Fall 2017

Computational Methods for Nonlinear Systems Analysis With Applications in Mathematics and Engineering

Geoffrey Kenneth Rose
Old Dominion University

COMPUTATIONAL METHODS FOR NONLINEAR SYSTEMS ANALYSIS WITH APPLICATIONS IN MATHEMATICS AND ENGINEERING

by

Geoffrey Kenneth Rose
B.S. May 1995, Old Dominion University
B.S. May 1999, Old Dominion University
M.E. May 2005, Old Dominion University

A Dissertation Submitted to the Faculty of
Old Dominion University in Partial Fulfillment of the
Requirements for the Degree of

DOCTOR OF PHILOSOPHY

AEROSPACE ENGINEERING

OLD DOMINION UNIVERSITY
December 2017

Approved by:

Brett A. Newman (Co-Chair)

Duc T. Nguyen (Co-Chair)

Julie Z. Hao (Member)

Gene J. Hou (Member)

ABSTRACT

COMPUTATIONAL METHODS FOR NONLINEAR SYSTEMS ANALYSIS WITH APPLICATIONS IN MATHEMATICS AND ENGINEERING

Geoffrey Kenneth Rose
Old Dominion University, 2017
Co-directors: Dr. Brett A. Newman
Dr. Duc T. Nguyen

An investigation into current methods and new approaches for solving systems of nonlinear equations was performed. Nontraditional methods for implementing arc-length type solvers were developed in search of a more robust capability for solving general systems of nonlinear algebraic equations. Processes for construction of parameterized curves representing the many possible solutions to systems of equations versus finding single or point solutions were established. A procedure based on these methods was then developed to identify static equilibrium states for solutions to multi-body-dynamic systems. This methodology provided for a pictorial of the overall solution to a given system, which demonstrated the possibility of multiple candidate equilibrium states for which a procedure for selection of the proper state was proposed. Arc-length solvers were found to identify and more readily trace solution curves as compared to other solvers making such an approach practical. Comparison of proposed methods was made to existing methods found in the literature and commercial software with favorable results. Finally, means for parallel processing of the Jacobian matrix inherent to the arc-length and other nonlinear solvers were investigated, and an efficient approach for implementation was identified. Several case studies were performed to substantiate results. Commercial software was also used in some instances for additional results verification.

ACKNOWLEDGMENTS

This work was funded by the Advanced Degree Program at NASA Langley Research Center.

NOMENCLATURE

$\boldsymbol{f(u)}$	System of nonlinear equations
$\boldsymbol{F}$	Reference vector
i	Iteration count
$\boldsymbol{I}$	Identity matrix
$\boldsymbol{I_{CM}}$	Inertia matrix
$\boldsymbol{J}$ or $\boldsymbol{K}$	Jacobian or tangent stiffness of $\boldsymbol{f(u)}$
$\boldsymbol{M}$	Mass matrix
$\boldsymbol{R}$	Residual vector
$\boldsymbol{u}$	Vector of unknown variables
Δ	Incremental change with respect to λ or $\boldsymbol{u}$
λ	Scaling parameter
$\boldsymbol{\Lambda}$	Vector of Lagrange multipliers
$\boldsymbol{\Phi}$	Vector of algebraic constraints

TABLE OF CONTENTS

Page

Page

LIST OF TABLES

Table Page

LIST OF FIGURES

Figure Page

Figure Page

Figure Page

CHAPTER 1

INTRODUCTION

1.1 BACKGROUND MOTIVATION

Solving systems of nonlinear equations lies at the core of many finite element analysis (FEA) and multi-body-dynamics (MBD) software codes. Solution strategies for equilibrium typically involve solving these equations iteratively through linearization or Taylor series expansion using some variant of a Newton-Raphson solver [1-6]. This technique requires computation of derivatives for the Jacobian or tangent stiffness matrix for construction of a local linear model about a known operating point or solution. Common variations available for this type solver include full [1-3], modified [1-3], quasi-Newton [1-3,7], inexact-Newton [8], tensor [7], and arc-length [1-3]. Full methods update the Jacobian at every iteration whereas modified methods hold the Jacobian constant or minimize updates to reduce the associated computational cost. This simplification results in increased iterations required for convergence or finding a solution to the nonlinear system but does so in a cheaper sense which typically increases solver speed or efficiency. Quasi-Newton methods, on the other hand, update the Jacobian using approximations and typically reduce iterations as compared to the modified method but can have issues with convergence as the Jacobian contains error. Line search methods [7] can be included to help with convergence but come with added cost. Inexact methods utilize an iterative versus a direct solver for the linearized system that avoids factorizing the Jacobian during iterations. This alteration can help with solver speed as systems become large. Tensor methods use an extended

form of a Taylor series expansion and supplement first-order linear models with approximations for second-order derivatives in an attempt to improve the local model. Iterations for convergence can be reduced, but this adds to computational expense per iteration. Arc-length solvers include an additional constraint equation with the Newton-Raphson method and can be more robust due to an additional unknown parameter used during search for a solution. They may also be referred to as continuation or path following methods due to their natural ability to follow an equilibrium path with changes in sign for slope or direction. Updates to the Jacobian in this case can be made at every iteration for a standard approach or using approximations for a quasi-Newton method. The Jacobian can be also be held constant or updated periodically for a modified approach.

Review of documentation for popular FEA [9-11] and MBD [12,13] commercial software codes revealed that the Newton-Raphson method with selective updating of the Jacobian is the standard solver for use in nonlinear static FEA and static equilibrium for MBD. Other FEA solver options included quasi-Newton and arc-length. Tensor methods were not identified as being used in FEA solvers but were found in a MBD code [12] as a solver option for static equilibrium. Arc-length solvers were used in FEA codes for post-buckling analysis of structures that "snap" into new geometric configurations from sudden changes in force-displacement relations, but these solvers were not being used otherwise. Trust-region type solvers [7,14] developed primarily for numerical optimization were found implemented as an advanced alternative to Newton-Raphson type solvers in the event of convergence failure in MBD [12,13] codes in search of static equilibrium. These type solvers were not found being implemented for general use in FEA, but scientific computing software MATLAB [15] included a version of this type solver specifically for solving general systems of nonlinear equations while not including

any versions of Newton-Raphson. Trust-region solvers are more robust than most Newton-Raphson type solvers as they are able to handle cases where the Jacobian becomes singular. Singularity typically occurs at limit points in an analysis where any of the independent variables go from increasing in magnitude to decreasing or vice versa. This behavior can cause Newton-Raphson type solvers to fail due to an ill conditioned Jacobian or search for solutions in a direction away from the equilibrium path. Arc-length solvers are unique as they are able to follow equilibrium paths by specifying iteration path slope which facilitates stepping over limit points to avoid this issue.

1.2 RESEARCH PROBLEM AND OBJECTIVES

Based on identification of solver types and use, arc-length type solvers were chosen for investigation and comparison to state-of-the-art solvers and methods being used in commercial software. Primary use of these type solvers for post-buckling structural analysis in FEA left other areas open to investigation for possible research contributions to solving general systems of nonlinear algebraic equations. The problem of finding static equilibrium was also identified as particularly challenging from developers of MBD codes [12,13] such that arc-length solvers were used to further develop solution methods to this specific class of problem as well. Comparisons of proposed methods using arc-length solvers to current practices were made through programming of a nonlinear solver suite using MATLAB and running a series of specific problems or case studies. Functions or subroutines for previously mentioned solver types were developed with exception to trust-region. The trust-region type solver inherent to the MATLAB software was used in this case. Several general and MBD based nonlinear systems

were chosen for evaluation with particular attention paid to robustness or whether specific solver types could identify a solution or maintain track of curves representing solutions to given systems. Case studies for identifying roots in nonlinear systems used to determine initial conditions for relative orbits for spacecraft were also performed. Path following solver strategies based on arc-length solvers were found to be more robust as compared to previous work where additional roots or solutions to the nonlinear systems were found.

The final part of the investigation involved identifying a more efficient means of numerically computing the Jacobian matrix through parallel processing. The Jacobian is a matrix of first-order partial derivatives inherent to linearization and nonlinear solver algorithms based on this principle. Computation and factorization of the matrix is performed during solver iterations resulting in a timing or speed bottleneck where increased solver efficiency is obtained through decreasing such operations. Performance gains through parallel processing typically become more apparent as systems increase in size; however, lower bounds to the size of systems to which this first becomes beneficial may not necessarily be known. A system of interconnected links was used as a benchmark problem to identify such lower bounds due to specific reference in a MBD user manual [12]. This particular system was identified as not exhibiting parallel processing performance gains for a given minimum scale and this scale was used to set a goal for parallel code speedup. Methods were then developed in MATLAB that demonstrated initial performance gains at even smaller scales. Results for a range of system sizes and processing methods were quantified and compared to non-parallel or serial processing results. The timing study was completed on a multi-core shared memory personal computer (PC) using MATLAB which included underlying parallel operations. These hidden operations

added to the challenge of achieving code speedup as the underlying parallel operations left little room for improvement.

The primary objective of this research effort was to identify state-of-the-art practices for solving systems of nonlinear equations and develop new methodologies for improvement of both robustness and efficiency. Focus of the study remained on developing solution strategies for systems of nonlinear algebraic equations and nonlinear differential and algebraic equations (DAEs) with regard to static equilibrium. User documentation of popular commercial software codes was included as part of a literature review for identifying what is considered to be state-of-the-art. Several sample problems or case studies were developed for comparison of proposed solver strategies to existing methods in terms of robustness or capability to solve a given problem and efficiency where the primary metric was solve time. Case studies for robustness involved finding solution to general mathematical systems of nonlinear algebraic equations and static equilibrium for nonlinear MBD systems. Efficiency was addressed through developing a method of parallel processing of the Jacobian matrix that demonstrated a timing speedup as compared to a serial version.

1.3 DISSERTATION OUTLINE

Chapter 2 contains an overview of literature used to support the study. Popular commercial software relevant to systems of nonlinear equations was identified and user documentation was consulted to identify solution procedures. These procedures are assumed to be state-of-the-art based on the assumption that developers strive to produce software that maximizes robustness, efficiency, or performance in general.

Chapter 3 covers use of the arc-length method for solving general systems of nonlinear equations and compares the solver in terms of robustness to other Newton-Raphson based solvers and MATLAB's *fsolve* [15] routine. Accomplished work in this area has been previously published and the reference has been included in the copyright notice in the Appendix. The metrics used for robustness are the solvers ability to identify a solution from an initial guess and ability to maintain track of a series of solutions along a path once a starting point was found.

Chapter 4 encompasses work in Chapter 3 but is specific to nonlinear equations for MBD systems and the search for static equilibrium. Accomplished work may become subject to copyright and reference has been included in the copyright notice in the Appendix. Candidate equilibrium configurations for sample systems were identified through plotting of the solution with respect to given independent variables. A procedure for selecting the proper equilibrium configuration based on energy and the use of solution curves is proposed. This procedure was shown to provide for a more comprehensive and systematic approach in identifying true equilibrium as compared to methods currently being used in commercial MBD software.

Chapter 5 addresses methods of parallel processing for computation of the Jacobian matrix inherent to Newton-Raphson based solvers. Accomplished work has been previously published and reference has been included in the copyright notice in the Appendix. MATLAB was used to complete this task for consistency with the previously developed nonlinear solver suite. A specific system of interconnected links was identified in MSC ADAMS user documentation [12] as a performance challenge by which to first demonstrate parallel processing speedup for a given minimum system size. The objective of achieving speedup of computer code was met and expected performance gains for MATLAB based applications running on shared memory personal computers were quantified.

Chapter 6 contains a case study addressing application of an arc-length method for solving spacecraft relative orbit determination equation sets. Path following of solution curves was demonstrated to be more robust in the identification of roots used for orbit initial conditions as compared to standard solver techniques. Identification of all roots is critical for this particular application as only one root represents initial conditions for an orbit of non-zero velocity and minimum energy.

Chapter 7 summarizes conclusions and findings. Recommended further research is identified including the need for implementing arc-length based solver schemes in a more automated manner for practical use.

CHAPTER 2

LITERATURE REVIEW

Solution to nonlinear systems of equations lies at the core of many computer software codes used in science and engineering. Such systems can often pose a significant challenge for finding and identifying solutions using existing mathematical tools. Review of nonlinear equation solver theory, parallel computing, and current software implementations was performed to help identify strengths and weaknesses of various solver strategies being used. Based on this review, solution procedures with possible areas for improvement were identified and used to define research objectives. This identification included evaluation and comparison of solver strategies used for general systems of nonlinear equations, strategies for obtaining equilibrium in MBD systems, and parallel computation of the Jacobian matrix identified as a speed bottleneck on shared memory personal computers. Work is documented in Chapters 3 through 6 with sample versions of MATLAB computer code contained in the appendix.

2.1 SOLVER OVERVIEW

The Newton-Raphson method and closely related techniques have been identified by Bathe [1] as the most frequently used iteration schemes for solution to nonlinear finite element based equations. Further stated is that the Newton-Raphson method represents the primary solution scheme for FEA. The major computational cost per iteration was identified as calculation and factorization of the tangent stiffness or Jacobian matrix and that the use of a

modified Newton-Raphson method can be effective in reducing this cost. A method that computes and factorizes the Jacobian once using a system's initial configuration and holds it constant during iterations is referred to as the "initial stress" method. Methods that update the Jacobian periodically during iterations are referred to as "modified" methods. Similar terminology is used in a text by de Borst et al. [3]. Computational cost associated with the Jacobian is noted similar to Bathe and it is assumed that limited variation of the Jacobian between subsequent iterations is what makes modified approaches practical. The slowing down of convergence or increase in iterations for the modified methods is noted as acceptable as it is offset by gains or performance in computation time. Cook et al. [2] state that computational cost is usually lowest by selectively updating the Jacobian. The initial stress method is here presented as a form of the modified Newton-Raphson method versus a unique procedure. Similar to Bathe, potential issues with convergence are identified due to lack of Jacobian updates during iterations.

Shabana [5] provides for several solution strategies that can be used for solving systems of nonlinear DAEs found in MBD codes. Solution strategies, whether static, kinematic, or dynamic, all involve use of the Newton-Raphson method. While systems of DAEs could be solved using only the Newton-Raphson method for a dynamic solution procedure, Shabana proposes using this for the constraint equations only followed by a direct numerical integration scheme for the dynamics portion. Shabana covers a variety of these direct integration procedures and processes that transform equations to a state space representation for use with these type solvers [4]. This study, however, will not cover dynamic solvers in detail; rather, focus will remain on solution strategies for identifying static equilibrium. Shabana [4] notes that it is desirable in many applications to obtain a static equilibrium configuration prior to a dynamic simulation. This desirability is due to differences between the as modeled and equilibrium

configurations that are likely to occur. Difficulty with obtaining solutions for static equilibrium are also addressed when Lagrange multipliers or constraint forces are included as unknowns. In his proposed algorithm for solving DAEs, he states that initial conditions must provide a good approximation of the exact initial configuration. This approximation would of course effect convergence and probability of finding a solution for any of the Newton-Raphson based solvers. Specific types or classes of Newton-Raphson methods are not discussed and reference to arc-length, continuation, or path following methods for static equilibrium is not made. This exclusion is also the case for solver strategies covered by Bauchau [6]. Mention of a modified Newton-Raphson method is discussed in reference to the possibility of considerable computational savings as compared to the standard or full method, but reference to other static type solvers for nonlinear equations is not made. Similar to Shabana, Bauchau also mentions carrying out a static equilibrium analysis prior to start of a dynamic simulation.

Quasi-Newton methods are an alternative to the full and modified forms of the Newton-Raphson method. These methods use an approximation for the Jacobian matrix by calculating it, or more specifically its inverse, in an inexact sense. Cook et al. [2] refer to the approximated Jacobian as the secant stiffness matrix versus tangent stiffness matrix as computation involves use of a previously known solution or point on an equilibrium path versus a single tangent point only. One of the most popular methods cited by de Borst et al. [3] is the Broyden-Fletcher-Goldfarb-Shanno (BFGS) [16-19] update. Caution is advised when using this method as convergence behavior deteriorates as compared to a Newton-Raphson method. Reports of erratic behavior and lack of numerical stability were also identified resulting in a decrease in popularity of quasi-Newton methods in more recent years. Although several quasi-Newton type methods exist, Bathe [1] reports that the BFGS method appears to be the most effective. Issues with

numerical stability are addressed through incorporation of a line search strategy that becomes an integral part of the overall solution procedure, but computational cost is increased as line searches scale candidate solutions in an iterative fashion in an attempt for convergence or further minimization of error. A detailed derivation of the BFGS method including a computational example can be found in a text by Komzsik [20].

Tensor methods use an extended form of a Taylor series expansion and augment first-order linear models with an approximation for the second-order term. Schnabel and Frank [21] introduced them as a new class of methods designed specifically for solving systems of nonlinear equations. They are intended to improve upon Newton-Rapson based methods and handle cases where the Jacobian is singular or ill conditioned. The second-order term, often called the Hessian, is formed by interpolating function values from previous iterations similar to what is done for quasi-Newton methods. According to Bouaricha and Schnabel [22], use of one or two past iteration points is sufficient. One of the major contributions of tensor methods according to Bouaricha [23] has been its greater robustness, and experimental results show that tensor methods consistently solve a wider range of problems as compared to the Newton-Raphson based methods. Most recent advancements for tensor methods appear to have been completed by Bader [24]. Focus is on large-scale systems and methods are referred to as tensor-Krylov similar to terminology used by Bouaricha [23] in an earlier publication. Krylov refers to a class of linear solvers named after Russian mathematician Aleksey Krylov [25]. Krylov solvers are typically used for large-scale systems as they avoid matrix factorization and solve linear systems iteratively to save on computational cost. Two of the most popular Krylov solvers are conjugate gradient (CG) [26] for use with symmetric matrices and generalized minimal residual (GMRES) [27] for use with nonsymmetric matrices.

Krylov solvers used in combination with the Newton-Raphson method lead to a specific class of solvers referred to as inexact-Newton methods. The term inexact is used as a solution to the linearized system performed during iterations of the Newton-Raphson method contains its own convergence criteria and therefore has some error. A detailed discussion on inexact-Newton methods can be found in a text by Kelly [8]; it also includes separate chapters on CG and GMRES methods. Solver naming convention is based on the type of linear solver being used such as Newton-CG or Newton-GMRES. A more general naming convention would be Newton-Krylov, which encompasses all Newton-based nonlinear solvers using Krylov subspace methods. Such naming convention can also be applied to tensor methods as well. According to GMRES developers Saad and Schultz [27], "One of the most effective iterative methods for solving large sparse symmetric positive definite linear systems of equations is a combination of the conjugate gradient method with some preconditioning technique." Preconditioning is used to reduce the condition number of a matrix to improve performance of iterations thereby helping to minimize the number of computations required for convergence. A detailed overview of preconditioning techniques and Krylov subspace methods can be found in a text by Saad [28].

Arc-length methods supplement the Newton-Raphson method with an additional constraint equation to define a path for iterations. The primary strength of arc-length methods is their ability to solve past limit points when traversing a solution curve or equilibrium path that changes slope or direction. Several variations of the arc-length method exist and naming convention is typically based on the type of constraint being used. Early development of the method is credited to Riks [29] and Wempner [30] where iterations are constrained to a normal plane. Crisfield [31] later proposed use of a circular path and Ramm [32] used a linearized version of Crisfield's constraint for an updated normal plane that provided for a "faceted" path

that mimics a curve. In later publications by Crisfield [33,34], the term cylindrical was used to define a path where one of the terms in the circular or spherical constraint was set to zero. This implementation was, in fact, the method used in his original publication [31] which is why the phrase “proposed circular” or “proposed spherical” path is often used. An elliptical path is presented in course notes by Felippa [35] where scalar coefficients are used as multipliers for terms in the spherical constraint equation. By setting these terms to one, the spherical path is recovered, whereas other values form an elliptical path. The prefix “hyper” is also appended to naming convention to reinforce use for multi-degree-of-freedom systems versus single-degree-of-freedom only. Felippa refers to both global and local hyperelliptic control options for defining the iteration path depending on the frame of reference being used.

Bathe and Dvorkin [36] provided for an additional constraint option by including an energy or work based equation in combination with a spherical constraint. The possibility of combining the full Newton-Raphson, modified Newton-Raphson, quasi-Newton, and line search methods in combination with arc-length constraints was also discussed. The solution scheme was acknowledged as being particularly effective near limit or collapse points with an overall objective of tracing the complete equilibrium path in an automated manner. This concept is relevant to numerical path following or continuation methods in mathematics. De Borst et al. [3] uses the phrase path following method as a similar descriptor for the arc-length method; however, connection to such theory in mathematics is not explicitly made. Previous texts by Crisfield [33,34], on which the later text by de Borst et al. is based, cover arc-length methods in greater detail and make the connection between these methods and related continuation methods or techniques in mathematics. This continuation terminology is used repeatedly in Volume 1 [33] and later replaced by path following in Volume 2 [34].

An overview of numerical path following or continuation methods presented from a mathematical perspective can be found in texts by Allgower and Georg [37,38]. Reference to works by authors who refined such procedures for FEA is included in the bibliography of the earlier publication, but specifics with regard to these references are not included in the text or later publications showing a general disconnect between the two vocations. Terminology varies slightly from what is used for FEA as arc-length methods are referred to as predictor-corrector or pseudo arc-length continuation methods. The term pseudo is best understood with respect to a single-degree-of-freedom system where a tangent line is used to approximate the length of an arc or curve. The point at the end of the tangent line is called a predictor for the next point on the curve, and Newton-Raphson iterations are then used as correctors until an intersection with the curve or a solution is found. The basic concept behind this principle is that a series of tangent lines or approximate arc-lengths serve as an ideal method for parameterization of a given curve. This parameterization is what enables tracking of the curve around limit or turning points where the Newton-Rapson method parameterized using only a fixed scalar for non-zero solutions would fail. The magnitude of the arc-length parameter is scaled in practice according to Bathe [1] based on the history of iterations between solution steps or previously found points on the solution curve. This magnitude could be large when the behavior of a solution curve is nearly linear and become small when behavior becomes nonlinear such as near a turning point. Crisfield [33] provides for a simple scaling equation using the ratio of desired to actual iterations required for convergence and refers to such an approach as required for a robust continuation method. He also warns that despite one's best attempt at automation, user intervention is often required and methods for restarting of the solver should be made available.

Felippa [35] summarizes the use of continuation or path following methods used in nonlinear structural mechanics or FEA as not just being a possible game but the only game. He refers to the engineering flavor of continuation methods as being less difficult to implement as those presented in mathematics where the objective of finding the roots in a nonlinear system analysis is replaced by following the physics. He comments on the lack of cross-fertilization between the math and engineering communities and notes a 1978 publication by H. B. Keller titled, "Global homotopies and Newton methods," that claims invention of the arc-length method years after Riks and Wempner first published it. Felippa opts for the predictor-corrector terminology for explaining the various forms of arc-length methods but notes that such terminology is far from standardized. Final comments in his course notes state that the last major advancement in arc-length methods was made by Riks and Wempner in the late 1960's and early 1970's and was later improved by Crisfield in the 1980's. Another significant improvement mentioned by de Borst et al. [3] is the partitioned versus direct solution procedure used to maintain symmetry and the banded nature of the Jacobian or tangent stiffness matrix. This procedure was used by Crisfield [31] and Ramm [32] and breaks the vector of unknown variables in the nonlinear system into two components for purpose of solving equations in an efficient manner.

Conn et al. [14] discuss the use of trust-region methods for solving systems of nonlinear equations although these methods are more commonly used for purpose of numerical optimization. Systems of nonlinear equations are used to construct what is called an objective function with a goal of finding a solution that minimizes this function. A model that approximates the objective function is formed and iterations are performed to continually update candidate solutions until convergence is achieved. Unlike the Newton-Raphson method, which

uses a linear model, a quadratic model that incorporates use of the Hessian or matrix of second-order partial derivatives from a Taylor series expansion is used. This fidelity increment adds a degree of robustness to the solver as convergence at limit or critical points that exhibit zero slope or a singular Jacobian can be achieved. However, warning is given as the solver may converge to a minimum and not a root or zero for a solution to the nonlinear equations. Nocedal and Wright [7] also include use of trust-region methods as a solver option for nonlinear equations. The phrase objective function is replaced by merit function to better differentiate between solvers used for optimization versus nonlinear equations. Merit functions are defined as scalar functions that indicate whether progress is being made towards finding a root. The most widely used merit function was identified as using the sum of squares of the nonlinear equations during iterations. Roots can be distinguished from minimums as they equate equations to all zeros versus some positive value. The most widely used quadratic model for solving nonlinear equations was identified as one that uses an approximate Hessian obtained by multiplying the transpose of the Jacobian by itself. Computation of the exact Hessian would be quite expensive and approximating it as a function of the Jacobian helps to reduce this cost. Crisfield [33] includes commentary on limit or critical points for continuation methods using arc-length solvers. He states that from an engineering viewpoint, the precise computation of limit points does not seem to be of practical importance. Although arc-length solvers can fail at limit points due to singularity of the Jacobian, he states that this was not found to be a significant problem as one appears never to arrive precisely at a limit point. There is always the option to reduce the arc-length parameter and restart the solver as well. Authors of FEA and MBD texts [1-6] did not include use of trust-region methods for solving nonlinear equations, likely due to their higher

computational expense as compared to Newton-Raphson methods and low probability that a root would also correspond to a limit point in an equilibrium solution.

2.2 PARALLEL COMPUTING

Barney [39], who authored an online tutorial in parallel computing for the Lawrence Livermore National Laboratory, defines parallel computing in the simplest sense as the simultaneous use of multiple computer resources to solve a computational problem. This process differs from more traditional serial computing where a problem is broken into a discrete series of instructions that are processed in order one at a time. Primary reasons cited for use of parallel computing are to save time, money or both, to solve larger, more complex problems, provide concurrency, to take advantage of non-local resources, and to make better use of underlying parallel hardware. Barney states that virtually all stand-alone computers today are parallel from a hardware perspective and that trends over the past 20+ years in network speed, distributed systems, and multi-processor computer architectures show that parallelism is the future of computing. Several programming models are covered in the tutorial where a few of the popular standards include OpenMP [40] where several central processor units (CPUs) share memory, message passing interface (MPI) [41] where memory is distributed among multiple CPUs, and Single Program Multiple Data (SPMD) [42] which can be a hybrid combination of both. He also notes an increasingly popular hybrid model that incorporates graphics processor units (GPUs) in addition to CPUs for parallel processing.

The primary intent of parallel programing cited by Barney is to decrease execution wall clock time. One of the simplest and most widely used metrics for parallel performance is

observed speedup being defined as serial wall clock time divided by parallel wall clock time. Caution is given regarding performance of short running parallel programs as there can be a decrease in performance due to issues such as task creation and communications overhead. A noted inhibitor to parallelism is input-output timing for the transfer of data. In regards to CPU and available memory, a table is included in the tutorial showing a decrease in speed for various memory types with respect to the CPU register with a baseline communication time of one nanosecond. Cache memory is shown to be 10X slower, main memory 100X slower, and magnetic disk memory 100,000,000X slower showing dramatic reduction in speed for memory locations further away from the cache. An older yet still relevant text on parallel computing by Grama et al. [43] makes note of impressive gains in CPU performance over a given decade while the ability of computer memory to feed data to processors has not kept up with their execution rate. This timing gap between processor and memory has led to a significant performance bottleneck diluting overall parallel performance.

Matloff [44], a University of California at Davis professor, maintains an open-source text available online for parallel programming and lists several issues that can effect or inhibit performance. He identifies the most central performance issue as being load balancing where the objective is to keep all processors as busy as much as possible and that communication considerations largely drive this issue. Also noted is that the phrase “embarrassingly parallel” has evolved over recent years from referring to parallel code in a simple or easy to implement sense to one of maintaining low communications overhead. He states that most users find their code often becomes slower on their first attempt to parallelize. The reason being lack of understanding how hardware works, at least at a high level.

2.3 COMMERCIAL SOFTWARE

Commercial finite element codes such as Abaqus [9], ANSYS [10], and MSC Nastran [11] all use some variant of a Newton-Raphson [1-3] type solver for solving systems of nonlinear equations. Nastran has several variations of arc-length solvers referred to as the Riks method [45,46] where iterations are constrained to a normal path, a modified Riks method [32] for an updated normal path, and Crisfield's method [31] for a circular path. Abaqus offers a modified Riks method [31,32,47] and ANSYS offers Crisfield's method [31]. Arc-length solvers are specifically used for post-buckling type analysis of structures in these finite element codes. Quasi-Newton or BFGS [16-19] updates for the Jacobian or tangent stiffness matrix are available as an option in Nastran and Abaqus where ANSYS refers to use of a secant matrix implying some other variation. Line searches [7] are also available in these codes where use of such a method is included by default in Abaqus when using BFGS updates. Iterative solvers are also available as part of an inexact-Newton solver where Nastran and ANSYS use a preconditioned conjugate gradient method [26] and Abaqus uses a more generic preconditioned Krylov [28] solver.

Multi-body-dynamics codes MSC ADAMS [12] and RecurDyn [13] both use Newton-Raphson as an initial method to search for static equilibrium. RecurDyn augments the solver with a trust region method [14] in the event singularity with the Jacobian is encountered and incorporates a line search procedure as well. ADAMS [12], on the other hand, offers a suite of static solver options in addition to Newton-Raphson including tensor-Krylov [24] and several optimization based solvers, one of which includes a line search strategy. These choices are some of the more robust nonlinear solver options found in commercial software where attempts to find

equilibrium can be made using an entire suite of solvers. No methods based on path following were found and solvers will generally provide point solutions in closest proximity to as modeled configurations when possible. Technical computing software MATLAB only offers optimization algorithms imbedded in its *fsolve* [15] routine for solving general systems of nonlinear equations. No variations of Newton-Raphson, tensor, or arc-length were found leaving this up to users for programming and implementation.

All of the referenced commercial software offers parallel operations with shared memory parallelism being the most common. Abaqus, ANSYS, and MALAB extend parallel operations to include use of GPUs while Nastran, ADAMS, and RecurDyn do not appear to have implemented this latest form of parallel computing technology. Typical subroutines that have been parallelized in FEA and MBD codes include matrix computation and factorization, linear solvers, and eigenvalue solvers. MATLAB offers an even larger variety of shared memory parallel operations [48-50] that are embedded in subroutines and occur by default without user intervention. This implementation may be referred to as implicit parallelism [48] and such operations are minimally referenced in MATLAB's user documentation [15]. This default can significantly add to the challenge of achieving speedup for explicit parallel code on shared memory computers where underlying parallel operations in serial versions of code are already in place.

CHAPTER 3

SOLVING GENERAL SYSTEMS OF EQUATIONS

Solving systems of nonlinear equations can be challenging and analysts are often required to provide an initial guess of the solution as a starting point for use in an iterative solver. Insight into approximate solutions leading to a good initial guess can usually be obtained if equations are representative of a physical system. However, this process may not be achievable for complex systems or when the analyst lacks familiarity or experience with the system. In this case, convergence may not be achieved if the initial guess is not close to the solution. A general nonlinear solver suite based on the arc-length method with these circumstances in mind was developed for the purpose of numerical experimentation and was found to be a useful alternative to the *fsolve* function inherent to the MATLAB software [51]. Due to the additional unknown variable and supplemental constraint equation used by the arc-length method, curves representing solutions to parameterized equation sets were found by embedding the solver in a loop. Restarts in the analysis were minimized as the arc-length method is capable of solving beyond local maxima or minima on smooth curves. Several examples are provided demonstrating the unique capabilities of arc-length solvers.

3.1 INTRODUCTION

Arc-length methods have successfully been used as a means for solving problems in structural analysis that involve tracking sudden changes in equilibrium paths or force-

displacement curves [30-32,45]. The collapse of a structure from an applied load for example may not necessarily involve total system failure but a sudden geometric change where the structure has "snapped" to a new configuration. Analysts may be interested in tracking the equilibrium path through the snapping event and want to determine how the structure behaves if continued loading is applied. Arc-length methods were developed for this purpose and are capable of tracking solutions beyond limit points such as points 2 or 3 on the sample curve in Fig. 1.

Starting from point 1 on Fig. 1, Newton-Raphson solvers are able to find points on the path up to limit point 2. Point 4, if found, would be the next available value of displacement at this specified force value leading to a break or discontinuity in path. The procedure could be restarted near point 4 using an estimated larger value for displacement in an attempt to find a new point on the path. Once a new starting point is found, additional points on the missing portion of the path could be found by reducing force until the lower limit at point 3 is encountered. Additional restarts in the analysis could be performed including use of MATLAB's *fsolve* routine [15] to help fill in the remaining section of the path between points 2 and 3 for specified levels of force.

Although such an approach could be used, it would be less robust as compared to a solver that could track the equilibrium path beyond limit points in a continuous manner. There is also the possibility of gathering an insufficient number of points needed to construct the path or missing sections containing abrupt turns. Arc-length methods, on the other hand, treat force as an unknown parameterizing variable extending the search for new points to the two-dimensional space represented by Fig. 1. When force is maintained as a specified variable in other methods, the search for new points is limited to a horizontal line crossing the vertical axis at the specified

value. When force is treated as a simultaneous unknown parameter, search procedures become multi-dimensional, making the arc-length method better suited for finding the different nonlinear solutions. Although Fig. 1 is representative of a single-degree-of-freedom system, the same holds true for multi-degree-of-freedom systems where any component of the displacement vector can be plotted against a factor used to scale force. Example problems studied in this chapter demonstrate robustness of arc-length methods through minimizing or avoiding the need for restarts and added capability to search for and successfully find solutions where other solvers may fail. Development of the solver suite provided a robust tool set for finding solutions to nonlinear systems of equations and visualizing results.

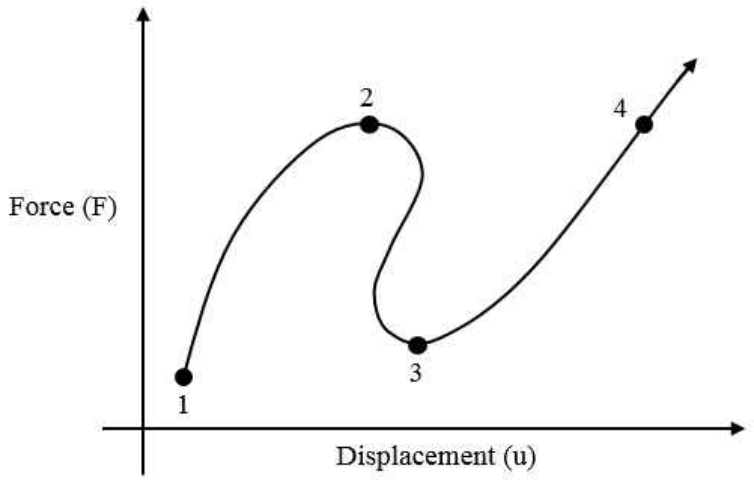

Figure 1. Equilibrium path exhibiting limit points and snap-through

3.2 SOLVER THEORY AND BACKGROUND

The objective in solving nonlinear systems involves finding $\boldsymbol{u}$ such that

$$\boldsymbol{f}(\boldsymbol{u}) = \boldsymbol{0} \tag{1}$$

In structural analysis, the problem is formulated as

$$\boldsymbol{f}(\boldsymbol{u}) - \lambda \boldsymbol{F} = \boldsymbol{0} \tag{2}$$

where the objective is to find a balance between internal system forces $\boldsymbol{f}(\boldsymbol{u})$ and a scaled value of applied force $\boldsymbol{F}$. Use of this format allows for generation of curves such as Fig. 1 where displacement can be found at various levels of force. This equation format was used to implement the arc-length solver for general sets of nonlinear equations where $\boldsymbol{F}$ is taken as unity or a column vector of ones denoted as $\boldsymbol{1}$. The refined objective is to find $\boldsymbol{u}$ such that $\boldsymbol{f}(\boldsymbol{u}) = \lambda \boldsymbol{1}$ where λ can be any constant that is common to the set of equations. By making λ an unknown parameter, a higher likelihood of finding a solution from an initial guess exists as λ can be any scalar value including zero. Once a solution has been found, it can be used as an initial guess for nearby values of λ making the process less random and more likely to find new solutions. The process can therefore be continued to construct curves representing all values of λ common to the equation set with regards to individual components of $\boldsymbol{u}$.

At the core of many solver methods is the Newton-Raphson scheme. A first-order Taylor series is applied to the system of nonlinear equations $\boldsymbol{f}(\boldsymbol{u})$ which "linearizes" the system at specific values of $\boldsymbol{u}$ resulting in

$$\boldsymbol{f}(\boldsymbol{u}) + \boldsymbol{K}\Delta\boldsymbol{u} - \lambda \boldsymbol{F} = \boldsymbol{0} \tag{3}$$

Matrix $\boldsymbol{K}$ is typically referred to as the Jacobian or tangent stiffness as it represents a tangent line to the equilibrium path at $\boldsymbol{u}$. The resulting set of equations may now be used as a local linear model to predict new values of $\boldsymbol{u}$ for a fixed value of λ. A measure of error or residual in the nonlinear equations at the predicted value of $\boldsymbol{u}$ is used to correct $\boldsymbol{u}$ and update the linear model for the next step. The process is repeated through iterations i until a specified tolerance on error is achieved or an iteration limit has been met to avoid the possibility of an infinite loop if the

solver were to diverge. A graphical representation of the procedure is shown in Fig. 2. The iteration path for this case follows a series of horizontal points starting with initial predicted values for $\boldsymbol{u}_0$ and $\boldsymbol{K}_0$ and corresponding corrected values $\boldsymbol{u}_i$ and $\boldsymbol{K}_i$ along the horizontal line $\lambda \boldsymbol{F}$ until converging with the equilibrium path.

