities were inputted to the function as was shown in Figure 4.6. The accelerations will need to be calculated and can be easily done by assembling the augmented matrix shown in (4.8). During each step the forces need to be added which include the spring, dampers or actuators for the augmented matrix formulation. Finally the function returns the velocities and calculated
accelerations to the ODE algorithm. The algorithm then calculates the position and velocities for the next time step and repeats the process over until the simulation time is complete.

### 4.2.1 Baumgarte Constraint Stabilization

One of the problems that can occur with longer simulations is the error that generates in the constraints themselves. Because the constraint forces are only used to solve for the accelerations, the error will accumulate over time, shown in Figure 4.8: Numerical error accumulation (Lankarani, 2010). This is mainly due to the discrete time steps used where reducing the time step may improve results. There is another method, where calculating the constraint matrix error and velocity error can be feed backed into correct the constraints.

![Figure 4.8: Numerical error accumulation (Lankarani, 2010)](image)

The Baumgarte method takes a page from control theory where the constraint matrix and its derivative are used along with a gain to feedback correct the forces in the augmented matrix. Figure 4.9: Baumgarte Constraint Stabilization shown shows how the feedback system will cause the error to stabilize around the exact solution instead of diverging over time.
Figure 4.9: Baumgarte Constraint Stabilization shown (Lankarani, 2010)

The equation shown below is the equation that was utilized and accomplished by simply subtracting from the current forces in the augmented matrix. The beta term is a gain that is applied directly to the constraint matrix error, for a perfect system the constraints would be zero. The alpha term is a gain that is applied to the derivative of the error (Shabana, 2010).

\[
\begin{bmatrix}
M & C_q^T \\
C_q & 0
\end{bmatrix}
\begin{bmatrix}
\ddot{q} \\
\dot{\lambda}
\end{bmatrix}
= \begin{bmatrix}
Q_e \\
Q_d - 2\alpha(C_q\dot{q} + C_\ell) - \beta^2 C
\end{bmatrix}
\]

(4.21)

The only issue with this type of constraint stabilization is choosing the alpha and beta values are somewhat arbitrary as it depends on the system and how it moves through time. It will be up to the designer to look to analyze the error and evaluate the alpha and beta for different simulations. Usually the initial values are one for alpha and beta.
5.1 Analysis Tools and Examples

Once the system has been solved, often is the question on how to present or analyze the data. While the data is saved to a .mat file allowing the data to be manipulated how the user sees fit, a graphical user interface was created in order to graph the generic variables of interest a user may want to see. The variables of interest are often the positions, velocities and accelerations of each body and for a dynamic analysis the constraint forces or body forces may also be of interest. Since sometimes a system can behave differently or perhaps an error was made in its formulation, the ability to graph the error overtime quickly may be useful to the user.

![Figure 5.1: The post-processor for planar systems](image)
Figure 5.1 shows the analysis interface for the planar simulations which is identical to the spatial interface. The interface will appear upon completion of a simulation allowing the user to quickly start analyzing the results. The options for both x and y axis allows the user to pick the body they want to reference and the variables to graph. Several options are included to allow quick comparison, such as turning off automatic redraw and changing the line color. To analyze the error the constraint matrix equation (3.25) and the derivative form of equation (3.27) are given as options to be graphed. These equations are such that for if the system was without error the matrices result would be zeros, which can be helpful to identify problems for when there not. Since it is a numerical approximation the error can be shown by graphing the results of these matrices. Also another form of error that is quickly available to the user is utilizing the Euler parameter constraint equation (3.74).

So far the data can be represented to the user using graph easily provided by the user interface. Although sometimes the graphs can be ambiguous, especially in spatial systems, where an animation could be used to play back the system allowing the user to quickly understand what is happening. An example of the animation is shown in figure Figure 5.2.
Figure 5.2: Animation progression of a 5 link system in free fall

This animation is of a five link chain that is extended along the X axis and allowed to free fall. The four frames capture the motion that was played back during the animation; coordinate systems are shown attached to each body. The large spheres show the location of either a spherical or revolute joint, where the smaller spheres show the body’s location. The excel sheet that describes the body and joint data is shown in Figure 5.3.
5.1.1 Example: Kinematics of a Fourbar Mechanism

In this example a fourbar system is simulated to demonstrate how the solver can be used to simulate a simple mechanism. The planar kinematic solver was used with the input link rotating half a revolution a second. Figure 5.4 show the representation of the mechanism drawn in Matlab. Notice the red point above the middle link in the mechanism, this is a point attached to the upper link. This example is to demonstrate how a point of interest such as the red point shown in the figure and be analyzed, the interest is in the path of this point as the system is rotates.
Figure 5.4: Representation of quick return mechanism

The excel file was populated with the system positions and constraint definitions to model the system shown in the above figure. Three main sheets were populated in order to fill out the system and are shown in Figure 5.5. These define the body initial parameters, constraint information and the driver required to move the system.
Figure 5.5: Excel sheet for fourbar kinematic system

To show how the system moves through time Figure 5.6 shows the motion of the system at certain points in time. The red point that is of interest can be seen moving with the middle link of the mechanism. This is expected and can be plotted in the main interface.
Figure 5.6: Animation of the quick return mechanism

Figure 5.7: Path of point attached to mechanism
Finally Figure 5.7 plots the path of the point as it moves with the system. When a point is told to be simulated extra options appear in the body tab that lets the user pick the parts of the point to graph. Simply by plotting the x coordinate of the point on the x axis and the y coordinate of the point on the y axis the path can then be shown.

The same system was modeled in Adams in order to verify the operation of the program. The x and y positions of the output link were plotted over five seconds, producing very similar results between the programs. Shown in Figure 5.8 is the plot generated from the Matlab program, the blue line is the plot of the y position and the red line is the plot of the x position. In Figure 5.9 is the same plot in Adams, the results are identical.

![Figure 5.8: Kinematic analysis of a fourbar (Matlab)](image-url)
5.1.2 Example: Dynamics of a Fourbar Mechanism

The same fourbar mechanism was taken from the previous example and used in a simple dynamic simulation. Each linkage was given a mass and inertia needed for the simulation. The system was then allowed to fall from rest with the only forces acting upon the system being gravity. The driving constraint was removed in order to under constrain the system, allowing the mechanism to free fall. The modified excel sheet can be seen in Figure 5.10.
The simulation was allowed to run for five seconds, which showed the crank never achieving a full rotation. Due to the system free falling the cranks motion was of a rocking motion, carrying the system with it. The animation can be seen in the frame taken in Figure 5.11. The forces that affect how fast the system moves and rotates are all generated by the masses and inertias of each link. The constraint forces keep the system connected together during the simulation. Gravity needs to be manually entered as shown in the excel sheet, but due to the nature of excel a relation to the mass can be easily made and auto fill the appropriate cell.
Figure 5.11: Animation of dynamic fourbar mechanism
Comparison with Adams was also done in order to verify the correct operation of the simulation. Unfortunately the inertia information for this system could not be exactly entered as describe and skewed some of the initial result. The graphs trends and patterns are identical from the Matlab simulation and Adams. The only difference is in the time it takes due to the intertias unable to be properly defined in Adams, they were only close. A redefinition of the system would be needed in order to properly define the system into Adams. The data is still useful showing the identical movement over time. The Matlab simulation is shown in Figure 5.12 and the Adams simulation is shown in Figure 5.13. Body one which was the input link was plotted where the blue lines represent the x displacement while the red lines represent the y displacement.

![Graph showing Matlab plot of dynamic fourbar](image)

Figure 5.12: Matlab plot of dynamic fourbar
5.1.3 Example: Vehicle Suspension - Ackermann analysis

This example shows an application in vehicle suspension systems where an arbitrary rack and pinion, typical of small FSAE race cars, is simulated. Ackermann steering is commonly implemented in low-mid speed cars to improve handling. When turning the wheels with a vehicle that has Ackermann steering implemented, the inside wheel will turn more than the outside wheel. The effect is increased with greater steering input, although when steering straight the wheels are still aligned straight. This is done due to the inside tire projects a smaller circle when in a turn, but also optimize tire geometry (slip angles) to achieve more grip at low speeds (Milliken, 1995). The main goal in this simulation will be to analyze how much the inside wheel turns versus the outside wheel.
Shown in Figure 5.14 is a SolidWorks sketch that was used to translate the geometry to be used in a planar kinematic study. All the details from body locations to joint coordinated can be measured from a CAD drawing such as the one shown. This information can be seen in the excel sheet used to define the system in Figure 5.15.

![Figure 5.14: Solidworks Sketch shown](image)

![Figure 5.15: Excel sheet for Ackermann simulation](image)
Figure 5.16: Matlab model shown

Figure 5.16 Shows the Matlab version of the system midway into the simulation, body numbers were drawn in, to make referencing to easier. Body three simulates the rack which is limited to sliding along the x axis only. Body four and five are the tie rods that transmit the motion to the uprights of the vehicle. Finally bodies one and two are the uprights which the holds the wheel and where typically the suspension is attached too. To simulate this system body three being the rack was slid horizontally and the angles of body one and two was recorded and saved into excel to be analyzed. An animation is shown in Figure 5.17.
Figure 5.17: Progression of Ackermann simulation
Figure 5.18: Ackerman shown, y-axis degrees, x axis translation of the rack

The data was recorded in Matlab and exported into excel, Figure 5.18 shows the manipulated data to achieve the data of interest. The x axis is the rack position and the y axis is the degrees of rotation, while the red line represents the angle of the inner wheel while the blue line represents the angle of the outer wheel. It is clear to see that as more steering input is inputted the inner wheel rotates more than the outer. The green line is the difference in degrees between the inner and outer wheel and shows the disparity as the rack travels. To see the body data and how the system was formulated see the appendix section of this paper.

5.1.4 Example: Dynamics of a 3D Crank Slider

This example is of a 3D crank slider system. A spatial dynamic analysis was ued to simulate the system. The difference between a normal planar crank slider and this example, is the crank rotates about the x axis instead of z, causing the crank to come out of x and y plane. A torque is applied to the crank to cause continued rotation. The slider moves due to being offset from the ground about being places one unit below the x axis. To see the system paramters the excel sheet is shown in Figure 5.19 and Figure 5.20 with all the body and constraint information.
**Figure 5.19: Body Information for 3D Crank Slider**

<table>
<thead>
<tr>
<th>BODY</th>
<th>Initial Position</th>
<th>Initial Rotation</th>
<th>Initial Velocity</th>
<th>INIT u</th>
<th>Mass</th>
<th>Principle Inertia</th>
<th>Body Forces</th>
<th>Body Torques</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>1 0 1 0 0 0 0</td>
<td>0 0 0 0 0 0</td>
<td>0 0 0 0 0 0</td>
<td>1.0</td>
<td>0.1 0.1 0.1</td>
<td>0 -32 0 0</td>
<td>2 0 0 0</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>2 1 0 0 0 0 0</td>
<td>0 0 323 1391</td>
<td>0 0 0 0 0 0</td>
<td>1.0</td>
<td>0.1 0.1 0.1</td>
<td>0 -32 0 0</td>
<td>0 0 0 0</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>3 -1 0 0 0 0 0</td>
<td>0 0 0 0 0 0</td>
<td>0 0 0 0 0 0</td>
<td>5.0</td>
<td>0.1 0.1 0.1</td>
<td>0 -160 0 0</td>
<td>0 0 0 0</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>4 -1 0 0 0 0 0</td>
<td>0 0 0 0 0 0</td>
<td>0 0 0 0 0 0</td>
<td>0.0</td>
<td>0.1 0.1 0.1</td>
<td>0 -160 0 0</td>
<td>0 0 0 0</td>
<td></td>
</tr>
</tbody>
</table>
In order to help visualize the 3D motion 6 frames were used to capture the animation for display in Figure 5.21. This shows the crank rotation along with the rotation and translation of the crank. The slider rotates due to a revolute joint that was used to attach the middle link to the slider.

The displacement along the x axis of the slider itself was plotted and shown in Figure 5.22. Because the simulation is a dynamic simulation where a torque was applied to the crank, the displacement of the slider is not uniform and changes as the system progresses through time. This effect can be seen at the very beginning as the system just begins to move.
Figure 5.21: Animation of the 3D crank slider
5.1.5 Example: Finite Element Method Integration

This simulation is different from the previous three, where the previous three used the general purpose features that were implemented in normal study, this example shows how code can be modified to include different fields of study. In this simulation a single pendulum is simulated where instead being attached to a rigid ground body, it is attached to the middle of a beam that is fixed at both ends.

Utilizing two dimensional beam elements equations from the Finite Element Method book (Logan, 2007) the equations were implemented into a separate Matlab function. To generate the forces that would act upon the bodies a stiffness matrix was used and multiplied by displacements of each node. The equation (5.1) shows how this can be accomplished. The function calculates the displacements and stiffness matrix, multiplies them to find the forces and then add the forces to the augmented matrix.
The equation (5.1) shows the stiffness matrix that was used in the simulation. The beam was constructed with twenty nodes and values for modulus of elasticity were 210 MPA, the area of beam was .1 m², while the length between each node was .2 meters and inertia of each element .000001 m⁴. Overall twenty nodes were used in the simulation and each end node was fixed.

\[
F = Kd
\]

(5.1)

The excel sheet was filled differently from normal, the first twenty bodies were treated as nodes for the 2D beam elements. With the rest of the sheet was filled normally for the swinging pendulum. The body and constraint information can be seen in Figure 5.23.
Shown in Figure 5.24 is the animation of the pendulum as it swings attached to the non-rigid beam. The pendulum weighed 3500Kg and in the second frame shown the figure shows the bending of the beam as the pendulum swings to the lowest point. The pendulum is two meters long and can be seen stretching further down due to the deflection of the beam.
Figure 5.24: Animation of pendulum attached to beam
Figure 5.25: Vertical displacement of center beam

In the Figure 5.25 the displacement plot of the middle node is shown. As the pendulum swung to its lowest point the displacement is shown to be about .1 meters. When the pendulum was at its highest point the beam was essentially unloaded and began to vibrate from the unloading. Unfortunately the beam elements cause the equations to become stiff, forcing the solver to take many time steps. Because of the many steps taken the program still saved all the data and took a few hours to simulate. The amount of data generated was also rather large which accumulated to be over one gigabyte. It is clear if rigid elements were to be implemented tighter control over the ODE algorithm and skipping steps that are saved would have to be implemented. As this was an example of how different elements can be injected into a system these improvements were not implemented.
CHAPTER 6
CONCLUSIONS AND RECOMMENDATIONS

6.1 Conclusions

The objective of this study was accomplished. These objectives were a multi-body simulation program was created that allowed the use of excel for inputting data, utilized Matlab for optimized user modifiable source and graphing support for analysis. The solver allows dynamic or kinematic simulations in two or three dimensions. Constraints were used to allow constrained systems with the most often seen constraints formulated and available in a general purpose fashion.

For spatial systems, the constraints were defined as they are in the code, with the inclusion of the Euler parameters for the kinematic analysis. The same formulations were used for planar systems with a similar but different definition for the constraints. These constraints were effectively integrated using tables provided from research. Each solver was able to use these matrices in order to run a simulation in either spatial or planar systems, that were kinematic or dynamics simulations each. Even though the spatial kinematic analysis is difficult to formulate for most analysis it is completed for academics and future work.

The modification and injection of elements from different fields outside of normal mechanics was shown in section 5.1.4 where a beam element was analyzed and in section 4.1.3 where a general purpose spring, damper and actuator forces where implemented. Due to the code with flexibility of being modified multiple fields of study can be included as long as a relation to the forces can be made.
It was shown in the examples in chapter five that these systems cannot only plot the paths and analyze certain aspects of a system, but can be done using CAD to find define system conditions and imported into Matlab. Also demonstrated in the Ackermann example was data that could be exported beyond Matlab and manipulated to analyze the variables of interest.