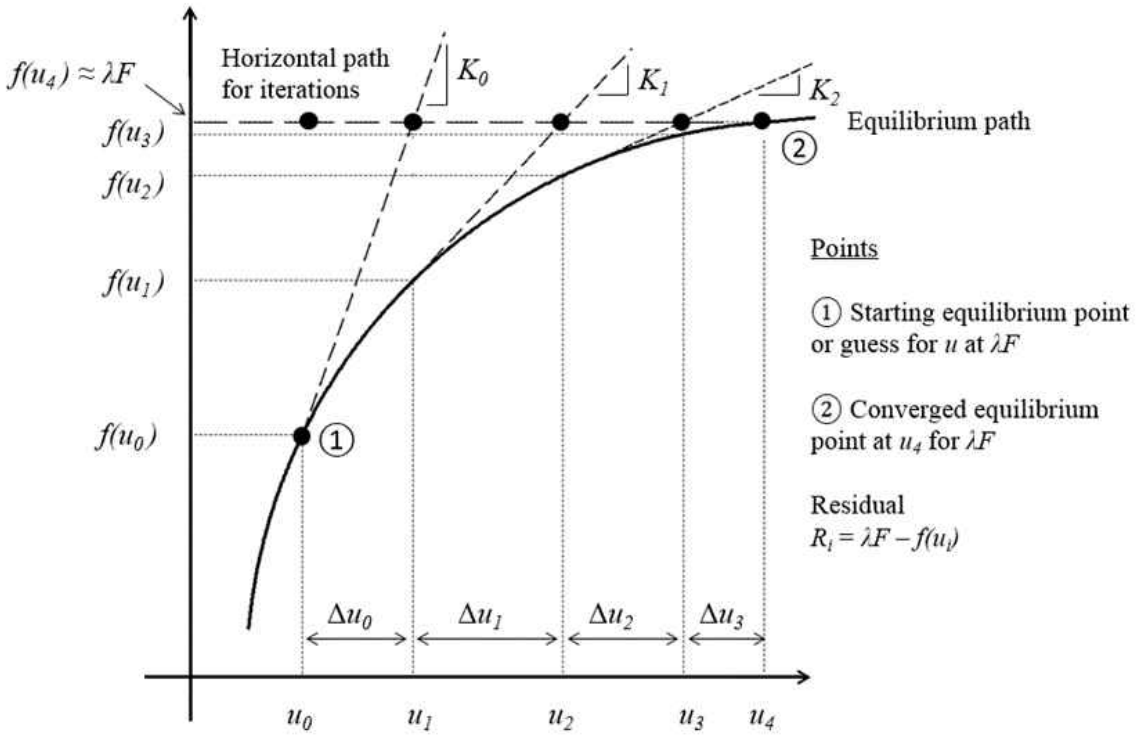

Figure 2. Newton-Raphson method for single-degree-of-freedom

Variations of the Newton-Raphson method exist to save on cost associated with computation and inversion of the tangent stiffness matrix when solving for $\Delta \boldsymbol{u}$. These variations include a modified Newton-Raphson method where $\boldsymbol{K}$ is held constant during iterations and a quasi-Newton method where a secant approximation for $\boldsymbol{K}$ is obtained by passing a line through two previously found points $(\boldsymbol{u}_i, \boldsymbol{f}(\boldsymbol{u}_i))$ on the equilibrium path. One of the most popular quasi-Newton methods is known as the Broyden-Fletcher-Goldfarb-Shanno or BFGS method [16-19]. Tensor methods [21] can also be considered a variation of the Newton-Raphson method as they

include second-order information from the Taylor series approximation of $\boldsymbol{f}(\boldsymbol{u})$. Instead of using a linear approximation of the nonlinear model, a quadratic approximation is used for an improved local model at points $\boldsymbol{u}_i$. Tensor methods will generally require fewer iterations to reach a solution as compared to Newton-Raphson methods due to the improved model. This increased convergence rate, however, comes at a cost due to added computation of the quadratic term. One example of a tensor solver implemented in commercial software can be found in the multi-body-dynamics analysis package MSC ADAMS [12].

The arc-length method differs from the Newton-Raphson method by incorporating to the procedure an additional constraint equation for the iteration path that allows λ to be treated as unknown. Graphical representation of two common variations of the constraint are shown in Fig. 3. A user specified arc-length L is provided, which controls the starting point of the iteration path used to search for a new point at the intersection with the equilibrium path. Early developments by Riks [45] and Wempner [30] were later updated by Ramm [32] to maintain symmetry of governing equations for finite element analysis. A normal path or hyperplane relative to arc-length L is used to search for new points on the equilibrium path for this case. Crisfield [31] made a further refinement and proposed using a circular or hyperspherical iteration path. The circular iteration path reduces λ by a larger amount as compared to the normal iteration path and would more likely intersect the equilibrium path near limit points for a given arc-length L. A detailed description of arc-length solvers and how they are implemented for nonlinear finite element analysis can be found in the MSC Nastran solution 400 user guide [11]. An overview of nonlinear solvers for finite element methods in general can be found in texts by Cook [2] and Bathe [1].

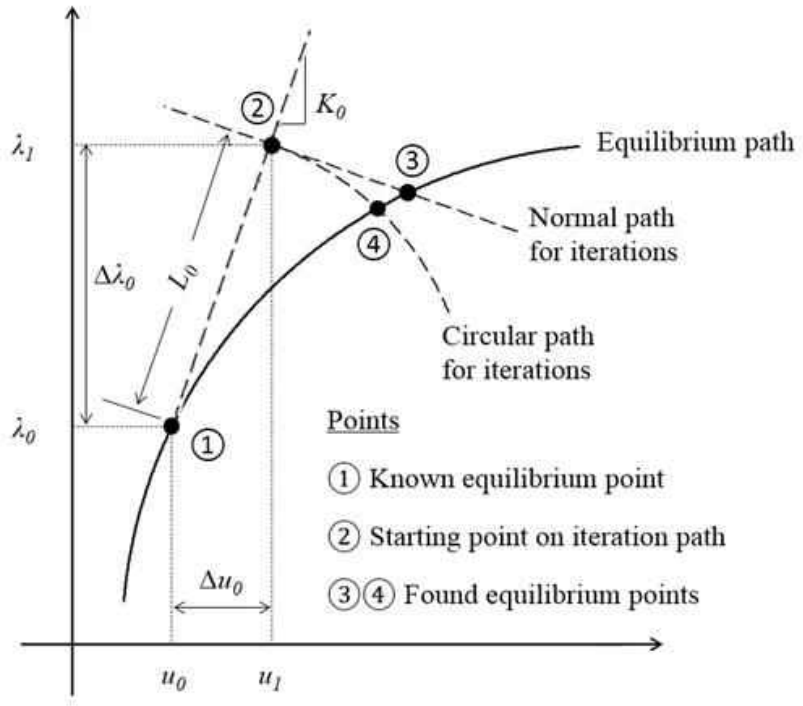

Figure 3. Arc-length method for single-degree-of-freedom

Solvers implemented with MATLAB's *fsolve* are based on trust-region methods used for numerical optimization where a minimization procedure is used to find roots of Eq. (1). Trust-region solvers are more robust than Newton-Raphson based solvers as they are able to handle cases where $\boldsymbol{K}$ is singular. Singularity becomes an issue near limit points as shown on Fig. 1 due to the zero or near zero slope condition of $\boldsymbol{K}$. Singularity of $\boldsymbol{K}$ will cause Newton-Raphson methods to fail or diverge from finding a solution due to the requirement of matrix inversion for finding updated values of $\boldsymbol{u}_i$. Further details on trust-region methods can be found in reference [14]. Although trust-region methods are more robust, they still do not have the capability of including λ as unknown due to the lack of a constraint equation for this variable in the algorithm. Similar to Newton-Raphson methods, a restart in analysis would be required for equilibrium path continuation when traversing limit points.

3.3 SOLVER SUITE DEVELOPMENT AND IMPLEMENTATION

Several solver variations were implemented using MATLAB for the purpose of numerical experiments involving systems of nonlinear equations. These solvers include Newton-Raphson, tensor, BFGS, and arc-length methods on a normal plane and sphere. Equation sets are defined using stand-alone functions or subroutines that are called by the solver. A main program or script file is used to run the solver and specify results formatting. A flowchart representation of the process is shown in Fig. 4. Names of the corresponding MATLAB files or m-files are identified on the chart, and some of these m-files are provided in the Appendices. The function used to define the system of nonlinear equations has the option for explicit definition of the Jacobian matrix. This definition was easy to implement for the small-scale demonstration problems used in this chapter but may not be practical if the system is large. An additional function is included to compute the Jacobian matrix numerically using a perturbation technique.

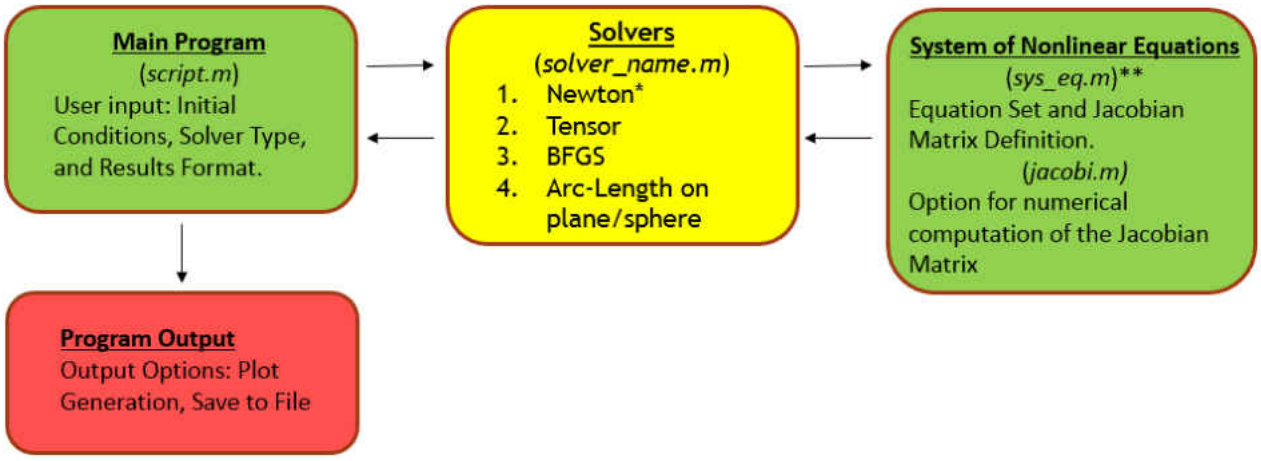

Figure 4. Program flowchart

3.4 DEMONSTRATION PROBLEMS

Two, three, and eleven degree-of-freedom (DOF) sample problems were chosen to demonstrate the arc-length solver capability and identify curves representing solutions to the equation sets. Eq. (4) consists of the two DOF system with an objective of characterizing the family of equilibrium pairs (u_1, u_2) parameterized by λ. Overlaid surface plots of f_1 and f_2 are shown in Fig. 5. For the case $\lambda = 0$, equilibrium solutions correspond to any intersections of the two surfaces that simultaneously occur at the level $f_1 = f_2 = 0$ in Fig. 5. No solutions exist for $\lambda = 0$. This conclusion is easily determined by finding the single polynomial equation for u_1 after eliminating u_2, by numerical factoring, and finally by noting all roots are complex numbers. For the case $\lambda \neq 0$, equilibrium solutions correspond to the intersections of the two surfaces in Fig. 5, and the corresponding level $f_1 = f_2 \neq 0$ determines the specific value of λ. Therefore, the vertical axis in Fig. 5 also denotes the value of λ for the intersection curve. A continuous family of solutions exists for $-1.0522 \leq u_1 \leq +1.0522$, $-0.2968 \leq u_2 \leq +1.2968$, and $+0.5000 \leq \lambda \leq +2.5497$. These results are easily determined by finding the quadratic equation for u_2 in terms of u_1 after enforcing $f_1 = f_2$ and by finding the range for u_1 where only real roots exist. Independent variables are plotted on the horizontal plane and function values are on the vertical axis. The solution is represented by the "shoe-shaped" curve representing the intersection of the two surfaces in Fig. 5.

$$\boldsymbol{f}(\boldsymbol{u}) - \lambda \mathbf{1} = \mathbf{0} \tag{4}$$

where

$$\boldsymbol{f}(\boldsymbol{u}) = \begin{Bmatrix} f_1(\boldsymbol{u}) \\ f_2(\boldsymbol{u}) \end{Bmatrix} = \begin{Bmatrix} u_1^6 + u_2^2 + 0.5 \\ u_1^2 + u_2 + 0.5 \end{Bmatrix}$$

The previous results are now used as a test case to validate the arc-length method algorithm and nonlinear solver suite code. Newton methods can be used initially and then switched to arc-length methods when λ reaches a local maximum or minimum on the curve. Alternately, arc-length methods can be used for the entire procedure. Specification of the arc-length parameter L will control spacing between points on the curve where specification of an incremental change in λ will control spacing between points for Newton methods. Several points consisting of combinations of ones and zeroes are easily identified by inspection of Eq. (4) and are used as starting locations for solvers that trace the equilibrium path. This accurate initialization will not necessarily be the case for more complex systems meaning an initial guess in the literal sense will be required. Although guessed points $(\boldsymbol{u}_0, \lambda_0)$ may not lie on the equilibrium curve, $\boldsymbol{f}(\boldsymbol{u}_0)$ and $\boldsymbol{K}_0$ would still be output to initialize the solver procedure. This procedure can be represented graphically where point ① on Figs. 2 and 3 no longer lies on the equilibrium curve and slope $\boldsymbol{K}_0$ is based on the fictitious point. An iteration path will still be established and the process will likely still converge to a point on the equilibrium path. The arc-length method using a circular iteration path may produce complex roots for initial converged values $(\boldsymbol{u}_1, \lambda_1)$ where this is typically not the case for known or previously computed initialization points. The appearance of complex roots is due to a quadratic equation used to enforce the circular constraint and the fact that guessed values will likely not fall on the equilibrium path where real number solutions exist. Through trial and error, it was found that taking the real component of the complex root for a subsequent guessed value provided satisfactory results for cases studied. Complex roots are not an issue for the other solvers unless they are inherited products of system $\boldsymbol{f}(\boldsymbol{u})$. In the event a complex solution is produced, it is rejected and new guesses are provided until a real solution is found.

When using the arc-length method with nearly arbitrary initialization, the equilibrium path or intersection curve in Fig. 5 is generated, exactly matching the known solution to a specified tolerance. Once a point was identified on the equilibrium path, both arc-length variants readily identified new points and followed the equilibrium path past limit points avoiding the need for restarts. Although arc-length methods can fail due to singularity issues similar to Newton methods, the procedure typically skips over and does not directly land on a limit point. If it does, an automatic reduction or increase in the specified arc-length could be made to deal with trouble locations. Arc-length methods were also able to find solutions for initial guess values of λ outside the solution range, such as $\lambda = 0$, since the method inherently varies λ. When $\lambda = 0$ is strictly enforced, surfaces intersect above the zero-plane and there is no solution to the equation set. Newton methods are limited to searching planes normal to the λ axis resulting in surface intersections at specified levels. When $\lambda = 0$ was specified as an initial guess for Newton methods, no solution was found as the algorithm did not converge. Attempts were also made using *fsolve* for the λ equal to zero initial guess and convergence could not be achieved.

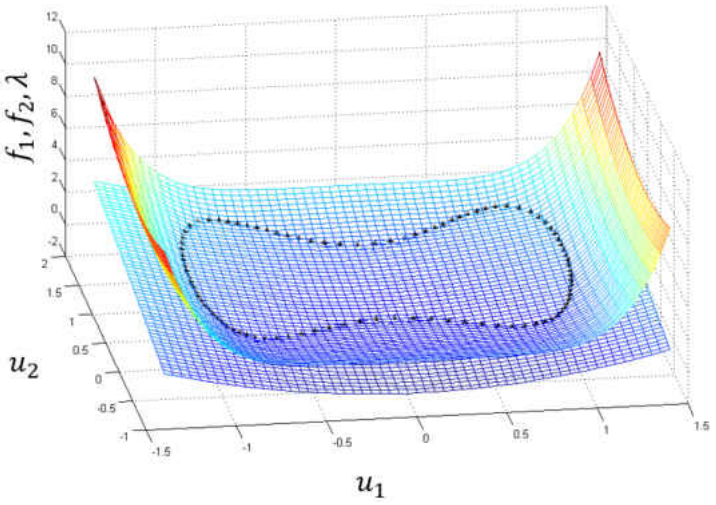

Figure 5. Intersecting surfaces (2 DOF)

A more complex three DOF system given in Eq. (5) was tested and results for selected variable u_1 are shown in Fig. 6. Solution accuracy was confirmed by back substituting into the nonlinear equation set and assessing the residuals. Curves generated for systems of three DOF and greater represent intersections of hypersurfaces in hyperspace and are best viewed graphically by plotting any of the selected degrees-of-freedom versus λ. Fig. 6 shows the sectional view of λ versus u_1. Inspection of the equations was required to establish allowable ranges for variables to avoid complex roots occurring from initial guesses needed to start the solvers. This treatment is due to square roots contained in Eq. (5) where it is seen that u_3 must be zero or a positive value and u_2 must be zero or a negative value. The solution curve was found not to close in this case, and the curves become asymptotic as λ continues to increase. If the objective was to find solutions for $\lambda = 0$, two zero-crossings can be identified. Monitoring for a sign change in λ was incorporated into the root extraction procedure to trigger the Newton-Raphson method and provide solutions precisely at $\lambda = 0$ where solutions are $(u_1 = 2, u_2 = -1, u_3 = 4)$ and $(u_1 = 0.6240, u_2 = -1.7880, u_3 = 5.9308)$ for Eq. (5). Due to λ being unknown in the arc-length method, zero-crossings will occur between positive and negative values of λ on the solution curve, and the Newton-Raphson method simply provides a small convenient adjustment to achieve the precisely desired condition.

$$\boldsymbol{f}(\boldsymbol{u}) - \lambda \mathbf{1} = \mathbf{0} \tag{5}$$

where

$$\boldsymbol{f}(\boldsymbol{u}) = \begin{Bmatrix} u_3^{1/2} + u_1^2 u_2^3 u_3 + u_3^{-1} + 4u_2 + 17.75 \\ (-u_3 u_2)^{1/2} + u_3^3 u_1 - u_2^{-2} + 3u_1 - 135 \\ u_1 u_2 u_3 + u_2^2 u_3 + u_1^2 u_3 - 3u_1 u_2 - 18 \end{Bmatrix}$$

Similar to Newton-methods, the *fsolve* algorithm was found to fail for initial guesses with λ values below the minimum range of -5 on Fig. 6. One of the biggest challenges for all solvers, including *fsolve*, was the possibility of producing complex values of $\boldsymbol{u}$. The *fsolve* algorithm appeared to be the most robust for not producing complex values from poor initial guesses but still lacked the capability to track the solution with relatively even spacing of points as compared to the arc-length method. In the near horizontal portion of the curve on Fig. 6 for example, a very fine increment in λ would be required when traversing upward. Solvers without arc-length control in this case can overshoot and lose track of the solution.

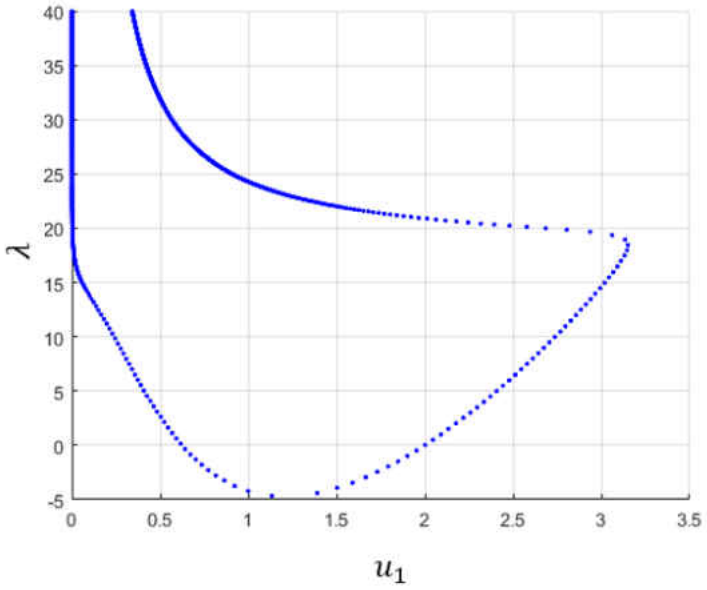

Figure 6. Solutions for λ versus u_1 (3 DOF)

An eleven DOF system given in Eq. (6) with results shown for a selected variable in Fig. 7 demonstrates the possibility of multiple complex shaped curves for solutions in hyperspace. Note the multi-values of u_2 for a single λ value on the curves. The larger closed curve is represented by a series of blue dots and the smaller by a series of small red circles to indicate found points. Equations are representative of the collapsing arch mechanical system found in

Chapter 4 and the initial value of $\boldsymbol{u}$ was based the known initial state or configuration of the system. Only a guessed value of λ would be required in this case. The *fsolve* algorithm, which included three variants using default settings, could not find a solution when $\lambda = 0$ was used as an initial guess. The algorithm could, however, find solutions for other values within the range shown on Fig. 7. Newton-based solvers were able to find a solution for the $\lambda = 0$ initial guess but jumped between curves for a $\lambda = 1$ initial guess. The arc-length method was able to trace both curves in this instance and a restart was only required for traversing the pointed section of the curve on the upper right-hand corner of Fig. 7. The curve was completed by changing the arc-length parameter from a positive to negative value to trace the curve in both clockwise and counter-clockwise directions from an initial starting point. The small curve could be traced in a single sweep.

$$\boldsymbol{f}(\boldsymbol{u}) - \lambda \mathbf{1} = \mathbf{0} \tag{6}$$

where

$$\boldsymbol{f}(\boldsymbol{u}) = \begin{Bmatrix} u_7 - u_9 \\ u_8 - u_{10} + mg \\ u_9 - k(u_3 + Lcos(u_6) - E) \\ u_{10} + u_{11} + mg \\ Lsin(u_5)(u_7 + u_9) - Lcos(u_5)(u_8 + u_{10}) \\ L\big(u_9 sin(u_6) - u_{10} cos(u_6) + u_{11} cos(u_6)\big) \ldots \\ + kLsin(u_6)\big(u_3 + Lcos(u_6 - E)\big) \\ u_1 - A_1 - Lcos(u_5) \\ u_2 - A_2 - Lsin(u_5) \\ u_3 - u_1 - L\big(cos(u_6) + cos(u_5)\big) \\ u_4 - u_2 - L\big(sin(u_6) + sin(u_5)\big) \\ u_4 + Lsin(u_6) \end{Bmatrix}$$

where

$$L = 5, A_1 = A_2 = 0, k = -0.1, mg = 1, E = 20\cos(\frac{\pi}{4})$$

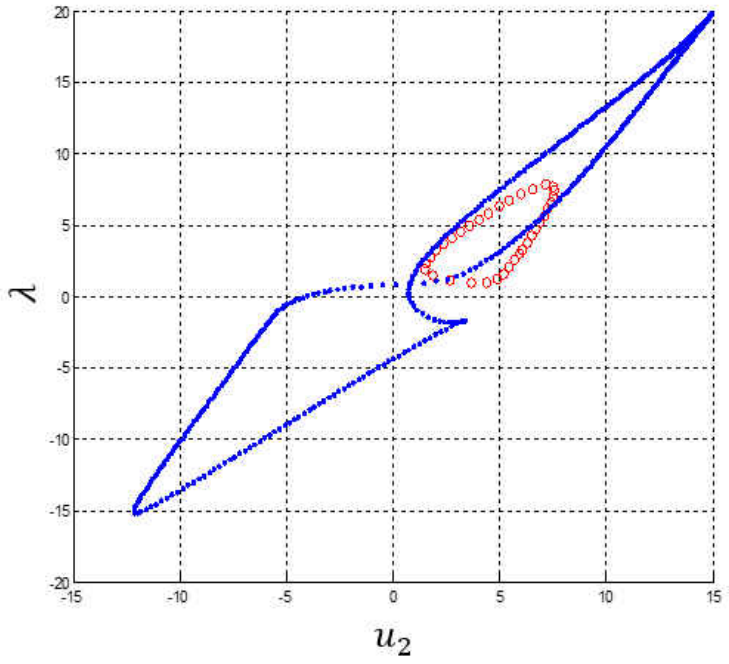

Figure 7. Solutions for λ versus u_2 (11 DOF)

3.5 CONCLUSIONS

Arc-length methods were found to be useful tools for solving systems of nonlinear equations and generating curves representing the many possible solutions for a given system. Newton-based solvers and MATLAB's *fsolve* would also be capable of generating similar curves but in a less robust manner. Arc-length solvers were found to minimize the need for restarts by continuing to track points on curves past and around limit points for smooth portions. Newton-based methods, on the other hand, would fail near limit points on curves and *fsolve* was also found to fail for cases studied if λ was outside the solution range. By readily tracing solution curves, arc-length methods helped identify variable bounds and zero-crossings of curves for the special solution to $\boldsymbol{f}(\boldsymbol{u}) = \boldsymbol{0}$. Without identifying solution curves or only discrete portions, critical or specific points could be missed. The *fsolve* algorithm appears more robust for finding

point solutions from poor initial guesses if λ is within the solution range as was the case for the three DOF system. The arc-length method was able to find solutions for guessed values of λ that were outside the solution range and worked well for the eleven DOF system where $\boldsymbol{u}$ was based on an initial physical state of the mechanical system. Without use of the arc-length method, solution curves would be difficult to trace due to likelihood of multiple restarts and the requirement for new guesses to search for new points on curves.

CHAPTER 4

EQUILIBRIUM FOR MULTI-BODY-DYNAMIC SYSTEMS

Determining states of static equilibrium for MBD systems can be challenging and may result in convergence failure for nonlinear static solvers. Analysts are often faced with uncertainty in regards to the quantity of candidate equilibrium states or whether a state of minimum potential energy was found. In the event of static solver failure or uncertainty with regards to a candidate solution, equilibrium could be obtained through a dynamic simulation which may require the addition of artificial damping. This method, however, can have significant computational expense as compared to static solution procedures. Using simple MBD systems representing a pendulum and two variations of a spring supported arch, arc-length solvers were found suitable for identifying equilibrium states through a robust production of static solution curves thereby avoiding dynamic simulation. Using these examples, a procedure for finding the correct equilibrium state for general systems is proposed.

4.1 INTRODUCTION

This chapter is an expansion of work documented in Ref. [51] and Chapter 3 where arc-length solvers [30,31,45] were applied to general systems of nonlinear equations in search of the many possible solutions for a parameterized system versus the special case of the zero parameter solutions. Arc-length solvers have the unique capability of following solution curves past turning or limit points and are less likely to fail or require restart as compared to other

parameterized solvers. Graphical representation of the total solution set was plotted using sectional views of the multi-dimensional hyperspace where any of the independent state variables could be plotted against the dependent variable denoted as λ. In the arc-length method, λ is treated as an unknown and computed from an additional constraint equation. Plotting the solution in this manner revealed a path or curve representing the intersection of hypersurfaces within the specified sectional view. Sectional views of systems studied demonstrated the possibility of closed curves, curves that self-intersect, multiple curves, and open curves that reach an asymptotic limit implying an intersection of surfaces that become parallel. Possibility of multiple roots or solutions was identified as curves tended to cross the λ equal to zero axis at more than just one location. Zero-crossings of the path were of particular interest for systems studied in this chapter as they represent candidate equilibrium states as equations were based on physical systems. Mechanical systems including a pendulum and variations of a spring supported v-shaped arch were used to develop theory and a proposed method for selecting equilibrium. A variety of selection criteria were used to identify equilibrium including potential energy, eigenvalues, and solution curves generated using an arc-length solver.

Equations of motion for the mechanical systems used in this study were derived using Lagrange's method [52]. Reducing the derived differential equations to first-order and coupling them with the system's algebraic constraint equations resulted in sets of differential and algebraic equations that could be solved numerically using various methods. This procedure can be automated and is central to commercial multi-body-dynamics software MSC ADAMS. A detailed explanation on how this procedure is implemented including derivation of equations for the pendulum used as the starting example can be found in Ref. [53]. Modeling of mechanical or other dynamic systems typically results in initial configurations or states that are not in

equilibrium. In the spring supported arch under the influence of gravity for example, the spring may not be exactly extended or compressed from its free length to balance the applied load. The search for equilibrium in this case can be done either dynamically with added damping for a decayed response towards the static configuration, or statically where time dependent terms in the governing equations are set to zero and an attempt to solve the resulting system is performed. The dynamic approach has an obvious computational expense due to implementing a nonlinear solver at every time step versus the cheaper static approach where the solver is implemented only once. The static solution, however, includes risk for converging to a rest-state that numerically satisfies equations but does not represent equilibrium. As noted in Ref. [53], the static solution for a pendulum may align with the gravity vector and converge to an upward pointing vertical or unstable configuration versus the stable downward configuration.

Finding equilibrium statically using arc-length solvers and previously mentioned selection criteria are the primary focus of this chapter. Arc-length solvers are typically used for tracking nonlinear events such as post-buckling or snap-through in structures and have been successfully implemented in commercial finite element codes such as MSC Nastran [11] and Abaqus [9]. They may also be referred to as numerical path following [38] or continuation methods [37] where such terminology may be more familiar to mathematicians. There appears to be little cross-fertilization between the mathematical and engineering communities as noted by Felippa [35] and use of arc-length solvers in general seems limited. Current implementation in multi-body-dynamics codes, including MSC ADAMS [12], could not be found from literature review. Based on previous work in Ref. [51], arc-length solvers will increase likelihood for finding a solution where other solvers may fail, and help identify the many candidate or numerically feasible equilibrium states using the generated solution curves. Note that arc-length

solvers and proposed methods will require many static solutions for construction of curves; however, this can be viewed as a compromise between a one-time static approach that may converge to an improper solution or fail and a full dynamic simulation.

Efficiency of arc-length or other Newton-Raphson based solvers is primarily a function of the computational cost associated with calculation and factorization of the tangent stiffness or Jacobian matrix and use of a modified Newton-Raphson method can be effective in reducing this cost [1]. Modified methods hold the Jacobian constant or only update it periodically during solver iterations or search for a solution. While this tends to increase the number of solver iterations required for convergence, overall computational cost and wall time can be significantly less. In addition to minimizing the computation and factorization of the Jacobian, known patterns, invariant terms, sparsity, and parallel operations can be taken advantage of as well. A detailed study relevant to computation of the Jacobian for multi-body-dynamic systems can be found in Chapter 5.

The structure of this chapter begins with a description of a parameterized Newton-Raphson solver followed by two variations of arc-length solvers that modify Newton-Raphson with an additional constraint. Governing equations for a pendulum and spring supported arch are presented along with solution curves obtained using an arc-length solver and the found candidate equilibrium states. The paper ends with a proposed method for selecting equilibrium and conclusions.

4.2 THEORY AND METHODS FOR PARAMETERIZED NEWTON-RAPHSON

A common approach for finding static equilibrium in multi-body-dynamic systems involves setting time dependent terms in governing DAEs to zero and solving the remaining nonlinear algebraic equations using some variant of a Newton-Raphson solver [53]. The general format for such systems may be written in compact form as in Eq. (1) where the objective is to find vector $\boldsymbol{u}$ that makes all equations within vector $\boldsymbol{f}$ equal to zero. Searching for other than zero solutions as part of a path following or continuation method requires inclusion of an additional parameter such that Eq. (1) can be redefined as given in Eq. (2) where scalar λ can be any real number and $\boldsymbol{F}$ is a reference vector set to all ones. Solving of Eq. (2) requires linearization about a local point $\boldsymbol{u}_i$ through a first-order Taylor series expansion resulting in

$$\boldsymbol{f}(\boldsymbol{u}) - \lambda \boldsymbol{F} \approx \boldsymbol{f}(\boldsymbol{u}_i) + \left(\frac{\partial \boldsymbol{f}}{\partial \boldsymbol{u}}\right)_i (\boldsymbol{u} - \boldsymbol{u}_i) - \lambda \boldsymbol{F} = \boldsymbol{0} \tag{7}$$

Unknown vector $\boldsymbol{u}$ will be referred to as the state vector and contains information on position, velocity, and constraint forces for the mechanical systems being modeled. Updating terms in Eq. (7), which includes the matrix of first-order partial derivatives for $\boldsymbol{f}$ and the incremental change or difference between $\boldsymbol{u}$ and $\boldsymbol{u_i}$, results in

$$\boldsymbol{f}(\boldsymbol{u}_i) + \boldsymbol{K}_i \Delta \boldsymbol{u}_i - \lambda \boldsymbol{F} = \boldsymbol{0} \tag{8}$$

Eq. (8) may now be solved using a Newton-Raphson method where a value for λ needs to be specified. A stepwise procedure for solving Eq. (8) is given in Subsection 4.2.1 with graphical representation shown in Fig. 2. Note that results only for λ equal to zero are admissible solutions for equilibrium as other values of λ modify governing equations with a scalar offset for non-zero solutions to equations. Static solution curves can be constructed through incremental variation of λ as part of a path following method in an attempt to identify additional equilibrium states.

Though plausible, such an approach would be difficult using Newton-Raphson due to failure at limit or turning points on solution curves which is better suited for arc-length solvers where λ is treated as an unknown and path following is more easily achieved.

4.2.1 STEPWISE PROCEDURE FOR PARAMETERIZED NEWTON-RAPHSON

1. Specify a value for λ and provide an estimate or initial guess for state $\boldsymbol{u}_i$ at $\lambda \boldsymbol{F}$. Use iteration count $i = 0$ to begin the process.
2. Calculate $\boldsymbol{K}_i$ or the matrix of first-order partial derivatives with respect to state variables in vector $\boldsymbol{u}_i$ where

$$\boldsymbol{K}_i = \begin{bmatrix} \frac{\partial f_1}{\partial u_1} & \cdots & \frac{\partial f_1}{\partial u_N} \\ \vdots & \ddots & \vdots \\ \frac{\partial f_N}{\partial u_1} & \cdots & \frac{\partial f_N}{\partial u_N} \end{bmatrix}$$

3. Calculate system vector $\boldsymbol{f}(\boldsymbol{u}_i)$.
4. Determine the residual or difference between $\lambda \boldsymbol{F}$ and $\boldsymbol{f}(\boldsymbol{u}_i)$ where $\boldsymbol{R}_i = \lambda \boldsymbol{F} - \boldsymbol{f}(\boldsymbol{u}_i)$.
5. Calculate $\Delta \boldsymbol{u}_i$, where $\Delta \boldsymbol{u}_i = \boldsymbol{K}_i^{-1} \boldsymbol{R}_i$.
6. Check if $\Delta \boldsymbol{u}_i$ is small with respect to $\boldsymbol{u}_i$. This check may be done by taking the ratio of vector norms or absolute values for single-degree-of-freedom systems and seeing if this is less than a user-specified error tolerance. Is $\|\Delta \boldsymbol{u}_i\| / \|\boldsymbol{u}_i\|$ less than the specified error tolerance?
7. If yes, stop, the solution has been obtained; otherwise, update both the iteration count and $\boldsymbol{u}_i$ and repeat the procedure starting with step 2. The updated value of $\boldsymbol{u}_i$ is obtained by adding $\Delta \boldsymbol{u}_i$ to the current value of $\boldsymbol{u}_i$ where $\boldsymbol{u}_{i+1} = \boldsymbol{u}_i + \Delta \boldsymbol{u}_i$.

8. If a solution has been obtained and search for other nearby solutions is desired as part of a path following procedure, restart the procedure beginning with step 1. Use the previously found solution as an initial guess and specify a new value for λ that represents a small change away from that solution.

4.3 THEORY AND METHODS FOR TWO VARIATIONS OF ARC-LENGTH

The arc-length method is similar to the Newton-Raphson method with the exception to λ being unknown. The variable $\Delta\lambda_i$ is introduced for use in incremental form and is defined as the difference between unknown and known values of λ at iterations $i+1$ and i where $\Delta\lambda_i = \lambda_{i+1} - \lambda_i$. Eq. (8) is now modified as

$$\boldsymbol{f}(\boldsymbol{u}_i) + \boldsymbol{K}_i\Delta\boldsymbol{u}_i - (\Delta\lambda_i + \lambda_i)\boldsymbol{F} = \boldsymbol{0} \tag{9}$$

At equilibrium, both $\Delta\lambda_i$ and $\Delta\boldsymbol{u}_i$ become very small such that the difference between $\lambda_i\boldsymbol{F}$ and $\boldsymbol{f}(\boldsymbol{u}_i)$ or residual $\boldsymbol{R}_i$ is minimized. Iterations are typically stopped when a user-specified error tolerance on $\Delta\boldsymbol{u}_i$ has been achieved. Solving of Eq. (9), however, requires an additional equation as there is now an additional unknown variable. The additional equation constrains iterations to a defined path with two common variations being a normal path [30,45] or circular path [31]. The starting location for the iteration path is based on a user-specified arc-length L that controls the magnitude of the initial $\Delta\boldsymbol{u}_i$ and $\Delta\lambda_i$ terms. The arc-length is made tangent to a known equilibrium point or alternately a guessed fictitious point using the tangent stiffness matrix $\boldsymbol{K}$, which can have either positive or negative slope based on the matrix determinant. Once the initial point at the end of the arc-length has been found, the residual is checked and iterations are performed along the specified path until convergence or a limit on iterations is

achieved. Fig. 3 shows how the arc-length is used to provide an initial guess or starting point on the iteration path. Both λ and $\boldsymbol{u}$ are varied through the process which extends the search for new points on the equilibrium path to the normal or circular path as shown in Fig. 3 versus the horizontal path used by the Newton-Raphson method in Fig. 2. This adjustment is what allows arc-length solvers to track equilibrium paths or solution curves for given systems that may suddenly change slope or direction in a more robust manner as compared to others solvers. Due to $\boldsymbol{F}$ being constant, it has been left off of the remaining figures for clarity. Known or starting points correspond to an iteration count of zero or point $(\boldsymbol{u}_0, \lambda_0)$ on figures. In the event the method fails to converge, the magnitude of the arc-length L can be reduced to a smaller value and the process repeated. This logic can be done by reducing the arc-length by a factor such as one half.

Arc-length methods may also be referred to as predictor-corrector or pseudo arc-length continuation methods [37,38]. The term pseudo is best understood with respect to the single-degree-of-freedom system shown in Fig. 3 where a tangent line L_0 of slope K_0 is used to approximate the length of the arc or curve between points ① and ③ or ① and ④ depending on the iteration path being used. Point ② at the end of the tangent line can also be referred to as a predictor for points ③ or ④ on the curve. Corresponding linear model iterations would then be performed as correctors along the iteration path until a converged solution or intersection with the equilibrium path is found. The basic concept behind numerical path following or continuation using this approach is that a series of tangent lines or approximate arc-lengths serve as an ideal method for parameterization of a given curve.

4.3.1 ARC-LENGTH METHOD USING NORMAL ITERATION PATH

Perhaps the most straightforward implementation of the arc-length method is to constrain iterations to a normal path. This path may also be referred to as a plane or hyperplane to emphasize use for multi-degree-of-freedom systems. Because the arc-length L is specified and tangent stiffness $\boldsymbol{K}$ can be calculated at known states, $\Delta\boldsymbol{u}_0$ and $\Delta\lambda_0$ can be calculated using length relations for a right triangle and Eq. (9). Terms $\boldsymbol{f}(\boldsymbol{u}_i)$ and $\lambda_i\boldsymbol{F}$ cancel for points lying on the equilibrium path resulting in the following set of equations.

$$\boldsymbol{K}_i\Delta\boldsymbol{u}_i - \Delta\lambda_i\boldsymbol{F} = \boldsymbol{0} \tag{10}$$

$$L_i^2 \overset{\text{def}}{=} (\Delta\lambda_i)^2 + (\Delta\boldsymbol{u}_i)^T\Delta\boldsymbol{u}_i \tag{11}$$

Subscripts in these equations are for i equal to zero as they are based on a known equilibrium point or initial configuration. The two equations can be solved using a second coincident triangle where the length of one edge is specified. Unknown variables are found based on equivalent length ratios as shown in Fig. 8 where numbered points correspond to those found on Fig. 3. Eq. (11) is commonly used to define arc-length L, which is a "distance" in the λ; $\boldsymbol{u}$ space with inconsistent dimension since $\Delta\boldsymbol{u}_i$ has possibly mixed dimensions of position; velocity; and force for mechanical applications while $\Delta\lambda_i$ is dimensionless. Alternately, a normalizing factor can be applied to the $\Delta\boldsymbol{u}_i$ product in Eq. (11) to render this term dimensionless and thereby define L in a consistent sense [1]. An update rule for arc-length values L_i can be specified based on local curvature of the equilibrium path, or simply held constant. Computations in the dissertation used fixed values of arc-length and automatically reduced this parameter by one-half in the event the solver failed to converge within a specified number of iterations.