The overall time taken to simulate systems shown in this thesis and used for testing was relatively short. The actual time to run a simulation depends on the type of analysis and data that is needed in return. Typically the time to run a simulation is on the order of half to a tenth of the length of the simulation depending on the parameters. It was observed in the example using non-rigid beam elements that stiff systems can cause the time to run a simulation to reach hours for only a second of simulation time.

6.2 Recommendations and Future Research

The following recommendations for future research and improvement can be made:

1. Additional constraints added that can extend the capabilities solvers. For planar systems constraints such as gears, racks and cams could be added to broaden the capabilities. In spatial systems constraints such as U-joints and more composites joints other than Spherical-Spherical joint that was included.

2. Inclusion of different fields such as FEA for flexible bodies, control theory and even robotics such as microcomputer simulation of adaptive systems. These can be implemented through injection to the augmented matrix like the spring and dampers in this paper.

3. Allowing the use of equations or empirical data to provide non-linear spring or dampers. This would allow increase in realism a system to utilize empirical data measured, allowing a theoretical system match actual systems.
4. Improved handling of Euler parameters as drivers, allowing increased flexibility in spatial
dynamic and kinematic simulations.

5. Addition of electro-mechanical elements.

6. Control elements, such as motors and their characteristics.

7. Running simulations using inverse dynamics, commonly used in robotics and automation.

8. Real-Time simulations for use in flight or vehicle simulators, using input from a user during
simulation of the system.
REFERENCES
LIST OF REFERENCES


APPENDIX
APPENDIX
Matlab Code and File List (In Order)

Motionsim.m – Main Program
PreProcess2D.m – Process Planar Excel Sheet
PreProcess3D.m – Process Spatial Excel Sheet
KinematicSim2D.m – Planar Kinematic Solver
DynamicSim2D.m – Main file for planar dynamic solver
RKCalcDyn2D.m – Solver file for ODE45 Algorithm, Planar Dynamics
BuildC2D.m – Construct the Constraint Matrix
BuildQd2D.m – Build Planar Qd matrix
BuildCt2D.m – Build Planar Ct matrix
BuildCtt2D.m – Build Planar Ctt Matrix
SpringDamperDyn2D.m – Calculate Spring Forces for Planar Systems
DecompJointSys2D.m – Decompose Joint Data for Planar Systems
DecompDriver2D.m – Decompose Driver Constraints for Planar System
DecompSpringSys2D.m – Decompose SDA Data for Planar Systems
evalpoints2D.m – Evaluate Point Information for Planar Systems
KinematicSim3D.m – Spatial Kinematic Solver
DynamicSim3D.m – Spatial Dynamic Solver
RKCalcDyn3D.m – ODE45 Function
BuildC3D.m – Constraint Matrix
BuildCq3D.m – Jacobian Matrix
BuildQd3D.m – Qd Matrix
BuildCt.m – Build Ct Matrix
BuildCtt.m – Build Ctt Matrix
BuildC3DKin.m – Build Constraint Matrix with Euler Parameters
BuildCq3DKin.m – Build Jacobian Matrix with Euler Parameters
BuildQd3DKin.m – Build Qd Matrix with Euler Parameters
BuildCttKin.m – Build Ctt Matrix with Euler Parameters
SpringDamperDyn3D.m – Calculates Spring Forces
DecompJointSys.m – Decompose Joint Variables 3D systems
DecompDriver.m – Decompose Driver Variables 3D systems
DecompSpringSys.m – Decompose SDA Variables 3D systems
evalpoints.m – Evaluate Points for Spatial Analysis
CalcDof.m – Calculate DOF of Systems
calcG.m – Calculate G Matrix
calcL.m – Calculate L Matrix
Skew.m – 3 Dimensional Skew-Symmetric Matrix
Skew4.m – 4 Dimensional Skew-Symmetric Matrix
twoperpvect3D.m – Two Perpendicular Vectors to Input Vector
unitvector.m – Find Unit Vector and Magnitude
Amat_angles.m – Construct Transformation Matrix from Angles
Amat_Euler.m – Find Euler parameters from Transformation Matrix
%This is the main program, it is designed as a function to be called easily
%and begin the process. Everything is handled through the excel file,
%which this function imports, finds the pre/post and solver information
%from file, and then uses to run the simulation.
clc

%set paths to enable simulation
path(path, strcat(pwd,'\Solver'));
path(path, strcat(pwd,'\Solver\3DSIM'));
path(path, strcat(pwd,'\Solver\2DSIM'));
path(path, strcat(pwd,'\Models'));
path(path, strcat(pwd,'\Results'));
path(path, strcat(pwd,'\PreProcess'));
path(path, strcat(pwd,'\PostProcess'));

%import excel file, Filename should point to the file in the same folder.
%if option to load directly from .mat file is selected, will skip
%the excel import and load directly

tic
if runmat
    fprintf('%s', 'Reading Mat File:  ');
    load(strcat('Models\',filename),'system','data');
else
    fprintf('%s', 'Reading Excel File:  ');
    data = importdata(filename);
    %Run the selected preprocessor, using the function handle from excel
    sheet
    preprocessor = eval(strcat('@', cell2mat(data.textdata.INFO(22,3))));
    system = preprocessor(data, filename);
    system.directory = path;
    %save to .mat
    save(strcat('Models\',system.info.name), 'system', 'data');
end

time = toc;
disp([' Time:  ', num2str(time), 's']);

if solver
    %Run the selected Solver, using the function handle from excel sheet
    solver = system.solver;
    solver(system);end

%Run the selected Postprocessor, using the function handle from excel sheet
if post
    postprocess = system.post;
    postprocess([system.info.name, 'RES.mat']);
end
end
function [ system ] = PreProcess2D( xls, filename )

%Preprocessor to process 2D Dynamic and Kinematic Simulations
%from the 2D Excel Sheet.

%Begin processing excel sheet import, decide simulation type
%and obtain solver and post functions
dynamic = xls.data.INFO(1,1);
if dynamic == 1
    disp('----GENERIC 2D DYNAMIC ANALYSIS SELECTED -----');
    fprintf('%s', 'Processing Data:  ');
    solver = eval(['@', cell2mat(xls.textdata.INFO(24,3))]);
    post = eval(['@', cell2mat(xls.textdata.INFO(25,3))]);
elseif dynamic == 0
    disp('----GENERIC 2D KINEMATIC ANALYSIS SELECTED -----');
    fprintf('%s', 'Processing Data:  ');
    solver = eval(['@', cell2mat(xls.textdata.INFO(27,3))]);
    post = eval(['@', cell2mat(xls.textdata.INFO(28,3))]);
else
    disp('----WARNING UNKNOWN ANALYSIS TYPE SELECTED -----');
end

%Process the info sheet of the excel file - all simulation solver data
data = xls.data.INFO;
%Process overall simulation information
if dynamic
    time = data(6,7);
    enablestabilization = data(7,7);
else
    time = data(6,2);
    timestep = data(7,2);
    NRiter = data(8,2);
    accuracy = data(9,2);
end
%Set the pre-configured units that will be used later
unittype = data(2,1);
if unittype == 1
    units = struct('time', 'seconds',...
                   'pos', 'in.',...
                   'vel', 'in./s',...
                   'acc', 'in./s^2',...
                   'force', 'lb',...
                   'torque', 'lb-in');
elseif unittype == 2
    units = struct('time', 'seconds',...
                   'pos', 'ft.',...
                   'vel', 'ft./s',...
                   'acc', 'ft./s^2',...
                   'force', 'lb',...
                   'torque', 'lb-ft');
elseif unittype == 3
    units = struct('time', 'seconds',...
                   'pos', 'M.',...
                   'vel', 'M./s',...
'acc', 'M./s^2', ...
'force', 'N', ...
'torque', 'N-M');
end

% Process body information and its initial parameters
% bodies stores the count, body is struct storing all the intitial parameters, all body information located in sheet 1 of excel.

%set the ground body as body 1, no 0 allowed in indexes
data = xls.data.BODIES;
body(1) = struct(
    'R', [0;0],
    'PHIZ', 0 ,
    'Rd', [0;0],
    'w', 0,
    'Rdd', [0;0],
    'wd', [0],
    'M', 1,
    'I', 1,
    'Force', [0;0],
    'Torque', 0);

%begin processing all other bodies with the + 1 shift due to ground body being 1.
bodies = 1;
for x = 1:30
    if data(x,1) > 0
        bodies = bodies+1;
        body(bodies) = struct(
            'R', data(x, 2:3),
            'PHIZ', data(x,4)*pi/180 ,
            'Rd', data(x, 5:6),
            'w', data(x, 7),
            'Rdd', [0;0],
            'wd', [0],
            'M', data(x, 8),
            'I', data(x, 9),
            'Force', data(x, 10:11),
            'Torque', data(x, 12));
    end
end
fprintf('%s', 
    [' BODIES: ' num2str(bodies)]);

%proces the constraint sheet
data = xls.data.CONSTRAINTS;
%First check for any constraints are currently listed
joints = 0;
for x = 1:30
    if data(x,1) > 0
        joints = joints+1;
    end
end
%Finally process the joints into solver format if exist. if no joints exist
%create a dummy variable to avoid errors in packaging later.
if joints == 0
joint = []; 
else 
    joints = 0;
    for x = 1:30
        if data(x,1) > 0
            joints = joints + 1;
            joint(joints) = struct(...
                'Type', data(x,1), ...
                'Bodyi', data(x,2)+1,...
                'Bodyj', data(x,3)+1,...
                'pi', data(x, 4:5), ... 
                'qi', data(x, 6:7), ...
                'pj',data(x, 8:9),...
                'qj', data(x, 10:11),...
                'L', data(x,12));
        end
    end
end
alpha = data(1, 13);
beta = data(1, 14);
fprintf('%s', 
    %Driving Constraints sheet, will need txt data due to eval used 
%in the simulations

    data = xls.data.DRIVINGCONST;
txtdata = xls.textdata.DRIVINGCONST;
    numdriver = 0;
    for x = 1:30
        if data(x,1)>0 
            numdriver = numdriver+1;
        end
    end
    if numdriver == 0;
        drivers = [];
    else
        index = 0;
        for x = 1:30
            if data(x,1)>0 
                index = index+1;
                %process f(t), check for txt or number and save
                if isnan(data(x,3))
                    ft = txtdata(x+2,3);
                else
                    ft = num2str(data(x,3));
                end
                %process fd(t), check for txt or number and save
                if isnan(data(x,4))
                    fdt = txtdata(x+2,4);
                else
                    fdt = num2str(data(x,4));
                end
                %process fdd(t), check for txt or number and save
                if isnan(data(x,5))

103
fddt = txtdata(x+2,5);
else
    fddt = num2str(data(x,5));
end
coord = txtdata(x+2,2)';

drivers(index) = struct('body', data(x,1)+1,...
                        'coord', coord,...
                        'ft', ft,...
                        'fdt', fdt,...
                        'fddt',fddt);

end
end
end
fprintf('%s', [' DRIVER:  ' num2str(numdriver)]);

%Process the Spring/Damper/Actuator sheet (SDA)
data = xls.data.SDA;

%check for springs or dampers
springdampers = 0;
for x = 1:12
    if data(x,1)==1
        springdampers = springdampers+1;
    end
end

%Process the information, if there are no spring dampers create a dummy variable
if springdampers == 0
    springdamp = [ ];
else
    springdampers =0;
    for x = 1:12
        if data(x,1)==1
            springdampers = springdampers+1;
            springdamp(springdampers) = struct(...
                'Bodyi', data(x,2) + 1,...
                'Bodyj', data(x,3) + 1,...
                'pi', data(x, 4:5)',...
                'pj', data(x,6:7)',...
                'initL', data(x,8)',...
                'k', data(x,9)',...
                'c', data(x,10)',...
                'f', data(x,11)');
        end
    end
end
fprintf('%s', [' SDA:  ' num2str(springdampers)]);

%begin searching the post processing option, for graph etc.
data = xls.data.POST;
txtdata = xls.textdata.POST;
umpts = 0;
for x = 1:12
    if data(x,1)==1
numpts = numpts+1;
end
end
axissize = txtdata(2,6);
bodysize = data(2,10);
%Process the information, if there are no spring dampers create a dummy
%variable
if numpts == 0
    numpts = [];
    points = [];
else
    numpts =0;
    for x = 1:12
        if data(x,1) > 0
            numpts = numpts+1;
            points(numpts) = struct(...
                'num', data(x,1)',
                'body', data(x,2)+1,
                'vect', data(x, 3:4'));
        end
    end
end
end

%Organize the general info of the system into a info structure, noting
%the differences between dynamic and kinematic.
if dynamic
    info = struct(...
        'name', filename(1:length(filename)-5),...
        'bodies', bodies,...
        'time', time,...
        'joints', joints,...
        'stabilize', enablestabilization,...
        'springdampers', springdampers,...
        'drivers', numdriver,...
        'dynamic', dynamic,...
        'units', units,...
        'alpha', alpha,...
        'beta', beta,...
        'numpts', numpts,...
        'axissize', axissize,...
        'bodysize',bodysize);
else
    info = struct(...
        'name', filename(1:length(filename)-5),...
        'bodies', bodies,...
        'time', time,...
        'timestep', timestep,...
        'NRiter', NRiter,...
        'accuracy', accuracy,...
        'joints', joints,...
        'drivers', numdriver,...
        'dynamic', dynamic,...
        'units', units,...
function numpts, axissize, bodysize;
end

%Construct System variables and package into "system" structure. This will %be packaged the very same way the solver will solve the system, using the %system variable. Packaged into a single variable, the entire system %information can be passed from one function to another.
if dynamic
    system = struct(...
        'info', info,...
        'body', body,...
        'joint', joint,...
        'springdamp', springdamp,...
        'eval', 0,...
        'drivers', drivers,...
        'solver', solver,...
        'post', post,...
        'points', points);
else
    system = struct(...
        'info', info,...
        'body', body,...
        'joint', joint,...
        'eval', 0,...
        'drivers', drivers,...
        'solver', solver,...
        'post', post,...
        'points', points);
end

%Display info to screen
disp(' '); disp('File Processed');
end
function [ system ] = PreProcess3D( xls, filename )
%Preprocessor to process 3D Dynamic\kinematic simulations
%from the 3D Excel Sheet. The excel sheet is already imported into the xls
%variable, with filename as the name of the file for reference later on.
%All data from the excel sheet is put into a single structure variable that
%the solvers use to process the system

% Process the information sheet from excel file
dynamic = xls.data.INFO(1,1);
data = xls.data.INFO;

% checking for a kinematic or dynamic simulation, and set the appropriate
%solver and post processor.
if dynamic == 1
    disp('----GENERIC 3D DYNAMIC ANALYSIS SELECTED -----');
    fprintf('%s', 'Processing Data:  ');
solver = eval(['@', cell2mat(xls.textdata.INFO(24,3))]);
post = eval(['@', cell2mat(xls.textdata.INFO(25,3))]);
elseif dynamic == 0
    disp('----GENERIC 3D KINEMATIC ANALYSIS SELECTED -----');
    fprintf('%s', 'Processing Data:  ');
solver = eval(['@', cell2mat(xls.textdata.INFO(27,3))]);
post = eval(['@', cell2mat(xls.textdata.INFO(28,3))]);
else
    disp('----WARNING UNKOWN ANALYSIS TYPE SELECTED -----');
end

%process overall simulation information, and pick up the needed variables
%for dynamic or kinematic sims.
if dynamic
    time = data(6,7);
    enablestabilization = data(7,7);
else
    time = data(6,2);
timestep = data(7,2);
NRiter = data(8,2);
accuracy = data(9,2);
end