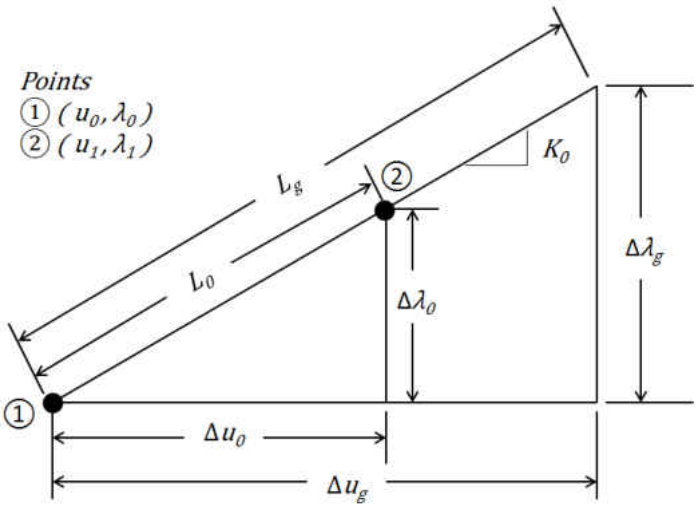

Figure 8. Calculation of $\Delta \boldsymbol{u}_0$ and $\Delta\lambda_0$ for single-degree-of-freedom system

The subscript g in Fig. 8 is used to denote the given or specified value $\Delta\lambda_g$, which is typically set to one. The sign of $\Delta\lambda_g$ depends on whether a positive or negative slope is used for the arc-length. For multi-degree-of-freedom systems this is based on the sign of the matrix determinant of $\boldsymbol{K}$. The ability to control the slope of the arc-length is what allows the arc-length method to change direction and traverse turning or limit points on the equilibrium path. $\Delta \boldsymbol{u}_g$ is calculated using Eq. (10) and L_g is calculated using Eq. (11) where subscript i is replaced with g. Length ratios can then be set up between the two triangles and unknown values of $\Delta \boldsymbol{u}_0$ and $\Delta\lambda_0$ are calculated using

$$\frac{\Delta\lambda_0}{L_0} = \frac{\Delta\lambda_g}{L_g} \tag{12}$$

$$\frac{\Delta \boldsymbol{u}_0}{L_0} = \frac{\Delta \boldsymbol{u}_g}{L_g} \tag{13}$$

Once the values of $\Delta \boldsymbol{u}_0$ and $\Delta\lambda_0$ are calculated using Eq. (12) and Eq. (13), the point at the end of the arc-length is found by adding these values to the previous known point. This step provides

a starting point for iterations where λ_i and $\boldsymbol{u}_i$ terms are updated throughout the procedure by adding incremental changes to previous estimates for the solution.

$$\boldsymbol{u}_{i+1} = \boldsymbol{u}_i + \Delta\boldsymbol{u}_i \tag{14}$$

$$\lambda_{i+1} = \lambda_i + \Delta\lambda_i \tag{15}$$

Corresponding iteration points $(\boldsymbol{u}_i, \lambda_i)$ are projected normal to the arc-length such that the dot product of the vectors used to define the arc-length and corresponding points on the iteration path is zero based on orthogonal orientation of vectors. The equilibrium and constraint equations used for solving unknown $\Delta\boldsymbol{u}_i$ and $\Delta\lambda_i$ are defined respectively as

$$\boldsymbol{K}_i\Delta\boldsymbol{u}_i = \boldsymbol{R}_i + \Delta\lambda_i\boldsymbol{F} \tag{16}$$

$$\lfloor(\Delta\boldsymbol{u}_0)^T\ \Delta\lambda_0\rfloor\begin{Bmatrix}\Delta\boldsymbol{u}_i\\ \Delta\lambda_i\end{Bmatrix} = 0 \tag{17}$$

The row vector in Eq. (17) defines the arc-length, which remains constant during iterations where the column vector defines the unknown locations on the path normal to the arc-length. The equations are typically solved by splitting $\Delta\boldsymbol{u}_i$ in Eq. (16) into two components $\Delta\boldsymbol{u}_i^I$ and $\Delta\boldsymbol{u}_i^{II}$, which are obtained based on known vector $\boldsymbol{F}$ and residual $\boldsymbol{R}_i$ at state $\boldsymbol{u}_i$. The purpose of this two-stage solution procedure is to maintain symmetry of the Jacobian or tangent stiffness matrix $\boldsymbol{K}$ [3]. Calculation of the $\Delta\boldsymbol{u}_i$ components is done by the following steps where $\Delta\lambda_i$ is temporarily set to one.

$$\boldsymbol{K}_i\Delta\boldsymbol{u}_i^I = \boldsymbol{F} \tag{18}$$

$$\boldsymbol{K}_i\Delta\boldsymbol{u}_i^{II} = \boldsymbol{R}_i \tag{19}$$

Vector $\Delta\boldsymbol{u}_i^{II}$ is based on the residual $\boldsymbol{R}_i$ and can be thought of as a predictor for the next value of $\Delta\boldsymbol{u}_i$, where $\Delta\boldsymbol{u}_i^I$ is a corrector to bring calculated points back to the iteration path. Scalar $\Delta\lambda_i$ must later be used in combination with $\Delta\boldsymbol{u}_i^I$ when recombining terms back into $\Delta\boldsymbol{u}_i$. Graphical

depiction of $\Delta \boldsymbol{u}_i^I$ and $\Delta \boldsymbol{u}_i^{II}$ and how they relate to the arc-length and normal iteration path are shown in Fig. 9 and Fig. 10 where the index $i = 0$ refers to the known or starting equilibrium point, $i = 1$ the first iteration point, $i = 2$ the second and so on. Labeled points correspond to ① for the point located at the end of the arc-length and start of the iteration path, ② for the predicted value of $\Delta \boldsymbol{u}_i$ using $\Delta \boldsymbol{u}_i^{II}$, and ③ for the corrected value of $\Delta \boldsymbol{u}_i$ using $\Delta \boldsymbol{u}_i^I$ and $\Delta \lambda_i$.

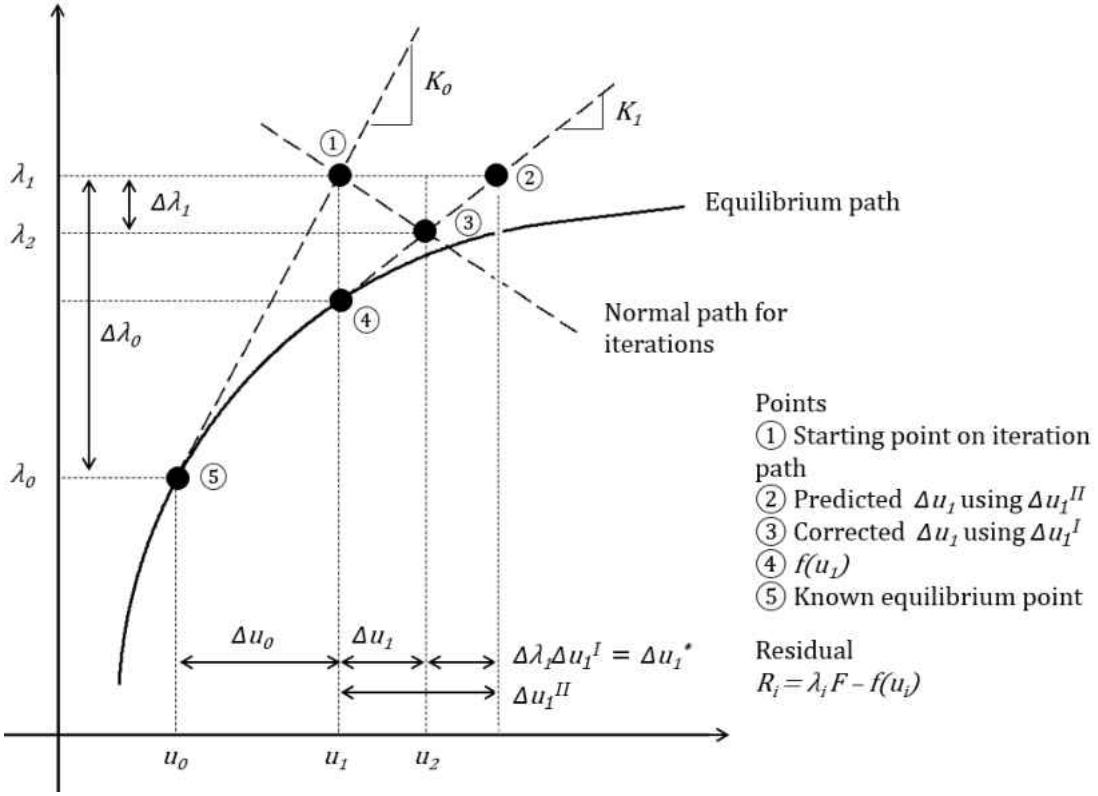

Figure 9. Points on normal iteration path for single-degree-of-freedom system

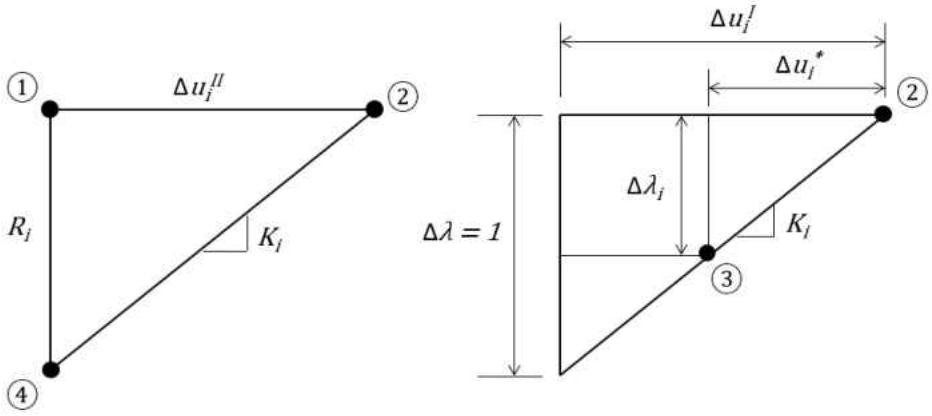

Figure 10. Components of $\Delta \boldsymbol{u}_i$ for single-degree-of-freedom system

The term $\Delta \boldsymbol{u}_i^*$ in Fig. 10 is solved using similar triangles and shows how $\Delta\lambda_i$ combines with $\Delta \boldsymbol{u}_i^I$ to bring $\Delta \boldsymbol{u}_i$ back to the iteration path. Scalar $\Delta\lambda$ was previously set to one for determination of $\Delta \boldsymbol{u}_i^I$ such that length ratios of equivalent triangles can be used to solve for $\Delta \boldsymbol{u}_i^*$ by

$$\frac{\Delta \boldsymbol{u}_i^*}{\Delta\lambda_i} = \frac{\Delta \boldsymbol{u}_i^I}{1} \tag{20}$$

$$\Delta \boldsymbol{u}_i^* = \Delta\lambda_i \Delta \boldsymbol{u}_i^I \tag{21}$$

Observe on Fig. 9 how $\Delta \boldsymbol{u}_i^{II}$ overshoots $\Delta \boldsymbol{u}_i$ requiring $\Delta \boldsymbol{u}_i^*$ or scaled $\Delta \boldsymbol{u}_i^I$ to bring calculated points back to the iteration path. Combining $\Delta \boldsymbol{u}_i^I$ and $\Delta \boldsymbol{u}_i^{II}$ terms back into $\Delta \boldsymbol{u}_i$ is done by adding the $\Delta \boldsymbol{u}_i^{II}$ and $\Delta \boldsymbol{u}_i^*$ terms where $\Delta\lambda_i$ scales $\Delta \boldsymbol{u}_i^I$ accordingly.

$$\Delta \boldsymbol{u}_i = \Delta \boldsymbol{u}_i^{II} + \Delta\lambda_i \Delta \boldsymbol{u}_i^I \tag{22}$$

$\Delta\lambda_i$ can now be solved using Eq. (17) where $\Delta \boldsymbol{u}_i$ is rewritten in terms of Eq. (22). The resulting expression for $\Delta\lambda_i$ after the substitution is

$$\Delta\lambda_i = \frac{-\Delta \boldsymbol{u}_0{}^T \Delta \boldsymbol{u}_i^{II}}{\Delta \boldsymbol{u}_0{}^T \Delta \boldsymbol{u}_i^I + \Delta\lambda_0} \tag{23}$$

where the negative sign for the expression becomes part of Eq. (22) for the final differencing of the $\Delta \boldsymbol{u}_i^{II}$ and scaled $\Delta \boldsymbol{u}_i^{I}$ terms.

The goal of the arc-length method is to minimize the $\Delta\lambda_i$ and $\Delta \boldsymbol{u}_i$ terms through an iterative process similar to the Newton-Raphson method. Once these terms are known, convergence is checked against a user-specified error tolerance. If convergence criteria are met, iterations are stopped and equilibrium or the solution is considered found. If not, $\boldsymbol{u}_i$ and λ_i are updated using Eq. (14) and Eq. (15) and the process is repeated. The following procedure in Subsection 4.3.1.1 summarizes the arc-length method on a normal path.

4.3.1.1 STEPWISE PROCEDURE FOR ARC-LENGTH METHOD ON NORMAL PATH

1. Specify an arc-length L to establish a search range for solutions away from a known or guessed point $(\boldsymbol{u}_i, \lambda_i)$. Use iteration count $i = 0$ to begin the process.
2. Calculate the tangent stiffness matrix $\boldsymbol{K}_i$ at state $\boldsymbol{u}_i$.
3. Find the sign of the matrix determinant of $\boldsymbol{K}_i$ to determine the slope of the tangent plane.
4. Solve for $\Delta \boldsymbol{u}_0$ and $\Delta\lambda_0$ using Eq. (12) and Eq. (13).
5. Find the point at the end of the arc-length or start of the iteration path using Eq. (14) and Eq. (15).
6. Calculate system vector $\boldsymbol{f}(\boldsymbol{u}_i)$ and tangent stiffness $\boldsymbol{K}_i$ at the new state $\boldsymbol{u}_i$ where i is updated for the next iteration count.
7. Determine the residual where $\boldsymbol{R}_i = \lambda_i \boldsymbol{F} - \boldsymbol{f}(\boldsymbol{u}_i)$.
8. Calculate $\Delta \boldsymbol{u}_i^{I}$ where $\Delta \boldsymbol{u}_i^{I} = \boldsymbol{K}_i^{-1} \boldsymbol{F}$.
9. Calculate $\Delta \boldsymbol{u}_i^{II}$ where $\Delta \boldsymbol{u}_i^{II} = \boldsymbol{K}_i^{-1} \boldsymbol{R}_i$.

10. Calculate $\Delta \boldsymbol{u}_i$ and $\Delta\lambda_i$ using Eq. (22) and Eq. (23).
11. Check if $\Delta \boldsymbol{u}_i$ is small with respect to $\boldsymbol{u}_i$ and $\Delta\lambda_i$ is small with respect to λ_i. This test may be done by taking the ratio of vector norms and seeing if they are less than a given user-specified error tolerance. Are $\|\Delta \boldsymbol{u}_i\|/\|\boldsymbol{u}_i\|$ and $|\Delta\lambda_i|/|\lambda_i|$ less than the specified error tolerance?
12. If yes, stop, the solution has been obtained; otherwise, update the iteration count and estimates for $\boldsymbol{u}_i$ and λ_i to improved values using Eq. (14) and Eq. (15) and repeat the procedure starting with step 6.
13. If a solution has been obtained and search for other nearby solutions is desired as part of a path following procedure, restart the procedure beginning with step 1. Use the previously found solution as an initial guess. Arc-length L can be held constant or reduced in length as needed for restart in the event of convergence failure.

4.3.2 ARC-LENGTH METHOD USING CIRCULAR ITERATION PATH

Use of a circular path for iterations versus a normal path requires modification of constraint Eq. (17). This path may also be referred to as a sphere or hypersphere. The arc-length L must be included in the constraint as it defines the radius of the circular path that remains constant during iterations and centered at point $(\boldsymbol{u}_0, \lambda_0)$. The process begins by defining the arc-length radius as a vector where

$$\boldsymbol{r}_0 = \begin{Bmatrix} \Delta \boldsymbol{u}_0 \\ \Delta\lambda_0 \end{Bmatrix} \tag{24}$$

Components of $\boldsymbol{r}_0$ are found using Eq. (12) and Eq. (13) for iteration count i equal to zero. The point located at the end of $\boldsymbol{r}_0$ or $(\boldsymbol{u}_1, \lambda_1)$ defines the start of the iteration path, which is similar to

the previous method. Corresponding points on the circular iteration path are then located using the current radius and to be determined incremental changes in $\boldsymbol{u}$ and λ.

$$\boldsymbol{r}_{i+1} = \boldsymbol{r}_i + \begin{Bmatrix} \Delta \boldsymbol{u}_{i+1} \\ \Delta \lambda_{i+1} \end{Bmatrix} = \begin{Bmatrix} \boldsymbol{r}_{i1} + \Delta \boldsymbol{u}_{i+1} \\ r_{i2} + \Delta \lambda_{i+1} \end{Bmatrix} \tag{25}$$

The constraint equation for the circular path is based on the constant magnitude of vector $\boldsymbol{r}$ or arc-length L and is defined using the dot product as

$$\boldsymbol{r}_{i+1} \cdot \boldsymbol{r}_{i+1} = L_i^2 \tag{26}$$

or

$$(r_{i2} + \Delta \lambda_{i+1})^2 + \boldsymbol{r}_{i1}{}^T \boldsymbol{r}_{i1} + 2\boldsymbol{r}_{i1}{}^T \Delta \boldsymbol{u}_{i+1} + \Delta \boldsymbol{u}_{i+1}{}^T \Delta \boldsymbol{u}_{i+1} = L_i^2 \tag{27}$$

through substitution of Eq. (25). Arc-length L can be written in terms of the sum of the squares of vector components as

$$L_i^2 = r_{i2}{}^2 + \boldsymbol{r}_{i1}{}^T \boldsymbol{r}_{i1} \tag{28}$$

Substituting Eq. (28) into Eq. (27) yields the resulting constraint equation used for the circular iteration path.

$$\Delta \lambda_{i+1}{}^2 + 2r_{i2} \Delta \lambda_{i+1} + 2\boldsymbol{r}_{i1}{}^T \Delta \boldsymbol{u}_{i+1} + \Delta \boldsymbol{u}_{i+1}{}^T \Delta \boldsymbol{u}_{i+1} = 0 \tag{29}$$

This equation in conjunction with Eq. (16) are used to solve for the unknown incremental changes in $\boldsymbol{u}$ and λ. Subscripts in Eq. (16) are updated to the $i + 1$ iteration count for compatibility with Eq. (29).

$$\boldsymbol{K}_{i+1} \Delta \boldsymbol{u}_{i+1} = \boldsymbol{R}_{i+1} + \Delta \lambda_{i+1} \boldsymbol{F} \tag{30}$$

Vector $\Delta \boldsymbol{u}_{i+1}$ is broken into two components as was done for the method on a normal path using Eq. (22) where subscripts are also updated.

$$\Delta \boldsymbol{u}_{i+1} = \Delta \boldsymbol{u}_{i+1}^{II} + \Delta \lambda_{i+1} \Delta \boldsymbol{u}_{i+1}^{I} \tag{31}$$

Substituting Eq. (31) into Eq. (29) yields the final expression for $\Delta \lambda_{i+1}$, which is quadratic in the unknown and having roots $(\Delta \lambda_{i+1})_1$ and $(\Delta \lambda_{i+1})_2$.

$$\left(1 + \Delta \boldsymbol{u}_{i+1}^{I}{}^{T} \boldsymbol{u}_{i+1}^{I}\right) \Delta\lambda_{i+1}{}^{2} + 2\left(r_{i2} + \boldsymbol{r}_{i1}{}^{T} \Delta \boldsymbol{u}_{i+1}^{I} + \Delta \boldsymbol{u}_{i+1}^{I}{}^{T} \Delta \boldsymbol{u}_{i+1}^{II}\right) \Delta\lambda_{i+1} \tag{32}$$

$$+ 2\boldsymbol{r}_{i1}{}^{T} \Delta \boldsymbol{u}_{i+1}^{II} + \Delta \boldsymbol{u}_{i+1}^{II}{}^{T} \Delta \boldsymbol{u}_{i+1}^{II} = 0$$

The correct value of $\Delta\lambda_{i+1}$ is found by looking at the value of the angle between known vector $\boldsymbol{r}_i$ and tentative vector $\boldsymbol{r}_{i+1}$. Selection is made by choosing the value of $\Delta\lambda_{i+1}$, which produces a maximum value of the cosine between the two vectors such that the new vector is closest to the current.

$$cos(\theta) = \frac{\boldsymbol{r}_i \cdot \boldsymbol{r}_{i+1}}{L_i^2} \tag{33}$$

Once values of $\Delta\lambda_{i+1}$ and $\Delta \boldsymbol{u}_{i+1}$ have been determined, convergence is checked and iterations are continued until a solution is found or specified criteria indicating divergence or failure stops the process. Graphical representation for the first two iteration points is shown in Fig. 11 and the procedure is summarized in Subsection 4.3.2.1. This method has an advantage over the normal path as it will more likely intersect a solution curve or equilibrium path that exhibits significant changes in slope for fixed values of arc-length. The disadvantage is the possibility of complex roots in Eq. (32). Work arounds can involve reducing the arc-length and repeating the procedure or switching to another variation of the method. For example, Ramm [32] developed a method that uses an updated normal path that mimics a curve to avoid this issue. However, complex roots were not a problem for cases studied with the exception of non-equilibrium or guessed solutions used to initiate the procedure. In this case, the normal iteration path could be used or new guesses supplied in an attempt to avoid complex roots. Another strategy used for this study was to accept only the real component of a complex root and let the algorithm proceed as if it were, in fact, real. In these initiation cases, complex roots were often eliminated within several iterations and some intersecting point with a static solution curve was

found. Although this strategy worked for these particular instances, it is probably best not to accept complex roots and restart the solver for corresponding solutions as part of a path following procedure.

The main differences in using either the normal or circular arc-length methods are typically the location of the initial found point on a solution curve when using a similar initial guess, and the spacing between points during path following. The circular method may tend to space points more closely to one another due to the curved iteration path but this can always be adjusted by specifying a larger value for arc-length. Both the circular and normal methods worked well for generating solution curves in this chapter.

4.3.2.1 STEPWISE PROCEDURE FOR ARC-LENGTH METHOD ON CIRCULAR PATH

1. Specify an arc-length L to establish a search range for solutions away from a known or guessed point $(\boldsymbol{u}_i, \lambda_i)$. Use iteration count $i = 0$ to begin the process.
2. Calculate the tangent stiffness matrix $\boldsymbol{K}_i$ at state $\boldsymbol{u}_i$.
3. Find the sign of the matrix determinant of $\boldsymbol{K}_i$ to determine the slope of the tangent plane.
4. Solve for $\Delta\boldsymbol{u}_0$ and $\Delta\lambda_0$ using Eq. (12) and Eq. (13). Define vector $\boldsymbol{r}_0$ using Eq. (24).
5. Find the point at the end of the arc-length or start of the iteration path using Eq. (14) and Eq. (15).
6. Calculate system vector $\boldsymbol{f}(\boldsymbol{u}_{i+1})$ and tangent stiffness $\boldsymbol{K}_{i+1}$ at state $\boldsymbol{u}_{i+1}$.
7. Determine the residual where $\boldsymbol{R}_{i+1} = \lambda_{i+1}\boldsymbol{F} - \boldsymbol{f}(\boldsymbol{u}_{i+1})$.
8. Calculate $\Delta\boldsymbol{u}_{i+1}^{I}$ where $\Delta\boldsymbol{u}_{i+1}^{I} = \boldsymbol{K}_{i+1}^{-1}\boldsymbol{F}$.
9. Calculate $\Delta\boldsymbol{u}_{i+1}^{II}$ where $\Delta\boldsymbol{u}_{i+1}^{II} = \boldsymbol{K}_{i+1}^{-1}\boldsymbol{R}_{i+1}$.

10. Calculate $\Delta \boldsymbol{u}_{i+1}$ and $\Delta\lambda_{i+1}$ using Eq. (31) and Eq. (32) for each root of $\Delta\lambda_{i+1}$.
11. Define tentative vectors $\boldsymbol{r}_{i+1}$ using Eq. (25). The new vector is chosen based on the maximum value of the cosine with the previous vector using Eq. (33).
12. Check if $\Delta \boldsymbol{u}_{i+1}$ is small with respect to $\boldsymbol{u}_{i+1}$ and $\Delta\lambda_{i+1}$ is small with respect to λ_{i+1}. This test may be done by taking the ratio of vector norms and seeing if they are less than a given user-specified error tolerance. Are $\|\Delta \boldsymbol{u}_{i+1}\|/\|\boldsymbol{u}_{i+1}\|$ and $|\Delta\lambda_{i+1}|/|\lambda_{i+1}|$ less than the specified error tolerance?
13. If yes, stop, the solution has been obtained; otherwise update estimates for $\boldsymbol{u}_{i+1}$ and λ_{i+1} to improved values and repeat the procedure starting with step 6 where i is updated for the next iteration count. Updated values are obtained using Eq. (14) and Eq. (15) where subscript i is updated for the current or $i + 1$ iteration count.
14. If a solution has been obtained and search for other nearby solutions is desired as part of a path following procedure, restart the procedure beginning with step 1. Use the previously found solution as an initial guess. Arc-length L can be held constant or reduced in length as needed for restart in the event of convergence failure.

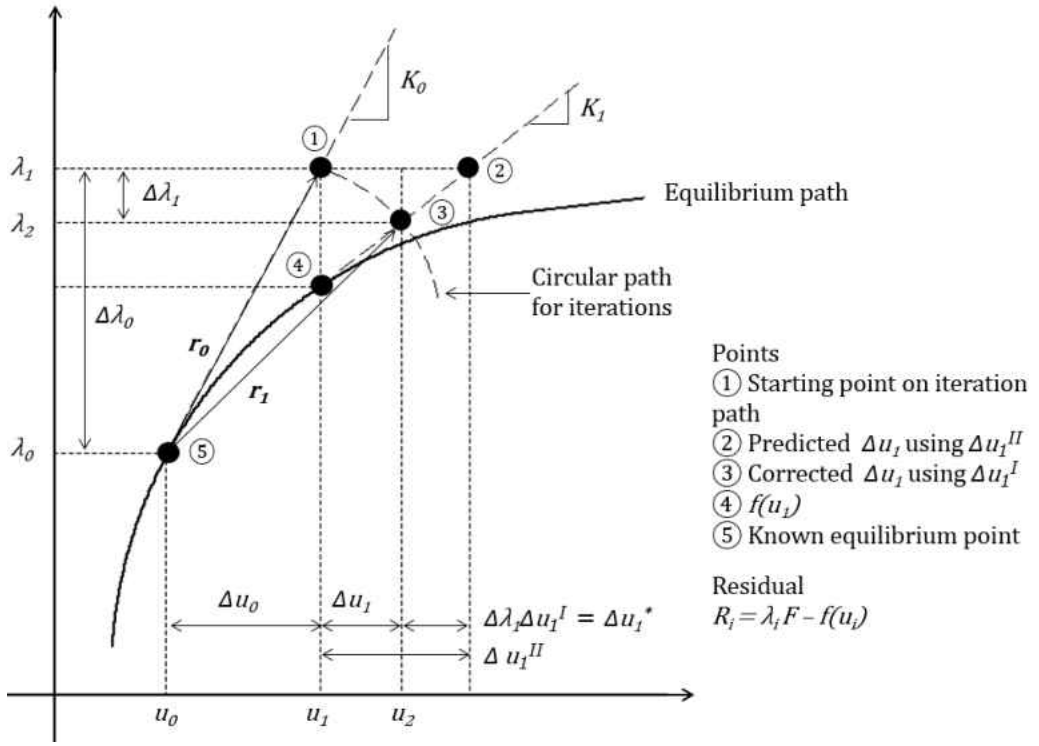

Figure 11. Points on circular iteration path for single-degree-of-freedom system

4.4 METHODS FOR DERIVING GOVERNING EQUATIONS

Governing equations for the pendulum and spring supported collapsible arch used in this study were derived using Lagrange's method or formulated using a method known as analytical mechanics [52]. Lagrange's equation may be written as

$$\frac{d}{dt}\left(\frac{\partial \mathbb{L}}{\partial \dot{\boldsymbol{u}}}\right) - \frac{\partial \mathbb{L}}{\partial \boldsymbol{u}} + \boldsymbol{\phi}_{\boldsymbol{u}}^{T}\boldsymbol{\Lambda} = \boldsymbol{Q} \tag{34}$$

where Lagrangian $\mathbb{L}$ is the difference between kinetic T and potential V energy for the system being modeled. Constraint equations are concatenated in column vector $\boldsymbol{\phi}$ where $\boldsymbol{\phi} = \boldsymbol{0}$ and $\boldsymbol{\phi_u}$ is the matrix of first-order partial derivatives with respect to components of state vector $\boldsymbol{u}$ containing the generalized degrees-of-freedom. The vector of constraint forces or Lagrange

multipliers are contained in vector $\boldsymbol{\Lambda}$. Non-conservative forces such as friction, damping, or applied forces are contained in vector $\boldsymbol{Q}$. Reducing differential terms in Eq. (34) to first-order and appending constraints $\boldsymbol{\phi}$ results in a set of differential and algebraic equations where Eq. (1) is rewritten as

$$\boldsymbol{f}(\boldsymbol{u}, \dot{\boldsymbol{u}}, t) = \boldsymbol{0} \tag{35}$$

to include differential components of state vector $\boldsymbol{u}$ and variable t, which represents time. Differential elements $\dot{\boldsymbol{u}}$ cannot be separated from $\boldsymbol{f}$ for this case, which is referred to as an implicit set of equations. Ordinary differential equations (ODEs), on the other hand, are of the form

$$\dot{\boldsymbol{u}} = \boldsymbol{f}(\boldsymbol{u}, t) \tag{36}$$

where $\boldsymbol{f}(\boldsymbol{u}, t)$ can be explicitly defined in terms of $\dot{\boldsymbol{u}}$, which is referred to as an explicit set of equations. DAEs can be thought of as an expanded form of ODEs where states $\boldsymbol{u}$ have been expanded into sets of redundant coordinates. DAEs for a single-degree-of-freedom pendulum constrained to a plane for example would contain variables (x, y, θ) for both position and orientation of the pendulum body where ODEs may only contain the variable θ for orientation, which defines position as well. Conversion of DAEs to ODEs is possible through index reduction where index is defined as the number of times select individual equations in Eq. (35) must be differentiated to recover underlying ODEs. This process may not always be practical and deriving DAEs for complex systems does have advantage over derivation of ODEs using vector mechanics based on Newton's laws. Derivation of DAEs for example is easily automated for computer implementation whereas a vector approach for deriving ODEs requires significant insight into manual construction of free body diagrams. DAEs in the form of Eq. (35) can be solved using a Newton-Raphson solver. Convergence criteria inherent to the solver alleviates the

need for small time steps needed to maintain accuracy for a dynamic solution. This behavior is not the case for explicit ODE solvers where unknown future states are entirely a function of past states $\boldsymbol{u}$ and much smaller time steps are needed to avoid accumulation of error. Based on solver type and parameter selection such as time step and error tolerance, solving DAEs can be faster and contain less error as compared to solving similar ODE systems explicitly. An overview of DAEs, ODEs, solution methods, index reduction, etc. can be found in Ref. [54].

The option for solving DAEs by Newton-Raphson is what makes arc-length methods a natural extension as an equilibrium solver where $\dot{\boldsymbol{u}}$ or derivative terms are set equal to zero leaving equations in the form of Eq. (1). Although DAEs could first be converted to ODEs for additional solver options, computational expense for such a conversion can be significant. Solving DAEs directly also has the advantage of minimizing the need for post processing of solutions to recover variables of interest including the Lagrange multipliers, which are the constraint forces for mechanical systems. A disadvantage of DAEs as compared to ODEs is that eigenvalues of the Jacobian for a linearized state about equilibrium do not follow the same rules for stability or natural frequency. Linearized ODE systems for example are considered stable if the real, or real components of all eigenvalues are less than or equal to zero where the imaginary component determines natural frequency [55]. This rule does not to apply to DAEs but patterns for real or non-complex eigenvalues at stable states were noted for cases studied. Based on literature review, straightforward stability rules using eigenvalues of linearized DAEs could not be found. Stability assessment of DAEs in general appears to be an area of ongoing research with recent work found in Ref. [56]. In this chapter, the quantity and type of eigenvalues for DAEs linearized about candidate equilibrium states are reported and patterns are identified. Numeric values are not reported as they have no physical meaning. However, stability

assessment using eigenvalues is performed only after converting systems to ODEs. This additional step is done to help further validate the proposed methodology using arc-length solvers.

Unlike ODEs, solving of DAEs also requires an estimate for the Lagrange multipliers or constraint forces as part of the initial conditions used to start the solver. Initial estimates for these constraint forces can be based on an initial guess, arbitrary values such as all zeros or ones, or obtained from an initial time step from a dynamic simulation assuming velocity for the various components were small or near zero. They can also be obtained in a more exact sense from what is referred to as an initial condition analysis in MSC ADAMS [53]. While all methods worked for cases studied, this may not hold true for larger, more complex systems. As is typical for any nonlinear equation solver, the better the initial guess, the more likely the solver will converge to a solution. Providing an initial guess for the constraint forces would be the simplest approach and could be followed by a single step dynamic solver attempt in the event of static solver failure. Dynamic solvers would be more likely to converge to a solution for similar sets of initial conditions as compared to static solvers as they inherently allow for rigid body motion. The preferred method for determining the initial constraint forces, however, may be to perform an initial condition analysis similar to ADAMS. In this case, the constraint equations are used to formulate a constrained optimization problem to determine a consistent set of initial states [12, 53]. This formulation is automatically done in ADAMS prior to starting any static or dynamic solution procedure.

There may also be cases where DAEs contain redundant constraints. While traversing of limit points that contain singular Jacobian matrices does not pose a problem for arc-length solvers, Jacobian matrices, which are rank deficient, inherently ill conditioned, or singular from

redundant constraints, would. MSC ADAMS solver for example does not tolerate redundant constraints and will subjectively delete them when encountered [12]. This auto preprocessing would imply time is better spent trying to eliminate redundant constraints manually from models in lieu of trying to solve systems with these type constraints left in place. If redundant constraints cannot easily be eliminated or path following of such a system were still desired, use of a tensor solver coupled with a geometric constraint equation for a "tensor-arc-length" solver may be an option. Though such a solver is not known to exist, tensor solvers do exist [24] and have been incorporated into MSC ADAMS as advanced solver options for static equilibrium [12]. These type solvers supplement the Newton-Raphson method with an approximation for the Hessian matrix or matrix of second-order partial derivatives from a Taylor series expansion and can handle cases where the Jacobian is singular or ill conditioned. Although optimization based solvers such as those contained in MATLAB's *fsolve* [15] routine are capable of handling singular Jacobians as well and could be parameterized, path following would be difficult as λ would need to be specified. When λ is treated as known, solver restarts with new initial guesses would be required at limit or turning points in solution curves and such solvers may have tendency to jump between sections of solution curves causing discontinuities where multiple solutions exist for given λ.

4.5 SINGLE-DEGREE-OF-FREEDOM PENDULUM

Equations for the single-degree-of-freedom pendulum shown in Fig. 12 being derived in Ref. [53] using Lagrange's method are

$$\boldsymbol{f}(\boldsymbol{u}, \ddot{\boldsymbol{u}}) = \begin{Bmatrix} m\ddot{x} + \Lambda_1 \\ m\ddot{y} + \Lambda_2 + mg \\ I_{CM}\ddot{\theta} + \Lambda_1 l sin(\theta) - \Lambda_2 l cos(\theta) \end{Bmatrix} = \mathbf{0} \tag{37}$$

State vector $\boldsymbol{u} = [x, y, \theta, \Lambda_1, \Lambda_2]^T$ where (x, y) denote position, orientation is θ, m represents pendulum mass, mass moment of inertia I_{CM} is with respect to the pendulum center of mass (CM), and g designates gravity. Point $A_{x,y}$ locates the pendulum pivot constraint relative to a global reference frame and l is the distance from pivot $A_{x,y}$ to CM. Constraint forces Λ_1, Λ_2 are in the x and y coordinate directions respectively. Specific values for constants in this case are $l = 0.127\ m$, $g = 9.807\ m/s^2$, $m = 0.4536\ kg$, and $I_{CM} = 2.463x10^{-3}\ kg \cdot m^2$.

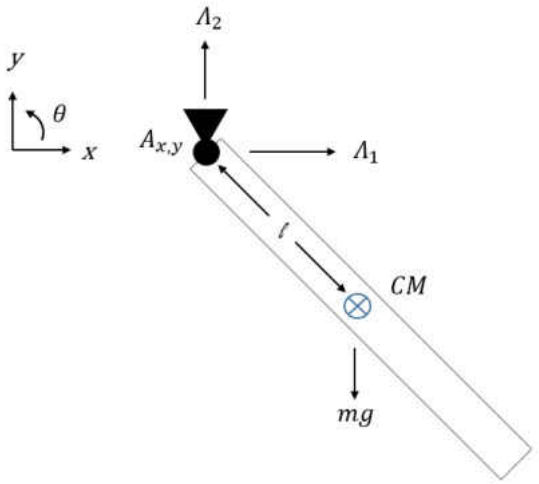

Figure 12. Single-degree-of-freedom pendulum

Constraint equations need to be appended to Eq. (37) and second-order derivatives reduced to first-order such that equations can be solved using the Newton-Raphson method. New variables $u = \dot{x}$, $v = \dot{y}$, and $w = \dot{\theta}$ are introduced which results in the final form of the DAEs where expanded state vector $\boldsymbol{u} = [u, v, w, x, y, \theta, \Lambda_1, \Lambda_2]^T$. Note that non-bolded u refers to the x component of velocity whereas bolded $\boldsymbol{u}$ refers to the state vector, which includes u and remaining system variables. System equations are

$$\boldsymbol{f}(\boldsymbol{u},\dot{\boldsymbol{u}}) = \begin{Bmatrix} m\dot{u} + \Lambda_1 \\ m\dot{v} + \Lambda_2 + mg \\ I_{CM}\dot{w} + \Lambda_1 lsin(\theta) - \Lambda_2 lcos(\theta) \\ \dot{x} - u \\ \dot{y} - v \\ \dot{\theta} - w \\ x - A_x - lcos(\theta) \\ y - A_y - lsin(\theta) \end{Bmatrix} = \mathbf{0} \tag{38}$$

The sparse nature of Eq. (38) due to redundant coordinates (x, y), which are functions of θ, is noted as seven of the eight equations contain only two variables each and the eighth equation contains four variables. This sparsity will also lead to a considerable amount of zeroes in the Jacobian or $\boldsymbol{K}$ matrix of $\boldsymbol{f}(\boldsymbol{u},\dot{\boldsymbol{u}})$. The static solution to Eq. (38) requires all velocity and acceleration terms be set and equal to zero where $\boldsymbol{f}(\boldsymbol{u},\mathbf{0}) = \boldsymbol{f}(\boldsymbol{u}) = \mathbf{0}$. This requirement contributes further to sparsity and provides the final format of equations used by the arc-length method in search of equilibrium.