%Process the preconfigured units for post processor analysis
unittype = data(2,1);
if unittype == 1
    units = struct('time', 'seconds',...
        'pos', 'in.',...
        'vel','in./s',...
        'acc','in./s^2',...
        'force','lb',...
        'torque','lb-in');
elseif unittype == 2
    units = struct('time', 'seconds',...
        'pos', 'ft.',...
        'vel','ft./s',...
        'acc','ft./s^2',...
        'force','lb',...
elseif unittype == 3
    units = struct('time', 'seconds', ...
        'pos', 'M.', ...
        'vel', 'M/s', ...
        'acc', 'M/s^2', ...
        'force', 'N', ...
        'torque', 'N-M');
end

% Process body information and its initial parameters
% bodies stores the count, body is struct storing all the initial
% parameters, all body information located in sheet 1 of excel.
data = xls.data.BODIES;
body(1) = struct(...
    'R', [0;0;0], ...
    'P', [1;0;0;0], ...
    'Rd', [0;0;0], ...
    'w', [0;0;0], ...
    'M', 1,...
    'I', [1;1;1],...
    'Force', [0;0;0],...
    'Torque', [0;0;0]);

bodies = 1;
for x = 1:30
    if data(x,1) > 0
        bodies = bodies+1;
        % if using Euler parameters import directly and calculate e0,
        % if using angles convert to Euler parameters.
        if data(1,8)
            e0 = sqrt( -data(x, 5)^2 - data(x, 6)^2 - data(x, 7)^2 +1);
            p = [e0;data(x, 5:7)'];
        else
            p = Amat_angles( data(x, 5)*pi/180, data(x, 6)*pi/180, data(x, 7)*pi/180);
            p = Amat_Euler(p);
        end
        body(bodies) = struct(...
            'R', data(x, 2:4)',...
            'P', p ,...
            'Rd', data(x, 8:10)' ,...
            'w', data(x, 11:13)' ,...
            'Rdd', [0;0;0],...
            'wd', [0;0;0],...
            'M', data(x, 14),...
            'I', data(x, 15:17)',...
            'Force', data(x, 18:20)',...
            'Torque', data(x, 21:23)');
    end
end
fprintf('%s', [' BODIES: ' num2str(bodies)]);
%process the constraint sheet in the excel file
data = xls.data.CONSTRAINTS;

%First check for any constraints are currently listed
joints = 0;
for x = 1:30
    if data(x,1) > 0
        joints = joints+1;
    end
end
%Finally process the joints into solver format if exist. if no joints exist
%create a dummy variable to avoid errors in packaging later.
if joints == 0
    joint = [];
else
    joints = 0;
    for x = 1:30
        if data(x,1) > 0
            joints = joints + 1;
            joint(joints) = struct(...
                'Type', data(x,1)',...
                'Bodyi', data(x,2)+1,...
                'Bodyj', data(x,3)+1,...
                'pi', data(x, 4:6)',....
                'qi', data(x, 7:9)',....
                'pj',data(x, 10:12)',....
                'qj',data(x, 13:15)',....
                'L', data(x,16));
        end
    end
end

alpha = data(1, 17);
beta = data(1, 18);

fprintf('%s', [' CONSTRAINTS:  ' num2str(joints)]);

%process the driving/simple constraint sheet
data = xls.data.DRIVINGCONST;
txtdata = xls.textdata.DRIVINGCONST;
%check/count driving constraints
numdriver = 0;
for x = 1:30
    if data(x,1)>0
        numdriver = numdriver+1;
    end
end
if numdriver == 0;
    drivers = [];
else
    index = 0;
    for x = 1:30
        if data(x,1)>0
            index = index+1;
            ...
%process f(t), check for text or numerical input
if isnan(data(x,3))
    ft = txtdata(x-1,3);
else
    ft = num2str(data(x,3));
end
%process fd(t), check for text or numerical input
if isnan(data(x,4))
    fdt = txtdata(x-1,4);
else
    fdt = num2str(data(x,4));
end
%process fdd(t), check for text or numerical input
if isnan(data(x,5))
    fddt = txtdata(x-1,5);
else
    fddt = num2str(data(x,5));
end
coord = txtdata(x-1,2)'

drivers(index) = struct('body', data(x,1)+1,...
    'coord', coord,...
    'ft', ft,...
    'fdt', fdt,...
    'fddt', fddt);
end
end
end
fprintf('%s', [' DRIVER: ' num2str(numdriver)]);

%Process the Spring/Damper/Actuator sheet.
data = xls.data.SDA;

%check for any elements exist
springdampers = 0;
for x = 1:12
    if data(x,1)==1
        springdampers = springdampers+1;
    end
end

%Process the information, if there are no spring dampers create a dummy variable
if springdampers == 0
    springdamp = [];
else
    springdampers =0;
    for x = 1:12
        if data(x,1)==1
            springdampers = springdampers+1;
            springdamp(springdampers) = struct(...
                'Bodyi', data(x,2)+1,...
                'Bodyj', data(x,3)+1,...
                'pi', data(x, 4:6)',...
                'pj', data(x,7:9)',...
                'initL', data(x,10)',...)
'k', data(x,11)',...
'c', data(x,12)',...
'f', data(x,13)');
end
end
end
fprintf('%s', [' SDA: ' num2str(springdampers)]);

%Process the post
data = xls.data.POST;
txtdata = xls.textdata.POST;
umpts = 0;
for x = 1:12
  if data(x,1)==1
    numpts = numpts+1;
  end
end
axissize = txtdata(2,6);
bodysize = data(2,10);
%Process the information, if there are no spring dampers create a dummy %variable
if numpts == 0
  numpts = [];
  points = [];
else
  numpts =0;
  for x = 1:12
    if data(x,1) > 0
      numpts = numpts+1;
      points(numpts) = struct(...
        'num', data(x,1)',....
        'body', data(x,2)+1,...
        'vect', data(x, 3:5)');
    end
  end
end
end

%Organize system general information, will be used in solver
if dynamic
  info = struct(...
    'name', filename(1:length(filename)-5),...
    'bodies', bodies,...
    'time', time,...
    'joints', joints,...
    'stabilize', enablestabilization,...
    'springdampers', springdampers,...
    'drivers', numdriver,...
    'units', units,...
    'numpts', numpts,...
    'alpha', alpha,...
    'beta', beta,...
    'dynamic', dynamic,...
    'axissize', axissize,...
    'bodysize',bodysize);
else
info = struct(...
    'name', filename(1:length(filename)-5),...
    'bodies', bodies,...
    'time', time,...
    'timestep', timestep,...
    'NRiter', NRiter,...
    'accuracy', accuracy,...
    'joints', joints,...
    'drivers', numdriver,...
    'units', units,...
    'numpts', numpts,...
    'dynamic', dynamic,...
    'axissize', axissize,...
    'bodysize',bodysize);
end

%Construct System variables and package into "system" structure. This will
%be packaged the very same way the solver will solve the system, using the
%system variable. Packaged into a single variable, the entire system
%information can be passed from one function to another.

if dynamic
    system = struct(...
        'info', info,...
        'body', body,...
        'joint', joint,...
        'springdamp', springdamp,...
        'eval', 0,...
        'drivers',drivers,...
        'solver', solver,...
        'post',post,...
        'points', points);
else
    system = struct(...
        'info', info,...
        'body', body,...
        'joint', joint,...
        'eval', 0,...
        'drivers',drivers,...
        'solver', solver,...
        'post',post,...
        'points', points);
end

%Display info to screen
disp(' ');
disp('-File Processed-');
end
function [ ] = KinematicSim2D( system )
%2D kinematic solver.

%initialize variables that will be used to save data
coord = struct('q', [ ], 'qd', [ ], 'qdd', [ ]); error = struct('Cerror', [ ], 'Cvelerror', [ ]); data = struct('coord', coord, 'error', error, 'T', [ ], 'points', [ ]); %begin simulation

disp(['Beginning 2D Kinematic simulation: ', system.info.name])

%run DOF analysis, check for proper system definition
system = CalcDof( system, 3 ); disp(['Body Dof: ' num2str(system.info.totdof) ' - Constraint DOF: ' ... num2str(system.info.c dof) ' = SYSTEM DOF: ' num2str(system.info.dof)])

if system.info.dof ~= 0
    disp('WARNING SYSTEM DOF NOT PROPERLY CONSTRAINED, SOLVER WILL FAIL')
    reply = input('Continue? y/n','s');
    if reply ~= 'y'
        return
    end
end
tic
%place initial estimate into a form to be used by the matrix equations.
q = [ ]; qd = [ ];
for x = 1:system.info.bodies
    q = [q; system.body(x).R; system.body(x).PHIZ];
    qd = [qd; system.body(x).Rd; system.body(x).w];
end

%START SIMULATION
index = 0;
for t = 0:system.info.timestep:system.info.time
    index = index+1;
    %Begin newton raphson iterations. while iterating through time, index holds %the current iteration number that is an integer.
    delta = 5;
    count = 0;
    while delta > system.info.accuracy
        %reload the system information for current iteration
        for x = 1:system.info.bodies
            m = 1 + (x-1)*3 ;
            system.body(x).R = q(m:m+1);
            system.body(x).PHIZ = q(m+2);
            system.body(x).Rd = qd(m:m+1);

        end

end
\[
\begin{align*}
\text{system.body(x).w} &= \text{qd}(m+2); \\
\text{phi} &= \text{q}(m+2); \\
\text{system.body(x).A} &= \begin{bmatrix} \cos(\phi), & -\sin(\phi); \\
\sin(\phi), & \cos(\phi) \end{bmatrix}; \\
\end{align*}
\]

%%%%%%%%% precompute joint data to reduce calculations %%%%%%%%%%%%%
for x = 1:system.info.joints
    system.joint(x).solverinfo = DecompJointSys2D(system, x, 1);
end

%if t == 0 store the initial configuration used in some calculations
%and constraints
if t == 0
    system.initial = q;
end

%build the constraint and its jacobian matrices for calculation
C = BuildC2D(system,t);
Cq = BuildCq2D(system);

%solve for q and find current accuracy by obtaining the delta, take
%max absolute value of delta to make sure all constraints meet the
%desired level of accuracy
temp = q;
q = Cq\(-C)+q;
deltaq = temp-q;
delta = max(abs(deltaq));

%check to insue the NR iterations have not exceeded a set maximum,
%if they have then abort the iteration, something may be wrong.
%count is the variable keeping count of the iterations
if count > system.info.NRiter
    disp('WARNING MAXIMUM NR ITERATIONS REACHED');
    break
else
    count = count+1;
end

end
%After the NR iterations the positions and information of the system is
%set, a last update of the matrices allows for calculations of the
%velocity and accelerations

%Make sure jacobian has update to latest iteration Make sure everything
%is current so nothing has changed
C = BuildC2D(system,t);
Cq = BuildCq2D(system);
Qd = BuildQd2D(system);
Ct = BuildCt2D(system,t);

%Capture the intial updated configuration of the system
if t == 0
    system.initial = q;
end

%VELOCITY ANALYSIS- With the Position analysis done, using the
%information found, calculated the velocities is straight forward.
qd = Cq\(-Ct); 

%Acceleration Analysis 
qdd = Cq\Qd; 

%save data for output 
data.coord.q(index,:) = q; 
data.coord.qd(index,:) = qd; 
data.coord.qdd(index,:) = qdd; 

data.error.Cerror(index,:) = C;  
data.error.Cvelerror(index,:) = Cq*qd-Ct; 

data.points(index,:) = evalpoints2D( system, qdd ); 

data.T(index) = t; 
end 

%Save data to NAMERES.mat 
save(strcat('Results\',system.info.name,'RES.mat'),'system', 'data'); 

calctime = toc; 
disp(['Calculation Complete  -  Time: ', num2str(calctime), 's']); 
end
function [ ] = DynamicSim2D( system )
%Prepares the 2d Dynamic simulation for calculation, invokes the
%ODE45 in solver

%Setup initial conditions for system
fprintf('%s', strcat('Run Simulation...', system.info.name));
disp(' ')
%run DOF analysis
system = CalcDof( system, 3 );
disp(['Body Dof: ' num2str(system.info.totdof) ' - Constraint DOF: ' ...
     num2str(system.info.cdof) ' = SYSTEM DOF: ' num2str(system.info.dof)])

%Organize variables into initial conditions and vectors for quicker
%computations later.
tic
massvect = [];
forcevect = [];
initpos = [];
initvel = [];
for x = 1:system.info.bodies
    initpos = [initpos;system.body(x).R; system.body(x).PHIZ];
    initvel = [initvel;system.body(x).Rd; system.body(x).w];
    massvect = [massvect; system.body(x).M*[1;1]; system.body(x).I];
    forcevect = [forcevect; system.body(x).Force; system.body(x).Torque];
end
system.forcevect = forcevect;
system.massvect = massvect;
system.initial = [initpos;initvel];

%SIMULATE using either the ode45 substitute for different algorithms or use
%ODESET to control error and solver.
[T,Y] = ode45(@RKCalcDyn2D, [0,system.info.time], system.initial, [],[]);
ComputationDuration = toc;
disp(['  Complete:', num2str(ComputationDuration),'s']);

%Once data is complete begin processing and organize data to .mat file for
%later analysis. First step setup variables
tic
fprintf('%s', 'Evaluating Results:');
coord = struct('q', [], 'qd', [], 'qdd', []);
forces = struct('body', [], 'joint', [], 'Qd', []);
error = struct('Cerror',[],'Cvelerror',[]);
data = struct('coord', coord, 'forces', forces, 'error', error, 'T', T,
              'points', [],);
data.coord.q = Y(:,1:system.info.totdof);
data.coord.qd = Y(:,1+system.info.totdof:2*system.info.totdof);

%Evaluate system data, by re-running the RKCALCDYN Algorithm in order to
%pull out extra data.
system.eval = 1;
for x = 1:length(T)
    Yd(x) = RKCalcDyn2D(T(x), Y(x,:), [], system);
    data.coord.qdd(x,:) = Yd(x).output(system.info.totdof+1:2*system.info.totdof);
    data.forces.body(x,:) = Yd(x).Qe;
    data.forces.joint(x,:) = Yd(x).lamda;
    data.forces.Qd(x,:) = Yd(x).Qd;
    data.error.Cerror(x,:) = Yd(x).Cerror;
    data.error.Cvelerror(x,:) = Yd(x).Cvelerror;
    data.points(x,:) = Yd(x).points;
end

%save to RES.mat file
save(strcat('Results\',system.info.name,'RES.mat'),'system', 'data');
compduration = toc;
disp(['  Complete - ', num2str(compduration),'s']);
end
function [ yd ] = RKCalcDyn2D( t, y, flag, system )
% Main function to return accelerations for current timestep in ODE algorithm. Variables are passed [q, qd] and returned as [qd, qdd]. The velocities are passed through, while accelerations need to be calculated.