4.5.1 RESULTS FOR SINGLE-DEGREE-OF-FREEDOM PENDULUM

Solution curves produced by the arc-length solver where point $A_{x,y}$ was set to $(0,0)$ are shown in Fig. 13. Curves were constructed using four runs of the solver in finite loops of 75 iterations each. Arc-length parameter L was set to plus and minus one for this case. The solver was initiated using pendulum initial conditions $\theta = 45\ deg$, $x = lcos(\theta)$, and $y = lsin(\theta)$. Guessed values were provided for λ where the objective for finding equilibrium is the λ equals to zero solution. A guessed value of λ equal to zero produced the solution curve with the zero-crossing at state B where a guessed value of λ equal to -10 produced the curve containing state A. Varying λ is what allows arc-length solvers to search for any scalar solution that satisfies the

governing equations and essentially "sweeps" the variable space in search of solution curves. Once a starting point or arbitrary scalar solution is found, the arc-length parameter L is used to control spacing or search for adjacent points used to construct curves. Specifying a positive or negative sign for L controls the slope or direction in which the search is initially performed. By running the solver through a loop, curves were readily produced by plotting found points in the plus and minus λ directions. Keeping the L parameter relatively small helps the solver to trace curves as they turn or change direction. Curves remained open in this case such that a path between candidate equilibrium states via a single solution curve does not exist. Continued plotting of points in the plus and minus λ directions would reveal asymptotic limits of curves towards vertical axes implying an intersection of parallel surfaces in the multi-dimensional space. Blue dots are used for positive λ values where red dots are used for negative λ.

Two candidate equilibrium states A and B for λ equals zero that satisfy static Eq. (38) are noted on Fig. 13. If the initial condition for the pendulum were specified close to the upward pointing vertical configuration, a Newton-Raphson solver was found to converge to an angle of $\pi/2$ or state A. Although this solution numerically satisfies equations for equilibrium, the system configuration is unstable. State B, on the other hand, places the pendulum at an angle of $-\pi/2$, which is the physically stable downward pointing configuration.

The path following procedure used for identifying these static equilibrium states is based on the assumption that equations representing multi-body systems have an unknown number of static solution curves and that each of these curves contains a finite number of roots. Guessed values of λ and arc-length L used in combination with initial conditions for state variables are used to start the procedure with the objective of finding any point or solution on a given solution curve. Once an arbitrary point is found, arc-length is then specified as part of a path following

procedure to control spacing between points and to construct the solution curve to identify the candidate equilibrium roots or λ equal to zero solutions. A more systematic approach for searching for the starting or arbitrary solution point could involve holding a guess for λ constant while varying arc-length for purpose of scaling a circular iteration path to cover an ever increasing range of variables. Note that geometrically this circular constraint corresponds to a hyperspherical constraint in multi-dimensional space [35]. The center point of the circle or hypersphere would then remain constant while the arc-length parameter scales the size of the hypersphere, which spans a given volume of hyperspace. Alternately, λ could be varied while holding arc-length constant, which will essentially move the center location of the hypersphere throughout the hyperspace. In the event there is only a single solution curve, such strategies will only tend to vary the location of the initial found point on the curve. If multiple solution curves exist, varying these parameters will help increase the likelihood that the additional curves are found.

There is no guarantee that the proposed procedure would identify all candidate equilibrium states as uncertainty would remain as to whether all possible solution curves containing a finite number of roots have been found. However, this uncertainty would be less as compared to state-of-practice solution methods that limit searches to λ equal to zero solutions only and require a good initial guess to converge to what is likely the closest proximity solution if possible. In general, there would be increased confidence that all candidate equilibrium states have been found through construction of solution curves and knowledge of the physical bounds or limits of state variables where roots or candidate states may be found through intersection with the λ equal to zero axis.

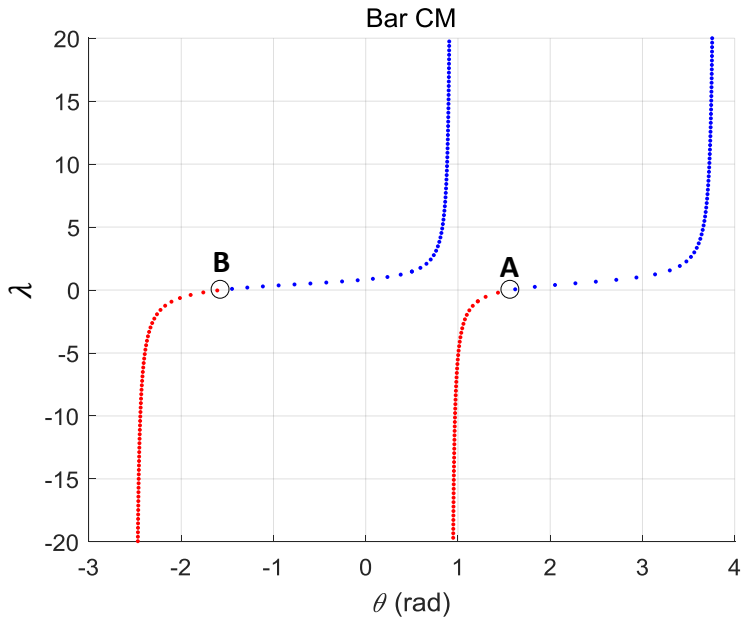

Figure 13. Solution curves for single-degree-of-freedom pendulum

A quantitative approach based on evaluation of potential energy can also be used for selecting true equilibrium in cases that may not be easily understood. A ground reference frame located below the pendulum center of mass would provide for a positive measure in height for comparison of energy between the two states. State B would be selected as it represents the state of lowest energy for the pendulum. An alternative to using energy for selection of equilibrium would be to assess eigenvalues for the linearized system about each state. The quantity and type of eigenvalues of the Jacobian for the two states using DAEs in Eq. (38) are in Table 1. Both equilibrium states A and B include positive real components indicating that stability rules for ODEs do not apply. While both states contain a mixture of complex conjugate and real eigenvalues, only state B has no positive or all negative real eigenvalues. This DAE eigenvalue pattern will be shown to remain consistent for stable configurations for the remaining cases studied. Stability rules based on this pattern are not generally implied and are considered as future work or outside of the scope of this dissertation, but the pattern could be used as an

indicator for stability where final determination would be made using eigenvalues from a similar ODE system. By converting DAEs to an ODE format, established rules using eigenvalues for assessment of stability can be applied.

Table 1
Pendulum DAE eigenvalue quantity

Type	State A	State B
+Re	1	0
−Re	3	2
+Re ± Im	2	3

Conversion of Eq. (38) to ODE format for assessment of eigenvalues is done by twice differentiating the constraints or the last two equations f_7, f_8 in the set. Results of the differentiation in column form are

$$\begin{Bmatrix} x = A_x + lcos(\theta) \\ \dot{x} = u = -lsin(\theta)\dot{\theta} = -lsin(\theta)w \\ \ddot{x} = \dot{u} = -lcos(\theta)w^2 - lsin(\theta)\dot{w} \\ y = A_y + lsin(\theta) \\ \dot{y} = v = lcos(\theta)\dot{\theta} = lcos(\theta)w \\ \ddot{y} = \dot{v} = -lsin(\theta)w^2 + lcos(\theta)\dot{w} \end{Bmatrix} \tag{39}$$

These equations are substituted into Eq. (38) eliminating individual equations f_4, f_5, f_7, f_8. Defining constraint forces Λ_1 and Λ_2 in terms of the first two equations f_1, f_2 in Eq. (38) eliminates these equations as well resulting in the following first-order ODE set.

$$\boldsymbol{f}(\boldsymbol{u}) = \begin{Bmatrix} (I_{CM} + ml^2)\dot{w} + mglcos(\theta) \\ \dot{\theta} - w \end{Bmatrix} = \mathbf{0} \tag{40}$$

Note how the mass moment of inertia includes the additional ml^2 term to account for the missing constraint equations as I_{CM} is defined relative to the pendulum center of mass and not pivot point $A_{x,y}$. Equilibrium states in Eq. (40) can be found be setting derivative terms $\dot{\theta}$ and $\dot{w}$ to zero and

solving the static problem. Values of $\pm\pi/2$ for θ provides the solution for $\boldsymbol{f}(\boldsymbol{u}) = \boldsymbol{0}$ and are similar to the results found by the arc-length solver using the DAEs.

Evaluation of eigenvalues for stability requires conversion of equations to state form followed by linearization about the candidate equilibrium states. The state or explicit form of Eq. (40) is

$$\begin{Bmatrix} \dot{\theta} \\ \dot{w} \end{Bmatrix} = \begin{Bmatrix} w \\ -mglcos(\theta)/(I_{CM} + ml^2) \end{Bmatrix} \tag{41}$$

where the state vector is defined as $\boldsymbol{u} = [\theta, w]^T$ and equilibrium states are $\boldsymbol{u}_A = [\pi/2,0]^T$, $\boldsymbol{u}_B = [-\pi/2,0]^T$. Linearization of Eq. (41) through Taylor series expansion results in

$$\begin{Bmatrix} \dot{\theta} - \dot{\theta}_S \\ \dot{w} - \dot{w}_S \end{Bmatrix} = \begin{bmatrix} 0 & 1 \\ mglsin(\theta_S)/(I_{CM} + ml^2) & 0 \end{bmatrix} \begin{Bmatrix} \theta - \theta_S \\ w - w_S \end{Bmatrix} \tag{42}$$

where subscript S refers to a specific state being A or B in this instance. Term $\boldsymbol{f}(\boldsymbol{u}_S)$ is not shown in the expansion as terms are zero at equilibrium. Eq. (42) represents a linear state space model for the pendulum where stability analysis using eigenvalues of the Jacobian or system matrix may be performed. Eigenvalues are reported in Table 2.

Table 2
Pendulum ODE eigenvalues, Re ± Im (Hz)

State A	State B
1.2097	0 + 1.2097i
–1.2097	0 – 1.2097i

These eigenvalues can be used for assessment of stability based on rules in Ref. [55] when they fall in a complex plane of real and imaginary axes. State A for the upward pointing configuration is unstable due to a positive real value where the zero real values for the complex conjugate roots in state B indicate a stable configuration with a natural frequency of 1.2097 Hz

for small displacements about equilibrium. Selection of state B for equilibrium using eigenvalues is seen to provide similar results to those obtained using energy.

4.6 SPRING SUPPORTED ARCH

The next system studied was the spring supported arch shown in Fig. 14. Both the collapsing and non-collapsing cases were evaluated by varying spring constant k_s. Spring constants were set to 17.5 N/m and 87.6 N/m for the two cases respectively. Bars are under the influence of gravity and mass and geometry of the bars are similar to the pendulum. Point $A_{x,y}$ represents a pinned connection to ground, $B_{x,y}$ is the center of mass of bar one with position (x_1, y_1) and orientation θ_1, $C_{x,y}$ is the pinned connection between bar one and bar two, $D_{x,y}$ is the center of mass of bar two with position (x_2, y_2) and orientation θ_2, and $E_{x,y}$ is a pin-slider connection attached to ground. One end of the spring is attached to ground while the other end attaches to the slider connection at $E_{x,y}$. Constraint forces $\boldsymbol{\Lambda}$ are broken into components and located at each joint. Vector notation for bar one shows how points in the system can be referenced relative to a global reference frame or ground. Vectors are not shown for other points for purpose of clarity on the figure.

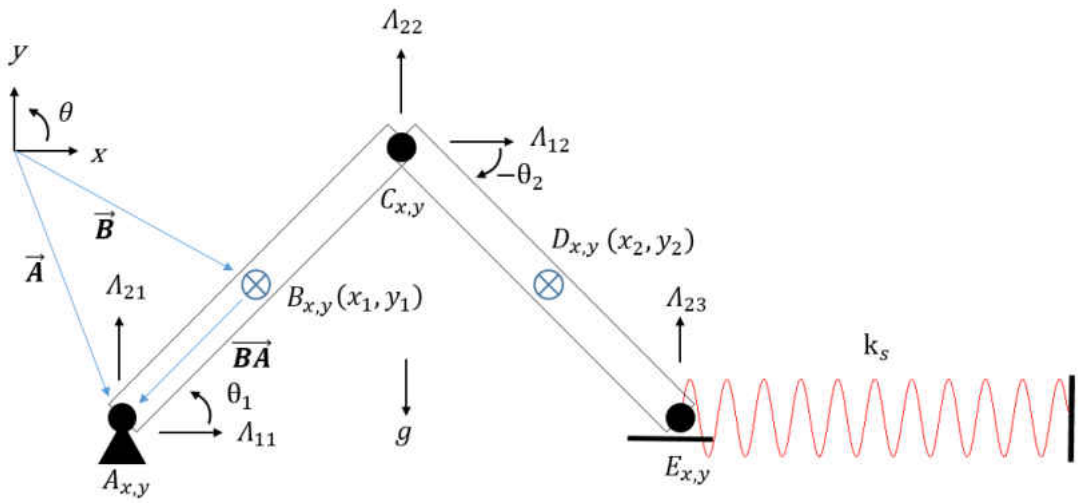

Figure 14. Spring supported arch

The Lagrangian $\mathbb{L}$ for the system is

$$\mathbb{L} = T - V \tag{43}$$

$$T = 0.5m(\dot{x}_1^2 + \dot{y}_1^2 + \dot{x}_2^2 + \dot{y}_2^2) + 0.5I_{CM}(\dot{\theta}_1^2 + \dot{\theta}_2^2)$$

$$V = mg(y_1 + y_2) + 0.5k_s(x_2 + lcos(\theta_2) - E_{x0})^2$$

which accounts for kinetic energy from translation and rotation of the bars, potential energy of the bars from gravity, and strain or potential energy from displacement of the free end of the spring with an initial position of E_{x0}. The three constraints can be expressed in vector format as

$$\vec{\boldsymbol{B}} + \overrightarrow{\boldsymbol{BA}} = \vec{\boldsymbol{A}} \tag{44}$$

$$\vec{\boldsymbol{D}} + \overrightarrow{\boldsymbol{DC}} + \overrightarrow{\boldsymbol{CB}} = \vec{\boldsymbol{B}}$$

$$\vec{\boldsymbol{E}} + \overrightarrow{\boldsymbol{ED}} = \vec{\boldsymbol{D}}$$

and further broken into components for obtaining an expression for $\boldsymbol{\phi}$. The first two expressions in Eq. (44) are broken into their x and y components where only the y component is required for point E in the third expression as the end of bar two is unconstrained in the x direction. Setting

vertical displacement E_y equal to zero for the horizontal slider constraint and breaking Eq. (44) into x and y components results in

$$\boldsymbol{\phi} = \begin{Bmatrix} x_1 - lcos(\theta_1) - A_x \\ y_1 - lsin(\theta_1) - A_y \\ (x_2 - x_1) - l(cos(\theta_2) + cos(\theta_1)) \\ (y_2 - y_1) - l(sin(\theta_2) + sin(\theta_1)) \\ y_2 + lsin(\theta_2) \end{Bmatrix} \tag{45}$$

Inserting Lagrangian $\mathbb{L}$ in Eq. (43) and constraints $\boldsymbol{\phi}$ in Eq. (45) into Lagrange's equation Eq. (34) where $\boldsymbol{Q}$ is a vector of zeroes provides for the following equation set used to describe the system. The state vector $\boldsymbol{u}$ for the two links and constraint force vector $\boldsymbol{\Lambda}$ are defined as $\boldsymbol{u} = [x_1, y_1, x_2, y_2, \theta_1, \theta_2]^T$ and $\boldsymbol{\Lambda} = [\Lambda_{11}, \Lambda_{21}, \Lambda_{12}, \Lambda_{22}, \Lambda_{23}]^T$ at this point in the derivation.

$$\boldsymbol{f}(\boldsymbol{u}, \ddot{\boldsymbol{u}}) = \begin{Bmatrix} m\ddot{x}_1 + \Lambda_{11} - \Lambda_{12} \\ m\ddot{y}_1 + \Lambda_{21} - \Lambda_{22} + mg \\ m\ddot{x}_2 + k_s(x_2 + lcos(\theta_2) - E_{x0}) + \Lambda_{12} \\ m\ddot{y}_2 + \Lambda_{22} + \Lambda_{23} + mg \\ I_{CM}\ddot{\theta}_1 + l(\Lambda_{11} + \Lambda_{12})sin(\theta_1) - l(\Lambda_{21} + \Lambda_{22})cos(\theta_1) \\ I_{CM}\ddot{\theta}_2 + l\big(\Lambda_{12}sin(\theta_2) - \Lambda_{22}cos(\theta_2) + \Lambda_{23}cos(\theta_2)\big) + \cdots \\ \cdots - k_s l(x_2 + lcos(\theta_2) - E_{x0})sin(\theta_2) \end{Bmatrix} = \boldsymbol{0} \tag{46}$$

Reducing the system to first-order through introduction of variables $u_j = \dot{x}_j$, $v_j = \dot{y}_j$ and $w_j = \dot{\theta}_j$ where subscript $j = 1,2$ for each bar and appending constraints $\boldsymbol{\phi}$ in Eq. (45) results in the following set of DAEs. The seventeen by one state vector is now defined as $\boldsymbol{u} = [u_1, v_1, u_2, v_2, w_1, w_2, x_1, y_1, x_2, y_2, \theta_1, \theta_2, \Lambda_{11}, \Lambda_{21}, \Lambda_{12}, \Lambda_{22}, \Lambda_{23}]^T$

$$\boldsymbol{f}(\boldsymbol{u},\dot{\boldsymbol{u}}) = \left\{ \begin{array}{c} m\dot{u}_1 + \Lambda_{11} - \Lambda_{12} \\ m\dot{v}_1 + \Lambda_{21} - \Lambda_{22} + mg \\ m\dot{u}_2 + k_s(x_2 + lcos(\theta_2) - E_{x0}) + \Lambda_{12} \\ m\dot{v}_2 + \Lambda_{22} + \Lambda_{23} + mg \\ I_{CM}\dot{w}_1 + l(\Lambda_{11} + \Lambda_{12})sin(\theta_1) - l(\Lambda_{21} + \Lambda_{22})cos(\theta_1) \\ I_{CM}\dot{w}_2 + l\big(\Lambda_{12}sin(\theta_2) - \Lambda_{22}cos(\theta_2) + \Lambda_{23}cos(\theta_2)\big) + \cdots \\ \cdots - k_s l(x_2 + lcos(\theta_2) - E_{x0})sin(\theta_2) \\ \dot{x}_1 - u_1 \\ \dot{y}_1 - v_1 \\ \dot{x}_2 - u_2 \\ \dot{y}_2 - v_2 \\ \dot{\theta}_1 - w_1 \\ \dot{\theta}_2 - w_2 \\ x_1 - lcos(\theta_1) - A_x \\ y_1 - lsin(\theta_1) - A_y \\ (x_2 - x_1) - l(cos(\theta_2) + cos(\theta_1)) \\ (y_2 - y_1) - l(sin(\theta_2) + sin(\theta_1)) \\ y_2 + lsin(\theta_2) \end{array} \right\} = \mathbf{0} \qquad (47)$$

The equation set was verified through dynamic simulation using a previously developed nonlinear solver suite written in MATLAB [51] and commercial software MSC ADAMS. Initial conditions for the bars were similar to Fig. 14 where θ_1 was set to 45 degrees and θ_2 to -45 degrees with zero initial rates. Point $A_{x,y}$ was set to the origin or (0,0) coordinate. Results for the x position of bar one versus time are shown in Fig. 15 and Fig. 16 for the collapsing and non-collapsing cases where MATLAB and ADAMS results are coincident giving the appearance of a single curve. Both the MATLAB and ADAMS solvers used a Newton-Raphson method where results for the simulation were found using a time step of 0.0005 seconds. Fig. 15 for the collapsing case shows bar one snapping through to an inverted orientation with the CM swinging past point $A_{x,y}$ in the horizontal axis, reversing direction, and snapping back past its original configuration in an oscillatory manner. Fig. 16, on the other hand, shows bar one oscillating

about a much smaller displacement as spring force is sufficient to prevent collapse from occurring.

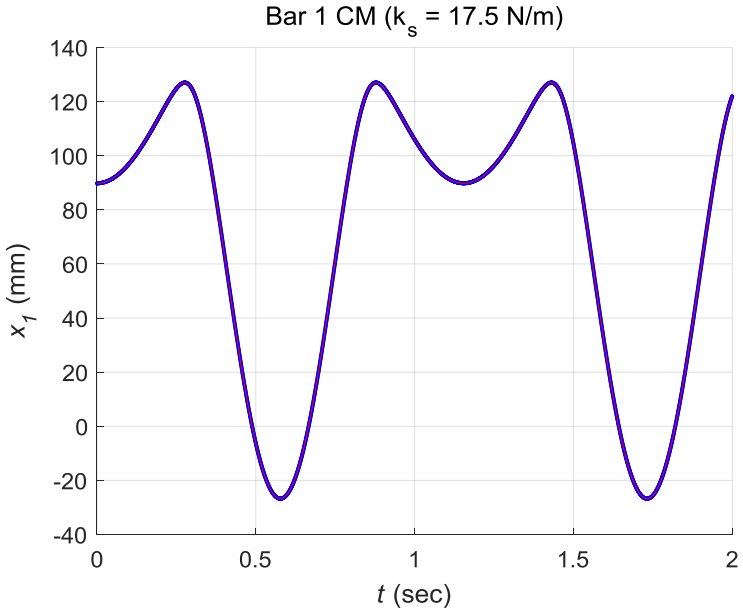

Figure 15. Dynamic simulation of collapsing arch

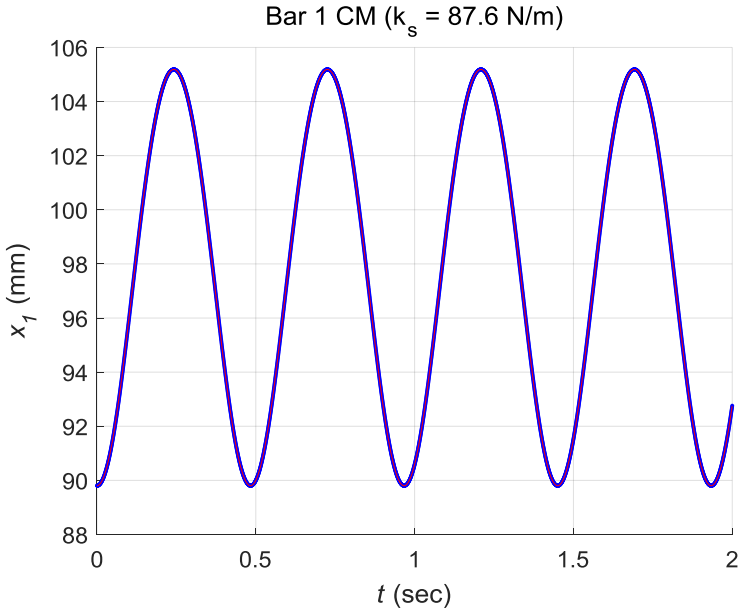

Figure 16. Dynamic simulation of non-collapsing arch

Candidate equilibrium states for the collapsing and non-collapsing cases were also verified using ADAMS and compared to those found using static solution curves produced by arc-length solvers. State vectors are recorded in tables in the following results and discussion sections. This recording was accomplished only after the list of candidate states obtained through path following of static solution curves was complete. Once these states were known, a similar ADAMS model was manually configured in sufficiently close proximity to each state providing necessary initial conditions that would cause available static solvers to converge to the desired configurations. A direct comparison of results for each state variable was then made to confirm similarity. For the non-collapsing case, the as modeled configuration was in closest proximity to true equilibrium such that the default Newton-Raphson solver in ADAMS converged to this solution. The collapsing case, however, proved more challenging as all Newton-Raphson based solvers failed while the more advanced solvers based on optimization algorithms converged to unstable pre-collapse configurations when using default settings. Failure of the Newton-Raphson solvers was due to several of the state variables having to pass through limit points or change direction in order to reach the inverted configuration. The x position of bar one on Fig. 15 for example must first increase from its initial configuration towards the horizontal limit prior to snapping through and decreasing towards the inverted configuration. Through trial and error, it was found that the ALIMIT parameter could be adjusted under the ADAMS solver settings to increase the allowed value for incremental displacement with respect to angular state variables. This setting enabled the solver to "jump over" or past limit points in this case and locate equilibrium. Setting the value too high, however, caused it to converge to other unstable configurations. Although such a procedure

could be used, uncertainty would still remain as to whether true equilibrium was found and a pictorial of solution curves with candidate equilibrium states at zero-crossings would also be missing.

4.6.1 RESULTS FOR COLLAPSING ARCH

Solution curves for the collapsing case produced by an arc-length solver using MATLAB are shown in Fig. 17 through Fig. 22. Plots were selected using position and orientation information from the bar one center of mass although any one or all of the state variables could have been selected. Multiple curves were identified for this case and were separated into two plots each for clarity of figures. As previously mentioned, curves may exhibit non-physical values for given variables at non-zero solutions for λ. This behavior is due to the fact that only the $\lambda = 0$ solutions are admissible for equilibrium while other values modify governing equations by including the scalar value. Although the non-zero λ solutions satisfy equations numerically, they are only used to construct static solution curves to follow or provide a path from one state to another. This formulation left only scalers λ and arc-length L as parameters that could be varied as part of the initial guess used to start solvers. By varying these parameters, initial found points at different locations on single curves or points on different curves could be found. The solvers could not traverse what appeared to be limit points located at the $\lambda = 20$ axes on Fig. 17 and Fig. 18 and at the $\lambda = -20$ axes on Fig. 20 and Fig. 21 during path following. These special points are asymptotic limits as new points could be continually found without ever crossing the limit. Spacing between points also became smaller for every new point found giving the appearance of a closed curve in these sectional views of hyperspace. In these

cases, the solver could be restarted to trace the remainder of the curve in opposite direction by changing the sign of arc-length L. Curves shown in Fig. 17 through Fig. 19 were traced in this manner.

DAE eigenvalue quantity for candidate equilibrium states using Eq. (47) are in Table 3; state vectors including the initial configuration specified as State I are in Table 4; strain or potential energy for the spring are in Table 5. Potential energy due to gravity was not included as it is altitude or elevation dependent and the datum for zero potential energy is arbitrary. If elevation were large, potential energy due to gravity would dominate the magnitude of the energy term and potentially mask or eliminate potential energy of the spring due to numerical precision or the number of significant digits used to represent the quantity.

State vectors in Table 4 were truncated to eliminate the first six velocity terms, which are zero for static solutions. Four candidate states A through D were found and identified on graphs. States C and D are unique as they were not found in an exact sense by the arc-length solver but implied graphically as locations for zero-crossings with bars in the vertical pointing down and up configurations. The velocity terms for this case, or first six state variables, were approximately zero but the vertical constraint forces significantly exceeded the weight of the mechanism for a non-physical solution. Closer investigation of these points revealed that curves became asymptotic as they approached the λ equals zero axis. The reciprocal of the matrix condition number for the Jacobian also became increasingly small indicating near singularity or an ill conditioned problem. Attempts to find an exact solution using MATLAB's *fsolve* routine [15], which is able to handle singularity of the Jacobian, also failed or produced warnings for possible error. An additional solving attempt was made by manually specifying a state vector for the exact geometric configurations of the implied solutions and inserting into $\boldsymbol{f}(\boldsymbol{u})$. This strategy

left equations containing velocity terms as non-zero further validating that an exact equilibrium solution does not exist at these locations.

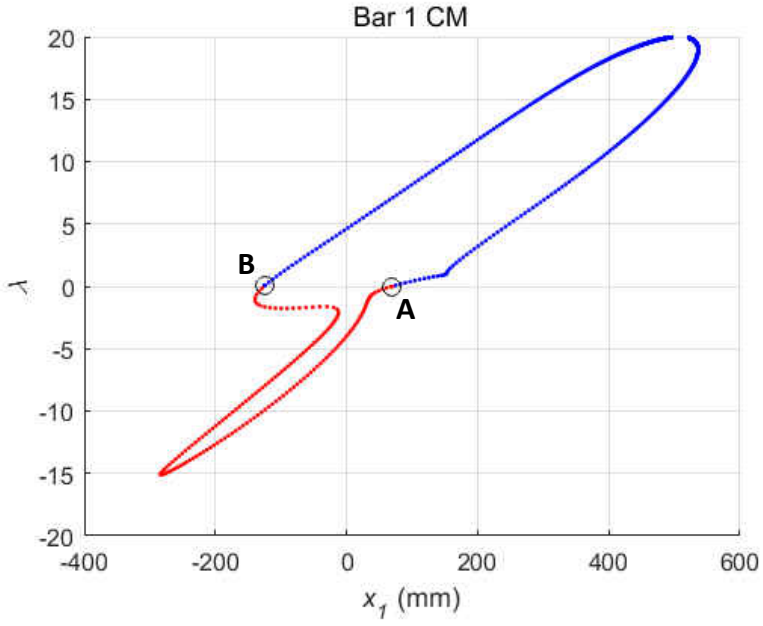

Figure 17. Solution curve for bar one x position, $k_s = 17.5\ N/m$

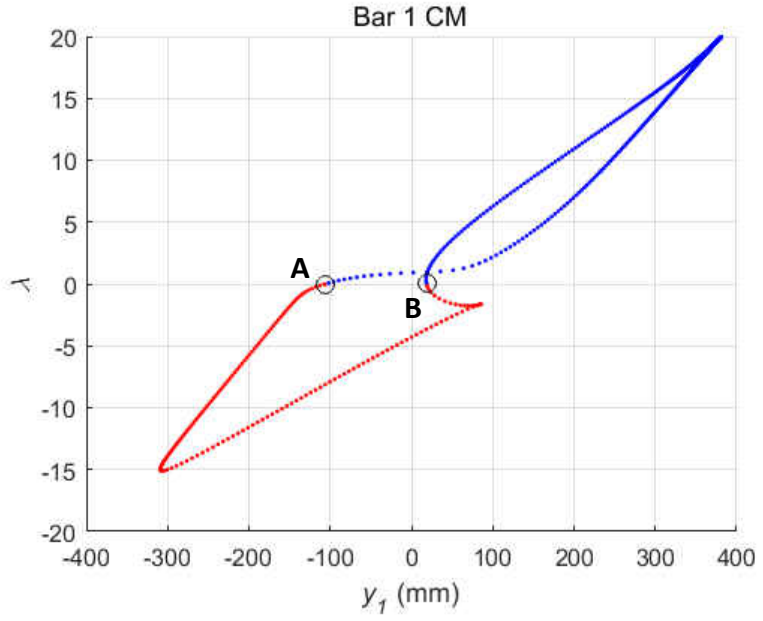

Figure 18. Solution curve for bar one y position, $k_s = 17.5\ N/m$

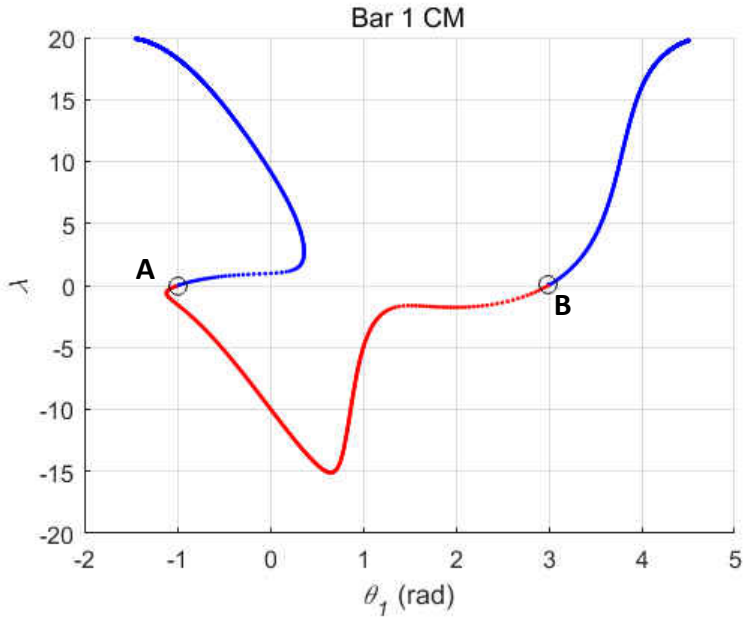

Figure 19. Solution curve for bar one angle, $k_s = 17.5\ N/m$

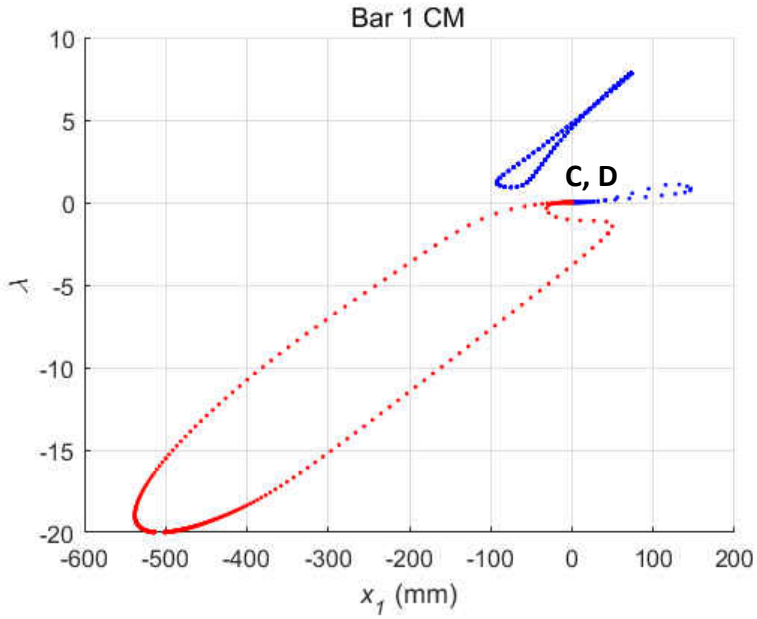

Figure 20. Additional solution curves for bar one x position, $k_s = 17.5\ N/m$

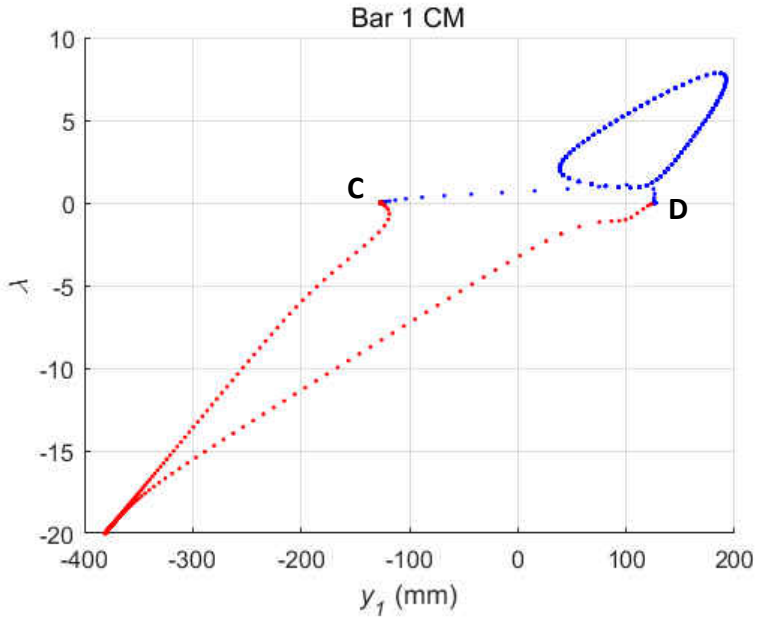

Figure 21. Additional solution curves for bar one y position, $k_s = 17.5\ N/m$

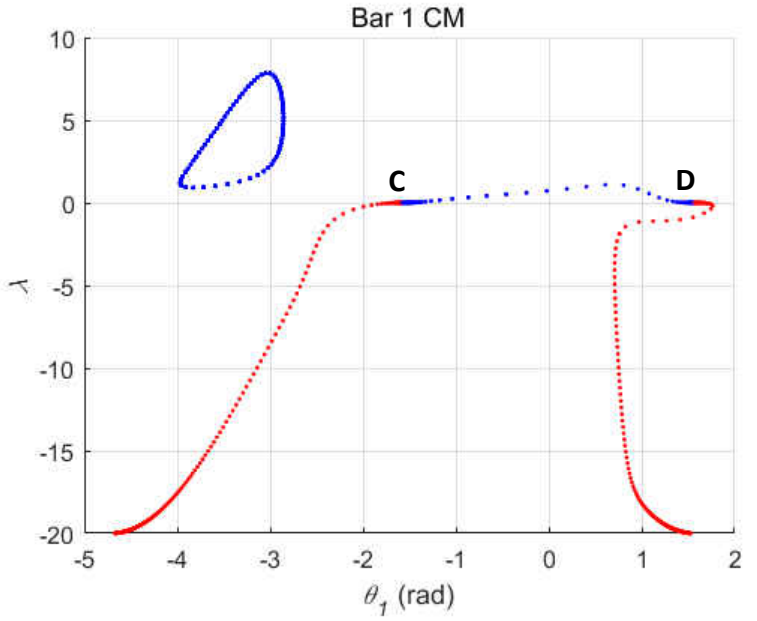

Figure 22. Additional solution curves for bar one angle, $k_s = 17.5\ N/m$

Table 3
Collapsing arch DAE eigenvalue quantity

Type	State A	State B	State C*	State D*
+Re	0	3	2	1
−Re	5	6	5	6
+Re ± Im	6	4	5	5

* Implied state

Table 4
Candidate equilibrium states for collapsing arch [1]

Variable	State I [2]	State A	State B	State C*	State D*
$x_1(mm)$	89.8017	69.1794	-125.6436	-0.4293	-0.3886
$y_1(mm)$	89.8017	-106.5047	18.5166	-127.0000	127.0000
$x_2(mm)$	269.4076	207.5383	-376.9284	-0.4293	-0.3886
$y_2(mm)$	89.8017	-106.5047	18.5166	-127.0000	127.0000
$\theta_1(rad)$	0.7854	-0.9947	2.9953	-1.5742	1.5739
$\theta_2(rad)$	-0.7854	0.9947	3.2879	1.5674	-1.5677
$\Lambda_{11}(N)$	-2.6663	1.4448	15.0919	6.2907	6.2907
$\Lambda_{21}(N)$	-4.4482	-4.4482	-4.4482	1847.2492	-2035.2284
$\Lambda_{12}(N)$	-1.9999	1.4448	15.0919	6.2907	6.2907
$\Lambda_{22}(N)$	-0.6663	-0.0000	-0.0000	1851.6974	-2030.7802
$\Lambda_{23}(N)$	-3.1151	-4.4482	-4.4482	-1856.1456	2026.3319

[1] Bar one x, y position and angle highlighted red
[2] Initial configuration
* Implied state

Table 5
Strain energy $(N \cdot m)$ for collapsing arch

State A	State B	State C*	State D*
0.0596	6.5031	1.1298	1.1298

* Implied state

Candidate states C and D could be eliminated based on failure to provide an exact solution. If this analysis had not been performed, they could, however, still be considered for further assessment. Prior to making a selection for equilibrium from the candidate states, a measure of how close the states are to the initial configuration was defined. A metric based on

the difference ratio of vector norms was used by defining the initial state vector as $\boldsymbol{u}_{\mathrm{I}}$ and candidate state vectors as $\boldsymbol{u}_{\mathrm{S}}$ where subscript "S" denotes the specific individual states and "I" denotes the initial state. The resulting expression is

$$ratio = \|\boldsymbol{u}_{\mathrm{I}} - \boldsymbol{u}_{\mathrm{S}}\|/\|\boldsymbol{u}_{\mathrm{I}}\| \tag{48}$$

Difference ratios using Eq. (48) for states A through D are shown in Table 6.

Table 6
Difference ratios with respect to state I for collapsing arch

State A	**State B**	**State C***	**State D***
0.9167	2.2149	10.3986	11.3990

* Implied state

Since A has the smallest ratio, it is closest to the as modeled initial configuration of the arch. State A is therefore selected as the starting point for evaluation among the candidate states with a strain energy of 0.0596 $(N \cdot m)$. Moving towards state B along the solution curves displayed using sectional plots in Fig 17 through Fig. 19 reveals an increase in strain energy such that state B is rejected for A. Both states C and D are even further away from state A, having higher strain energy and are also rejected. State A is therefore selected as equilibrium being consistent with results obtained from a dynamic simulation with added damping in ADAMS. Real eigenvalues using DAEs from Eq. (47) in Table 3 are shown to be all negative for state A where rejected states include positive real eigenvalues. Patterns for complex conjugate pairs show positive real component eigenvalues only.

Conversion of Eq. (47) to ODE format for stability assessment using eigenvalues uses a similar approach as was done for the pendulum where constraint equations, f_{13} through f_{17} in this case, are twice differentiated and substituted back into the equation set. The process can be simplified by observing that $\theta_2 = -\theta_1$ and $y_2 = y_1$from the constraints. This results in the

following set of first-order ODEs written in explicit form. The state vector for the reduced system is $\boldsymbol{u} = [\theta_1, w_1]^T$.

$$\begin{Bmatrix} \dot{\theta}_1 \\ \dot{w}_1 \end{Bmatrix} = \begin{Bmatrix} w_1 \\ \frac{-4ml^2\sin(2\theta_1))w_1^2+8k_s l^2 sin(2\theta_1)-4E_{x0}k_s lsin(\theta_1)-2mglcos(\theta_1)}{(10ml^2-8ml^2cos^2(\theta_1)+2I_{CM})} \end{Bmatrix} \tag{49}$$

Candidate equilibrium states are found by setting velocity and acceleration terms in Eq. (49) to zero and solving for θ_1 in the remaining equation.

$$f_2(\theta_1) = 8k_s l^2 sin(2\theta_1) - 4E_{x0}k_s lsin(\theta_1) - 2mglcos(\theta_1) = 0 \tag{50}$$

Graphical representation of Eq. (50) plotted between $\pm\pi$ with similar found states to the original DAE system is shown in Fig. 23. Implied states C and D that were previously dismissed do not show up on this figure as zero-crossings further validating they are, in fact, not candidates for equilibrium.

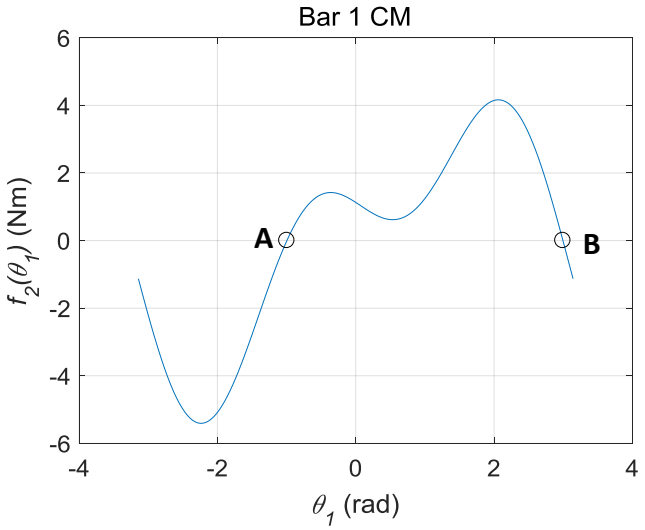

Figure 23. ODE static solution curve for bar one angle, $k_s = 17.5\ N/m$

Eq. (49) is linearized about states $\boldsymbol{u}_A = [-0.9947,0]^T$, $\boldsymbol{u}_B = [2.9953,0]^T$ and eigenvalues are reported in Table 7. Equations for the linearized system are not shown as was for the pendulum

due to the extensive algebraic expressions in the system matrix. Results were validated using ADAMS by manually configuring the arch near the given states and running a Newton-Raphson based equilibrium solver such that it would converge to the closest available equilibrium configuration. A linearization was then performed; it output similar eigenvalues. The linearization procedure used by ADAMS is based on a state space reduction method for reducing the governing DAEs to a set of minimal states [57]. This approach essentially recovers the underlying ODEs as redundant variables are eliminated and eigenvalues can then be used for stability assessment and determination of natural frequency. State A is shown to be stable with a natural frequency of 1.3741 Hz with state B being unstable. State A therefore represents true equilibrium similar to the previous assessment using strain energy. Real DAE eigenvalues were all negative for this state as an indicator for stability as well.

Table 7
Collapsing arch ODE eigenvalues, Re ± Im (Hz)

State A	State B
0 + 1.3741i	3.0528
0 – 1.3741i	–3.0528

4.6.2 RESULTS FOR NON-COLLAPSING ARCH

Solution curves for the non-collapsing case produced by an arc-length solver using MATLAB are shown in Fig. 24 through Fig. 29. Figures showing a detailed view of the solution centered near the λ equal to zero axis contain an additional section of curve that was left off of the larger, complete solution plots for clarity of figures. Six candidate equilibrium states were found where states E and F were implied zero-crossings similar to those in the previous case. Although this case is somewhat trivial as the initial and resultant stable equilibrium

configurations provide for a closest proximity solution for any nonlinear solver, plots produced by the arc-length solver are unique and will help further validate the proposed procedure for identifying equilibrium. DAE eigenvalue quantity, state vectors, strain energy stored in the spring, and difference ratios of candidate states S with respect to initial state I are shown in Tables 8 through 11 respectively.

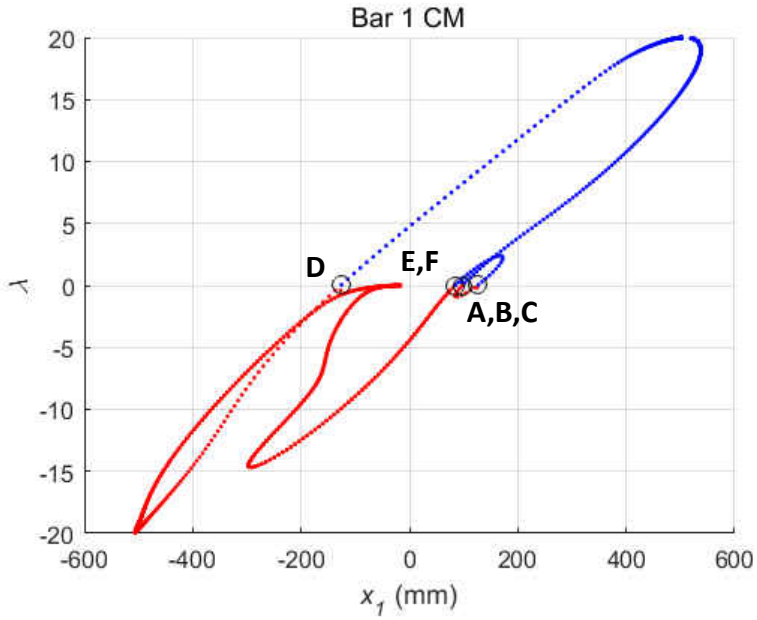

Figure 24. Total solution curve for bar one x position, $k_s = 87.6\ N/m$

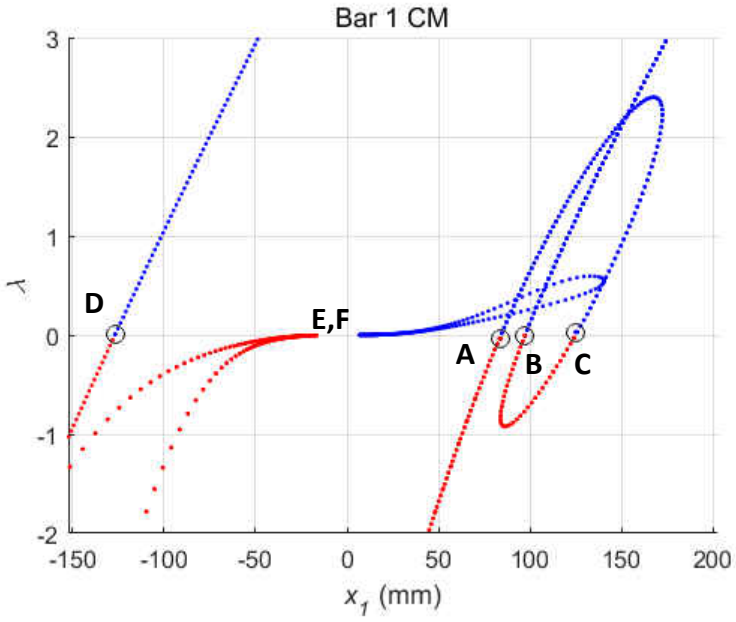

Figure 25. Partial solution curve for bar one x position, $k_s = 87.6\ N/m$

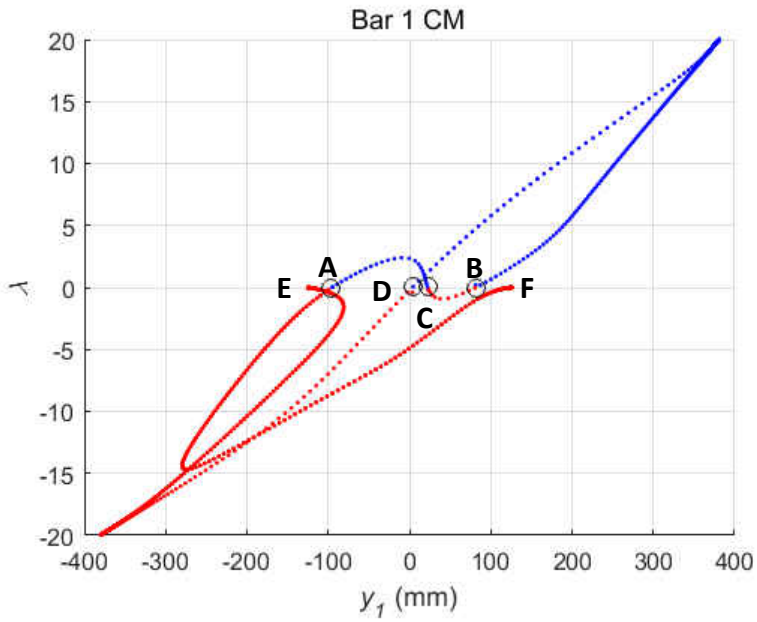

Figure 26. Total solution curve for bar one y position, $k_s = 87.6\ N/m$

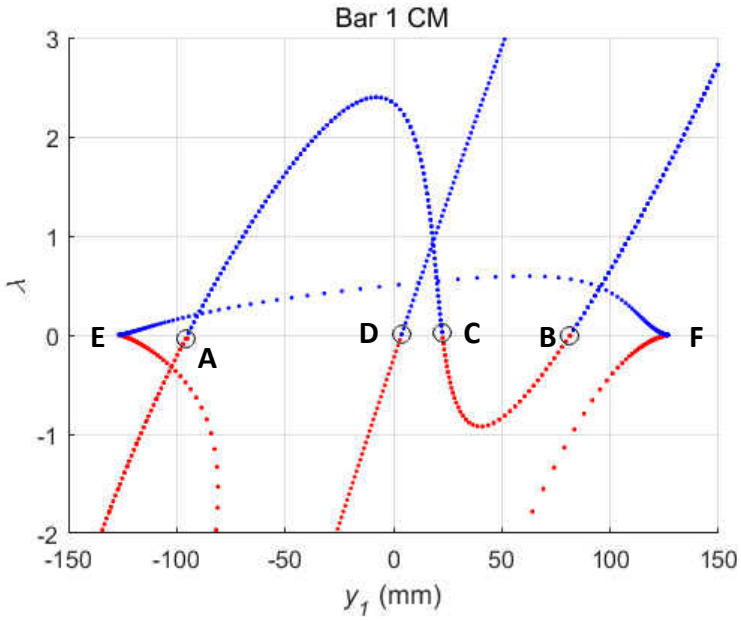

Figure 27. Partial solution curve for bar one y position, $k_s = 87.6\ N/m$

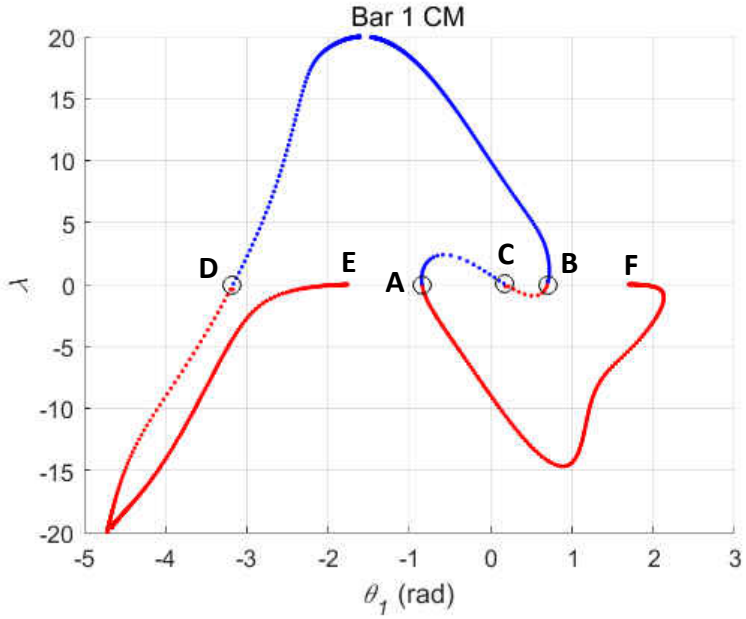

Figure 28. Total solution curve for bar one angle, $k_s = 87.6\ N/m$

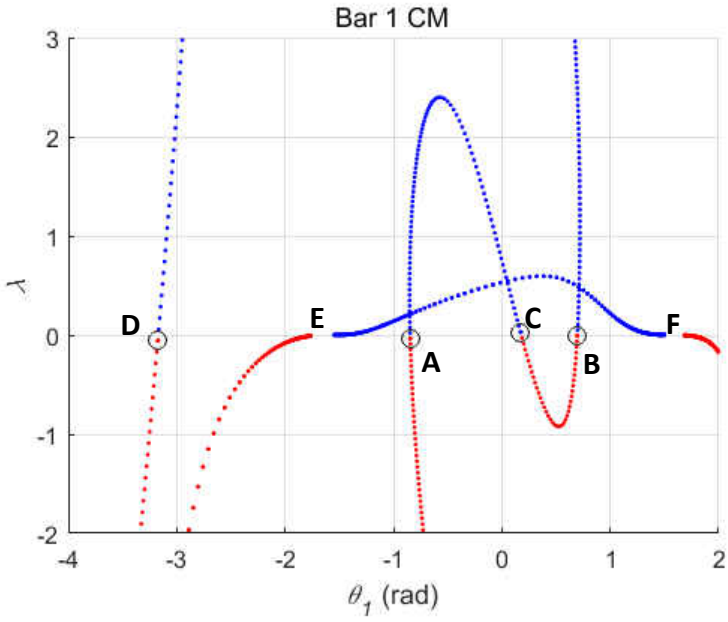

Figure 29. Partial solution curve for bar one angle, $k_s = 87.6\ N/m$

Table 8
Non-collapsing arch DAE eigenvalue quantity

Type	State A	State B	State C	State D	State E*	State F*
+Re	0	0	1	3	2	1
−Re	5	5	6	6	5	6
+Re ± Im	6	6	5	4	5	5

* Implied state

Table 9
Candidate equilibrium states for non-collapsing arch [1]

Variable	State I [2]	State A	State B	State C	State D	State E*	State F*
$x_1(mm)$	89.8017	84.1807	97.3887	124.9807	-126.9467	-1.4605	-1.9329
$y_1(mm)$	89.8017	-95.0925	81.5137	22.5603	3.7186	-126.9924	126.9848
$x_2(mm)$	269.4076	252.5446	292.1686	374.9396	-380.8374	-1.4630	-1.9279
$y_2(mm)$	89.8017	-95.0925	81.5137	22.5603	3.7186	-126.9924	126.9848
$\theta_1(rad)$	0.7854	-0.8462	0.6969	0.1786	-3.1709	-1.5823	1.5860
$\theta_2(rad)$	-0.7854	0.8462	-0.6969	-0.1786	3.1709	-4.7239	-1.5555
$\Lambda_{11}(N)$	-2.6663	1.9688	-2.6574	-12.3211	75.9169	31.4543	31.4529
$\Lambda_{21}(N)$	-4.4482	-4.4482	-4.4482	-4.4482	-4.4482	2,726.4447	-2,066.9384
$\Lambda_{12}(N)$	-1.9999	1.9688	-2.6574	-12.3211	75.9169	31.4543	31.4529
$\Lambda_{22}(N)$	-0.6663	-0.0000	0.0000	0.0000	0.0000	2,730.8929	-2,062.4902
$\Lambda_{23}(N)$	-3.1151	-4.4482	-4.4482	-4.4482	-4.4482	-2,735.3412	2,058.0420

[1] Bar one x, y position and angle highlighted red
[2] Initial configuration
* Implied state

Table 10
Strain energy $(N \cdot m)$ for non-collapsing arch

State A	State B	State C	State D	State E*	State F*
0.0221	0.0403	0.8669	32.9097	5.6494	5.6490

* Implied state

Table 11
Difference ratios with respect to state I for non-collapsing arch

State A	State B	State C	State D	State E*	State F*
0.8426	0.0860	0.4726	2.2655	15.2658	11.5164

* Implied state

Starting with the difference ratios in Table 11, state B is found to be in closest proximity to the initial configuration state I. Solution curves in Fig. 24 through Fig. 29 show states in the order of B-C-A-F and B-D-E when following the curves or paths on figures in either direction when starting from state B. The arc-length solver was run four times to find arbitrary starting values on curves and then run through a finite loop in the plus and minus arc-length directions to

construct curves. Fig. 26 for example exhibits non-smooth or sharp portions to the curve that the arc-length solver could not trace past. For these cases, the solver behaved as if near an asymptotic limit as new points were continually found, but distance covered on the graph became less and less. The smother, self-intersecting loop on Fig. 25, on the other hand, was traced in a single run of the solver loop. Note that information on all figures is being solved simultaneously as these are individual components of state vector $\boldsymbol{u}$. Figures are essentially sectional views with respect to individual states in the multi-dimensional space or hyperspace; this is why they appear different from one another. States E and F are referred to as implied due to the appearance of possible zero-crossings on figures. Reciprocals of the condition number of the Jacobian for these states are very small indicating near singularity such that they are likely asymptotic limits and not equilibrium states. As was done previously, both states are still included as candidates for equilibrium.

The process of identifying equilibrium begins with state B, which is closest to the initial configuration. Following the B-C-A-F path on solution curves, state C is found to have higher strain energy such that it is rejected. The next state A has lower strain energy; however, the system must first pass through the higher energy state C from the lower state B such that it too is rejected. This conclusion is drawn on both understood physical behavior of the simple system and through following the path from one state towards another on the solution curves. In a physical sense, the bars would have to pass through the horizontal configuration prior to snapping through to the inverted position. The gravity load in this case is not sufficient to overcome spring force leaving the arch in a non-collapsing configuration. The next state F has significantly higher strain energy, which eliminates this state as well. Remaining states are evaluated by traveling in the opposite direction from state B on path B-D-E. Both D and E are of

higher strain energy such that they are rejected. State B is therefore selected as equilibrium. State A is noted as a physically admissible and stable equilibrium, although it would be impossible to reach from the initial configuration without inclusion of additional force needed to compress the spring. Note that by following the path between states on solution curves, rejection of states C and D would eliminate any proceeding states as it is not possible to pass through higher energy configurations without application of additional force. Similar to the case of the collapsing arch, real eigenvalues using DAEs from Eq. (47) in Table 8 are shown to be all negative for stable states A and B where rejected states include positive real eigenvalues.

Eq. (49) and Eq. (50) remain similar for the ODEs used to describe the system where only spring constant k_s is updated for the non-collapsing case. Graphical results for equilibrium states using these equations are plotted in Fig. 30 with similar found states to the original DAE system. The angle for state D is reported in a positive sense with the $\pm\pi$ limits of the plot. The angle is reported in a negative sense for the DAE system in Table 9 but is, in fact, the same angle. Plotting the static solution curve within these limits provides for a consistent order among states when comparing the ODE and DAE systems. Implied states E and F do not show up as zero-crossings further validating they are not candidates for equilibrium.

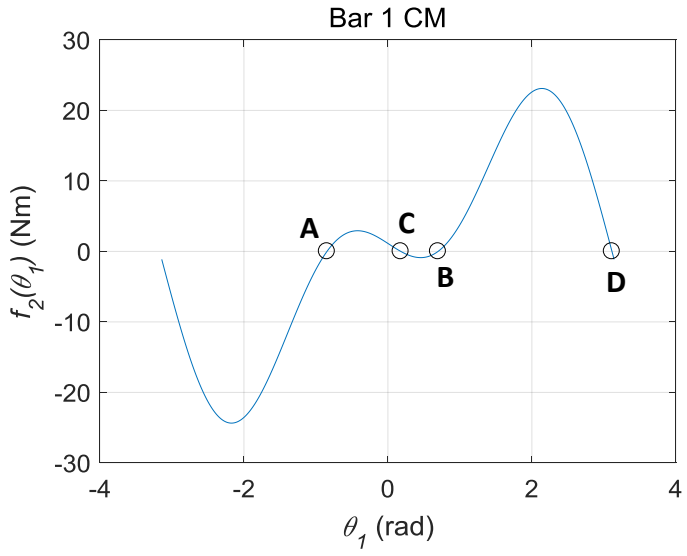

Figure 30. ODE static solution curve for bar one angle, $k_s = 87.6\ N/m$

Eigenvalues for the linearized states using the ODEs are shown in Table 12. These results were also validated using ADAMS similar to the previous case. State B that was identified as equilibrium and state A that was identified as physically possible for equilibrium are both shown to be stable with natural frequencies of 2.0925 Hz and 2.6187 Hz respectively. Dismissed states C and D are both shown to be unstable due to positive, real eigenvalues.

Table 12
Non-collapsing arch ODE eigenvalues, Re ± Im (Hz)

State A	State B	State C	State D
0 + 2.6187i	0 + 2.0925i	2.5851	7.0582
0 – 2.6187i	0 – 2.0925i	–2.5851	–7.0582

4.7 PROPOSED PROCEDURE FOR IDENTIFYING STATIC EQUILIBRIUM

Based on methods and results for the pendulum, collapsing arch cases, and non-collapsing arch cases, the following procedure for identifying equilibrium for general systems is proposed.

This procedure is based on path following of static solution curves for nonlinear systems of equations being derived from and representing physics-based systems.

1. Select any single or a combination of state variables for plotting of static solution curves using arc-length solvers.
2. Identify where solution curves cross the λ equal to zero axis and label these as candidate equilibrium states accordingly.
3. Determine the difference ratios between the as modeled initial state I and found candidate states S using Eq. (48). Choose the state with the smallest ratio as the initial candidate equilibrium state for consideration.
4. If additional candidate equilibrium states are present on a solution curve, identify the order in which states occur by following the curve or path in either direction starting from the initial candidate state.
5. Using this order and starting with the initial candidate state, accept or reject remaining states based on change in potential energy. States leading to an increase in potential energy would be rejected along with any remaining states for a given direction along a solution curve. New states would be accepted when leading to a decrease in potential energy for a given direction along a solution curve. If a path between states does not exist, consider all states simultaneously. Equilibrium is the state that reduces potential energy to a minimum with respect to a given static solution curve or minimizes potential energy in the event a relation among states via a solution curve does not exist.
6. Optionally convert equations to a linearized state space format and extract eigenvalues for each individual state. These eigenvalues can be used as a secondary metric to verify stability at the chosen equilibrium.

4.8 CONCLUSIONS

Arc-length solvers were used to successfully identify the many possible equilibrium states for nonlinear systems representing a pendulum and two variations of a spring supported arch. Graphical representation of static solution curves was accomplished through plotting of selected state variables against a common variable λ. Crossings of solution curves at the λ equal to zero axis identified candidate equilibrium states and gave insight into the overall quantity of states. A difference ratio between the as modeled and candidate equilibrium configurations provided for a metric to identify the starting or initial state for consideration. The order in which states were evaluated was then established through following of solution curves from one state to another. A procedure for selecting equilibrium using this order and requirement for reducing potential energy was proposed and confirmed plausible for cases studied. Eigenvalues of the linearized governing equations were used as a secondary metric to verify stability at the chosen equilibrium states for each system. The quantity and type of eigenvalues of the Jacobian for linearized DAEs was reported and patterns of all negative real or non-complex values were shown to be an indicator for stability. Final determination of stability was made after conversion of these equations to ODEs in state space format such that established rules using eigenvalues could be applied. Governing equations, candidate equilibrium state vectors, and eigenvalues from linearized ODEs were compared to those obtained using MSC ADAMS commercial software for further validation. The proposed method for identifying equilibrium through path following of static solution curves offers an alternative to dynamic simulation that includes potential computational cost savings depending on system size and complexity. Identifying

equilibrium in this manner is more robust than using single point solution procedures that may converge to a state that numerically satisfies but does not physically represent equilibrium or never achieves convergence, especially near limit points.

CHAPTER 5

PARALLEL PROCESSING OF THE JACOBIAN

Demonstrating speedup for parallel code on a multi-core shared memory PC can be challenging in MATLAB due to underlying parallel operations that are often opaque to the user. These hidden operations can limit potential for improvement of serial code even for the so-called embarrassingly parallel applications. One such application is the computation of the Jacobian matrix inherent to most nonlinear equation solvers. Computation of this matrix represents the primary bottleneck in nonlinear solver speed such that commercial finite element analysis and multi-body-dynamics codes attempt to minimize such computations. A timing study using MATLAB's Parallel Computing Toolbox [48] was performed for numerical computation of the Jacobian [58]. Several approaches for implementing parallel code were investigated while only the single program multiple data method (MATLAB's command *spmd*) using composite objects provided positive results. Parallel code speedup is demonstrated but the goal of linear speedup through the addition of processors was not achieved due to PC architecture.

5.1 INTRODUCTION

Most PCs available on the market today come equipped with multi-core processors where cores share a common memory [44,48]. Programming on these systems is typically done via threading, which is a special case of an operating system process whereby threads share memory [44]. Multithreading or Intel's proprietary version called *hyperthreading* is also commonplace

and allows for resource duplication within a given central processing unit core [44]. Such computer architecture is what enables programming languages to exploit thread-parallel operations. Use of this technology where parallel operations are carried out autonomously without any user input or code modifications is often referred to as implicit [48] or multithreaded parallelism [49] where such operations are an integral part of the software. MATLAB software uses multithreaded parallelism by default for many of its trigonometric and linear algebraic operations [48,49]. A partial list of these functions including linear equation solvers, matrix factorization methods, etc. can be found on the MathWorks user support website [50]. This default means serial versions of MATLAB code are typically running lower level parallel operations that users may be unaware of and have little or no control over. These operations can be validated in a qualitative sense through monitoring of the CPU usage history plots using Windows Task Manager or a similar program. A small serial program run using an Intel Core i7 chip for example showed use of only a single processor, while a much larger or more computationally intensive program showed use of all available processors. Although the Windows Task Manager showed a total of eight available processors for this chip, it should be noted that this is a quad-core processor with eight available threads meaning four of the processors are non-physical. MATLAB still allows users to specify the number of threads being used through the *maxNumCompThreads* command [15]. Warning has, however, been issued by MathWorks that this feature will be removed in a future release, implying multithreading is the intended normal software environment.

Even though MATLAB exploits use of multi-core processors for serial programming, code can potentially be further improved for speed through use of the Parallel Computing Toolbox [48]. This toolbox enables use of explicit parallelism where specific tasks can be

directed to specific processors. Reference to underlying parallelism for serial code could not be found in the Parallel Computing Toolbox documentation [15] while only a single reference to "built-in parallelism provided by the multithreaded nature of many of the underlying MATLAB libraries" was found in a later version. This lack of information may leave users unaware of underlying parallelism in serial code leading to high expectations for speedup of parallel versions. According to a professor who specializes in computer science, a common scenario of first-time developers of parallel code is to find out it is actually slower than the serial version, which he attributes to lack of understanding of how computer hardware works, at least at a high level [44]. Establishing serial MATLAB or any computer code with underlying parallelism as the de facto standard by which to gage parallel code performance can significantly add to the challenge of achieving speedup. This character can be true even for the so-called embarrassingly parallel applications as underlying parallelism may leave little room for code improvement. Users should also be aware that unlike distributed memory systems, the addition of processors for parallel computing on shared memory systems does not necessarily provide linear type improvement for speedup where doubling the number of processors doubles computational speed and so on.

MATLAB users who maintain or develop their own versions of nonlinear FEA or MBD software codes may wish to speedup computations using the Parallel Computing Toolbox. For Newton-Raphson based solvers, the major cost per iteration lies in computation of the Jacobian matrix [1] where it is often referred to as the tangent stiffness matrix in the FEA literature. Increasing the speed at which this computation is performed can have a dramatic effect on the overall solution time, especially for dynamic simulations where the matrix is not only computed during solver iterations but also at time steps during the simulation as well. One of the solver

options in MBD software MSC ADAMS for example contains heuristics to help minimize the number of times computation of the Jacobian is performed as this represents the most time consuming part of a simulation [12]. Candidate algorithms for parallel computation of the Jacobian should be gaged for performance relative to a similar serial version. One of the simplest and most widely used metrics to gage parallel performance is observed speedup being defined as serial execution divided by parallel execution time in terms of total elapsed or wall-clock time [39]. This metric can be accomplished in MATLAB using the *tic* and *toc* functions. MATLAB also offers a function for measuring CPU time but does not recommend using it on systems capable of *hyperthreading* as the *tic* and *toc* functions are more reliable [15].

5.2 METHODS FOR COMPUTING THE JACOBIAN

The Jacobian is a matrix of first-order partial derivatives resulting from the linearization or Taylor series expansion of a set of nonlinear equations about a known point or solution. This matrix provides for a local linear model about the known point that can be used to predict nearby points in the nonlinear model. Computation of this matrix is fundamental to most nonlinear solver algorithms and is performed on an iterative basis until a converged solution to the nonlinear model is found. In commercial FEA codes such as Nastran [11] and Abaqus [9], the Jacobian or tangent stiffness matrix is part of a Newton-Raphson type solver. Due to computational expense, effective solution strategies often minimize computation or hold the Jacobian constant during iterations for a modified Newton-Raphson approach [1]. Commercial MBD software MSC ADAMS [12] also uses a Newton-Raphson type solver for dynamics and only updates the Jacobian if convergence is not achieved within a finite number of iterations.

Development of efficient algorithms for computation of the Jacobian or derivatives in general is paramount to nonlinear equation solvers as this tends to dominate the total computational time for obtaining solutions.

Several methods for computing derivatives needed to construct the Jacobian are available. Review of popular FEA [9,11] and MBD [12] software documentation indicates that obtaining derivatives numerically by finite difference is still the standard approach being used. A goal set by developers of MSC ADAMS is to eventually eliminate the need for numerical differentiation [59] due to high computational cost. By finite difference, derivatives of an individual function f with respect to an independent variable x are obtained by applying a small change or perturbation to x. Variable h can be used as a perturbation parameter and is added to x to represent this change. The resulting expression for the derivative of $f(x)$ or $f'(x)$ by a forward finite difference is

$$f'(x) \approx \frac{f(x+h)-f(x)}{h} \tag{51}$$

which represents an approximation to the derivative by the calculus definition as it does not include the limit expression for h tending to zero. Observe that h cannot become too small due to limits of numerical precision on computers and possibility of dividing by a value close to zero. Take for example a sample function $f(x) = x^3 + 2x + 1$ with an exact or analytical derivative of $f'(x) = 3x^2 + 2$. Using Eq. (51) for estimation of the derivative about $x = 1$ and varying h by a factor of 10 between $10^0$and 10^{-20} results in Fig. 31 for the percent error of Eq. (51) with respect to the analytical derivative. Results were obtained using MATLAB with double precision representation of floating point numerical values. Error for this case was minimized for $h = 10^{-8}$ and the procedure broke down or failed for $h \leq 10^{-16}$ where $f(x)$ and $f(x + h)$ became numerically equivalent after the 15th decimal place or a maximum of 16 significant

digits. The numerator in Eq. (51) became zero for these instances resulting in 100% error. Additional information on this method including error can be found in Ref. [60].

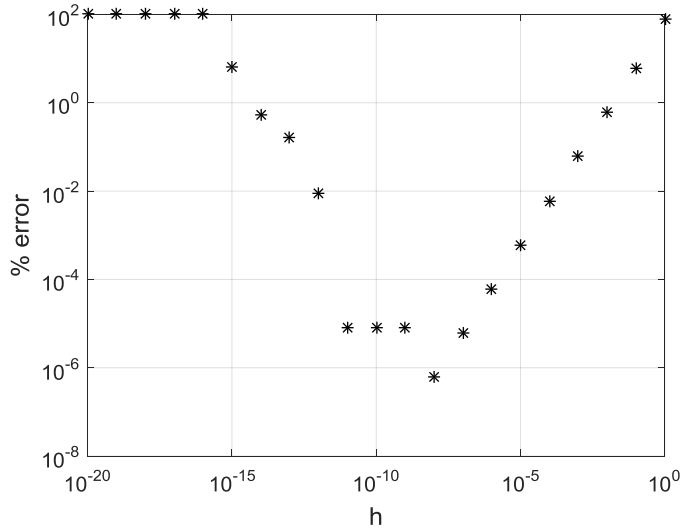

Figure 31. Percent error vs. parameter h for given function f

An alternative to obtaining derivatives numerically by finite difference is symbolic differentiation. In this case, the symbolic expression for $f(x)$ would be differentiated using rules of calculus to obtain a new symbolic expression for $f'(x)$. Numerical values of x can then be substituted into $f'(x)$ for specific values of the derivative with an accuracy of 16 significant digits when using double precision. The result for $f'(1)$ in this case would be 5 followed by a decimal with fifteen zeros. The value obtained by finite difference, on the other hand, is 4.999999969612644, which exhibits error in the eighth decimal place for $h = 10^{-8}$. Although symbolic differentiation can be used to obtain derivatives in an exact sense, computational overhead for manipulating symbolic expressions using calculus based rules would limit this procedure to small problems to avoid excess solve time. Further, some functions may lack analytic description being modeled by tabular data requiring table lookup or interpolation

procedures which cannot be differentiated symbolically. A comprehensive list of computer programs capable of manipulating symbolic math expressions including their capabilities can be found at *https://en.wikipedia.org/wiki/List_of_computer_algebra_systems*.

A third alternative to obtaining derivatives is automatic differentiation. The algorithm for computing derivatives in this case uses existing computer programs or subroutines for computation of a function f and supplements them with a new routine for computation of f'. Derivatives are not subject to approximation error and are produced in an exact sense similar to the symbolic method. Automatic differentiation seems to be gaining favor based on the amount of research and computer codes being generated. Developing efficient, robust algorithms for large-scale applications has been identified as a research challenge by a developer using MATLAB [61] and favorable timing results in comparison to finite difference have been obtained for a specific class of problem by developers using C++ [62]. MSC did a study for integrating ADIFOR [63] into the FORTRAN version of ADAMS but it was not stated to having been adopted [12] implying computational overhead exceeded that of finite difference for this general purpose commercial software. A community portal with information on software, conferences, and workshops dedicated to the subject matter can be found at *http://www.autodiff.org*.

Calculation of derivatives for components of the Jacobian matrix were made using the finite difference method in both serial and parallel code versions for this study. This decision was based on ease of implementing various parallel versions for evaluating speedup and likelihood it remains the most practical approach for computing derivatives in FEA and MBD programs. Equations for a repeating link or chain system were chosen for computing the Jacobian due to scalability and a specific reference in the *LSOLVER* section of the MSC

ADAMS solver manual [12]. Better performance is claimed when using an available sparse matrix solver with parallel capability for systems of 5000 degrees-of-freedom and larger with exception to some models like simply-connected long chains. This trend set a goal for positive margin on speedup for linkage systems under 5000 DOF for parallel computation of the Jacobian using MATLAB code. Although numerical accuracy of derivatives in the Jacobian may be of concern, highly accurate results for Newton-Raphson type solvers are not required. The modified Newton-Raphson method, for example, may hold the Jacobian constant, without any updates during iterations, and the BFGS method [16-19] avoids explicit computation of the Jacobian by only computing an approximate update during solver iterations.