% Initialize and setup values to be used in the algorithm.
qd = y(system.info.totdof+1:2*system.info.totdof);
for x = 1:system.info.bodies
    % iterate through the different bodies and update each body information to current step
    m = 1 + (x-1)*3;
    system.body(x).R = y(m:m+1,1);
    system.body(x).PHIZ = y(m+2,1);
    m = m + system.info.totdof;
    system.body(x).Rd = y(m:m+1,1);
    system.body(x).w = y(m+2,1);
    % calculate the transformation matrices for each body
    phi = system.body(x).PHIZ;
    system.body(x).A = [cos(phi), -sin(phi); sin(phi), cos(phi)];
end

% precompute joint data to reduce calculations
for x = 1:system.info.joints
    system.joint(x).solverinfo = DecompJointSys2D( system, x, 1);
end

% CONSTRAINTS AND GAMMA
% Calculate the constraint matrices and their derivative matrices, assemble Qd for the RHS of constraint forces
C = BuildC2D(system, t);
Cq = BuildCq2D(system);
Qd = BuildQd2D(system);
Ct = BuildCt2D(system, t);
Ctt = BuildCtt2D(system, t);
if ~isempty(Qd)
    Qd = Qd + Ctt;
end

% Constraint Stabilization implementation
alpha = system.info.alpha;
beta = system.info.beta;
Cvelerror = (Cq*qd - Ct);
Cerror = C;
if ~isempty(Qd) && system.info.stabilize
    Qd = Qd - 2*alpha*Cvelerror - beta^2*C; % stabilization equation.
end

% Assembling augmented matrix
% assemble augmented form of the matrix, with the constraint and mass matrices into one large matrix
i = size(Cq,1);
MATRIX = [diag(system.massvect), Cq';
            Cq, zeros(i,i)];

% fill out bodie forces and moments, inject any elements such as spring
% dampers etc.
Qe = system.forcevect + SpringDamperDyn2D(system);

% append constraint RHS (Qd) to force vector (Qe) to form RHS of Augmented
% matrix equation
RHS = [Qe; Qd];

%Solve system of equations, find accelerations and constraint lambda's
ANS = MATRIX\RHS;

%%%%%%%%%%%%%%%%%%%%%%%%%%ASSEMBLE OUTPUT%%%%%%%%%%%%%%%%%%%%%%%%%%
% construct the output to be used in the ODE45 Algorithm, follows form of
% [qd; qdd]
OUTPUT = [qd;ANS(1:system.info.totdof)];

% is system is in ODE45 algorithm simply pass velocities, if data is being
% process send out solver data.
if system.eval
    points = evalpoints2D(system, ANS);
    yd = struct('points',points,'output', OUTPUT, 'Cerror', Cerror,...
                 'Cvelerror', Cvelerror,'Qe', Qe, 'Qd', Qd,...
                 'lamda', [ANS((system.info.bodies*3):length(ANS))',0]);
else
    yd = OUTPUT;
end
end
function [ C ] = BuildC2D( system, t )

%Construct C, constraint matrix
%preallocate matrix, and setup ground body since it is hardcoded
C = zeros(system.info.c dof,1);
C(1:3,1) = [system.body(1).R; system.body(1).PHIZ];

%begin filling out constraint matrix
index = 4;
for x = 1:system.info.joints

unpack constraint information
[i,j,Ai,Aj,Ri,Rj,si,sj,sgi,Si,Sj,Sqi,...
   phi,phi,j,Rd,wi,wj,Aphi,Aphi,j] = system.joint(x).solverinfo{:};

if system.joint(x).Type == 1   %Rev Joint
   C(index:index+1, 1) = Ri + Ai*si - Rj-Aj*sj;
   index = index +2;
elseif system.joint(x).Type == 2   %trans Joint
   %locate initial angles for use
   theti =system.initial((i*3));
   thetj =system.initial((j*3));

   %Convert sq into a perp vector
   ni = sqi - si;
   ni = [cos(90*pi/180), -sin(90*pi/180); sin(90*pi/180),
       cos(90*pi/180)]*ni;

   %calc some differences to make solving easier
   Rij = Ri + Sqi-Rj -Sj;
   Hi = Ai*(ni);
   %fill in constraint matrix
   C(index, 1) = Hi'*Rij;
   C(index+1, 1) = phi - phi,j - (theti-thetj);
   index = index +2;

end
end

%fill in driver data for the constraint matrix
for x = 1:system.info.drivers

[ body,R,X,Y,coord,ft,fdt,fddt ] = DecompDriver2D( system, x );

if coord == 'X'
   C(index,1) = X  - eval(ft);
elseif coord == 'Y'
   C(index,1) = Y - eval(ft);
elseif coord == 'PHI'
   C(index,1) = system.body(body).PHIZ-eval(ft);
end
index = index +1;
end
function [ Cq ] = BuildCq2D( system )

% Construct Cq, jacobian of the system

% Construct the jacobian matrix of the system

Cq = zeros(system.info.cdof - system.info.drivers, system.info.bodies*3);
Cq(1:3,1:3) = eye(3,3);

% Reconstruct the constraint matrix with initial guesses
index = 4;
for x = 1:system.info.joints

    [ i,j,Ai,Aj,Ri,Rj,si,sj,si,Si,Sj,Sqi,...
        phi,phij,Rd,j,Rdi,wi,wj,Aphi,Aphij ] = system.joint(x).solverinfo{:};

    if system.joint(x).Type == 1   % Rev Joint
        ni = 1+(i-1)*3;
        nj = 1+(j-1)*3;
        % for the first body i
        Cq(index:index+1, ni:ni+2) = [eye(2,2), Aphi*si];
        Cq(index:index+1, nj:nj+2) = [-eye(2,2), -Aphij*sj] ;
        index = index +2;
    elseif system.joint(x).Type == 2   % trans Joint
        ni = 1+(i-1)*3;
        nj = 1+(j-1)*3;

        % Transform sqi to a vector that's perpendicular to the axis
        normi = sqi - si;
        normi = [cos(90*pi/180), -sin(90*pi/180); sin(90*pi/180), cos(90*pi/180)]*normi;

        % process the normal vectors
        hi = (normi);
        Hi = Ai*(normi);

        % Computer the difference between body coordinates, find D
        Rij = Ri + Si - Rj - Sj;

        % Fill in joint constraints for body i
        Cq(index, ni:ni+2) = [Hi', Rij'*Aphi*hi+Hi'*Aphi*si];
        Cq(index+1, ni:ni+2) = [ 0,0,1];

        % Fill in joint constraints for body j
        Cq(index, nj:nj+2) = [-Hi', -Hi'*Aphij*sj];
        Cq(index+1, nj:nj+2) = [ 0,0,-1];

        index = index +2; % increments joint location in Cq
    end
end

% Process driving constraints
for x = 1:system.info.drivers

    [ body,R,X,Y,coord,ft,fdt,ffddt ] = DecompDriver2D( system, x );

end

121
m = 1+(body-1)*3;
if coord == 'X'
    Cq(index,m) =1;
elseif coord == 'Y'
    Cq(index,m+1) = 1;
elseif coord == 'PHI'
    Cq(index,m+2) = 1;
end
index = index +1;
end
end
function [ Qd ] = BuildQd2D( system )
%Construct the Qd matrix, that will be the vector of forces to solve for
%constraint equations

%preallocated Qd matrix, make sure long enough for addition with Ctt
Qd = zeros(system.info.cdof,1);

index = 4;
for x = 1:system.info.joints
    [ i,j,Ai,Aj,Ri,Rj,si,sj,si,si,Si,Sj,Sqi,...
        phi,phi,phi,phi,phi,phi,phi,phi,phi,...
        Rij,rij,Rdi,rij,rij,rij,rij,rij,...
        wi,wj,Aphii,Aphij,phii,phij,Rdij,Rdij,...
    ] = system.joint(x).solverinfo{:};

    if system.joint(x).Type == 1  %Rev Joint
        %Calculate the Qd for rev joint.
        Qd(index:index+1,1) = Ai*si*wi^2 - Aj*sj*wj^2;
        index = index +2;
    elseif system.joint(x).Type == 2  %trans Joint
        %compute normal vector of sqi
        ni = sqi - si;
        ni = [cos(90*pi/180), -sin(90*pi/180); sin(90*pi/180),
            cos(90*pi/180)]*ni;
        %compute differences for calculations
        hi = (ni);
        Hi = Ai*hi;

        Rij = Ri + Si-Rj -Sj;
        %compute the Qd terms for the RHS of the constraints
        Qd(index,1) = -2*wi*(Aphii*hi)'*(Rdi-Rdij)...
            -wi^2*(si'*hi-Rij'*Hi)...
            + 2*wi*wj*(Aphii*hi)'*Aphij*sj...
            -wj^2*Hi'*Sj;
        Qd(index+1) = 0;
        index = index +2;
    end
end
end
function [ Ct ] = BuildCt2D( system,t )

%%%Construct Ct

%Since constraints dont rely on t, it will mostly be used for driver %constraints

Ct = zeros(system.info.cdof,1);

index = system.info.cdof-system.info.drivers;
for x = 1:system.info.drivers
    [ body,R,X,Y,coord,ft,fdt,fddt ] = DecompDriver2D( system, x );

        if coord == 'X'
            Ct(index,1) = -eval(fdt);
        elseif coord == 'Y'
            Ct(index,1) = -eval(fdt);
        elseif coord == 'PHI'
            Ct(index,1) = -eval(fdt);
        end
    index = index +1 ;
end
end
function [ Ctt ] = BuildCtt2D( system, t )
%construct the matrix Ctt for second derivative of t
Ctt = zeros(system.info.cdof,1);

index = system.info.cdof-system.info.drivers+1;
for x = 1:system.info.drivers
    [ body,R,X,Y,coord,ft,fdt,fddt ] = DecompDriver2D( system, x );

    if strcmp(coord, 'X')
        Ctt(index,1) = eval(fddt);
    elseif strcmp(coord, 'Y')
        Ctt(index,1) = eval(fddt);
    elseif strcmp(coord, 'PHI')
        Ctt(index,1) = eval(fddt);
    end

    index = index +1 ;
end
end
function [ springforces ] = SpringDamperDyn2D( system)

%This function handles the spring/dampers that can be placed between 2
%points on 2 different bodies. The vector loop is calculated from a point
%on body i to a point on body j, the magnitude and unit vector is
%calculated. Using hooks law the force is calculated, multiplied by unit
%vector and applied to the points. If the point is a given distance from Cg
%the torque is also applied by the cross of the force vector and point
%vector.

%Damping is done a simular way, but using the velocity loop instead to
%obtain the unit vector and velocity. A damping coeficiant of c is used to
%control the gain of damping.

%Once everything is calculated, the extra forces etc generated and a matrix
%constructed to the same specifications as the right hand side matrix in
%order to add easily.

%create a matric with same dimensions as RHS for easy addition
springforces = zeros(system.info.bodies*3,1);

%Begin the loop between different springs
for x = 1:system.info.springdampers

%Calculated and insert all the body information into easy to read/use
%variables

[ i,j,pi,pj,initialL,K,c,Adj,Adi,Ai,Aj,...
   Ri,Rj,Rdi,Rdj,factuator ] = DecompSpringSys2D( system, x );

%calculate the spring position vector, and velocity vector
springvect = Rj + Aj*pj- Ri - Ai*pi;
springvectd = Rdj + Adj*pj  - Rdi - Adi*pi;
%Use the unitvector function that was created to find the magnitude and
%unit vector of the position and velocity.
[ springL,springuvect] = unitvector(springvect);
[ springvel,springuvectd] = unitvector(springvectd);

%Calculate the spring and damping forces from the magnitudes then
%multiply them by their corresponding unit vectors. Then add the 2
%vectors to a Force vector
spforcesp = (initialL-springL)*K;
spforcedamp = -c*springvel;
spforcevect = spforcesp*springuvect + spforcedamp*springuvectd+factuator;

%calculate forces/moments on body i, Ensuring the proper direction
forcesi = -spforcevect;
momenti = pi(1)*spforcevect(2)-pi(2)*spforcevect(1);
%calculate forces on body j, ensuring the proper direction
forcesj = spforcevect;
momentj = pj(1)*spforcevect(2)-pj(2)*spforcevect(1);

%Fill in the springforces matrix with their respective bodies, and then
%output from function for later use. With careful attention to the
%ground

m = 1+(i-1)*3;
n = 3+(i-1)*3;
springforces(m:n) = springforces(m:n) + [forcesi; momenti];

m = 1+(j-1)*3;
n = 3+(j-1)*3;
springforces(m:n) = springforces(m:n) + [forcesj; momentj];

end

end
DecompJointSys2D.m – Decompose Joint Data for Planar Systems

function [i,j,Ai,Aj,Ri,Rj,pi,pj,qi,Qi,phii,phij,Rdj,Rdi,wi,wj,Aphii,Aphij] = DecompJointSys2D( system,x,pack )
%converts all constraint data into easy to read and use data in constraint
%files, Mirrors equations for ease of debug and addition

%start with body data and gen information
i = system.joint(x).Bodyi;
j = system.joint(x).Bodyj;
pi = system.joint(x).pi;
pj = system.joint(x).pj;
qi = system.joint(x).qi;

%calculate for body i and j, different conditions
phii = system.body(i).PHIZ;
Ai = [cos(phii), -sin(phii); sin(phii), cos(phii)];
Ri = system.body(i).R;
Rdi = system.body(i).Rd;
wi = system.body(i).w;

phij = system.body(j).PHIZ;
Aj = [cos(phij), -sin(phij); sin(phij), cos(phij)];
Rj = system.body(j).R;
Rdj = system.body(j).Rd;
wj = system.body(j).w;

%calculate extra information
Aphii = [-sin(phii), -cos(phii); cos(phii), -sin(phii)];
Aphij = [-sin(phij), -cos(phij); cos(phij), -sin(phij)];
Pi = Ai*pi;
Qi = Ai*qi;
Pj = Aj*pj;

%packs into single variable if requested for faster computation
if pack
    tempi = i;
    i = {tempi,j,Ai,Aj,Ri,Rj,pi,pj,qi,Qi,phii,phij,Rdj,Rdi,wi,wj,Aphii,Aphij};
end
end
DecompDriver2D.m – Decompose Driver Constraints for Planar System

function [ body,R,X,Y,coord,ft,fdt,fddt ] = DecompDriver2D( system, x )
%decompose driver information to easy to read and use variables for use in
%filling constraint matrices.

body = system.drivers(x).body;
R = system.body(body).R;
X = R(1);
Y = R(2);

coord = system.drivers(x).coord;

ft = system.drivers(x).ft;
fdt = system.drivers(x).fdt;
fddt = system.drivers(x).fddt;
end
DecompSpringSys2D.m – Decompose SDA Data for Planar Systems

function [ i,j,pi,pj,initialL,K,c,Adj,Adi,Ai,Aj,Ri,Rj,Rdi,Rdj,f ] = DecompSpringSys2D( system, x )
% used in spring/damper systems in order to proved easy to use variables
i = system.springdamp(x).Bodyi;
j = system.springdamp(x).Bodyj;
pi = system.springdamp(x).pi;
pj = system.springdamp(x).pj;
initialL = system.springdamp(x).initL;
K = system.springdamp(x).k;
c = system.springdamp(x).c;
f = system.springdamp(x).f;

Ai = system.body(i).A;
Aj = system.body(j).A;
Ri = system.body(i).R;
Rj = system.body(j).R;
Rdi = system.body(i).Rd;
Rdj = system.body(j).Rd;

phi = system.body(i).PHIZ;
Adi = [-sin(phi), -cos(phi); cos(phi), -sin(phi)];
phi = system.body(j).PHIZ;
Adj = [-sin(phi), -cos(phi); cos(phi), -sin(phi)];
end
evalpoints2D.m – Evaluate Point Information for Planar Systems

function [ points ] = evalpoints2D( system,ANS )
%Compute the points of interest in the, use calculated data to find the
%position, velocity and accelerations.

q = []; qd = []; qdd = [];
%iterate through points of interest
for x = 1:system.info.numpts
    body = system.points(x).body;
    vect = system.points(x).vect;

    m = 1+(body-1)*3;
    q = [q,system.body(body).R + system.body(body).A*vect];
    qd = [qd,system.body(body).Rd +
         system.body(body).w*system.body(body).A*vect];
    qdd = [qdd,ANS(m:m+1) - system.body(body).w^2*system.body(body).A*vect
           + ANS(m+2)*system.body(body).A*vect ];
end
    points = [q',qd',qdd',0];
end
function [ ] = KinematicSim3D( system )