5.3 EQUATION THEORY AND BACKGROUND

Equations for the linkage system used in this study were derived using Lagrange's method [52]. This derivation results in a set of nonlinear DAEs used for computation of the Jacobian matrix. Equations can be represented in compact form where $\boldsymbol{u}$ is understood to contain a mix of space and time dependent variables as represented by Eq. (1) and linearized about a known state $\boldsymbol{u}_i$ using a first-order Taylor series expansion for solution by the Newton-Raphson method.

$$\boldsymbol{f}(\boldsymbol{u}) \approx \boldsymbol{f}(\boldsymbol{u}_i) + \left(\frac{\partial \boldsymbol{f}}{\partial \boldsymbol{u}}\right)_i (\boldsymbol{u} - \boldsymbol{u}_i) = \boldsymbol{0} \tag{52}$$

Bolded terms in Eq. (52) are used to represent vectors where $\boldsymbol{u}$ is the vector of unknown variables and $\boldsymbol{f}(\boldsymbol{u})$ is the system of DAEs. Vector $\boldsymbol{u}$ is often referred to as the state vector and contains variables for position, velocity, and constraint forces for each link in the system. The

derivative term in Eq. (52) is the Jacobian with the following expanded or matrix format for N unknown variables or DOF.

$$\left(\frac{\partial \boldsymbol{f}}{\partial \boldsymbol{u}}\right)_i = \begin{bmatrix} \frac{\partial f_1}{\partial u_1} & \cdots & \frac{\partial f_1}{\partial u_N} \\ \vdots & \ddots & \vdots \\ \frac{\partial f_N}{\partial u_1} & \cdots & \frac{\partial f_N}{\partial u_N} \end{bmatrix}_i \tag{53}$$

Eq. (53) shows that a system containing N-DOF will have NxN or N^2 derivatives in the Jacobian. Calculation of every individual derivative may not be required, however, as individual equations in Eq. (52) can be organized in a manner such that the Jacobian will have a known pattern. This organization is true for mechanical systems in general and sparsity or zero-entries in the Jacobian resulting from linearization of governing DAEs can be taken advantage of as well. Details on the derivation of equations using this approach for a single link or pendulum including pattern forming of the Jacobian can be found in Ref. [53]. The single link has eight unknown variables for this case as motion is constrained to a plane. A similar planar constraint was used for the multi-link system in this study where total DOF is obtained by multiplying the number of links by eight. Variables or DOF for each link consist of two for position, one for orientation, their corresponding derivatives, and two for the constraint forces.

Governing equations for the multi-link systems were produced using a MATLAB function or subroutine based on a repeating pattern for systems of two links and greater. Serial and parallel subroutines with options for sparse versus dense formulations were then developed for timing of numerical computation of the Jacobian for a varying number of links. Validation of computer code was performed for a two link system under the influence of gravity using a previously developed nonlinear software suite capable of simulating dynamic systems [51]. Results for the horizontal constraint force versus time for the grounded connection with links

initially configured as an upside down "V" are shown in Fig. 32. Blue dots on the figure were found using MATLAB and the red line was found using MSC ADAMS.

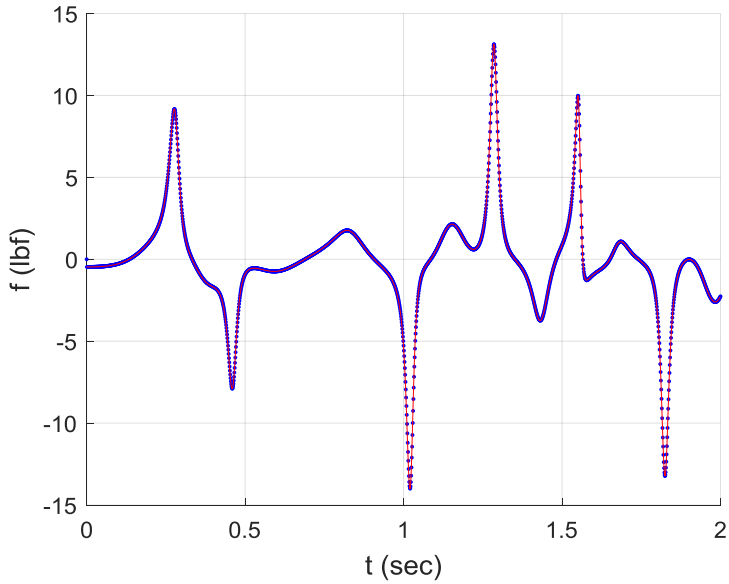

Figure 32. Constraint force vs. time for double link system

The 16x16 Jacobian was small enough in this case where hand or symbolic computation of derivatives could be performed with reasonable effort. A function with expressions for derivative terms was then developed for computation of the Jacobian in an exact sense for comparison to a serial numerical version in terms of solution time for the two second simulation shown in Fig. 32. The total solution or wall time for the MATLAB simulation was 0.45 seconds using the explicit definition of the Jacobian versus a 5 second solution time for computation of derivatives numerically by finite difference. The time step used for the simulation was 0.001 seconds and convergence was achieved within 3 to 4 iterations per time step using a Newton-Raphson type solver where the Jacobian was updated at every iteration. The over tenfold increase in solution time between the two simulations demonstrates the high cost associated with numerical computation of the Jacobian. Switching to a modified Newton-Raphson method

where the Jacobian was calculated numerically only once per time step and held constant increased iterations for convergence up to 9 in some instances but reduced the solution time to 1.67 seconds. This behavior further reinforces that computation of the Jacobian should be minimized to avoid excessive solution times in general. Note that the explicit definition of the Jacobian provided for an idealized case for timing results. However, such an approach would not be practical for large systems and would require use of a numerical procedure.

5.4 SERIAL CODE IMPLEMENTATION

A simplified version of MATLAB code used to numerically compute the Jacobian matrix in a serial fashion is shown in Fig. 33. Function *ser_jacobi* is defined to output Jacobian matrix $\boldsymbol{J}$ using state $\boldsymbol{u}_i$ as input. Code is "vectorized" in the sense that the matrix is computed a column at a time with a single for-loop verses element-wise using a double for-loop. Column entities in Eq. (53) show equations $\boldsymbol{f}$ being differentiated with respect to a given element of vector $\boldsymbol{u}$ such that perturbations applied to specific elements of $\boldsymbol{u}$ can be used to compute entire columns of $\boldsymbol{J}$. Vectorization is a key concept in MATLAB programming as it simplifies code, allows users to take advantage of underlying subroutines inherent to the programming language, and will likely perform computations in the most efficient manner. The column-wise implementation of Eq. (51) is shown on row twelve of Fig. 33. The $(:, j)$ operator is used to designate all row entities of the j^{th} column in $\boldsymbol{J}$ being a difference in perturbed vector $\boldsymbol{f}(\boldsymbol{u}_p)$ and original vector $\boldsymbol{f}(\boldsymbol{u}_i)$ with all entities being divided by h. Additional information on code vectorization can be found in the *Vectorization* section of the MATLAB user documentation [15].

```
1.    function J = ser_jacobi(u_i)
2.
3.    N    = length(u_i);              % number of elements in u
4.    fu_i = sys_eq(u_i);              % f(u) at state i
5.    h    = 1e-8;                     % perturbation parameter
6.    J    = zeros(N,N);               % pre-allocate memory for J
7.    u_p  = u_i;                      % initialize perturbation vector
8.
9.    for j = 1:N
10.       u_p(j) = u_p(j) + h;         % perturb element j in u
11.       fu_p   = sys_eq(u_p);        % f(u) at perturbed state
12.       J(:,j) = ( fu_p - fu_i )/h; % Jacobian column j derivatives
13.       u_p(j) = u_i(j);             % reset element to original value
14.   end
```

Figure 33. Serial Jacobian computation using MATLAB

Code in Fig. 33 is specific to computation of the full Jacobian matrix or all matrix entities and storing them in a dense format that includes any zero entities. Such computation can be expensive for large systems and a significant reduction in computational cost can be achieved by taking advantage of known patterns and sparsity. Through proper arrangement of state variables in $\boldsymbol{u}$, the Jacobian for the multi-link systems has the following block matrix format consistent with the general format given in Ref. [53]. Zeros sub-matrices are due to Lagrange's method being used to derive governing equations, which results in large sets of equations in redundant coordinates and considerable sparsity for the Jacobian.

$$\boldsymbol{J} = \begin{bmatrix} \frac{1}{dt}\boldsymbol{M} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \boldsymbol{\Phi}_p^T \\ \mathbf{0} & \frac{1}{dt}\boldsymbol{I}_{\boldsymbol{CM}} & \mathbf{0} & [\boldsymbol{\Phi}_\varepsilon^T \boldsymbol{\Lambda}]_\varepsilon & \boldsymbol{\Phi}_\varepsilon^T \\ -\boldsymbol{I} & \mathbf{0} & \frac{1}{dt}\boldsymbol{I} & \mathbf{0} & \mathbf{0} \\ \mathbf{0} & -\boldsymbol{I} & \mathbf{0} & \frac{1}{dt}\boldsymbol{I} & \mathbf{0} \\ \mathbf{0} & \mathbf{0} & \boldsymbol{\Phi}_p & \boldsymbol{\Phi}_\varepsilon & \mathbf{0} \end{bmatrix} \tag{54}$$

Components of $\boldsymbol{J}$ include diagonal sub-matrices $\boldsymbol{M}$, $\boldsymbol{I}_{\boldsymbol{CM}}$, and $\boldsymbol{I}$ being mass, inertia, and identity matrices respectively. Term dt applied to these matrices is the time step or increment used between states for dynamic simulation. Constraint equations are stored in vector $\boldsymbol{\Phi}$ where

$\boldsymbol{\Phi} = \mathbf{0}$ and subscripts p and ε are used to denote partial derivatives with respect to position and orientation variables respectively. Finally, the constraint forces or Lagrange multipliers are stored in column vector $\boldsymbol{\Lambda}$. Sub-matrices for mass, inertia and identity do not change for constant dt or within a given time step and are invariant. Standalone identity matrices are invariant by definition. This invariance leaves only sub-matrices containing $\boldsymbol{\Phi}$ for numerical computation, which dramatically reduces the amount of computational overhead and size of the for-loop in Fig. 33. A more efficient strategy for computation of the Jacobian would now involve pre-allocation and construction of a sparse matrix with invariant terms followed by computation of the $\boldsymbol{\Phi}$ sub-matrix blocks in the last row, and the row two, column four block locations of Eq. (54). Previously calculated $\boldsymbol{\Phi}$ blocks in the last row can then be transposed and inserted into the last column of Eq. (54).

The need for sparse versus dense format of the Jacobian is driven by both computer memory for storage and computational cost of factorization. The Jacobian must be factorized each time a new version is computed as it is part of a linear system being solved during iterations of Newton-Raphson based solvers. Eliminating the storage of zeros and the processing of zero entities in sparse computational algorithms can have dramatic effects on efficiency and become more apparent as systems increase in size. Table 13 for example shows the wall time needed to solve a sample linear system $\Delta \boldsymbol{u} = \boldsymbol{J}^{-1}\boldsymbol{R}$ where $\boldsymbol{J}$ is stored in both sparse and dense formats for timing comparison. Variable $\Delta \boldsymbol{u}$ denotes an incremental change in state vector $\boldsymbol{u}$, $\boldsymbol{R}$ is a residual vector set to all ones, and Jacobian $\boldsymbol{J}$ has been factorized into lower and upper triangular elements versus taking the inverse for solution. The speed factor in Table 13 is a multiplier of how many times faster the sparse solver is compared to the dense, and the non-zero (NZ) ratio is the number of non-zero terms divided by the total or N^2 number of terms in $\boldsymbol{J}$. Numerical values

in the table indicate that sparsity is significant and large performance gains in solution time can be expected by using the sparse matrix format and solver. A detailed overview of sparse matrices and sparse matrix operations in MATLAB can be found in Ref. [64].

Table 13
Solution times using sparse and dense Jacobian (sec)

Links	DOF	sparse	dense	factor	NZ ratio
200	1600	0.003	0.08	27.64	2.0E-03
400	3200	0.006	0.56	90.21	1.0E-03
600	4800	0.009	1.40	148.57	6.8E-04
800	6400	0.012	3.15	252.23	5.1E-04
1000	8000	0.016	5.87	372.81	4.1E-04
1200	9600	0.019	9.84	516.66	3.4E-04
1400	11200	0.023	15.06	664.97	2.9E-04
1600	12800	0.027	23.10	853.97	2.5E-04
1800	14400	0.031	32.24	1052.13	2.3E-04
2000	16000	0.034	43.29	1262.09	2.0E-04

5.5 PARALLEL CODE IMPLEMENTATION

Parallel processing of computational algorithms in MATLAB can be implemented using either parallel for-loops, *parfor*, or by *spmd*. Parallel for-loops work only for the simplest of algorithms and each loop must be totally independent from all others. The perturbed vector $\boldsymbol{u}_p$ inside the for-loop shown in Fig. 33 is updated element-wise over the course of loop iterations such that a parallel for-loop cannot be used for computing the Jacobian in this manner. The single program multiple data or *spmd* option, however, is more versatile and allows for specific tasks to be assigned to specific processors. Once a parallel job is started in MATLAB, one processor is assigned the role of client while the remaining processors are assigned the role of workers. Computation of the Jacobian can be accomplished by dividing the for-loop in Fig. 33 over a specified number of processors using *spmd*. This specification requires creation of an

indexing array used to identify the start and finish column identification numbers based on desired matrix partitions. However, changes to the serial code are minimal making this method easy to implement.

The Jacobian can be stored using either distributed arrays, codistributed arrays or composite objects when using the *spmd* option. Arrays are considered as distributed or codistributed as viewed from the perspective of the client or worker processors. Distributed arrays are created on the client where codistributed arrays are created on the workers themselves. Positive timing results for writing and updating elements of these type arrays could not be obtained. This lack of improvement may be due to the client-worker relation where writing new elements to workers causes a similar update to be performed on the client. However, explicit reference to how writing of elements to these arrays is performed could not be found in documentation and users do not have access to the underlying C-code used to write MATLAB software. Composite objects, on the other hand, produced positive results for computing the Jacobian in parallel. These objects exist on workers and have the same variable name on all workers but store different data. The downside of composite objects is that they must be converted back into a single matrix for use in computations in their entirety. Parallel computation of the Jacobian using composites for example will be stored in independent groups of columns on workers. If the Jacobian is then needed for use in a linear system $\boldsymbol{J\Delta u = R}$, it will need to be converted into matrix form.

A parallel version of the for-loop used to calculate the Jacobian in Fig. 33 is shown in Fig. 34. Code is again specific to computation of the full Jacobian matrix in dense form. This instruction provides for the most compact, readable version of code to demonstrate *spmd* parallelization. Computation of the index array, *index*, for parsing of the Jacobian is not shown

on the figure. Variable *w* is used to designate specific worker or processor identifications. The *labindex* function is used to distribute tasks being calculation of specific columns of the Jacobian to specific workers. Column identification numbers all start with one for composite objects on workers and an additional variable *k* is used to distinguish between column identification numbers for the entire Jacobian and sections being stored in composite objects. Computation of the **Φ** blocks only would require additional indexing for start and finish row identification numbers versus processing of all rows as shown in Fig. 34. Final assembly of the Jacobian using dense or sparse format would then be carried out after the *spmd* block of code is complete.

```
spmd

    J = zeros(N,C);     % pre-allocate and distribute composite across workers
    u_p = u_i;          % initialize perturbation vector
    fu_i = sys_eq(u_i); % f(u) at state i

    for w = 1:NP                                    % loop over workers
        if labindex == w                            % specific worker w tasks
            for j = index(w*2-1):index(w*2)         % loop over column range in J
                u_p(j) = u_p(j) + h;                % perturb element j in u
                fu_p = sys_eq(u_p);                 % f(u) at perturbed state
                if w == 1                           % worker 1 only
                    J(:,j) = (fu_p-fu_i)/h;         % Jacobian partition derivatives
                else                                % all other workers
                    k = 2*labindex - 2;             % shift column ID to 1 for composite
                    J(:,j-index(k)) = (fu_p-fu_i)/h; % Jacobian partition derivatives
                end
                u_p(j) = u_i(j);                    % reset to original
            end
        end
    end

end
```

Figure 34. Parallel Jacobian computation using MATLAB

5.6 CODE TIMING RESULTS

The timing of computer code was accomplished using the 2015b version of MATLAB software and is reported using wall time. The *tic* and *toc* functions in MATLAB behave similar to a stopwatch where *toc* provides for the total elapsed or wall time since the last initiation of *tic*.

The *cputime* function offers an alternative but was not used due to potential for misleading results. Additional explanation of the two timing methods can be found in the *Measure Performance of Your Program* section of the MATLAB user documentation [15]. Here, the *tic* and *toc* functions are stated to be more reliable than *cputime* and significant difference in reported times can occur due to *hyperthreading* where instructions are processed in parallel on a single processor. Wall time may be considered a more conservative approach for characterizing performance of computer code as it includes all communications overhead associated with parallel operations.

Wall timing results for processing of the Jacobian are shown in Tables 14 through 16. Each table includes a timing comparison of serial to parallel code for a given number of links using a varying number of processors (NP). The size of the Jacobian matrix or number of rows and columns is equal to the DOF number. Wall time is reported in seconds where an associated speedup factor defined as the serial divided by parallel time is used to indicate performance. Results were obtained using a Windows 7 laptop computer with an Intel i7-3720QM processor and available 16 GB RAM (gigabytes of random access memory). A maximum of 8 threads or processors were available to MATLAB as workers and timing is initially reported using maximum resources. This decision was based on identifying the smallest DOF system with positive performance or a speedup factor greater than one with maximum parallel communications overhead. The number of links was then varied in an increasing manner until the speedup factor no longer demonstrated significant gains in performance. At this point, use of computational resources is considered maximized with no additional bandwidth available for further performance gains. Lines across the center of tables are used to denote this breakpoint.

The number of processors was then decreased while holding the DOF constant showing an expected decrease in the speedup factor due to the reduction of computational resources.

Table 14 considers computation of all entities or the full Jacobian matrix using composite objects only and saves them using dense format; this includes zero terms. Positive performance with a speedup factor of 1.2 occurs for a 200 link, 1600 DOF system. As the number of DOF continues to increase, performance is seen to level off at 8000 DOF with a maximum speedup factor of 3.8. Note that linear speedup could not be obtained as the addition of processors does not come with additional memory. Decreasing the number of processors while holding DOF constant at 8000, then provides for a minimum speedup factor of 1.7 when using only two processors. Computations used for the Jacobian in Table 15 were similar to those in Table 14 with the exception of inclusion of time to convert the composite object to a double precision matrix. The conversion is simple but cost is significant as seen by the overall reduction in speedup factor when compared with corresponding values in Table 14. Positive margin for speedup now requires a 3200 DOF versus 1600 DOF system and performance levels out at 6400 DOF versus 8000 DOF when using 8 processors.

Table 16 provides results for a pre-allocated sparse Jacobian with invariant sub-matrices and computation of the constraints or blocks containing $\mathbf{\Phi}$ only. Composite objects are used for the constraint blocks and time to convert to sparse double precision format is included as well. Gains for parallel performance are seen for systems up to 6400 DOF. Wall time is the lowest as compared to other methods and use of sparse format will provide a significant speed advantage during a linear solution phase as shown in Table 13. This procedure for computing the Jacobian would be considered the most practical and recommended as it takes advantage of known

patterns, sparsity, and conversion to double precision matrix format for use in solving a linear system.

Table 14
Calculation of full Jacobian, dense composite format (sec)

Links	DOF	NP	serial	parallel	factor
200	1600	8	0.6	0.5	1.2
400	3200	8	3.3	1.6	2.1
600	4800	8	9.7	3.1	3.1
800	6400	8	18.6	5.3	3.5
1000	8000	8	30.5	8.1	3.8
1200	9600	8	45.3	12.0	3.8
1000	8000	8	30.5	8.1	3.8
1000	8000	6	30.0	9.1	3.3
1000	8000	4	30.2	11.5	2.6
1000	8000	2	30.2	18.0	1.7

Table 15
Calculation of full Jacobian, dense matrix format (sec)

Links	DOF	NP	serial	parallel	factor
200	1600	8	0.6	0.8	0.7
400	3200	8	3.2	2.5	1.3
600	4800	8	9.9	5.3	1.9
800	6400	8	19.0	9.2	2.1
1000	8000	8	32.3	15.5	2.1
1000	8000	8	32.3	15.5	2.1
1000	8000	6	32.3	16.3	2.0
1000	8000	4	32.2	17.7	1.8
1000	8000	2	32.4	24.8	1.3

Table 16
Calculation of block Jacobian, sparse matrix format (sec)

Links	DOF	NP	serial	parallel	factor
200	1600	8	0.2	0.4	0.6
400	3200	8	1.3	0.8	1.6
600	4800	8	3.9	1.6	2.4
800	6400	8	7.8	2.7	2.9
1000	8000	8	11.8	4.0	2.9
1000	8000	8	11.8	4.0	2.9
1000	8000	6	11.3	4.9	2.3
1000	8000	4	11.3	5.9	1.9
1000	8000	2	11.4	8.4	1.4

5.7 CONCLUSIONS

Successful development of explicitly defined parallel code for computing the Jacobian matrix was completed using MATLAB. The *spmd* method using composite objects was found to be the only procedure that produced positive results while use of the sparse as compared to dense format provided for dramatic speed improvements for solutions to linear systems. Speedup of parallel code was demonstrated on a shared memory PC and compared to serial code with underlying parallel operations using wall time. This comparison provided for a most conservative estimate for parallel code speedup as underlying parallel operations are integral to MATLAB and wall time includes parallel communications overhead. Linear type parallel speedup could not be achieved using the chosen performance metrics and computer architecture, which are quite common and may represent a typical MATLAB environment. Performance gains were demonstrated, however, and an approximate three times speedup for the recommended sparse format, double precision Jacobian matrix was achieved. The goal of demonstrating speedup for systems under 5000 DOF was also achieved being most applicable to smaller scale FEA or MBD problems that can run efficiently on PCs. MATLAB users running

nonlinear FEA and MBD codes on PCs should expect significant performance gains when using sparse matrix operations and marginal parallel performance gains for systems on the order of 3200 DOF and greater.

CHAPTER 6

SPACECRAFT RELATIVE ORBIT DETERMINATION CASE STUDY

A numerical path following procedure using an arc-length solver is applied to nonlinear algebraic equations sets used to determine initial conditions for spacecraft relative motion in planar and space or three-dimensional orbits. Multiple roots or solutions to such equations are known to exist based on previous work where MATLAB's *fsolve* routine was used to identify solutions. Previous work is revisited and two additional roots for the planar orbit system are found. Parameterized solution curves produced by the arc-length solver provide for a graphical representation of the overall solution and increase likelihood that all roots are found. Identification of all roots is critical as only one represents initial conditions for an orbit of non-zero velocity and minimum energy.

6.1 INTRODUCTION

Orbit determination procedures are used to predict relative motion of one moving body with respect to another through use of a series of measurements and mathematical models [65]. Recent work involves use of Volterra multi-dimensional convolution theory for prediction of this motion [66]. In this reference, one body is designated as the chief while the other is designated as a deputy. A series of measurements are performed that locate the deputy relative to the chief at discrete times and are used to construct a set of nonlinear measurement equations. These equations are coupled quadratic polynomials of the general form

$$\boldsymbol{f}_i(x_0, y_0, z_0, \dot{x}_0, \dot{y}_0, \dot{z}_0) = \mathbf{0} \tag{55}$$

$$\boldsymbol{g}_i(x_0, y_0, z_0, \dot{x}_0, \dot{y}_0, \dot{z}_0) = \mathbf{0}$$

where integer i provides a unique identifier for individual equations in the set. Unknown variables represent initial conditions for position (x_0, y_0, z_0) and velocity $(\dot{x}_0, \dot{y}_0, \dot{z}_0)$ respectively. Specific details regarding structure of equations can be found in Ref. [66] where constants used to define the equations include the Earth standard gravitational parameter $\mu = 3.986x10^5\ km^3/s^2$ and chief mean radius $R_C = 7100\ km$.

Solving of the measurement equations shown in Eq. (55) provides for the initial conditions needed to construct a set of trajectory equations used to determine relative motion between the chief and deputy. Trajectory equations are also defined in Ref. [66]. As these equations depend entirely on the set of initial conditions used, choosing the right set becomes critical. When multiple roots or solutions exist, this choice is based on the set of initial conditions that produces an orbit of non-zero and minimum relative specific energy (e). Equations used to calculate energy are based on the relative motion dynamics and reference frames defined in Ref. [67].

$$e = (V_D^2 - V_C^2)/2 - \mu(1/R_D - 1/R_C) \tag{56}$$

$$R_D = \sqrt{(R_C + x_0)^2 + y_0^2 + z_0^2}$$

$$V_D = \sqrt{(\dot{x}_0 - ny_0)^2 + (V_C + \dot{y}_0 + nx_0)^2 + \dot{z}_0^2}$$

$$V_C = \sqrt{\mu/R_C}$$

$$n = \sqrt{\mu/R_C^3}$$

Subscripts C and D designate chief and deputy respectively where R is radius, V is velocity and n is the chief mean motion. This energy calculation was not used in Ref. [66] as found roots produced relative orbits that were of obvious higher energy as compared to the expected solution. This excess energy trait was not the case for one of the additional found roots identified when using solution curves making the need to compute and assess energy in a systematic process necessary.

In addition to using arc-length solvers for identifying roots, stopping or termination criteria was also defined in the event solution curves continued on a path towards infinity. These curves are best plotted by selecting any or all of the independent variables and plotting them with respect to the common scalar solution λ on the vertical axis. Solution curves that do not close or remain open and increasing towards infinity need to be terminated at some point and designated as not turning back towards the $\lambda = 0$ axis for an additional root. This termination can be based on asymptotic or linear type behavior in the solution that may develop for continuously increasing or decreasing λ. Trends in solution curves towards vertical, horizontal, or oblique type asymptotes can be identified in a qualitative sense as viewed on plots within a given sectional view of hyperspace. Although path following of the solution could be stopped based on observation, more definitive criteria based on change in slope was used to terminate the procedure. Due to the multi-degree-of-freedom nature and coupling between equations for given systems, the behavior of all independent variables should be considered simultaneously when evaluating for asymptotes.

One of the simplest methods for identifying asymptotic behavior involves monitoring of the Jacobian or tangent stiffness matrix $\boldsymbol{K}$ for change at selected λ locations or $\Delta\lambda$ increments along a given solution curve. Row vectors $\boldsymbol{k}_{ij}$ of the Jacobian matrix represent change in slope

with respect to independent variables for individual equations or representative hypersurfaces contained in the nonlinear system. Subscript i designates the row number where bolded $\boldsymbol{j}$ designates all column entities with a given row. As equations are coupled and share a common solution, it may be possible to evaluate change using only a single row; however, all rows were chosen for evaluation and provided consistent results. Linear or asymptotic behavior for large and continually increasing or decreasing λ is assumed as change between individual rows of the Jacobian become continually less. A metric based on difference ratio denoted by $diff$ can be established where both a sampling increment $\Delta\lambda$ and a minimum difference used to designate asymptotic behavior will need to be specified.

$$diff = \left\| \left[\boldsymbol{k}_{ij}\right]_{\lambda+\Delta\lambda} - \left[\boldsymbol{k}_{ij}\right]_{\lambda} \right\| / \left\| \left[\boldsymbol{k}_{ij}\right]_{\lambda} \right\| \tag{57}$$

Subscripts located after row vector brackets in Eq. (57) are used to indicate specific points on the solution curve for values of λ. If the relation between two points λ and $\lambda + \Delta\lambda$ on a solution curve share the same slope or became perfectly linear, $diff$ would be equal to zero. A zero value for $diff$ may not be practical to achieve or may require excessive solver iterations during path following and need only be considered small for determination of asymptotic behavior.

6.2 PLANAR ORBIT

Two-dimensional relative motion between two bodies occurs when the deputy's motion lies in the orbital plane of the chief defined by the xy axes of the local-vertical local-horizontal (LVLH) reference frame attached to the chief. In the planar case, Eq. (55) represents four quadratic homogeneous polynomial equations in terms of four unknowns (x_0, y_0) and $(\dot{x}_0, \dot{y}_0)$. Specific orbital conditions and the four measurement times are documented in Ref. [66].

Solution curves for the planar orbit are shown in Figs. 35 through 38 for solution curve 1 and Figs. 39 through 42 for solution curve 2. Roots or candidate solutions for deputy initial orbit conditions are designated using capital letters A through F on curves where they cross the $\lambda = 0$ axis. Although plotting of a single variable only is required for a pictorial of the solution in a given sectional view of hyperspace, all four variables were chosen so that differences between plots could be observed. While some curves self-intersect, others do not and provide for more obvious separation between roots on plots. The order in which roots occur when following a given curve is consistent, but order does not matter for this particular application as the objective lies in finding the root, which minimizes relative specific energy. An all zero or trivial solution where the deputy coincides with the chief exists but is rejected based on the resulting zero energy condition. Specific values for roots obtained using solution curve 1 and their corresponding relative specific energy values are in Tables 17 and 18. Similar values for solution curve 2 are in Tables 19 and 20. Using these tables, state E is chosen based on minimum energy while trivial state D and other higher energy states are rejected.

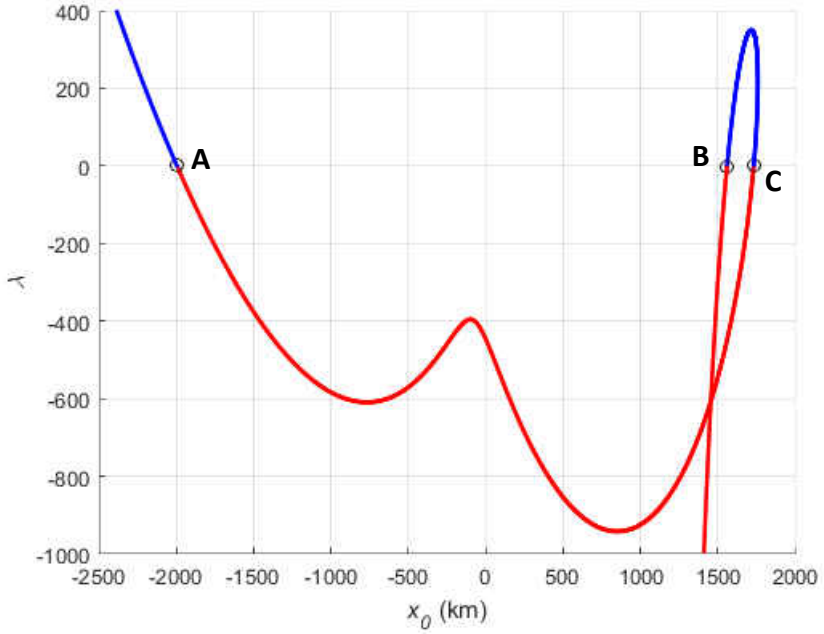

Figure 35. Solution curve 1 for deputy x_0

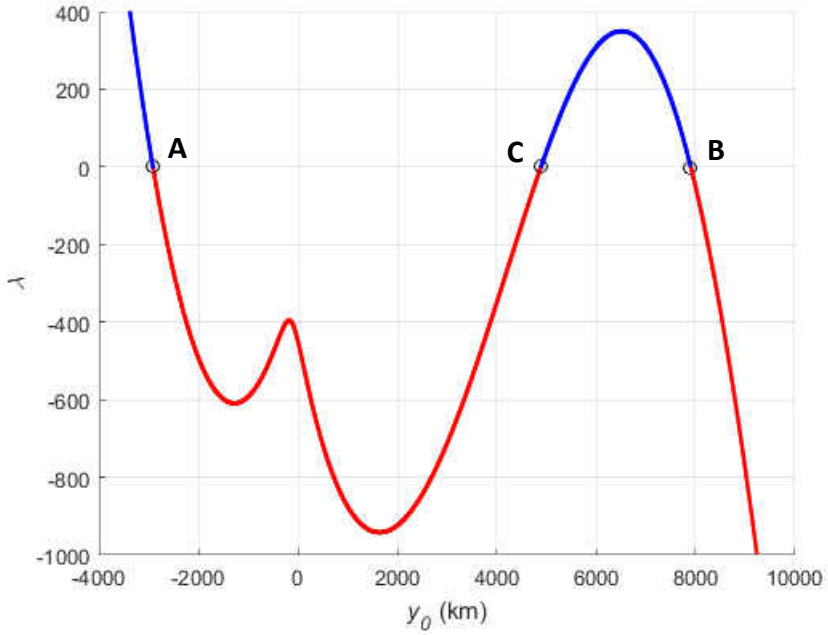

Figure 36. Solution curve 1 for deputy y_0

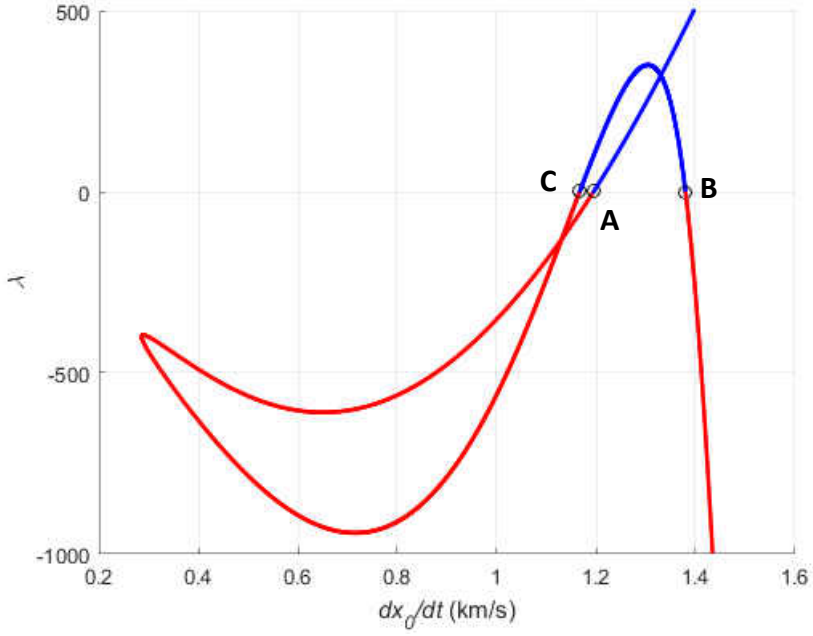

Figure 37. Solution curve 1 for deputy $\dot{x}_0$

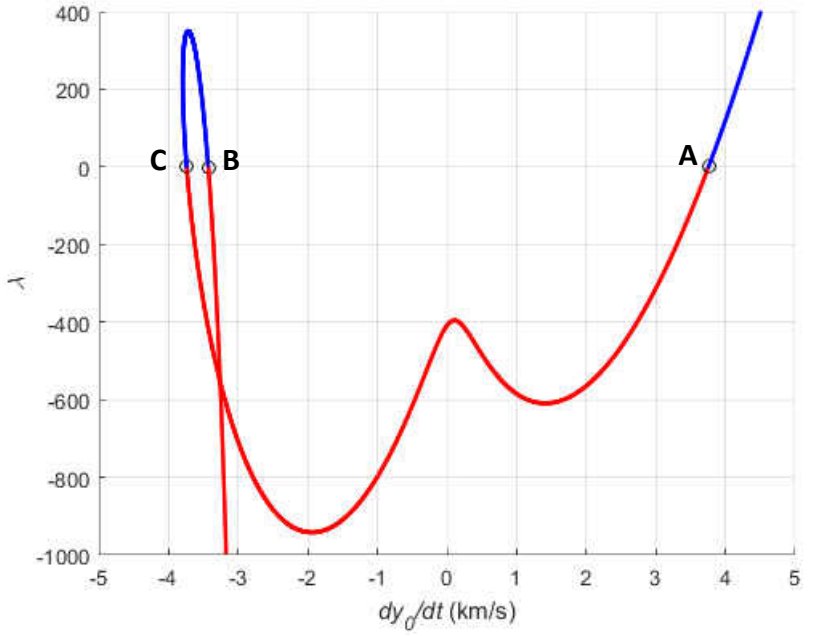

Figure 38. Solution curve 1 for deputy $\dot{y}_0$

Table 17
Candidate states, curve 1

Variable	State A	State B	State C
$x_0(km)$	-1996	1561	1733
$y_0(km)$	-2916	7902	4890
$\dot{x}_0(km/s)$	1.195	1.380	1.167
$\dot{y}_0(km/s)$	3.758	-3.431	-3.741

Table 18
Relative specific energy (km^2/s^2), curve 1

State A	State B	State C
11.199	34.585	12.134

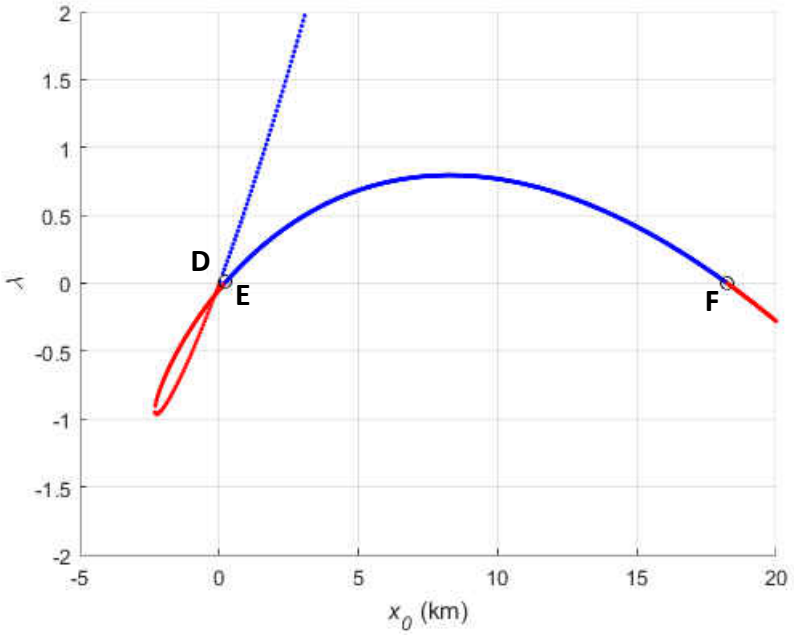

Figure 39. Solution curve 2 for deputy x_0

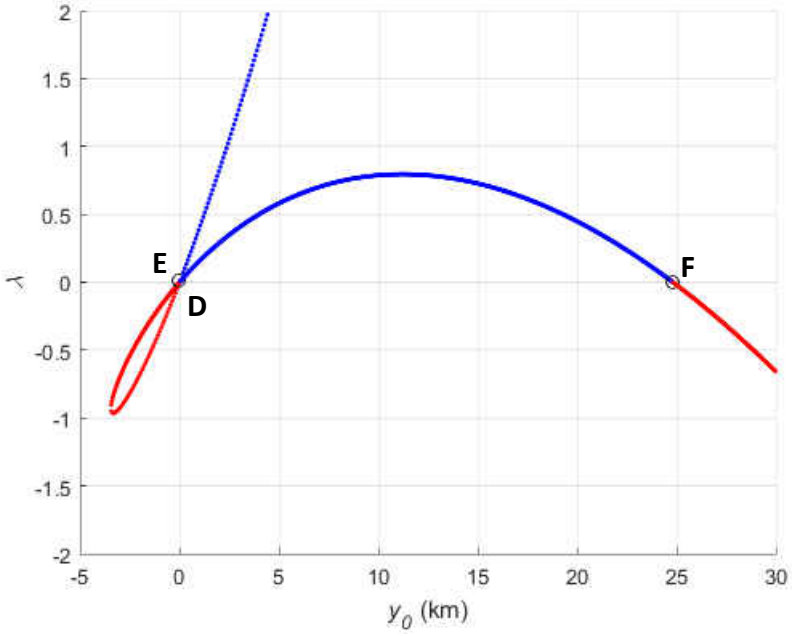

Figure 40. Solution curve 2 for deputy y_0

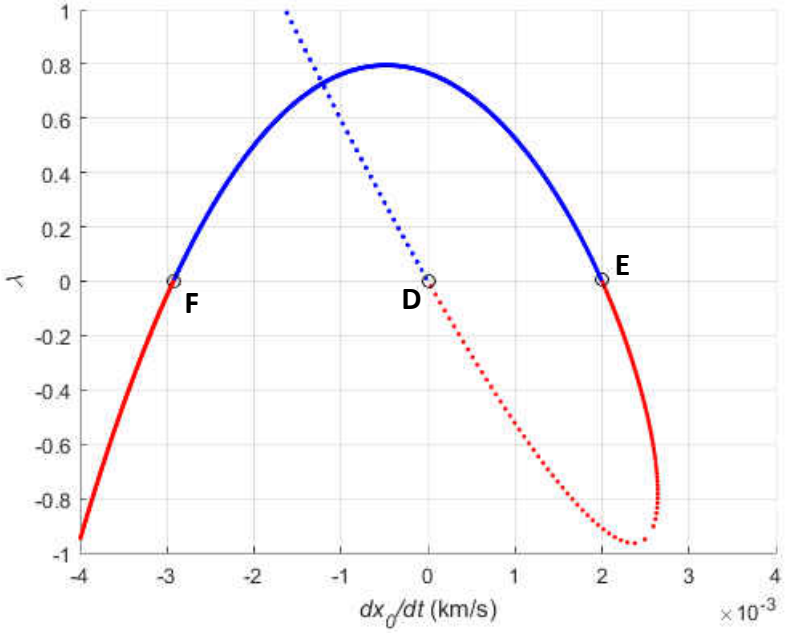

Figure 41. Solution curve 2 for deputy $\dot{x}_0$

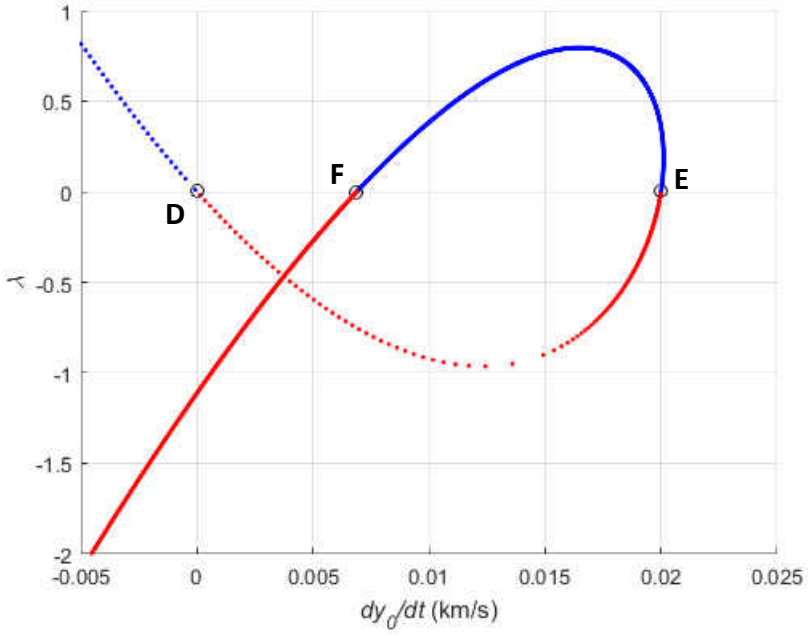

Figure 42. Solution curve 2 for deputy $\dot{y}_0$

Table 19
Candidate states, curve 2*

Variable	State D	State E	State F
$x_0(km)$	0	0.200	18.25
$y_0(km)$	0	0.000	24.81
$\dot{x}_0(km/s)$	0	0.002	-0.003
$\dot{y}_0(km/s)$	0	0.020	0.007

* Determined orbit highlighted red

Table 20
Relative specific energy (km^2/s^2), curve 2*

State D	State E	State F
0	0.153	0.341

* Determined orbit highlighted red

Extended plots for solution curve 1 are shown in Figs. 43 through 46 and Figs. 47 through 50 for solution curve 2. Values for λ exceed $\pm 10^4$ such that development of linear or

asymptotic type behavior can be observed. At these large values for λ, the common scalar solution to all equations is indicative of continually increasing towards infinity and not turning back towards the $\lambda = 0$ axis for possibility of an additional root. In these sectional views of hyperspace, asymptotes primarily appear as oblique with the exception of the x_0 and $\dot{y}_0$ solution curves, which appear to approach vertical asymptotes in the increasing $-\lambda$ direction. The likely presence of asymptotes indicates there are an infinite number of non-zero λ solutions to this system with only a finite number of $\lambda = 0$ roots for the given solution curves. For this particular system, two solution curves containing three roots each were found.