%KinematicSim 3d Solver

calculate degrees of freedom, and check for properly constraint system
disp(["Beginning 3D Kinematic simulation: ' system.info.name])
disp('')
%run DOF analysis
system  = CalcDof( system, 7 );
disp(["Body DoF: ' num2str(system.info.totdof) ' - Constraint DoF: ' ...
     num2str(system.info.cdof) ' = SYSTEM DoF: ' num2str(system.info.dof)])

if system.info.dof ~= 0
    disp('WARNING SYSTEM DOF NOT PROPERLY CONSTRAINTED, SOLVER WILL FAIL')
    reply = input('Continue? y/n','s');
    if reply ~= 'y'
        return
    end
end

tic
%place initial estimate into a form to be used by the matrix equations.
coord = struct('q', [], 'qd', [], 'qdd', []);
error = struct('Cerror',[],'Cvelerror',[]);
data = struct('coord', coord, 'error', error, 'T', [], 'points', []);

q = [];
qd = [];
for x = 1:system.info.bodies
    q = [q; system.body(x).R; system.body(x).P];
    qd = [qd; system.body(x).Rd; system.body(x).w];
end

%Begin Simulation
index = 0;
for t = 0:system.info.timestep:system.info.time
    index = index+1;
    %Using NR method, find the position at the current time step. delta is
%the current accuracy, count is the number of iterations past
    delta = 5;
    count = 0;
    while delta > system.info.accuracy
        %reload the system information for current iteration
        for x = 1:system.info.bodies
            m = 1 + (x-1)*7 ;
            n = 1 + (x-1)*6 ;
            system.body(x).R = q(m:m+2);
            system.body(x).P = q(m+3:m+6);
            system.body(x).Rd = qd(n:n+2);
            system.body(x).w = qd(n+3:n+5);
            system.body(x).A = euler_amat(system.body(x).P);
            system.body(x).SkewW = Skew(system.body(x).w);
        end
for x = 1:system.info.joints
    system.joint(x).solverinfo = DecompJointSys( system, x, 1 );
end

% if t == 0 store the initial configuration used in some calculations
% and constraints
if t == 0
    system.initial = q;
end

% build the constraint and its jacobian matrices for calculation
Ckin = BuildC3DKin(system,t);
Cqkin = BuildCq3DKin(system);

% solve for q and find current accuracy by obtaining the delta, take
% max absolute value of delta to make sure all constraints meet the
% desired level of accuracy
temp = q;
q = Cqkin\(-Ckin)+q;
deltaq = temp-q;
delta = max(abs(deltaq));

% check to insue the NR iterations have not exceeded a set maximum,
% if they have then abort the iteration, something may be wrong.
% count is the variable keeping count of the iterations
if count > system.info.NRiter
    disp(['WARNING MAXIMUM NR ITERATIONS REACHED: Time: ' num2str(t)]);
    disp(num2str( system.body(1).P'*system.body(1).P))
    break
else
    count = count+1;
end

end

% After the NR iterations the positions and information of the system is
% set, a last update of the matrices allows for calculations of the
% velocity and accelerations
for x = 1:system.info.bodies
    m = 1 + (x-1)*7 ;
    n = 1 + (x-1)*6 ;
    system.body(x).R = q(m:m+2);
    system.body(x).P = q(m+3:m+6);
    system.body(x).Rd = qd(n:n+2);
    system.body(x).w =  qd(n+3:n+5);
    system.body(x).A = euler_amat(system.body(x).P);
    system.body(x).SkewW = Skew(system.body(x).w);
end
for x = 1:system.info.joints
    system.joint(x).solverinfo = DecompJointSys( system, x, 1 );
end

% update some Euler constraints for next system
for x = 1:system.info.bodies
    system.body(x).Pd = qd(m+3:m+6);
end
%Calculate Matrices to be used for analysis
C = BuildC3DKin(system,t);
Cq = BuildCq3DKin(system);
Ct = BuildCt(system,t);
Qd = BuildQd3DKin(system);
Ctt = BuildCttKin(system,t);
%
%Capture the initial updated configuration of the system
if t == 0
    system.initial = q;
end
%
% VELOCITY ANALYSIS- With the Position analysis done, using the
% information found, calculated the velocities is straight forward.
qd = Cqkin\(-Ct);
%
%Solve for Accelerations
qdd = Cq\(Qd-Ctt);
%
% Compile data into standard format, in the data variable, also with raw
% data.
data.coord.q(index,:) =  q;
data.coord.qd(index,:) =  qd;
data.coord.qdd(index,:) =  qdd;

data.error.Cerror(index,:) = C;
data.error.Cvelerror(index,:) = Cq*qd-Ct;

data.points(index,:) = evalpoints2D( system, qdd );

data.T(index) = t;
end
save(strcat('Results\',system.info.name,'RES.mat'),'system', 'data');
calctime = toc;
disp(['Calculation Complete  -  Time: ', num2str(calctime), 's']);
end
function [ ] = DynamicSim3D( system, post, saveexcel )
%Dynamic 3D solver
fprintf('%s', strcat('Run Simulation...', system.info.name));

%Process the degrees of freedom
disp(' ')
system  = CalcDof( system, 6 );
disp([}'Body Dof: ' num2str(system.info.totdof) ' - Constraint DOF: ' ...'
num2str(system.info.c dof) ' = SYSTEM DOF: ' num2str(system.info.dof)]);

%setup initial condition and vectors for use in the solver
tic
massvect = [];
forcevect = [];
initpos = [];
initvel = [];
for x = 1:system.info.bodies
    initpos = [initpos;system.body(x).R; system.body(x).P];
    initvel = [initvel;system.body(x).Rd; system.body(x).w];
    massvect = [massvect; system.body(x).M*[1;1;1]; system.body(x).I];
    forcevect = [forcevect; system.body(x).Force; system.body(x).Torque];
end
system.forcevect = forcevect;
system.massvect = massvect;
system.initial = [initpos;initvel];

%SIMULATE using ODE45, use a different solver or ODESET to control ODE
%solver
[T,Y] = ode45(@RKCalcDyn3D, [0,system.info.time], system.initial, [],[], system);
ComputationDuration = toc;
disp([}'Complete:', num2str(ComputationDuration),'s']);

%Process the results suchs as forces, errors etc
tic
fprintf('%s', 'Evaluating Results:');
coord = struct('q', [], 'qd', [], 'qdd', []);
forces = struct('body', [], 'joint', [], 'Qd', []);
error = struct('Cerror',[],'Cvelerror',[]);
data = struct('coord', coord, 'forces', forces, 'error', error, 'T', T,
'points', []);
data.coord.q = Y(:,1:system.info.bodies*7);
data.coord.qd = Y(:,1+system.info.totdof:2*system.info.totdof);
data.coord.qdd = Yd(x).output(system.info.totdof+1:2*system.info.totdof);

%Evaluate extra data and store in data variables
system.eval = 1;
for x = 1:length(T)
    Yd(x) = RKCalcDyn3D(T(x), Y(x,:)', [], system);
    data.coord.qdd(x,:) = Yd(x).output(system.info.totdof+1:2*system.info.totdof);
data.forces.body(x,:) = Yd(x).Qe;
data.forces.joint(x,:) = Yd(x).lamda;
data.forces.Qd(x,:) = Yd(x).Qd;

data.error.Cerror(x,:) = Yd(x).Cerror;
data.error.Cvelerror(x,:) = Yd(x).Cvelerror;

data.points(x,:) = Yd.points;
end

%save data and system information
save(strcat('Results\',system.info.name,'RES.mat'),'system', 'data');
compduration = toc;
disp([' Complete - ', num2str(compduration),'s']);
end
function [ yd ] = RKCalcDyn3D(t, y, flag, system)
% 3D Dynamic simulation solver, input data is the current time and system
% information [q, qd], this function outputs the derivative [qd, qdd] for the
% ODE45 algorithm.

%%%%%%%%%%%%%INITIALIZE AND SETUP CURRENT ITERATION VALUES%%%%%%%%%%%%%
% fill out system info from previous RK update, the parameters needed
% for further calculation, as long as updating the system struct.
qd = y(1+7*system.info.bodies:7*system.info.bodies+6*system.info.bodies);
qdp = [];
for x = 1:system.info.bodies

% iterate through the different bodies. m updates the corresponding
% parameters in the y matrix to its particular body. where the
% coordinates, euler parameters, velocities, and angular velocities
% are stored for calculation
m = 1 + (x-1)*7;
system.body(x).R = y(m:m+2,1);  % 1:3
system.body(x).P = y(m+3:m+6,1);  % 4:7
m = 1+7*system.info.bodies + (x-1)*6;
system.body(x).Rd = y(m:m+2,1);  % 8:10
system.body(x).w = y(m+3:m+5,1);  % 11:13

% find the transformation matrix from helper functions
system.body(x).A = euler_amat(system.body(x).P);

% calculate the derivative of Euler parameters for output
system.body(x).Pd = .5*calcL(system.body(x).P)'*system.body(x).w;  % calc
qdp = [qdp;y(m:m+2,1); .5*calcL(system.body(x).P)'*system.body(x).w];

% precalculate the Skew of the angular velocity matrix to save
% computations
system.body(x).SkewW = Skew(system.body(x).w);
end

% pre compute joint information, to be used later in building matrices. Do
% this early, so not to repeat multiple times with other matrices saving
% time.
for x = 1:system.info.joints
    system.joint(x).solverinfo = DecompJointSys( system, x, 1 );
end

%%%%%%%%%%%%CONSTRAINTS%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Calculate the constraint matrices for the corresponding joints
% for the system. Seperated into a different fuction due to the number of
% constraints needed to be handeled
Cq = BuildCq3D( system, t );
Ctt = BuildCtt(system,t);
Qd = BuildQd3D(system,t);
C = BuildC3D(system,t);
Ct = BuildCt(system,t);

% calculate the Qd matrix for the RHS portion of the constraints, this is
% from the derivations of Cq*qdd=Qd, where Qd = -Ctt -d/dq(Cq*qd)Qd -
% 2*Cqt*qd.. where all but the -d/dq(Cq*qd)Qd is zero.
if ~isempty(Qd)
    Qd = Qd + Ctt;
end

% % constraint stabilization, Using the constraint stabilization techniques
% the alpha and beta can be set for each spherical joinig, and then added to
% its corresponding gamma for correction. Alpha and Beta controlling the
% system response
alpha = system.info.alpha;
beta = system.info.beta;
Cerror = C;
Cvelerror = (Cq*qd - Ct);
if ~isempty(Qd) && system.info.stabilize
    Qd = Qd - 2*alpha*Cvelerror - beta^2*C; % new method for later
end

% assemble augmented form of the matrix, with the constraint and mass
% matrices into one large matrix
i = size(Cq,1);
MATRIX = [diag(system.massvect), Cq';
          Cq, zeros(i,i)];

% Build the right hand side of the equation, or force matrix
Qe = system.forcevect + SpringDamperDyn3D(system);
RHS = [Qe; Qd];

% solve system of equations for rdd and wd, using the simple inverse method
% LU factorization or sparse matrices may later be used to expedite
% calculation
ANS = MATRIX\RHS;

% construct the output of the derivatives for continuing numerical
% approximation rd and wd calculated from the RK algorithm, while the new
% accelerations are placed into an OUTPUT variables.
OUTPUT = [qdp;ANS(1:system.info.bodies*6)];

% return to sender, OUTPUT is the variable given to yd to be sent out of the
% function, unless data is being extracted for analysis
if system.eval
points = evalpoints( system );
yd = struct('points',points, 'output', OUTPUT, 'Cerror', Cerror, 'Cvelerror', Cvelerror, 'Qe', Qe, 'Qd', Qd, 'lambda', [ANS((system.info.bodies*6+6):length(ANS))'],0));
else
   yd = [OUTPUT];
end
end
function [ C ] = BuildC3D( system,t )
%construct the constraint matrix for the 3D dynamic analysis

%preallocate constraint matrix
C = zeros(system.info.cdof,1);
C(1:6,1) = [system.body(1).R; system.body(1).P(2:4)];

%fill in constraint matrix
index = 7;
for x = 1:system.info.joints
    [ i,j, Ri, Rj, Rdi, Rdj, Ai,Aj, wi,wj, pi,pj,qi,qj,Pi,Pj,...
    Qi,Qj,Pdi,Pdj,Qdi,Qdj,pdi,pdj,qdi,qdj ] =
    system.joint(x).solverinfo{:};
    if system.joint(x).Type == 1   %spherical
        C(index:index+2, 1) = Ri + Pi - Rj-Pj;
        index = index +3;
    elseif system.joint(x).Type ==2 %rev
        %calculate the perpindicular vectors
        Si = Ai*(pi - qi);
        Sj = Aj*(pj - qj);
        [ W_vect, V_vect ] = twoperpvect3D( Si );
        %fill in constraints
        C(index:index+2, 1) = Ri + Pi - Rj-Pj;
        C(index+3,1) = W_vect'*Sj;
        C(index+4,1) = V_vect'*Sj;
        index = index +5;
    elseif system.joint(x).Type ==3 %cyl
        %calculate the vectors and perpindicular vectors, and the loop
        Si = Ai*(pi - qi);
        Sj = Aj*(pj - qj);
        Pij = Rj + Aj*pj -Ri-Ai*pi;
        [ W_vect, VVect ] = twoperpvect3D( Si );
        C(index,1) = W_vect'*Pij;
        C(index+1,1) = V_Vect'*Pij;
        C(index+2,1) = W_vect'*Sj;
        C(index+3,1) = V_Vect'*Sj;
        index = index +4;
    elseif system.joint(x).Type ==4 %Spherical-Spherical
        %calculate the distance vector
        d = Rj+ Pj-Ri - Pi;
        %dd = Rdj + Pdj - Rdi - Pdi ;
        L = system.joint(x).L;
        C(index,1) = d'*d-L^2;
        index = index +1;
    elseif system.joint(x).Type ==5 %prismatic
        %calculate needed vectors
        Si = Ai*(pi - qi);
Sj = Aj*(pj - qj);

Pij = Rj + Aj*pj -Ri-Ai*pi;
[ W_vect, V_vect ] = twoperpvect3D( Si );
[ Wj_vect, Vj_vect ] = twoperpvect3D( Sj );

%perp type 2
C(index,1) = W_vect'*Pij;
C(index+1,1) = V_vect'*Pij;

%perp type 1
C(index+2,1) = W_vect'*Sj;
C(index+3,1) = V_vect'*Sj;
C(index+4,1) = V_vect'*Wj_vect;
index = index +5;
end
end

%Now append driving constraints
for x = 1:system.info.drivers
    [coord,R, X,Y,Z,ft,fdt,fddt,body] = DecompDriver( system, x );
    e1 = system.body(body).P(4);
e2 = system.body(body).P(3);
e3 = system.body(body).P(2);

    if coord == 'X'
        C(index,1) = X  - eval(ft);
    elseif coord == 'Y'
        C(index,1) = Y  - eval(ft);
    elseif coord == 'Z'
        C(index,1) = Z  - eval(ft);
    elseif coord == 'PHIZ'
        C(index,1) =  e1 - eval(ft);
    elseif coord == 'PHIX'
        C(index,1) =  e2 - eval(ft);
    elseif coord == 'PHIY'
        C(index,1) =  e3 -eval(ft);
    end

    index = index +1;
end
end
function [ Cq ] = BuildCq3D( system,t )
%Build the jacobian matrix for 3D Dynamic Solver

%preallocated jacobian matrix and body info
Cq = zeros(system.info.cdof - system.info.drivers, system.info.bodies*6);
Cq(1:6,1:6) = eye(6,6);

%build the rest of the matrix
index = 7;
for x = 1:system.info.joints
    %%% Load common variables between joints into simple variables for
    %%% calculation
    [ i,j, Ri, Rj, Rdj, Ai,Aj, wi,wj, pi,pj,qi,qj,Pi,Pj,...
        Qi,Qj,Pdi,Pdj,Qdi,Qdj,pdi,pdj,qdi,qdj ] =
        system.joint(x).solverinfo{:};

    if system.joint(x).Type == 1  %for joint type of spherical
        m = 1+(x-1)*3;
        ni = 1+(i-1)*6;
        nj = 1+(j-1)*6;
        %for body i
        Cq(index:index+2,ni:ni+5) = [eye(3,3), -Skew(Pi)*Ai];
        %for body j
        Cq(index:index+2,nj:nj+5) = [-eye(3,3), Skew(Pj)*Aj];
    end

    elseif system.joint(x).Type == 2 %rev
        ni = 1+(i-1)*6;
        nj = 1+(j-1)*6;
        Si = Ai*(pi - qi);
        Sj = Aj*(pj - qj);
        [ W_vect, V_vect ] = twoperpvect3D( Si );

        %for body i
        Cq(index:index+2,ni:ni+5) = [eye(3,3), -Ai*Skew(pi)]; %spherical
        Cq(index+3,ni:ni+5) = [0,0,0,-Sj'*Skew(W_vect)*Ai]; %perp type 1
        Cq(index+4,ni:ni+5) = [0,0,0,-Sj'*Skew(V_vect)*Ai]; %perp type 1
        %for body j
        Cq(index:index+2,nj:nj+5) = [-eye(3,3), Skew(Pj)*Aj];
        Cq(index+3,nj:nj+5) = [0,0,0,-W_vect'*Skew(Sj)*Aj];
        Cq(index+4,nj:nj+5) = [0,0,0,-V_vect'*Skew(Sj)*Aj];

    end

    elseif system.joint(x).Type == 3 %cylindrical
        ni = 1+(i-1)*6;
        nj = 1+(j-1)*6;
        Si = Ai*(pi - qi);
        Sj = Aj*(pj - qj);

        index = index + 5;
    end
end
\[ P_{ij} = R_j + A_j p_j - R_i - A_i p_i; \]

\[ [ W_{\text{vect}}, V_{\text{vect}} ] = \text{twooperpvect3D}( S_i ); \]

\%for body i
\%Perp Type two
Cq(index,ni:ni+5) = \[-W_{\text{vect}}',-(Pij + Pi)'*\text{Skew}(W_{\text{vect}})*A_i\];
Cq(index+1,ni:ni+5) = \[-V_{\text{vect}}',-(Pij + Pi)'*\text{Skew}(V_{\text{vect}})*A_i\];
\%Perp Type one
Cq(index+2,ni:ni+5) = [0,0,0,-Sj'*\text{Skew}(W_{\text{vect}})*A_i];
Cq(index+3,ni:ni+5) = [0,0,0,-Sj'*\text{Skew}(V_{\text{vect}})*A_i];

\%for body j
\%Perp Type two
Cq(index,nj:nj+5) = \[W_{\text{vect}}', - W_{\text{vect}}'*\text{Skew}(Pj)*A_j\];
Cq(index+1,nj:nj+5) = \[V_{\text{vect}}', -V_{\text{vect}}'*\text{Skew}(Pj)*A_j\];
\%Perp Type one
Cq(index+2,nj:nj+5) = [0,0,0,-W_{\text{vect}}'*\text{Skew}(Sj)*A_j];
Cq(index+3,nj:nj+5) = [0,0,0,-V_{\text{vect}}'*\text{Skew}(Sj)*A_j];

index = index + 4;

elseif system.joint(x).Type == 4 \%for joint type of S-S
ni = 1+(i-1)*6;
nj = 1+(j-1)*6;

\%for body i
Cq(index,ni:ni+5) = \[-2*d', 2*d'*A_i*\text{Skew}(p_i)\];
\%for body j
Cq(index,nj:nj+5) = \[2*d', -2*d'*A_j*\text{Skew}(p_j)\];

index = index + 1;

elseif system.joint(x).Type == 5 \%prismatic
ni = 1+(i-1)*6;
nj = 1+(j-1)*6;

Si = A_i*(p_i - q_i);
Sj = A_j*(p_j - q_j);

Pij = R_j + A_j p_j - R_i - A_i p_i;
[ W_{\text{vect}}, V_{\text{vect}} ] = \text{twooperpvect3D}( S_i );
[ W_{j}\_\text{vect}, V_{j}\_\text{vect} ] = \text{twooperpvect3D}( S_j );

\%for body i
\%Perp Type two
Cq(index,ni:ni+5) = \[-W_{\text{vect}}',-(Pij + Pi)'*\text{Skew}(W_{\text{vect}})*A_i\];
Cq(index+1,ni:ni+5) = \[-V_{\text{vect}}',-(Pij + Pi)'*\text{Skew}(V_{\text{vect}})*A_i\];
\%Perp Type one
Cq(index+2,ni:ni+5) = [0,0,0,-Sj'*\text{Skew}(W_{\text{vect}})*A_i];
Cq(index+3,ni:ni+5) = [0,0,0,-Sj'*\text{Skew}(V_{\text{vect}})*A_i];
Cq(index+4,ni:ni+5) = [0,0,0,-W_{j}\_\text{vect}'*\text{Skew}(V_{\text{vect}})*A_i];

\%for body j
\%Perp Type two
Cq(index,nj:nj+5) = [W_vect', - W_vect'*Skew(Pj)*Aj];
Cq(index+1,nj:nj+5) = [V_vect', -V_vect'*Skew(Pj)*Aj];
%Perp Type one
Cq(index+2,nj:nj+5) = [0,0,0,-W_vect'*Skew(Sj)*Aj];
Cq(index+3,nj:nj+5) = [0,0,0,-V_vect'*Skew(Sj)*Aj];
Cq(index+4,nj:nj+5) = [0,0,0,-V_vect'*Skew(Wj_vect)*Aj];

index = index + 5;
end
end

%driving constraints for Cq to be added in.
for x = 1:system.info.drivers
    [coord,R, X,Y,Z,ft,fdt,fddt,body] = DecompDriver( system, x );
    m = 1+(body-1)*6;
    if coord == 'X'
        Cq(index,m) = 1;
    elseif coord == 'Y'
        Cq(index,m+1) = 1;
    elseif coord == 'Z'
        Cq(index,m+2) = 1;
    elseif strcmp(coord, 'PHIZ')
        Cq(index,m+5) = 1;
    elseif strcmp(coord, 'PHIY')
        Cq(index,m+4) = 1;
    elseif strcmp(coord, 'PHIX')
        Cq(index,m+3) = 1;
    end
    index = index + 1;
end
end
function [ Qd ] = BuildQd3D( system,t )
%Build the Qd matrix for the 3D Dynamic simulation

%preallocated Qd matrix
Qd =zeros(system.info.c dof,1);

%fill out Qd matrix
index = 7;
for x = 1:system.info.joints
    %%% Load common variables between joints into simple variables for
    %%% calculation
    [ i,j, Ri, Rj, Rdj, Ai,Aj, wi,wj, pi,qi,qj,pi,Pj,...
     Qj,Pj, Pdj,Qdj,pqi,qdj,SkewWi,SkewWj ] =
     system.joint(x).solverinfo{ };

    m = 1+(x-1)*3;
    n = 3 + (x-1)*3;
    if system.joint(x).Type == 1  %for joint type of spherical
        SWi = Skew(Ai*wi);
        SWj = Skew(Aj*wj);
        Qd(index:index+2,1) = -Ai * SkewWi *SkewWi *(pi) +  Aj * SkewWj
        *SkewWj* (pj);
        index = index +3;
    elseif system.joint(x).Type == 2  %for joint type of rev
        %Compute extra vectors and simplify
        Sj = Aj*(pj - qj);
        Sdj = Aj*(pdj - qdj);
        si = (pi - qi);
        [ w_vect, v_vect ] = twoperpvect3D( si );
        W_vect=Ai* w_vect;
        V_vect = Ai*v_vect;
        Wd_vect = Ai*Skew(wi)*w_vect;
        Vd_vect = Ai*Skew(wi)*v_vect;

        %fill in Qd matrix
        Qd(index:index+2,1) = -Ai * SkewWi *(pdi) +  Aj * SkewWj * (pdj);
        Qd(index+3,1) = -2*Wd_vect'*Sdj +
        Wd_vect'*Ai*SkewWi*Ai'*Sj+Sdj'*Aj*SkewWj*Aj'*W_vect; %wi
        Qd(index+4,1) = -2*Vd_vect'*Sdj +
        Vd_vect'*Ai*SkewWi*Ai'*Sj+Sdj'*Aj*SkewWj*Aj'*V_vect; %vi
        index = index +5;
    elseif system.joint(x).Type == 3
        %compute extra variables for Qd matrix
        Sj = Aj*(pj - qj);
        Sdj = Aj*(pdj - qdj);
        si = (pi - qi);
        Plj = Rj + Aj*pj -Ri-Ai*pi;
        Pdj = Rdj + Aj*SkewWj*pj -Rdi-Ai*SkewWi*pi;
        [ w_vect, v_vect ] = twoperpvect3D( si );
        W_vect=Ai* w_vect;
        V_vect = Ai*v_vect;
        wd_vect = SkewWi*w_vect;
        vd_vect = SkewWi*v_vect;
        Wd_vect = Ai*wd_vect;
Vd_vec = Ai*vd_vec;

%fill in Qd matrix
%Type 2 perp
Qd(index+1,1) = -2*Pdij'*Wd_vec - Pij'*Ai*SkewWi*wd_vec +
W_vec'**(Ai*SkewWi*pdj - Aj*SkewWj*pdj); %wi
Qd(index+2,1) = -2*Pdij'*Vd_vec - Pij'*Ai*SkewWi*vd_vec +
V_vec'**(Ai*SkewWi*pdj - Aj*SkewWj*pdj);
%Type one perp
Qd(index+3,1) = -2*Wd_vec'*Sdj +
Wd_vec'*Ai*SkewWi*Ai'*Sj+Sdj'*Aj*SkewWj*Aj'*W_vec;
Qd(index+4,1) = -2*Vd_vec'*Sdj +
Vd_vec'*Ai*SkewWi*Ai'*Sj+Sdj'*Aj*SkewWj*Aj'*V_vec;

index = index +4;
elseif system.joint(x).Type == 5
%calculate the extra vectors
Sj = Aj*(pj - qj);
Sdj = Aj*(pdj - qdj);
si = (pi - qi);
sj = (pj - qj);
Pij = Rj + Aj*pj -Ri-Ai*pi;
Pdij = Rdj + Aj*SkewWj*pj -Rdi-Ai*SkewWi*pi;
[wj_vec, vj_vec] = twoperpvect3D( si );
[wj_vec, vj_vec] = twoperpvect3D( sj );
Wd_vec = Ai*wd_vec;
Vd_vec = Ai*vd_vec;
wdj_vec = SkewWj*wj_vec;
Wdj_vec = Aj*wdj_vec;
Wj_vec = Aj*wj_vec;

%fill in Qd matrix
%Type 2 perp
Qd(index+1,1) = -2*Pdij'*Wd_vec - Pij'*Ai*SkewWi*wd_vec +
W_vec'**(Ai*SkewWi*pdj - Aj*SkewWj*pdj); %wi
Qd(index+2,1) = -2*Pdij'*Vd_vec - Pij'*Ai*SkewWi*vd_vec +
V_vec'**(Ai*SkewWi*pdj - Aj*SkewWj*pdj);
%Type one perp
Qd(index+3,1) = -2*Wd_vec'*Sdj +
Wd_vec'*Ai*SkewWi*Ai'*Sj+Sdj'*Aj*SkewWj*Aj'*W_vec;
Qd(index+4,1) = -2*Vd_vec'*Sdj +
Vd_vec'*Ai*SkewWi*Ai'*Sj+Sdj'*Aj*SkewWj*Aj'*V_vec;

elseif system.joint(x).Type == 4
%for joint type of S-S
%calculate the D vector and derivative
d = Rj+ Pj-Ri - Pi;
dd = Rdj + Pdj - Rdi - Pdi ;
%fill in Qd matrix
Qd(index+1,1) = -2*dd'*dd+2*d'**(Ai*SkewWi*pdj - Aj*SkewWj*pdj);
index = index +1;
function [ Ct ] = BuildCt( system,t )
%Build the Ct matrix for 3d Dynamic simulations
Ct = zeros(system.info.cdof,1);

index = system.info.cdof-system.info.drivers;
for x = 1:system.info.drivers
    [coord,R, X,Y,Z,ft,fdd,fddt,body] = DecompDriver( system, x );

    Ct(index,1) =-eval(fdt);
    index = index +1 ;
end
end
function [ Ctt ] = BuildCtt( system,t )
%Build Ctt matrix for 3D Dynamic Simulation
Ctt = zeros(system.info.cdof,1);

index = system.info.cdof - system.info.drivers;
for x = 1:system.info.drivers
    [coord,R,X,Y,Z,ft,fdt,fddt,body] = DecompDriver( system, x );
    Ctt(index,1) = eval(fddt);
    index = index +1 ;
end
end
BuildC3DKin.m – Build Constraint Matrix with Euler Parameters

function [ C ] = BuildC3DKin( system, t )
%Constraint matrix for 3d Kinematic simulations using Euler parameters

%preallocate constraint matrix
C = zeros(system.info.cdof,1);
C(1:6,1) = [system.body(1).R; system.body(1).P(2:4)];

%Fill constraint matrix
index = 7;
for x = 1:system.info.joints
    [ i, j, Ri, Rj, Rdi, Rdj, Ai, Aj, wi, wj, pi, pj, qi, qj, Pi, Pj, ... 
      Qi, Qj, Pdi, Pdj, Qdi, Qdj, pdi, pdj, qdi, qdj ] = 
    system.joint(x).solverinfo{:};
    if system.joint(x).Type == 1 %spherical
        C(index:index+2, 1) = Ri + Pi - Rj-Pj;
        index = index +3;
    elseif system.joint(x).Type == 2 %rev
        %calculate vectors
        Si = Ai*(pi - qi);
        Sj = Aj*(pj - qj);
        [ W_vect, V_vect ] = twoperpvect3D(Si);
        %fill in matrix
        C(index,1) = W_vect'*Pij;
        C(index+1,1) = V_vect'*Pij;
        C(index+2,1) = W_vect'*Sj;
        C(index+3,1) = V_vect'*Sj;
        index = index +4;
    elseif system.joint(x).Type == 3 %cyl
        %calculate extra variables
        Si = Ai*(pi - qi);
        Sj = Aj*(pj - qj);
        Pij = Rj + Aj*pj -Ri-Ai*pi;
        [ W_vect, V_vect ] = twoperpvect3D(Si);
        %fill in matrix
        C(index+1,1) = W_vect'*Pij;
        C(index+2,1) = V_vect'*Pij;
        C(index+3,1) = W_vect'*Sj;
        C(index+4,1) = V_vect'*Sj;
        index = index +5;
    elseif system.joint(x).Type == 4 %ss
        %calculate the vectors
        d = Rj+ Pj-Ri - Pi;
        %dd = Rdj + Pdj - Rdi - Pdi ;
        L = system.joint(x).L;
        %fill in constraint matrix
        C(index,1) = d'*d-L^2;
        index = index +1;
    elseif system.joint(x).Type == 5 %prismatic
%calculate extra vectors
Si = Ai*(pi - qi);
Sj = Aj*(pj - qj);
Pij = Rj + Aj*pj -Ri-Ai*pi;
[ W_vect, V_vect ] = twoperpvect3D( Si );
[ Wj_vect, Vj_vect ] = twoperpvect3D( Sj );

%fill in constraint matrix
%perp type 2
C(index,1)        = W_vect'*Pij;
C(index+1,1)        = V_vect'*Pij;
%perp type 1
C(index+2,1)        = W_vect'*Sj;
C(index+3,1)        = V_vect'*Sj;
C(index+4,1)        = V_vect'*Wj_vect;
index = index +5;
end
end

%Fill in the final constraint, for the jacobian that is the euler
%parameters constraints to make bodies form a 7x7 matrix for fully
%constrained
for x = 1:system.info.bodies
    C(index,1) = system.body(x).P'*system.body(x).P - 1;
    index = index + 1;
end