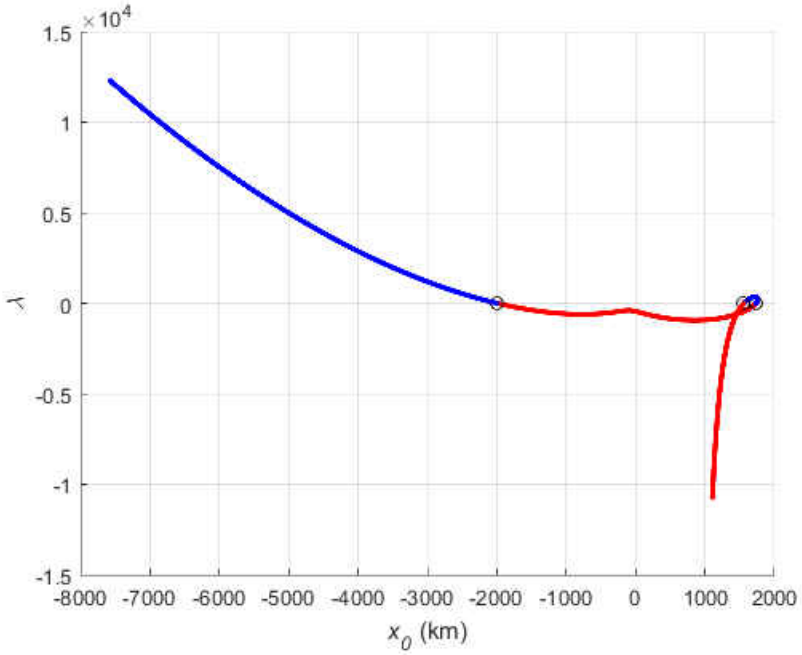

Figure 43. Extended solution curve 1 for deputy x_0

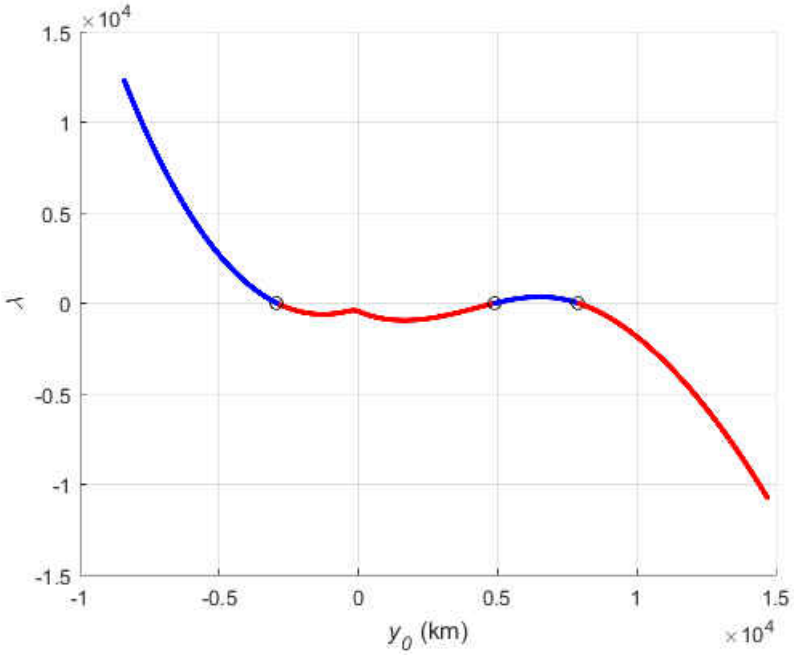

Figure 44. Extended solution curve 1 for deputy y_0

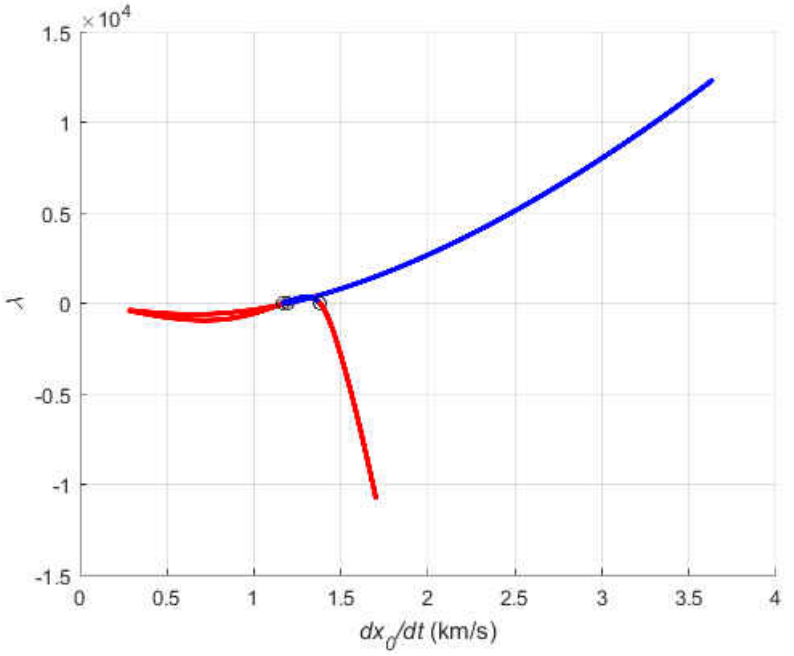

Figure 45. Extended solution curve 1 for deputy $\dot{x}_0$

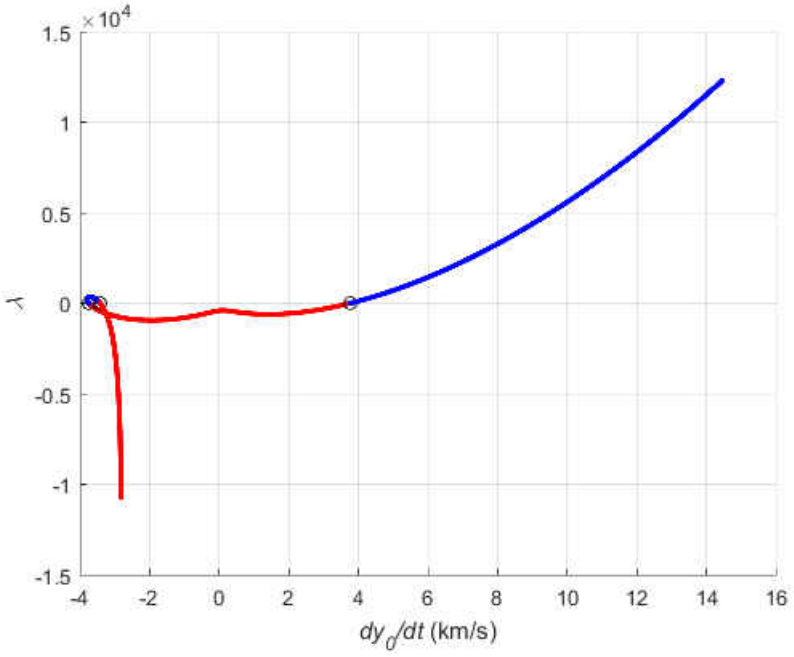

Figure 46. Extended solution curve 1 for deputy $\dot{y}_0$

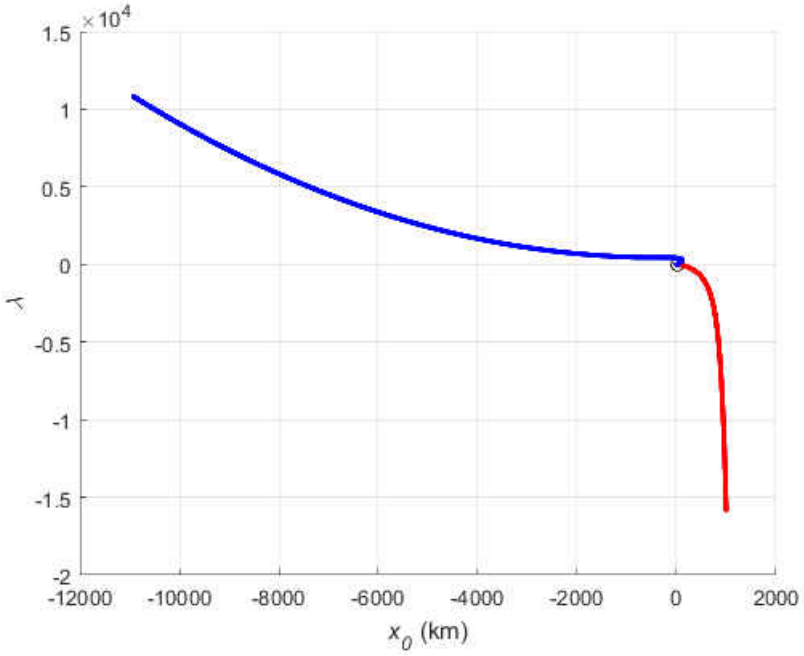

Figure 47. Extended solution curve 2 for deputy x_0

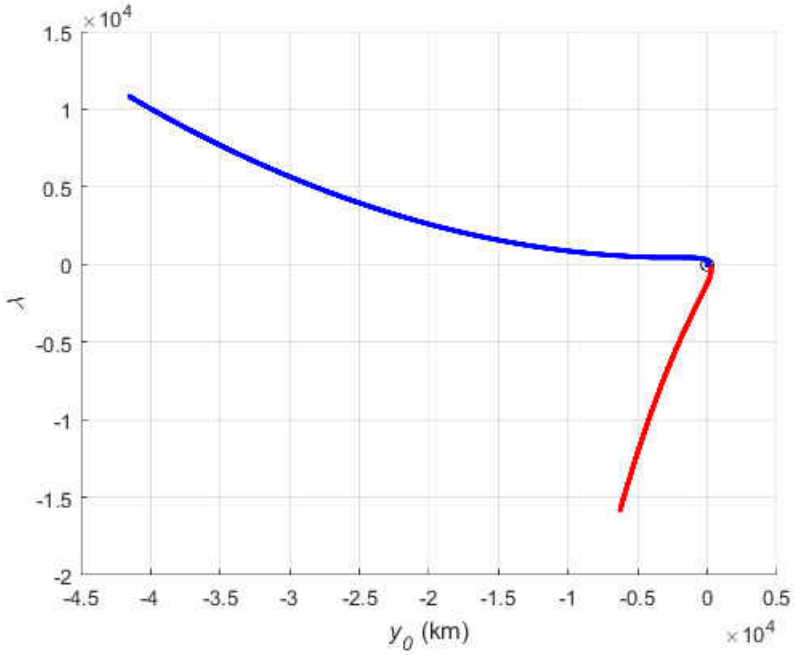

Figure 48. Extended solution curve 2 for deputy y_0

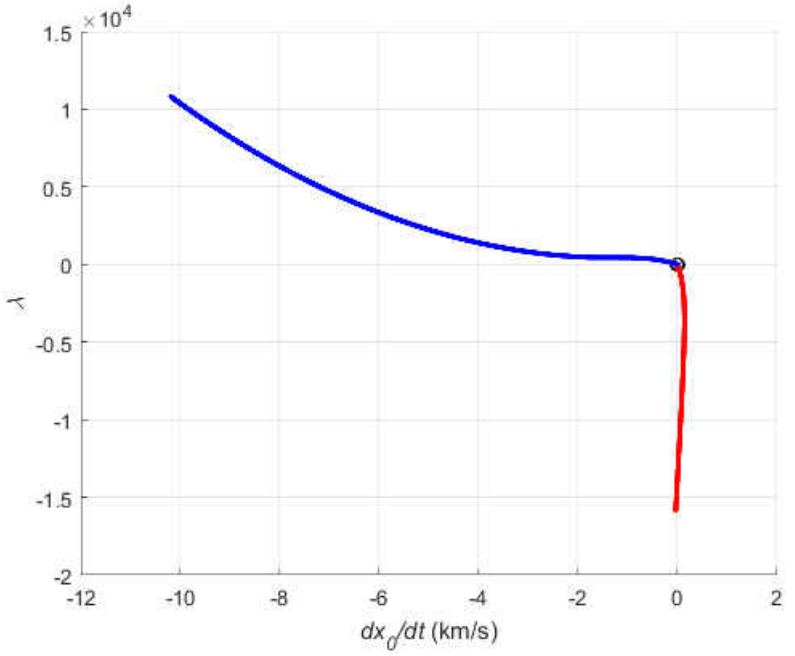

Figure 49. Extended solution curve 2 for deputy $\dot{x}_0$

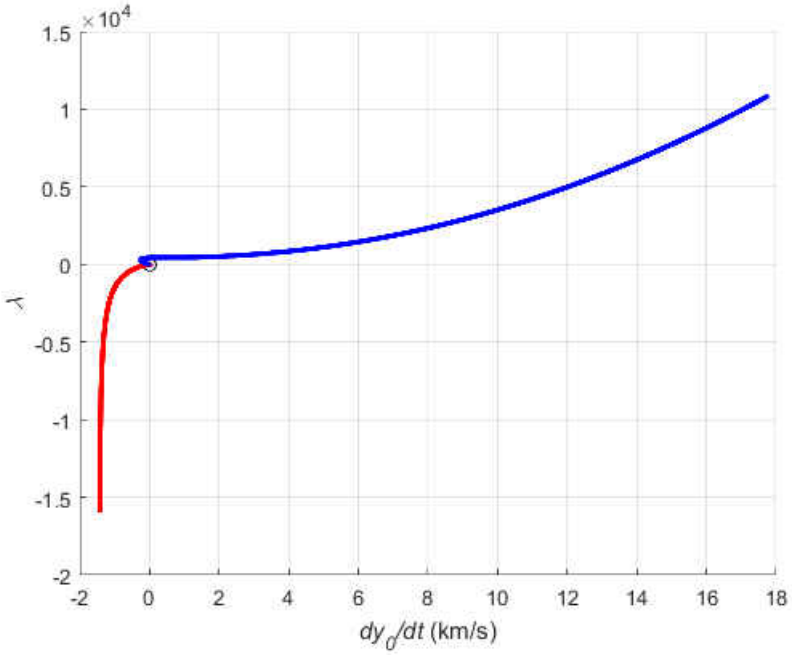

Figure 50. Extended solution curve 2 for deputy $\dot{y}_0$

Plots of difference ratios using Eq. (57) for rows in the Jacobian matrix are shown in Figs. 51 through 54. Computations begin at the $\lambda = \pm 500$ axes, continue toward the $\lambda = \pm 2 \times 10^4$ axes, and cover what appears to be asymptotic regions of the extended solution curve plots. Curves were sampled at locations that provided for a $\Delta\lambda$ increment of approximately 100 for construction of plots.

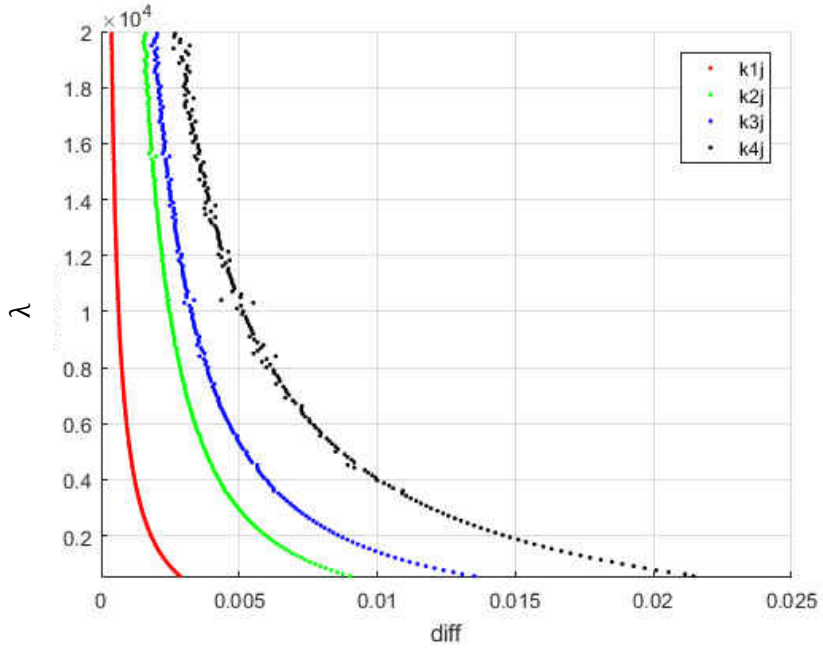

Figure 51. Solution curve 1 difference ratio for positive λ

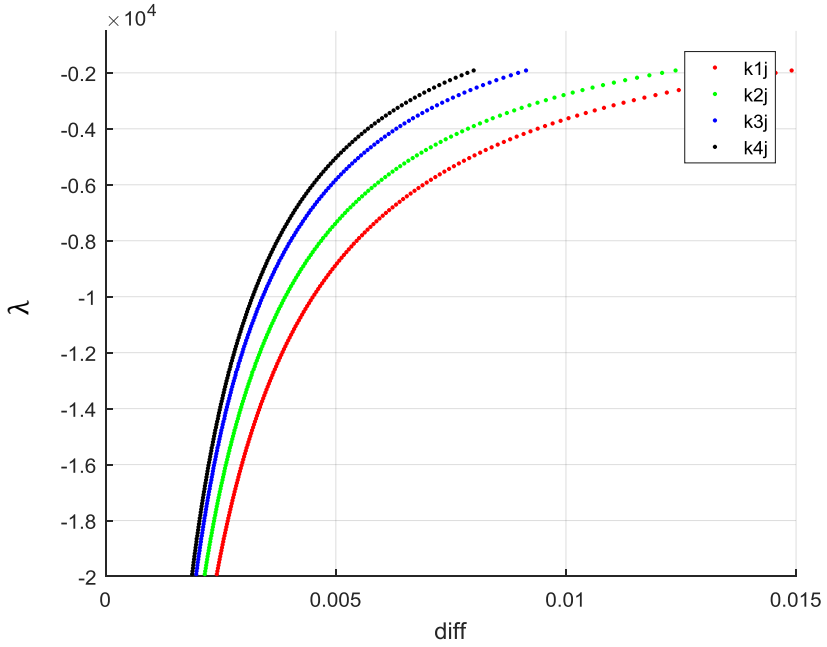

Figure 52. Solution curve 1 difference ratio for negative λ

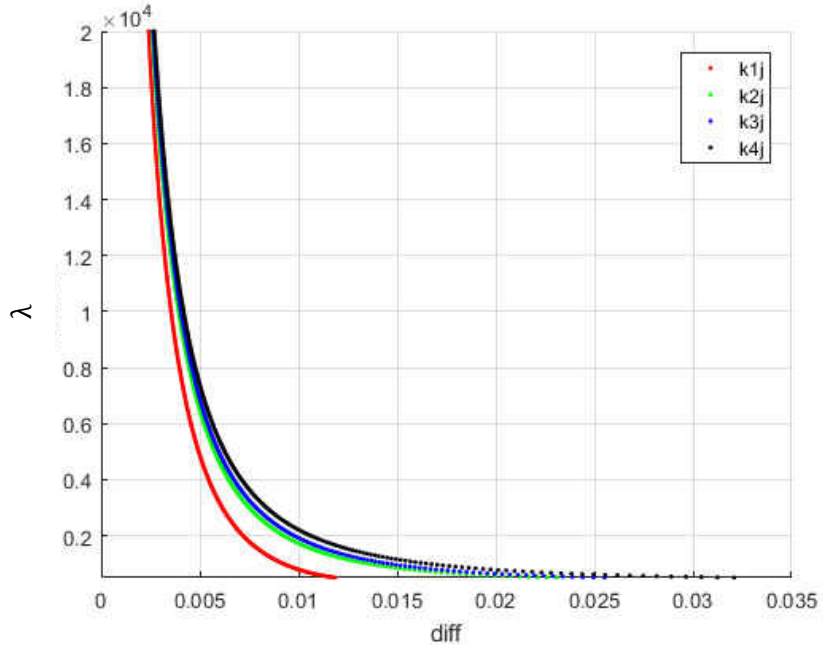

Figure 53. Solution curve 2 difference ratio for positive λ

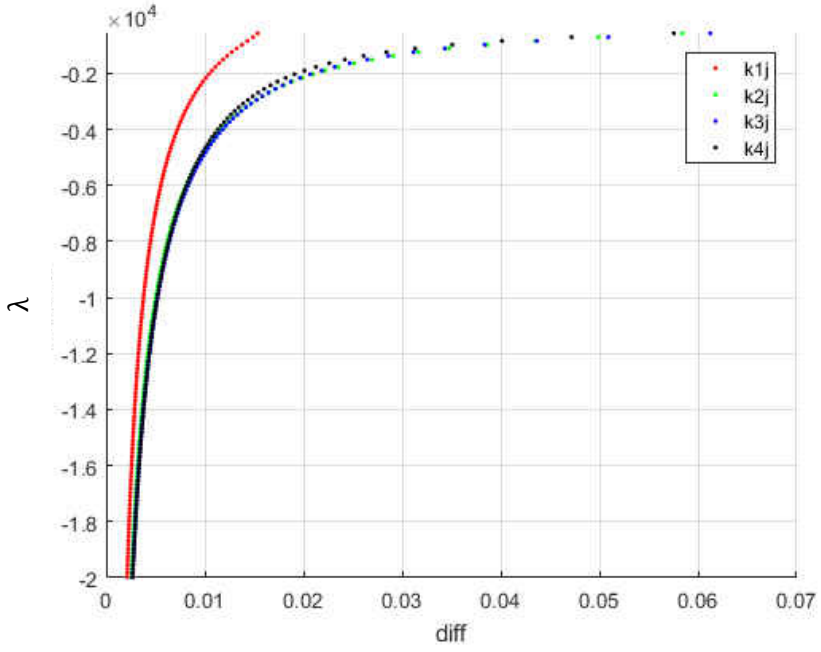

Figure 54. Solution curve 2 difference ratio for negative λ

Tendency for the difference ratio to approach zero for increasing λ is apparent on Figs. 51 through 54. The procedure was terminated when this ratio or $diff$ became less than 0.005 or half of a percent when using the specified $\Delta\lambda$ increment. Plots could be made to appear more dramatic or have a sharper turning radius near the vertical $diff = 0$ axis if the process were started closer to the $\lambda = 0$ axis where changes in slope are more significant. Although this region of the plot could be included, evaluation of changes in the Jacobian for asymptotes is intended for the more linear portions of the solution curves with large and continually increasing or decreasing λ. Decreasing the value of $\Delta\lambda$ will also have an effect on the magnitude of $diff$. Using curve 2 for example, $diff$ can be decreased by nearly an order of magnitude or 10X through a 5X reduction in the $\Delta\lambda$ sampling increment, which essentially shifts the curve left towards the $diff = 0$ axis. Best practice in evaluation of asymptotes should therefore involve selection of a $\Delta\lambda$ increment that represents a significant change in the solution and avoids approaching the limiting case of a zero increment with coincident points or zero difference. In Figs. 51 through 54, the beginning of $diff$ to decrease in a more gradual manner towards zero can be seen in the $|\lambda| < 1$ x 10^4 range. The selected $\Delta\lambda$ increment used to construct figures represents a 1% change between solution points in this region, provides for sufficient separation between points, and identifies relatively small changes in the Jacobian or the development of linear type behavior. Continued path following of the solution for increasing or decreasing λ could be performed to further strengthen the argument of linear behavior, but this would come at the expense of increased computations for additional points on the solution curve.

6.3 SPACE ORBIT

Three-dimensional relative motion between the two bodies occurs when the deputy's motion lies off of the chief orbital plane due to additional position and velocity components z and $\dot{z}$ in the LVLH reference frame. Note the deputy trajectory frequently crosses or intersects this plane momentarily as it orbits "above" and "below" the chief. In this space case, Eq. (55) represents six quadratic homogeneous polynomial measurement equations for the six unknowns (x_0, y_0, z_0) and $(\dot{x}_0, \dot{y}_0, \dot{z}_0)$. Specific orbital conditions and measurement times are documented in Ref. [68]. A single solution curve for the general orbit case is shown in Figs. 55 through 60. A total of seven roots or candidate solutions for deputy initial orbit conditions are designated using capital letters A through G on the figures. Specific values for roots and their corresponding relative specific energy values are shown in Tables 21 and 22. Similar to the planar orbit, an all zero or trivial solution exists and is rejected based on having zero energy. Using these tables, state A is chosen based on minimum energy while trivial state C and other higher energy states are rejected. While state variables x_0, y_0 and their derivatives for the nontrivial, lowest energy state appear somewhat similar for the planar and space orbits, energy for the general three-dimensional orbit is approximately an order of magnitude or ten times less than the two-dimensional case. Inclusion of the z_0 variable and its derivative is also shown to produce states of much higher energy as compared to the planar orbit.

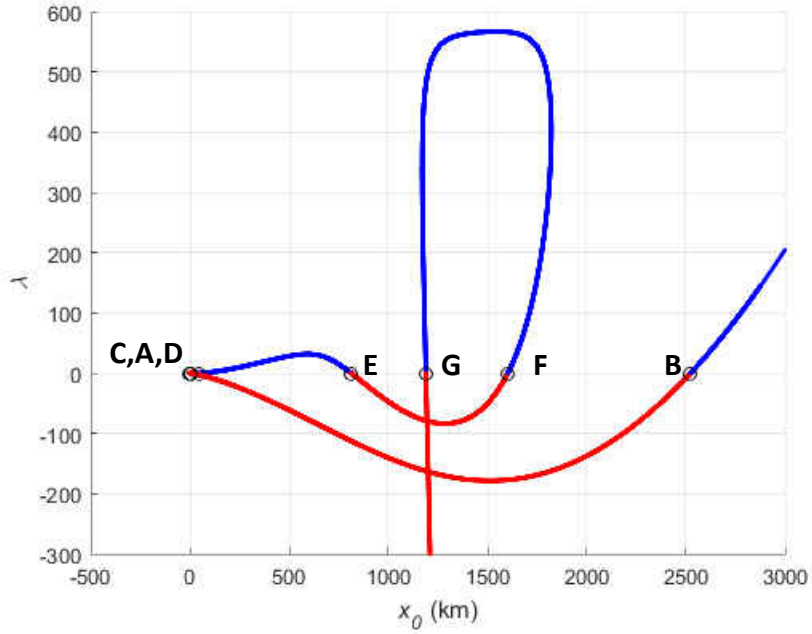

Figure 55. Solution curve for deputy x_0

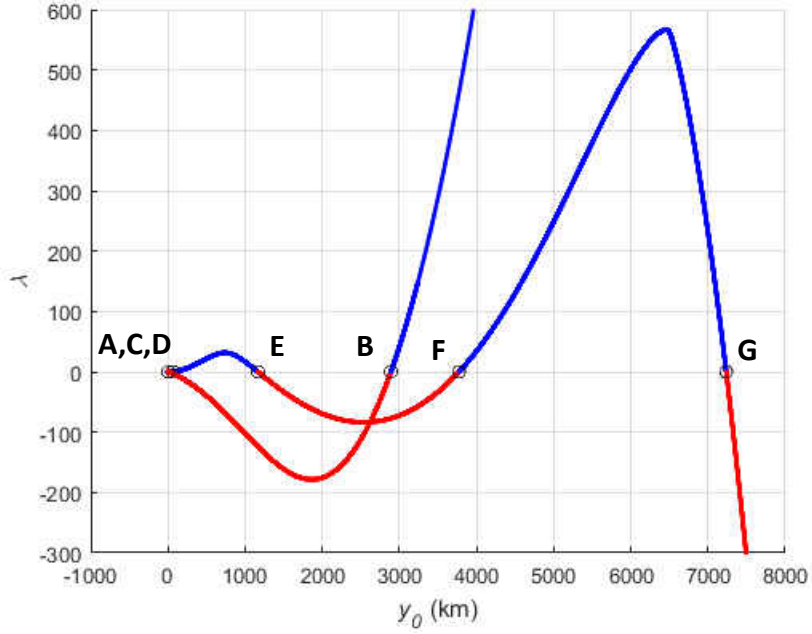

Figure 56. Solution curve for deputy y_0

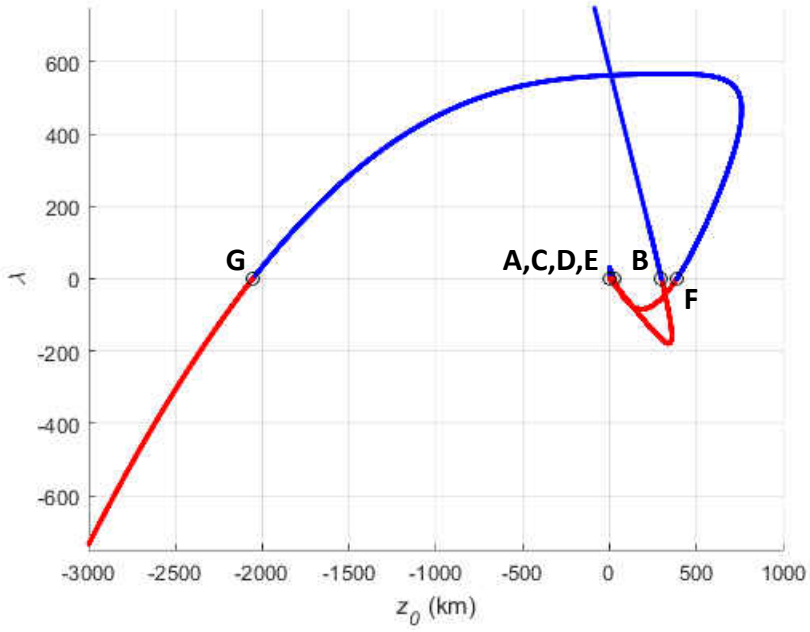

Figure 57. Solution curve for deputy z_0

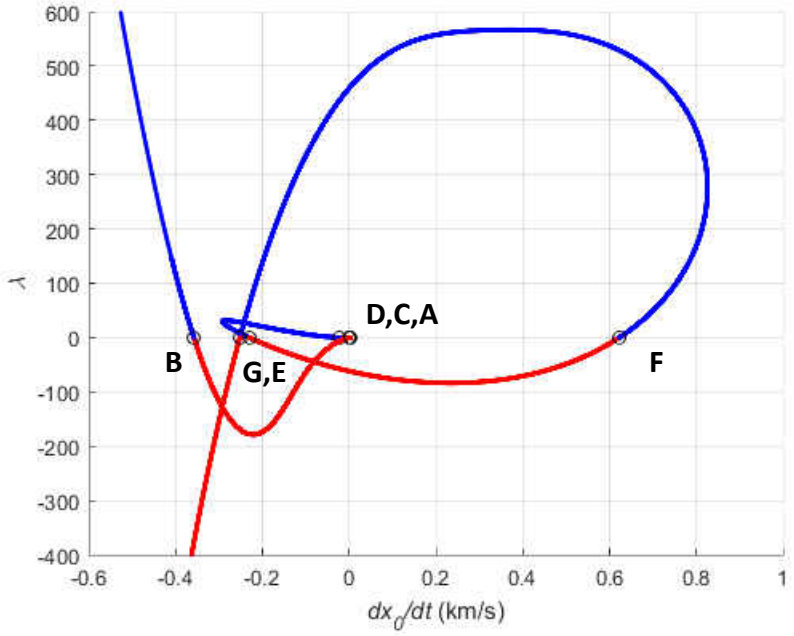

Figure 58. Solution curve for deputy $\dot{x}_0$

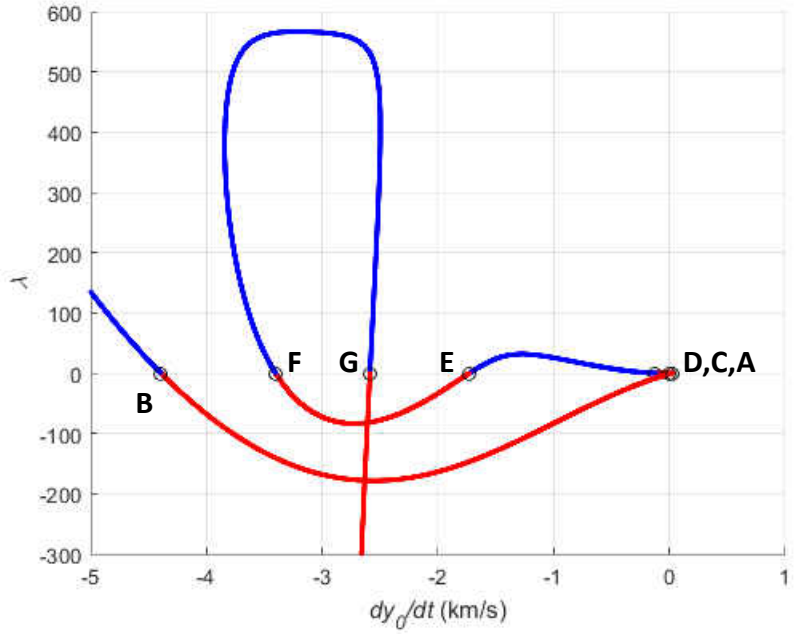

Figure 59. Solution curve for deputy $\dot{y}_0$

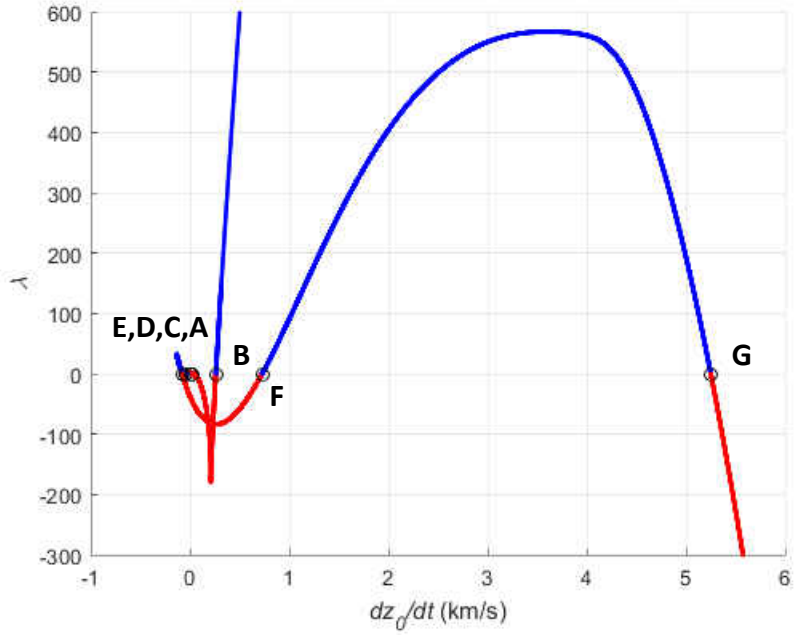

Figure 60. Solution curve for deputy $\dot{z}_0$

Table 21
Candidate states*

Variable	State A	State B	State C	State D	State E	State F	State G
$x_0(km)$	0.200	2523	0	44.79	812.4	1602	1189
$y_0(km)$	0.000	2890	0	60.04	1166	3772	7235
$z_0(km)$	0.000	299.2	0	2.853	24.23	386.7	-2055
$\dot{x}_0(km/s)$	0.002	-0.358	0	-0.023	-0.230	0.622	-0.252
$\dot{y}_0(km/s)$	0.020	-4.393	0	-0.121	-1.728	-3.404	-2.587
$\dot{z}_0(km/s)$	0.020	0.260	0	-0.037	-0.080	0.727	5.249

* Determined orbit highlighted red

Table 22
Relative specific energy (km^2/s^2)*

State A	State B	State C	State D	State E	State F	State G
0.018	43898	0	4.427	275.7	73280	2127953

* Determined orbit highlighted red

Extended plots for the solution curve are shown in Figs. 61 through 66. Values for λ exceed $\pm 10^4$ such that development of linear or asymptotic type behavior can be observed on the figures. As is the case for the planar orbit solution, asymptotes primarily appear as oblique with the exception of the x_0 and $\dot{y}_0$ solution curves which appear to approach vertical asymptotes in the $-\lambda$ direction. These asymptotes again imply existence of an infinite number of $\lambda \neq 0$ solutions to the relative motion equation set in the three-dimensional setting, but only seven $\lambda = 0$ solutions were discovered.