%Now fill in driving constraints
for x = 1:system.info.drivers
    [coord,R, X,Y,Z,ft,fdt,fddt,body] = DecompDriver( system, x );
    PHI2 = system.body(body).P(4);
    PHIY = system.body(body).P(3);
    PHIX = system.body(body).P(2);
    if coord == 'X'
        C(index,1) = X  - eval(ft);
    elseif coord == 'Y'
        C(index,1) = Y  - eval(ft);
    elseif coord == 'Z'
        C(index,1) = Z  - eval(ft);
    elseif strcmp(coord, 'e1')
        e1 = eval(ft);
        C(index,1) = PHIX - e1;
    elseif strcmp(coord, 'e2')
        e2 = eval(ft);
        C(index,1) = PHIY - e2;
    elseif strcmp(coord, 'e3')
        e3 = eval(ft);
        C(index,1) = PHI2 - e3;
    end
    index = index +1;
end
end
BuildCq3DKin.m – Build Jacobian Matrix with Euler Parameters

function [ Cq ] = BuildCq3DKin( system,t )
%Calculate jacobian based on Euler parameters for 3d Kinematic solver

%preallocate matrix
Cq = zeros(system.info.c dof, system.info.bodies*7);
Cq(1:3,1:3) = eye(3,3);
Cq(4:6,5:7) = eye(3,3);

%fill in matrix
index = 7;
for x = 1:system.info.joints
  %%% Load common variables between joints into simple variables for
  %%% calculation
  [ i, j, Ri, Rj, Ai, Aj, wi, wj, pi, pj, qi, qj, Pi, Pj,...
   Qj, Qj, Pdi, Pdj, Qi, Qdj, pdi, pdj, qdi, qdj ] =
   system.joint(x).solverinfo();

  %process constraints
  if system.joint(x).Type == 1  %for joint type of spherical
    ni = 1+(i-1)*7;
    nj = 1+(j-1)*7;
    ei = system.body(i).P;
    ej = system.body(j).P;

    %for body i
    Ci = 2*(calcG(ei)*skew4(pi) + pi*ei');
    Cq(index:index+2,ni:ni+6) = [eye(3,3), Ci]; %spherical portion

    %for body j
    Cj = 2*(calcG(ej)*skew4(pj) + pj*ej');
    Cq(index:index+2,nj:nj+6) = [-eye(3,3), -Cj]; %spherical portion

    index = index+3;
  elseif system.joint(x).Type == 2 %rev
    ni = 1+(i-1)*7;
    nj = 1+(j-1)*7;
    ei = system.body(i).P;
    ej = system.body(j).P;

    %load up variables needed for calcs
    si = (pi - qi);
    sj = (pj - qj);
    [ w_vect, v_vect ] = twoperpvect3D( si );
    Si = Ai*(pi - qi);
    Sj = Aj*(pj - qj);
    W_vect = Ai*w_vect;
    V_vect = Ai*v_vect;

    %for body i
\[
Ci = 2 \times (\text{calcG}(ei) \times \text{skew4}(pi) + pi' \times ei'); \\
Cq(index:index+2,ni:ni+6) = [\text{eye}(3,3), Ci]; \text{ %spherical portion}
\]

\[
Ci = 2 \times (\text{calcG}(ei) \times \text{skew4}(w\_vect) + w\_vect' \times ei'); \\
Cq(index+3,ni:ni+6) = [0,0,0,Sj' \times Ci]; \text{ %perp type 1}
\]

\[
Ci = 2 \times (\text{calcG}(ei) \times \text{skew4}(v\_vect) + v\_vect' \times ei'); \\
Cq(index+4,ni:ni+6) = [0,0,0,Sj' \times Ci]; \text{ %perp type 1}
\]

\%for body j
\[
Cj = 2 \times (\text{calcG}(ej) \times \text{skew4}(pj) + pj' \times ej'); \\
Cq(index:index+2,nj:nj+6) = [-\text{eye}(3,3), -Cj]; \text{ %spherical portion}
\]

\[
Cj = 2 \times (\text{calcG}(ej) \times \text{skew4}(sj) + sj' \times ej'); \\
Cq(index+3,nj:nj+6) = [0,0,0,W\_vect' \times Cj]; \\
Cq(index+4,nj:nj+6) = [0,0,0,V\_vect' \times Cj];
\]

\]

index = index + 5;
\]

\%for body i
\]

\%Perp Type two
\[
Ci = 2 \times (\text{calcG}(ei) \times \text{skew4}(w\_vect) + w\_vect' \times ei'); \\
Bi = 2 \times (\text{calcG}(ei) \times \text{skew4}(pi) + pi' \times ei'); \\
Cq(index,nj:nj+6) = [-W\_vect', -W\_vect' \times Bi + Pij' \times Ci]; \\
Cq(index+1,nj:nj+6) = [-V\_vect', -V\_vect' \times Bi + Pij' \times Ci];
\]

\%Perp Type one
\[
Ci = 2 \times (\text{calcG}(ei) \times \text{skew4}(v\_vect) + v\_vect' \times ei'); \\
Cq(index+2,nj:nj+6) = [0,0,0,Sj' \times Ci]; \text{ %perp type 1}
\]

\[
Ci = 2 \times (\text{calcG}(ei) \times \text{skew4}(v\_vect) + v\_vect' \times ei'); \\
Cq(index+3,nj:nj+6) = [0,0,0,Sj' \times Ci]; \text{ %perp type 1}
\]

\%for body j
\]

\%Perp Type two
\[
Bj = 2 \times (\text{calcG}(ej) \times \text{skew4}(pj) + pj' \times ej'); \\
Cq(index,nj:nj+6) = [W\_vect', W\_vect' \times Bj]; \\
Cq(index+1,nj:nj+6) = [V\_vect', V\_vect' \times Bj];
\]
\% Perp Type one
\[ C_j = 2*(\text{calcG}(e_j)*\text{skew4}(s_j) + s_j*e_j'); \]
\[ C_q(\text{index}+2,n_j:n_j+6) = [0,0,0,W\_vect'*C_j]; \]
\[ C_q(\text{index}+3,n_j:n_j+6) = [0,0,0,V\_vect'*C_j]; \]

\text{index} = \text{index} + 4;
\text{elseif} \ \text{system.joint(x).Type == 4 } \text{\% for joint type of S-S}
\text{ni} = 1+(i-1)*7;
\text{nj} = 1+(j-1)*7;
\text{ei} = \text{system.body(i).P};
\text{ej} = \text{system.body(j).P};
\text{d} = R_j + P_j - R_i - P_i;
\text{%dd} = R_dj + P_dj - R_di - P_di ;
\text{% for body i}
\text{Bi} = 2*(\text{calcG}(e_i)*\text{skew4}(p_i) + p_i*e_i');
\text{Cq(index,n_i:n_i+6) = [-2*d', -2*d'*Bi];}
\text{% for body j}
\text{Bj} = 2*(\text{calcG}(e_j)*\text{skew4}(p_j) + p_j*e_j');
\text{Cq(index,n_j:n_j+6) = [2*d', 2*d'*Bj];}
\text{index} = \text{index} + 1;
\text{elseif} \ \text{system.joint(x).Type == 5 } \text{\% prismatic}
\text{ni} = 1+(i-1)*7;
\text{nj} = 1+(j-1)*7;
\text{ei} = \text{system.body(i).P};
\text{ej} = \text{system.body(j).P};
\text{% calculate extra variables for calc}
\text{Si} = A_i*(p_i - q_i);
\text{Sj} = A_j*(p_j - q_j);
\text{Pij} = R_j + A_j*p_j - R_i - A_i*p_i;
\text{si} = (p_i - q_i);
\text{sj} = (p_j - q_j);
\text{[ w\_vect, v\_vect ] = twoperpvect3D( si );}
\text{W\_vect = A_i*w\_vect;}
\text{V\_vect = A_i*v\_vect;}
\text{[ wj\_vect, vj\_vect ] = twoperpvect3D( sj );}
\text{Wj\_vect = A_j*wj\_vect;}
\text{Vj\_vect = A_j*vj\_vect;}
\text{% for body i}
\text{% Perp Type two}
\text{Ci} = 2*(\text{calcG}(e_i)*\text{skew4}(w\_vect) + w\_vect*e_i');
\text{Bi} = 2*(\text{calcG}(e_i)*\text{skew4}(p_i) + p_i*e_i');
\text{Cq(index,n_i:n_i+6) = [-W\_vect', -W\_vect'*Bi + Pij'*Ci];}
Ci = 2*(calcG(ei)*skew4(v_vect) + v_vect*ei');
Cq(index+1,ni:ni+6) = [-V_vect', -V_vect'*Bi + Pij'*Ci];

% Perp Type one
Ci = 2*(calcG(ei)*skew4(w_vect) + w_vect*ei');
Cq(index+2,ni:ni+6) = [0,0,0,Sj'*Ci]; % perp type 1

Ci = 2*(calcG(ei)*skew4(v_vect) + v_vect*ei');
Cq(index+3,ni:ni+6) = [0,0,0,Sj'*Ci]; % perp type 1

Ci = 2*(calcG(ei)*skew4(v_vect) + v_vect*ei');
Cq(index+4,ni:ni+6) = [0,0,0,Wj_vect'*Ci];

% for body j
% Perp Type two
Bj = 2*(calcG(ej)*skew4(pj) + pj*ej');
Cq(index,nj:nj+6) = [W_vect', W_vect'*Bj];
Cq(index+1,nj:nj+6) = [V_vect', V_vect'*Bj];

% Perp Type one
Cj = 2*(calcG(ej)*skew4(sj) + sj*ej');
Cq(index+2,nj:nj+6) = [0,0,0,W_vect'*Cj];
Cq(index+3,nj:nj+6) = [0,0,0,V_vect'*Cj];

Cj = 2*(calcG(ej)*skew4(wj_vect) + wj_vect*ej');
Cq(index+4,nj:nj+6) = [0,0,0,V_vect'*Cj];

index = index + 5;
end
driver Cq, append to current matrix, fill in proper spot of Cq
for x = 1:system.info.bodies
    ni = 1+(x-1)*7;
    Cq(index,ni:ni+6) = [0,0,0,2*system.body(x).P'];
    index = index + 1;
end

% Fill in the final constraint, for the jacobian that is the euler parameters Cq to make bodies form a 7x7 matrix for fully constrained
for x = 1:system.info.bodies
    ni = 1+(x-1)*7;
    Cq(index,ni:ni+6) = [0,0,0,2*system.body(x).P'];
    index = index + 1;
end

%driving Cq, append to current matrix, fill in proper spot of Cq
for x = 1:system.info.drivers
    [coord,R, X,Y,Z,ft,fdt,fddt,body] = DecompDriver( system, x );
    m = 1+(body-1)*7;
    if coord == 'X'
        Cq(index,m) = 1;
    elseif coord == 'Y'
        Cq(index,m+1) = 1;
    elseif coord == 'Z'
        Cq(index,m+2) = 1;
    end
elseif strcmp(coord, 'PHIZ')
    Cq(index,m+6) = 1;
elseif strcmp(coord, 'PHIY')
    Cq(index,m+5) = 1;
elseif strcmp(coord, 'PHIX')
    Cq(index,m+4) = 1;
elseif strcmp(coord, 'e3')
    Cq(index,m+6) = 1;
elseif strcmp(coord, 'e2')
    Cq(index,m+5) = 1;
elseif strcmp(coord, 'e1')
    Cq(index,m+4) = 1;
end

index = index +1 ;
end
end
function [ Qd ] = BuildQd3DKin( system,t )
%Construct the Qd matrix for Kinematic 3D simulation using Euler parameters

%preallocate Qd matrix
Qd =zeros(system.info.cdof,1);
index = 7;
for x = 1:system.info.joints
  %%% Load common variables between joints into simple variables for
  %%% calculation
  [ i,j, Ri, Rj, RdI, Rdj, Ai,Aj, wi,wj, pi,pj,qi,qj,Pi,Pj,...
    Qi,Qj,Pdi,Pdj,Qdi,Qdj,pdi,pdj,qdi,qdj,SkewWi,SkewWj ] =
  system.joint(x).solverinfo{:};

  %Begin Qd calculations
  m = 1+(x-1)*3;
  n = 3 + (x-1)*3;
  if system.joint(x).Type == 1  %for joint type of spherical
    %calculate the G and L matrices
    Gid = calcG(system.body(i).Pd);
    Lid = calcL(system.body(i).Pd);
    Gjd = calcG(system.body(j).Pd);
    Ljd = calcL(system.body(j).Pd);

    %solve for Qd portion
    Hip = -2*Gid*Lid'*pi;
    Hjp = -2*Gjd*Ljd'*pj;
    Qd(index:index+2,1) =  Hip-Hjp;
    index = index +3;
  elseif system.joint(x).Type == 2  %for joint type of rev
    %calculate extra vectors
    Sj = Aj*(pj - qj);
    Sdj = Aj*(pdj - qdj);

    si = (pi - qi);
    sj = (pj - qj);

    [ w_vect, v_vect ] = twoperpvect3D( si );
    W_vect=Ai* w_vect;
    V_vect = Ai*v_vect;
    Wd_vect = Ai*Skew(wi)*w_vect;
    Vd_vect = Ai*Skew(wi)*v_vect;

    Gid = calcG(system.body(i).Pd);
    Lid = calcL(system.body(i).Pd);
    Gjd = calcG(system.body(j).Pd);
    Ljd = calcL(system.body(j).Pd);

    %fill in spherical portion
\[
Hip = -2*Gd*Ld'*pi;
Hjp = -2*Gjd*Ljd'*pj;
Qd(index:index+2,1) = Hip-Hjp;
\]

\% 2 perp vectors
\[
Hi = -2*Gid*Lid'*w_vect;
Hj = -2*Gjd*Ljd'*sj;
Qd(index+3,1) = -2*Wd_vect'*Sdj + W_vect'*Hj + Sj'*Hi; \%wi
Hi = -2*Gid*Lid'*v_vect;
Hj = -2*Gjd*Ljd'*sj;
Qd(index+4,1) = -2*Vd_vect'*Sdj + V_vect'*Hj + Sj'*Hi; \%vi
index = index +5;
\]

\textbf{elseif} system.joint(x).Type == 3
\%
\textbf{calculate vectors}
Sj = Aj*(pj - qj);
Sdj = Aj*(pdj - qdj);

\[\begin{align*}
&Pi = Rj + Aj*pj - Ri - Ai*pi; \\
Pdij = Rdj + Aj*SkewWj*pj - Rdi - Ai*SkewWi*pi; \\
&[ w_vect, v_vect ] = twoperpvect3D( si ); \\
&W_vect = Ai*w_vect; \\
&V_vect = Ai*v_vect; \\
&wd_vect = SkewWi*w_vect; \\
&vd_vect = SkewWi*v_vect; \\
&Wd_vect = Ai*wd_vect; \\
&Vd_vect = Ai*vd_vect;
\end{align*}\]

Gid = calcG(system.body(i).Pd);
Lid = calcL(system.body(i).Pd);

Gjd = calcG(system.body(j).Pd);
Ljd = calcL(system.body(j).Pd);

\%
\textbf{Type 2 perp}
Hi = -2*Gid*Lid'*w_vect;
Hib = -2*Gid*Lid'*pdi;
Hjb = -2*Gid*Lid'*pdj;
Qd(index,1) = -2*Pdij'*Wd_vect + Pij'*Hi + W_vect'**(Hjb-Hib); \%wi
Hi = -2*Gid*Lid'*v_vect;
Qd(index+1,1) = -2*Pdij'*Vd_vect + Pij'*Hi + V_vect'**(Hjb-Hib);

\%
\textbf{Type one perp}
Hi = -2*Gid*Lid'*w_vect;
Hj = -2*Gjd*Ljd'*sj;
Qd(index+2,1) = -2*Wd_vect'*Sdj + W_vect'*Hj + Sj'*Hi;

Hi = -2*Gid*Lid'*v_vect;
Hj = -2*Gjd*Ljd'*sj;
Qd(index+3,1) = -2*Vd_vect'*Sdj + V_vect'*Hj + Sj'*Hi;

index = index +4;
elseif system.joint(x).Type == 5

%Calc extra vectors
Sj = Aj*(pj - qj);
Sdj = Aj*(pdj - qdj);

si = (pi - qi);
sj = (pj - qj);

Pij = Rj + Aj*pj -Ri-Ai*pi;
Pdij = Rdj + Aj*SkewWj*pj -Rdi-Ai*SkewWi*pi;
[ w_vect, v_vect ] = twoperpvect3D( si );
[ wj_vect, vj_vect ] = twoperpvect3D( sj );
W_vect=Ai* w_vect;
V_vect = Ai*v_vect;
wdj_vect = SkewWj*wj_vect;
Wdj_vect = Aj*wdj_vect;
Wj_vect = Aj*wj_vect;

Gid = calcG(system.body(i).Pd);
Lid = calcL(system.body(i).Pd);

Gjd = calcG(system.body(j).Pd);
Ljd = calcL(system.body(j).Pd);

%Type 2 perp
Hi = -2*Gid*Lid'*w_vect;
Hib = -2*Gid*Lid'*pdi;
Hjb = -2*Gid*Lid'*pdj;
Qd(index,1) = -2*Pdij'*Wd_vect + Pij'*Hi + W_vect'*(Hjb-Hib); %wi
Hi = -2*Gid*Lid'*v_vect;
Qd(index+1,1) = -2*Pdij'*Vd_vect + Pij'*Hi + V_vect'*(Hjb-Hib);

%Type one perp
Hi = -2*Gid*Lid'*w_vect;
Hj = -2*Gjd*Ljd'*sj;
Qd(index+2,1) = -2*Wd_vect'*Sdj + W_vect'*Hj + Sj'*Hi;
Hi = -2*Gid*Lid'*v_vect;
Hj = -2*Gjd*Ljd'*sj;
Qd(index+3,1) = -2*Vd_vect'*Sdj + V_vect'*Hj + Sj'*Hi;
Hi = -2*Gjd*Ljd'*wj_vect;
Qd(index+4,1) = -2*Vd_vect'*Wd_j_vect + V_vect'*Hj + Wj_vect'*Hi;
index = index +5;

elseif system.joint(x).Type == 4  %for joint type of S-S
%calculate needed vector loops and G/L matrices

Gid = calcG(system.body(i).Pd);
Lid = calcL(system.body(i).Pd);
\begin{verbatim}
Gjd = calcG(system.body(j).Pd);
Ljd = calcL(system.body(j).Pd);

Hi = -2*Gid*Lid'*pi;
Hj = -2*Gjd*Ljd'*pj;
Qd(index,1) = -2*dd'*dd+2*d'*(Hj-Hi);
index = index +1;
end

end

for x = 1:system.info.bodies
    Qd(index) = system.body(x).Pd'*system.body(x).Pd;
    index = index+1;
end
%Lengthen Qd to take in account Driver constraints.
for x = 1:system.info.drivers
    Qd(index) = 0;
    index = index+1;
end
end
\end{verbatim}
function [ Ctt ] = BuildCttKin( system,t )
%build Ctt matrix for 3d Kinematic Simulation

Ctt = zeros(system.info.cdo,1);
index = system.info.cdo-system.info.drivers;

for x = 1:system.info.drivers
    [coord,R, X,Y,Z,ft,fdt,fddt,body] = DecompDriver( system, x );
    Ctt(index,1) = eval(fddt);
    index = index +1 ;
end
end
function [ springforces ] = SpringDamperDyn3D( system )

%This function handles the spring/dampers that can be placed between 2
%points on 2 different bodies. The vector loop is calculated from a point
%on body i to a point on body j, the magnitude and unit vector is
%calculated. Using hooks law the force is calculated, multiplied by unit
%vector and applied to the points. If the point is a given distance from Cg
%the torque is also applied by the cross of the force vector and point
%vector.

%Damping is done a simular way, but using the velocity loop instead to
%obtain the unit vector and velocity. A damping coefficient of c is used to
%control the gain of damping.

%Once everything is calculated, the extra forces etc generated and a matrix
%constructed to the same specifications as the right hand side matrix in
%order to add easily.

%create a matric with same dimensions as RHS for easy addition
springforces = zeros(system.info.bodies*6,1);
%Begin the loop between different springs
for x = 1:system.info.springdampers
    %Calculated and insert all the body information into easy to read/use
    %variables
    [ i,j,pi,pj,initialL,K,c,wj,wi,Ai,Aj,...
        Ri,Rj,Rdi,Rdj,actuator ] = DecompSpringSys( system, x );
    %calculate the spring position vector, and velocity vector
    springvect = Rj + Aj*pj- Ri - Ai*pi;
    springvectd = Rdj + Aj*Skew(wj)*pj  - Rdi - Ai*Skew(wi)*pi;
    %Use the unitvector function that was created to find the magnitude and
    %unit vector of the position and velocity.
    [ springL,springuvect] = unitvector(springvect);
    [ springvel,springuvectd] = unitvector(springvectd);
    %Calculate the spring and damping forces from the magnitudes then
    %multiply them by their corresponding unit vectors. Then add the 2
    %vectors to a Force vector
    spforcep = (initialL-springL)*K;
    spforcedamp = -c*springvel;
    spforcevect = spforcep*springuvect + spforcedamp*springuvectd+factuator;
    %calculate forces/moments on body i, Ensuring the proper direction
    forcesi = -spforcevect;
    momenti = cross(pi,-spforcevect);
    %calculate forces on body j, ensuring the proper direction
    forcesj = spforcevect;
    momentj = cross(pj,spforcevect);
    %Fill in the springforces matrix with their respective bodies, and then
    %output from function for later use. With careful attention to the
    %ground
m = 1+(i-1)*6;
n = 6+(i-1)*6;
springforces(m:n) = springforces(m:n)+ [forcesi; momenti]);

m = 1+(j-1)*6;
n = 6+(j-1)*6;
springforces(m:n) = springforces(m:n)+ [forcesj; momentj]);

end

end
DecompJointSys.m – Decompose Joint Variables 3D systems

function [ i, j, Ri, Rj, Rdi, Rdj, Ai, Aj, wi, wj, pi, pj, qi, qj, Pi, Pj, Qi, Qj, Pdi, Pdj, Qdi, Qdj, pdi, pdj, qdi, qdj, P, Q, Pdi, Pdj, Qdi, Qdj, pdi, pdj, qdi, qdj, SkewWi, SkewWj ] = DecompJointSys( system, x, pack )
% simplifies and reduces code to decompose a joint into its basic elements
% for calculations in multiple functions.
i = system.joint(x).Bodyi;
j = system.joint(x).Bodyj;
pi = system.joint(x).pi;
pj = system.joint(x).pj;
qi = system.joint(x).qi;
qj = system.joint(x).qj;

wj = system.body(j).w;
wi = system.body(i).w;
Ai = system.body(i).A;
Aj = system.body(j).A;
Ri = system.body(i).R;
Rj = system.body(j).R;
Rdi = system.body(i).Rd;
Rdj = system.body(j).Rd;
SkewWi = system.body(i).SkewW;
SkewWj = system.body(j).SkewW;

pdi = SkewWi*pi;
pdj = SkewWj*pj;
qdi = SkewWi*qi;
qdj = SkewWj*qj;
Pdi = Ai*pdi;
Pdj = Aj*pdj;
Qdi = Ai*qdi;
Qdj = Aj*qdj;
Pi = Ai*pi;
Pj = Aj*pj;
Qi = Ai*qi;
Qj = Aj*qj;

% if request pack in single variable to later use
if pack
    tempi = i;
    i = {tempi, j, Ri, Rj, Rdi, Rdj, Ai, Aj, wi, wj, pi, pj, qi, qj, Pi, Pj, Qi, Qj, Pdi, Pdj, Qdi, Qdj, pdi, pdj, qdi, qdj, P, Q, Pdi, Pdj, Qdi, Qdj, pdi, pdj, qdi, qdj, SkewWi, SkewWj};
end
end
function [coord, R, X, Y, Z, ft, fdt, fddt, body] = DecompDriver( system, x )
%simply driver constraints by inputting into easy to use variables
body = system.drivers(x).body;

R = system.body(body).R;
X = R(1);
Y = R(2);
Z = R(3);

coord = system.drivers(x).coord;

ft = system.drivers(x).ft;
fdt = system.drivers(x).fdt;
fddt = system.drivers(x).fddt;
end
DecompSpringSys.m – Decompose SDA Variables 3D systems

function [ i, j, pi, pj, initialL, K, c, wj, wi, Ai, Aj, Ri, Rj, Rdi, Rdj, f ] = DecompSpringSys( system, x )
% simplify spring data, to be easily extracted when needed
i = system.springdamp(x).Bodyi;
j = system.springdamp(x).Bodyj;
pi = system.springdamp(x).pi;
pj = system.springdamp(x).pj;
initialL = system.springdamp(x).initL;
K = system.springdamp(x).k;
c = system.springdamp(x).c;
f = system.springdamp(x).f;

wj = system.body(j).w;
wi = system.body(i).w;
Ai = system.body(i).A;
Aj = system.body(j).A;
Ri = system.body(i).R;
Rj = system.body(j).R;
Rdi = system.body(i).Rd;
Rdj = system.body(j).Rd;

end
function [ points ] = evalpoints( system,ANS )
%compute points of interest, based of system data.

q = []; 
qd = []; 
qdd = []; 
for x = 1:system.info.numpts 
    m = 1+(body-1)*6; 
    q = [q,system.body(body).R + system.body(body).A*vect]; 
    qd = [qd,system.body(body).Rd + system.body(body).A*system.body(body).SkewW*vect]; 
    qdd = [qdd,ANS(m:m+2) + system.body(body).A*system.body(body).SkewW*system.body(body).SkewW*vect ... + system.body(body).A*Skew(ANS(m+3:m+5))*vect]; 
end 
points = [q,qd,qdd,0]; 
end
CalcDof.m – Calculate DOF of Systems

```matlab
function [ system ] = CalcDof( system, dim )
% calculates the degrees of freedom for different coordinates and updates
% the system variable.

% begin processing DOF analysis
cdof = 0;
if dim == 3
    bdof = 3;  % number of dof per body
    cdof = 3;
    % calc DOF by constraints
    for x = 1:system.info.joints
        if system.joint(x).Type == 1
            cdof = cdof +2;
        elseif system.joint(x).Type ==2
            cdof = cdof +2;
        end
    end

    cdof = cdof + system.info.drivers;
elseif dim == 6
    bdof = 6;% number of dof per body
    cdof = 6;
    % calc dof by constraints
    for x = 1:system.info.joints
        if system.joint(x).Type == 1
            cdof = cdof +3;
        elseif system.joint(x).Type ==2
            cdof = cdof +5;
        elseif system.joint(x).Type ==3
            cdof = cdof +4;
        elseif system.joint(x).Type ==4
            cdof = cdof +1;
        elseif system.joint(x).Type ==5
            cdof = cdof +5;
        end

    end
    cdof = cdof + system.info.drivers;
elseif dim == 7
    bdof = 7;% number of dof per body
    cdof = 7;
    % calc dof by constraints
    for x = 1:system.info.joints
        if system.joint(x).Type == 1
            cdof = cdof +3;
        elseif system.joint(x).Type ==2
            cdof = cdof +5;
        elseif system.joint(x).Type ==3
            cdof = cdof +4;
        elseif system.joint(x).Type ==4
            cdof = cdof +1;
        elseif system.joint(x).Type ==5
            cdof = cdof +5;
        end
```
end
cdof = cdof + system.info.bodies-1;
cdof = cdof + system.info.drivers;
end
totdof = system.info.bodies*bdof;

dof = totdof - cdof;
%store data back in system variable
system.info.dof = dof;
system.info.totdof = totdof;
system.info.cdof = cdof;
end
calcG.m – Calculate G Matrix

```matlab
function [ gmat ] = calcG( e )
    % calculate the skew of e1, e2, e3
    % calculate the 2nd half of g
    gmat = [-e(2:4), Skew(e(2:4)) + e(1)*eye([3 3]) ]; % put together
    end
```

calcL.m – Calculate L Matrix

function [ Lmat ] = calcL( e )
%calculate L matrix from Euler parameters
    temp2 = Skew(e(2:4));             %calculate the skew of e1, e2, e3
    temp = -temp2 + e(1)*eye([3 3]); %calculate the 2nd half of L
    Lmat = [-e(2:4), temp];          %put together the L matrix
end
Skew.m – 3 Dimensional Skew-Symmetric Matrix

function [ matrix ] = Skew( v )
%calculate the 3D skew of the vector v.
matrix = [0 -v(3) v(2);      %calculates the skew matrix
          v(3) 0 -v(1);      %from an inputed vector
          -v(2) v(1) 0];
end
function [ out ] = Skew4( a )
%Calculate the 4D Skew of vector A
out = [0, -a';
     a, -Skew(a)];
end
function \[ N1, N2 \] = twoperpvect3D( S )

% calculate the orthogonal triad to vector S
% from dot s'*d = 0, there are infinite possibilities, 1,1 is used for y
% and z to calculate a perpendicular vector.
N1 = \[0;0;0\];
N2 = \[0;0;0\];

if S(1) > 0 || S(1) < 0
    Nx = (+S(2)+S(3))/S(1);
    % construct N1 and normalize
    N1 = \[Nx; -1; -1\];
    % find the second unit vector 90 normal to both.
    N2 = Skew(N1)*S;%cross(N1,S);
elseif S(2) > 0 || S(2) < 0
    Ny = (+S(3)+S(1))/S(2);
    % construct N1 and normalize
    N1 = \[-1; Ny; -1\];
    % find the second unit vector 90 normal to both.
    N2 = Skew(N1)*S;%cross(N1,S);
elseif S(3) > 0 || S(3) < 0
    Nz = (+S(1)+S(2))/S(3);
    % construct N1 and normalize
    N1 = \[-1; -1; Nz\];
    % find the second unit vector 90 normal to both.
    N2 = Skew(N1)*S;%cross(N1,S);
end

% use the previous function to find the unit quickly.
[mag, N1] = unitvector(N1);
[mag, N2] = unitvector(N2);

end
function [ magnitude, uvector] = unitvector( vect )
%Finds the unit vector of the input vector and magnitude, outputs it for
%use

magnitude = norm(vect);

uvector = vect/magnitude;

if magnitude == 0
    if length(vect) == 3
        uvector = [0;0;0];
    else
        uvector = [0;0];
    end
end
end
function [ Amat ] = Amat_angles(x,y,z)
%find the transformation matrix from a series of angles about the x,y,z
    a = [1 0 0;               %calculate the X rotation
        0 cos(x) -sin(x);
        0 sin(x) cos(x)];

    b = [cos(y) 0 sin(y);     %calculate the Y rotation
        0 1 0;
        -sin(y) 0 cos(y)];    %calculate the Z rotation

    c = [cos(z) -sin(z) 0;
        sin(z) cos(z) 0;
        0 0 1];
    Amat = a*b*c;       %multiply together for trans matrix
end
function [ eulerang ] = Amat_Euler ( Amat )

%Find the euler angles from the transformation matrix
tracea = Amat(1,1) + Amat(2,2) + Amat(3,3);  %trace of A matrix
e0 = sqrt((tracea + 1)/4);                   %calc e0
e1 = (Amat(3,2) - Amat(2,3))/(4*e0);          %calc e1
e2 = (Amat(1,3) - Amat(3,1))/(4*e0);          %calc e2
e3 = (Amat(2,1) - Amat(1,2))/(4*e0);          %calc e3
eulerang = [e0; e1; e2; e3];                 %assemble to vector
end