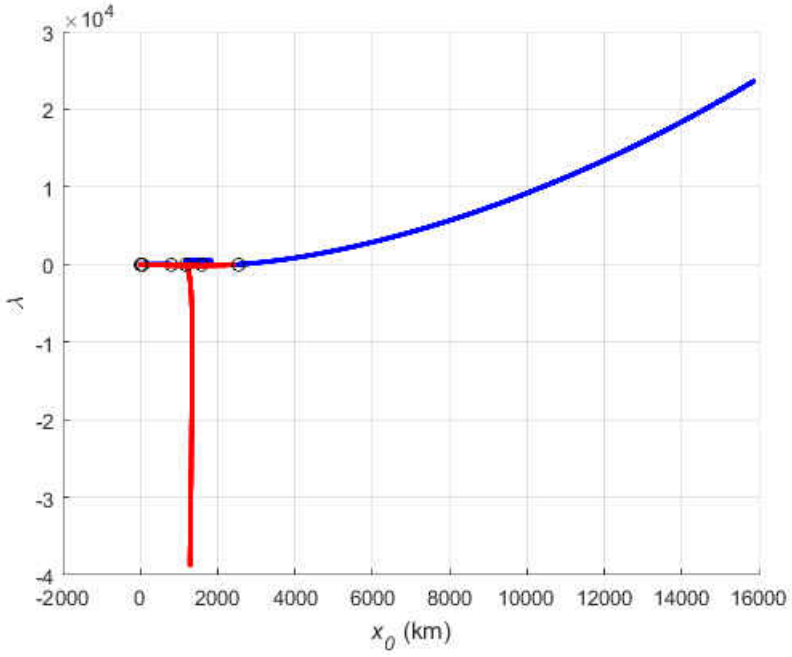

Figure 61. Extended solution curve for deputy x_0

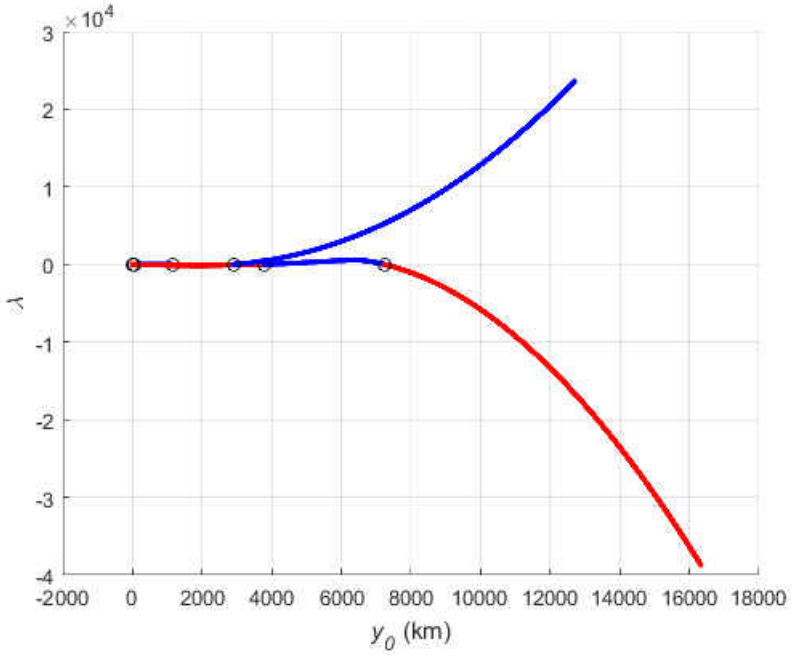

Figure 62. Extended solution curve for deputy y_0

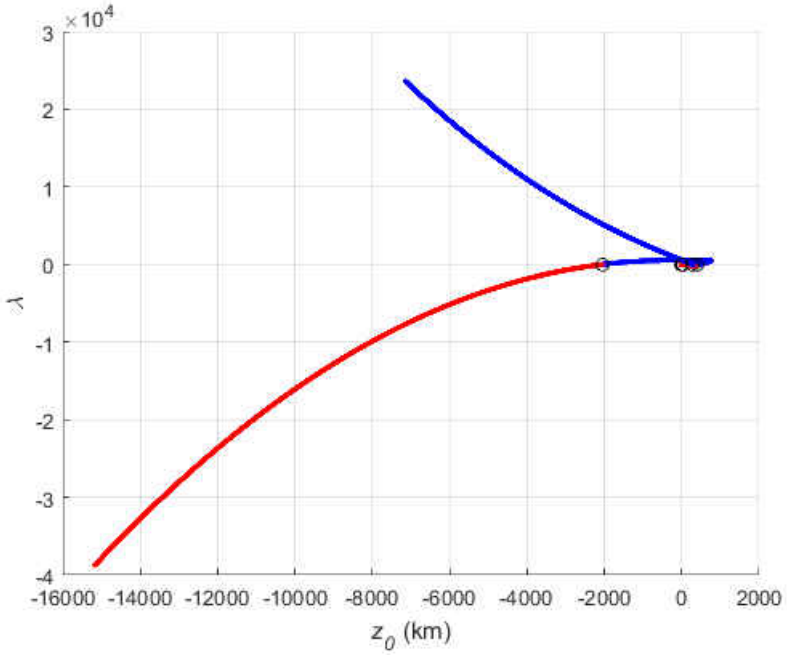

Figure 63. Extended solution curve for deputy z_0

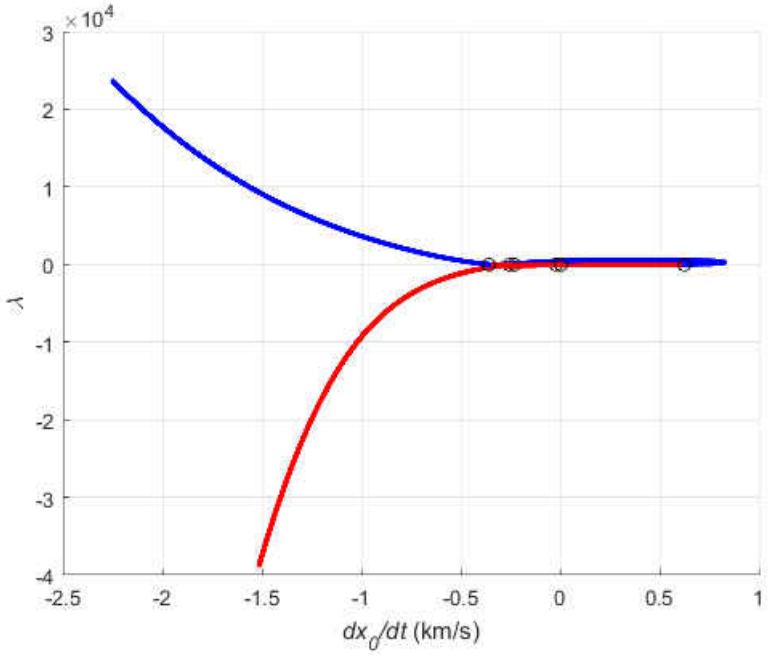

Figure 64. Extended solution curve for deputy $\dot{x}_0$

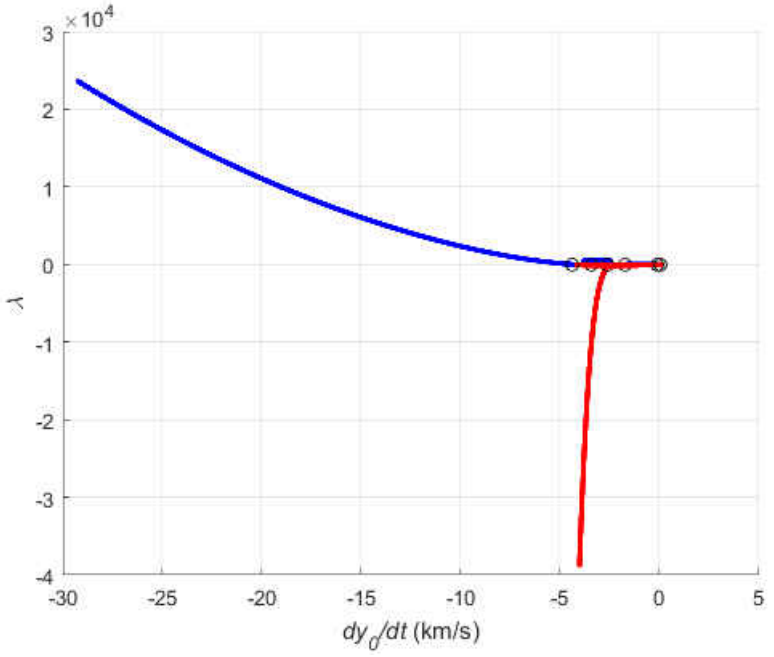

Figure 65. Extended solution curve for deputy $\dot{y}_0$

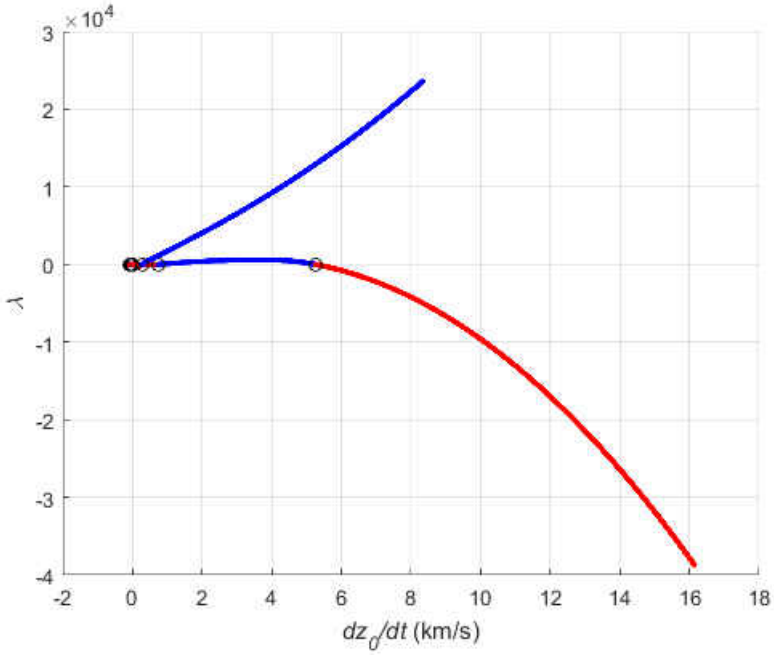

Figure 66. Extended solution curve for deputy $\dot{z}_0$

Plots of difference ratios using Eq. (57) for rows in the Jacobian matrix are shown in Figs. 67 and 68. Plots cover what appear to be asymptotic regions of the extended solution curves and were constructed using a $\Delta\lambda$ increment of approximately 100. Similar to the planar orbit case, the space orbit case shows difference ratios for all rows of the Jacobian are seen to approach zero for continuously increasing or decreasing λ.

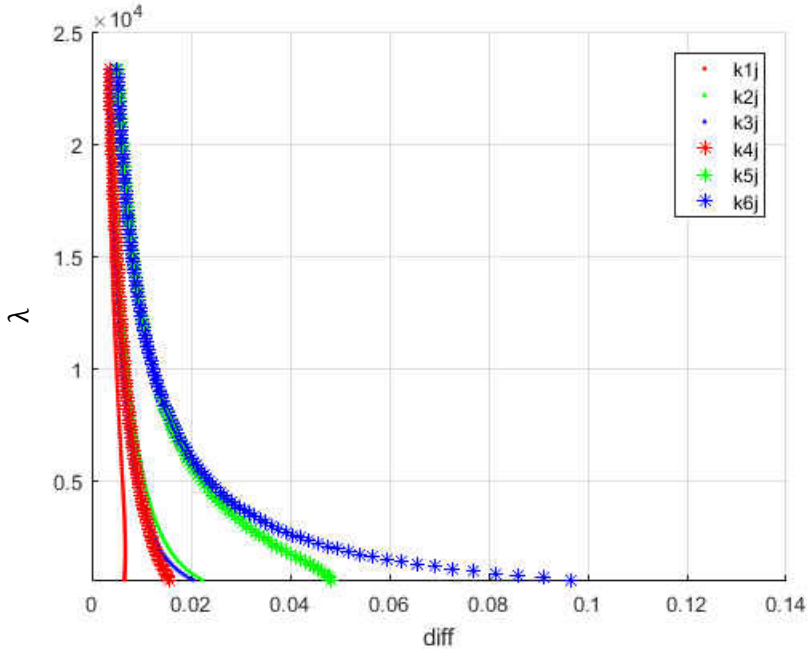

Figure 67. Difference ratio for positive λ

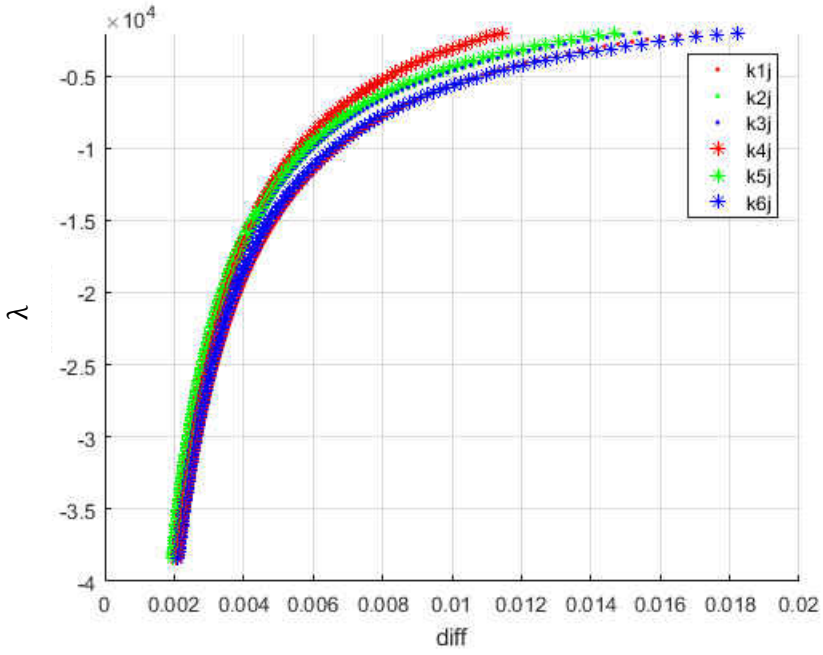

Figure 68. Difference ratio for negative λ

6.4 CONCLUSIONS

A numerical path following procedure based on the arc-length method was successfully applied to nonlinear equation sets for purpose of finding roots representing initial conditions for orbit determination. The procedure was demonstrated to be more robust as compared to searching for individual roots as two additional roots were found for the planar orbit case. The primary advantage of using arc-length solvers over other nonlinear solvers that treat the common scalar solution as known is the increased likelihood that some point or solution on a given solution curve will be found as compared to finding an individual root. Once an arbitrary solution point is found, path following of the solution curve is performed to identify the associated roots. Problems were thus transformed from searching for individual roots to searching for individual solution curves containing a finite number of roots. The path following

procedure was terminated based on a proposed difference metric using the Jacobian matrix to identify the development of asymptotic or linear type behavior.

CHAPTER 7

CONCLUSIONS AND FURTHER RESEARCH

Specific conclusions are contained in Chapters 3 through 6 to maintain consistency with previous or pending published work. In Chapter 3, arc-length solvers were found to be more robust as compared to other parametrized nonlinear solvers in terms of minimizing restarts in the event of solver failure and provided for a broader range by which to search for solutions due to the parameter being treated as unknown. In Chapter 4, arc-length solvers were used to construct static solution curves as part of a path following technique to identify the many possible equilibrium states for several mechanical systems. A procedure was proposed for identification of true equilibrium providing for a more comprehensive methodology as compared to point solution methods currently found in commercial software. In Chapter 5, a method for parallel processing of the Jacobian matrix using MATLAB was established and speedup was achieved in comparison to a serial version with underlying parallel operations on a shared memory PC. In Chapter 6, a case study was performed where path following based on the arc-length method was used to identify roots in nonlinear systems used for initial relative orbit determination.

Recommended further research is to establish simple rules for stability assessment of physical systems using eigenvalues from differential and algebraic equation sets. Such rules exist for ordinary differential equation sets as to where eigenvalues fall in a complex plane, but computational expense for conversion from DAE to ODE format can be significant. Patterns of DAE eigenvalues for stable configurations were identified in Chapter 4 making such an investigation appear plausible. Another recommendation is to develop a user friendly interface

making path following techniques for identification of equilibrium and solving of general systems of nonlinear equations more practical. This enhancement is seen as a necessary step if such a procedure were ever adopted in commercial software. Users should be able to easily plot selected variables and specify a range for the arc-length and unknown scalar parameter lambda needed to initialize the procedure. Methods for varying arc-length, stopping, reversing, or restarting the solver during path following of solution curves should also be provided.

BIBLIOGRAPHY

[1] K. J. Bathe, Finite Element Procedures, Klaus-Jürgen Bathe, 2006.

[2] R. D. Cook, D. S. Malkus, M. E. Plesha, R. J. Witt, Concepts and Applications of Finite Element Analysis, fourth ed., Wiley, 2002.

[3] R. de Borst, M. A. Crisfield, J. J. C. Remmers, C. V. Verhoosel, Non-linear Finite Element Analysis of Solids and Structures, second ed., Wiley, 2012.

[4] A. Shabana, Computational Dynamics, third ed., Wiley, 2010.

[5] A. Shabana, Dynamics of Multibody Systems, fourth ed., Cambridge University Press, 2013.

[6] O. Bauchau, Flexible Multibody Dynamics (Solid Mechanics and Its Applications), Springer, 2011.

[7] J. Nocedal, S. Wright, Numerical Optimization, second ed., Springer, 2006.

[8] C. Kelley, Iterative Methods for Linear and Nonlinear Equations, SIAM, 1995.

[9] Abaqus Analysis Users Guide, Dassault Systèmes Simulia Corp., 2014.

[10] ANSYS Mechanical APDL Theory Reference, ANSYS, Inc., 2013.

[11] MSC Nastran Nonlinear User's Guide (SOL 400), MSC Software Corporation, 2016.

[12] About ADAMS Solver, MSC Software Corporation, 2016.

[13] RecurDyn / Solver Theoretical Manual, FunctionBay, Inc., 2012.

[14] A. R. Conn, N. I. M. Gould, P. L. Toint, Trust-Region Methods, SIAM, 2000.

[15] MATLAB R2015b Documentation, The MathWorks, Inc., 2015.

[16] C. G. Broyden, The convergence of a class of double-rank minimization algorithms, J. Inst. Math. Appl. 6 (1970) 76-90.

[17] R. Fletcher, A new approach to variable metric algorithms, Comp. J. 13 (1970) 317-322.

[18] D. Goldfarb, A family of variable metric updates derived by variational means, Math. Comp. 24 (1970) 23-26.

[19] D. F. Shanno, Conditioning of quasi-Newton methods for function minimization, Math. Comp. 24 (1970) 647-656.

[20] L. Komzsik, What Every Engineer Should Know About Computational Techniques of Finite Element Analysis, first ed., CRC Press, 2005.

[21] R. B. Schnabel, P. D. Frank, Tensor methods for nonlinear equations, J. Numer. Anal., 21, (1984) 815-843.

[22] A. Bouaricha, R. B. Schnabel, Tensor methods for large sparse systems of nonlinear equations, J. Math. Progr., 82 (1998) 377-400.

[23] A. Bouaricha, Tensor-Krylov methods for large nonlinear equations, J. Comput. Optim. Appl., 5 (1996) 207-232.

[24] B. W. Bader, Tensor-Krylov methods for solving large-scale systems of nonlinear equations, in: Sandia Report SAND2004-1837, 2004.

[25] A. Krylov, Professor Krylov's Navy: Memoir of a Naval Architect, Magnet Publishing, 2014.

[26] M. Hestenes, E. Steifel, Methods of conjugate gradient for solving linear systems, J. Res. Natl. Bur. Stand., 49 (1952) 409-436.

[27] Y. Saad, M. Schultz, GMRES: A generalized minimal residual algorithm for solving nonsymmetric linear systems, J. Sci. Stat. Comp., 7 (1986) 856-869.

[28] Y. Saad, Iterative Methods for Sparse Linear Systems, second ed., SIAM, 2003.

[29] E. Riks, The application of newton's method to the problem of elastic stability, J. Appl. Mech., 39 (1972), 1060-1066.

[30] G. A. Wempner, Discrete approximations related to nonlinear theories of solids, Int. J. Solids Struct. 7 (1971) 1581-1599.

[31] M. A. Crisfield, A fast incremental / iterative solution procedure that handles snap-through, Comput. Struct. 13 (1981) 55-62.

[32] E. Ramm, Strategies for tracing the nonlinear response near limit points, in: Proceedings of the Europe-U.S. Workshop on Nonlinear Finite Element Analysis in Structural Mechanics, 1981.

[33] M. A. Crisfield, Non-linear Finite Element Analysis of Solids and Structures, Volume 1: Essentials, Wiley, 1991.

[34] M. A. Crisfield, Non-linear Finite Element Analysis of Solids and Structures, Volume 2: Advanced Topics, Wiley, 1997.

[35] Nonlinear finite element methods, course notes for ASEN 6107, http://www.colorado.edu/engineering/cas/courses.d/NFEM.d/Home.html, (accessed 03.17.2017).

[36] K. J. Bathe, E. Dvorkin, On the automatic solution of nonlinear finite element equations, Comput. Struct., 17 (1983) 871-879.

[37] E. L. Allgower, K. Georg, Numerical Continuation Methods: An Introduction, Springer Series in Computational Mathematics, Vol. 13, Springer Berlin Heidelberg, 1990.

[38] E. L. Allgower, K. Georg, Numerical Path Following, Handbook of Numerical Analysis, Vol. V, Techniques of Scientific Computing (Part 2), Elsevier Science, 1997.

[39] B. Barney, Introduction to Parallel Computing, Lawrence Livermore National Laboratory, 2016. https://computing.llnl.gov/tutorials/parallel_comp/, (accessed 02.06.17).

[40] OpenMP, http://www.openmp.org/, (accessed 03.22.2017).

[41] MPI Forum, http://mpi-forum.org/, (accessed 03.22.2017).

[42] F. Darema, SPMD Computational Model, Encyclopedia of Parallel Computing, 2011, pp. 1933-1943.

[43] A. Grama, G. Karypis, V. Kumar, A. Gupta, Introduction to Parallel Computing, second ed., Pearson, 2003.

[44] N. Matloff, Programming on Parallel Machines, University of California, Davis, 2016, http://heather.cs.ucdavis.edu/~matloff/158/PLN/ParProcBook.pdf, (accessed 02.06.17).

[45] E. Riks, An incremental approach to the solution of snapping and buckling problems, Int. J. Solids Struct., 15 (1979) 529-551.

[46] E. Riks, Some computational aspects of the stability analysis of nonlinear structures, Comp. Meth. Appl. Mech. Eng., 47 (1984) 219-259.

[47] G. Powell, J. Simons, Improved iterative strategy for nonlinear structures, Int. J. Numer. Meth. Eng., 17 (1981) 1455-1467.

[48] MATLAB Parallel Computing Toolbox Tutorial, http://www.bu.edu/tech/support/research/training-consulting/online-tutorials/matlab-pct/, (accessed 02.06.17).

[49] C. Moler, Parallel MATLAB: Multiple processors and multiple cores, MathWorks, 2007, https://www.mathworks.com/company/newsletters/articles/parallel-matlab-multiple-processors-and-multiple-cores.html, (accessed 02.06.17).

[50] MATLAB Answers, http://www.mathworks.com/matlabcentral/answers/95958-which-matlab-functions-benefit-from-multithreaded-computation, (accessed 02.06.17).

[51] G. Rose, D. Nguyen, B. Newman, Implementing an arc-length method for a robust approach in solving systems of nonlinear equations, in: IEEE Southeast Conference, 2016.

[52] L. Meirovitch, Methods of Analytical Dynamics, Dover Publications Inc., 2003.

[53] D. Negrut, A. Dyer, ADAMS/Solver Primer, MSC Software Corporation, 2004.

[54] U. M. Ascher, L. R. Petzold, Computer Methods for Ordinary Differential Equations and Differential-Algebraic Equations, SIAM, 1998.

[55] L. Meirovitch, Fundamentals of Vibrations, McGraw-Hill, 2001.

[56] M. Menrath, Stability criteria for nonlinear fully implicit differential-algebraic systems, PhD Dissertation, University in Cologne, Germany, 2011.

[57] D. Negrut, J. Ortiz, On an Approach for the Linearization of the Differential Algebraic Equations of Multibody Dynamics, in: Proceedings of the ASME/IEEE International Conference on Mechatronic and Embedded Systems and Applications, 2005.

[58] G. Rose, D. Nguyen, B. Newman, Parallel Computation of the Jacobian Matrix for Nonlinear Equation Solvers Using MATLAB, NASA/TM-2017-219655, 2017.

[59] J. Ortiz, Introduction to Adams/Solver C++, Charts from the 2011 Adams user meeting, Munich, Germany, 2011.

[60] W. Press, S. Teukolsky, W. Vetterling, B. Flannery, Numerical Recipes in Fortran 77: The Art of Scientific Computing, second ed., Press Syndicate of the University of Cambridge, 1997.

[61] R. Neidinger, Introduction to automatic differentiation and MATLAB object-oriented programming, SIAM Review, Vol. 52, No. 3, (2010) 545-563.

[62] R. Bartlett, D. Gay, E. Phipps, Automatic differentiation of C++ codes for large-scale scientific computing, in: International Conference on Computational Science, 2006, pp. 525-532.

[63] ADIFOR, http://www.anl.gov/technology/project/adifor-automatic-differentiation-fortran-77, (accessed 02.06.17).

[64] J. Gilbert, C. Moler, R. Schreiber, Sparse matrices in MATLAB: Design and implementation, J. Matrix Anal. Appl. 13 (1992) 333-356.

[65] P.R. Escobal, Methods of Orbit Determination, John Wiley & Sons, 1965.

[66] B. Newman, T.A. Lovell, E. Pratt, Second order nonlinear initial orbit determination for relative motion using Volterra theory, Adv. Astronaut. Sci., 152 (2014) 1253-1272.

[67] M.T. Stringer, B. Newman, T.A. Lovell, A. Omran, Analysis of a new nonlinear solution of relative orbital motion, Proceedings of the 23rd International Symposium on Space Flight Dynamics, 2012.

[68] B. Newman, T.A. Lovell, E. Pratt, E. Duncan, Quadratic hexa-dimensional solution for relative orbit determination - revisited, Adv. Astronaut. Sci., 155 (2015) 3359-3376.

APPENDIX A

MATLAB CODE FOR NEWTON-RAPHSON METHOD

```
function [u] = newton(u_0,lambda) % Newton-Raphson method

u_i   = u_0;                      % initial guess
conv  = 0;                        % convergence criteria, 1 = yes, 0 = no
iter  = 0;                        % starting iteration count

    while conv == 0               % iteration starts to determine new u

        [K_i,fu_i,F]   = sys_eq(u_i);     % system data for displacement u_i
        R_i            = lambda*F - fu_i; % imbalance at state (i)
        du_i           = (K_i)\R_i;       % du_i = inv(K_i)*R_i;

        if norm(du_i)/norm(u_i) < 1.0e-8
            conv = 1;             % converged
        elseif iter > 50          % set iteration limit
            break;
        else
            u_i  = u_i + du_i;
            iter = iter + 1;
        end

    end

if conv == 1                      % converged
    u = u_i;                      % new equilibrium displacement
else
    [row,col] = size(u_i);
    u = NaN*ones(row,col);        % NaN for convergence failure
end
```

APPENDIX B

MATLAB CODE FOR ARC-LENGTH METHODS

```
function [u,lambda] = arclength(arcL,u_0,lambda_0,method)

% Arc-length method
% arcL     - user specified arc-length
% u_0      - guessed or initial state
% lambda_0 - guessed or initial parameter

[K_0,~,F] = sys_eq(u_0);  % output system data for point u_0

[row,col]    = size(u_0);

%% find starting iteration point at end of arc-length

detK_0       = det(K_0);                % matrix determinant
del_lambda_g = sign(detK_0);            % +/- slope for arc-length
du_g         = K_0\(del_lambda_g*F);    % du_g = inv(K_0)*del_lambda_g*F;
arcL_g       = sqrt(del_lambda_g^2 + du_g'*du_g);
del_lambda_0 = (arcL/arcL_g)*del_lambda_g;
du_0         = (arcL/arcL_g)*du_g;
lambda_i     = lambda_0 + del_lambda_0; % lambda at end of arc-length
u_i          = u_0 + du_0;              % u at end of arc-length
r_i          = [du_0; del_lambda_0];    % store vector defining arc-length
L_squared    = r_i'*r_i;                % arc-length magnitude

%% begin iterations to determine equilibrium point

conv = 0;                               % convergence criteria
iter = 1;                               % starting iteration count

while conv == 0

    r_p          = r_i;                 % previous vector for arc-length
    [K_i,fu_i,F] = sys_eq(u_i);         % output system data for point u_i
    R_i          = lambda_i*F - fu_i;   % imbalance at state (i)
    du_I         = K_i\F;               % du_I  = inv(K_i)*F;
    du_II        = K_i\R_i;             % du_II = inv(K_i)*R_i;

% %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

if method == 1                  % spherical iteration path

    c(1)         = 1 + du_I'*du_I;
    c(2)         = 2*(r_p(row + 1) + r_p(1:row)'*du_I + du_I'*du_II);
    c(3)         = 2*r_p(1:row)'*du_II + du_II'*du_II;
    del_lambda_t = roots(c);
```

```
    if isreal(del_lambda_t) == 0
        del_lambda_t = real(del_lambda_t); % use with caution
        fprintf('Complex roots for iteration %g\n',iter)
    end
    du_t1        = du_II + del_lambda_t(1)*du_I;
    du_t2        = du_II + del_lambda_t(2)*du_I;
    r_t1         = r_p + [du_t1; del_lambda_t(1)];
    r_t2         = r_p + [du_t2; del_lambda_t(2)];
    cos_theta1   = (r_p'*r_t1)/L_squared;
    cos_theta2   = (r_p'*r_t2)/L_squared;

    if cos_theta1 > cos_theta2
        dui         = du_t1;
        del_lambdai = del_lambda_t(1);
    else
        dui         = du_t2;
        del_lambdai = del_lambda_t(2);
    end

    du_i         = dui;
    del_lambda_i = del_lambdai;
    r_i          = r_p + [du_i; del_lambda_i]; % updated vector

elseif method == 2              % normal iteration path

    del_lambda_i = -(du_0'*du_II)/(du_0'*du_I + del_lambda_0);
    du_i         = du_I * del_lambda_i + du_II;

end

% %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

    if norm(du_i)/norm(u_i) < 1.0e-6 || abs(del_lambda_i)/abs(lambda_i) <
1.0e-6
        conv     = 1;           % converged
    elseif iter > 100
        break;                  % exit loop for specified iteration limit
    else
        u_i      = u_i + du_i;
        lambda_i = lambda_i + del_lambda_i;
        iter     = iter + 1;
    end

end

if conv == 1                    % converged
    u      = u_i;               % new equilibrium displacement
    lambda = lambda_i;          % new equilibrium load factor
    %R_i
else
    u      = NaN*ones(row,col); % NaN for convergence failure
    lambda = NaN;
end
```

APPENDIX C

MATLAB CODE FOR SOLVING A NONLINEAR SYSTEM

```
% Sample code for solving nonlinear system of equations
% USER INPUT: 3DOF example

dlambda = 1;          % search increment for RHS solution
u       = [2;-2;2];   % initial guess or known state
lambda  = 0;          % initial guess or known RHS solution
N       = 5;          % search limit

figure(1); hold on; xlabel('u1'); ylabel('lambda');
figure(2); hold on; xlabel('u2'); ylabel('lambda');
figure(3); hold on; xlabel('u3'); ylabel('lambda');

%% Implement Newton-Raphson method

for i = 1:N
    lambda = lambda + dlambda;      % next value for parameter
    u_0 = u;                        % initial guess for displacement
    [u] = newton(u_0, lambda);
    if isnan(u) == 1
        disp('Convergence failure, switching to arc-length')
        break;
    elseif isreal(u) == 0
        disp('Complex root, switching to arc-length')
        break;
    end
    figure(1); plot(u(1),lambda,'ro'); % plot point
    figure(2); plot(u(2),lambda,'ro'); % plot point
    figure(3); plot(u(3),lambda,'ro'); % plot point
end

u = u_0;                          % final state obtained from NR method
lambda = lambda - dlambda;        % final RHS solution from NR method

%% Implement the arc-length method
method = 1;           % 1 - sphere, 2 - plane
arcL   = 0.5;         % specified arc-length to follow solution curve
N      = 200;

if method == 1
    disp('Perform arclength on sphere')
elseif method == 2
    disp('Perform arc-length on normal plane')
end

u_0 = u;              % guess based on previous state
lambda_0 = lambda;    % guess based on previous RHS
[u,lambda] = arclength(arcL,u_0,lambda_0,method);
%arcL = -arcL;        % uncomment and re-run to follow opposite direction
```

```
iter = 0;
for i = 1:N
    iter = iter + 1;
    u_0 = u;              % guess based on previous state
    lambda_0 = lambda;    % guess based on previous RHS
    [u,lambda] = arclength(arcL,u_0,lambda_0,method);
    if sign(lambda) ~= sign(lambda_0)
        u_zero = newton(u_0, 0) % perform NR at zero-crossings
    end
    figure(1); plot(u(1),lambda,'ro'); % plot point
    figure(2); plot(u(2),lambda,'ro'); % plot point
    figure(3); plot(u(3),lambda,'ro'); % plot point
end
```

APPENDIX D

MATLAB CODE FOR DEFINING A NONLINEAR SYSTEM

```
function [K,fu,F] = sys_eq(u) % Define system of nonlinear equations
% System can be general or based on finite element model. K is tangent
% stiffness or Jacobian matrix. This is the slope of tangent hyperplane at
% displacement u. fu is system value at state u. F is a reference vector
% for scalar lambda.

% 3DOF example

K = zeros(3,3);

K(1,1) = 2*u(1)*(u(2)^3)*u(3);
K(1,2) = 3*(u(2)^2)*(u(1)^2)*u(3) + 4;
K(1,3) = 0.5*(u(3)^-0.5) + (u(1)^2)*(u(2)^3) - (u(3)^-2);
K(2,1) = (u(3)^3) + 3;
K(2,2) = 0.5*((-u(3)*u(2))^-0.5)*(-u(3)) + 2*(u(2)^-3);
K(2,3) = 0.5*((-u(3)*u(2))^-0.5)*(-u(2)) + 3*(u(3)^2)*u(1);
K(3,1) = u(2)*u(3) + 2*u(1)*u(3) - 3*u(2);
K(3,2) = u(1)*u(3) + 2*u(2)*u(3) -3*u(1);
K(3,3) = u(1)*u(2) + (u(2)^2) + (u(1)^2);

fu = zeros(3,1);

fu(1) = (u(3)^0.5) + (u(1)^2)*(u(2)^3)*u(3) + (u(3)^-1) + 4*u(2) + 17.75;
fu(2) = ((-u(3)*u(2))^0.5) + (u(3)^3)*u(1) - (u(2)^-2) + 3*u(1) - 135;
fu(3) = u(1)*u(2)*u(3) + (u(2)^2)*u(3) + (u(1)^2)*u(3) - 3*u(1)*u(2) -18;

F = [1;1;1];
```

APPENDIX E

MATLAB CODE FOR PARALLEL JACOBIAN COMPUTATION

```
function J = par_jacobi_a(u_i,NP)

N = size(u_i,1);
C = ceil(N/NP);
index = ones(1,2*NP);
index(end) = N;
j = 2;

for i =1:NP-1
    index(j) = i*C;
    index(j+1) = index(j) + 1;
    if index(j) > N
        fprintf('invlid matrix partition of %g for NP = %g\n',index(j),NP)
        return
    elseif index(j+1) > N
        fprintf('invlid matrix partition of %g for NP = %g\n',index(j+1),NP)
        return
    end
    j = j + 2;
end
del = 1e-6;

spmd
    J = zeros(N,C); % distribute partitions across labs
    uperturb = u_i;
    fu_i = sys_eq(u_i);

    for j = 1:NP
        if labindex == j
            for i = index(j*2-1):index(j*2)
                uperturb(i) = uperturb(i) + del;
                fu_p = sys_eq(uperturb);
                if j == 1
                    J(:,i) = ( fu_p - fu_i )/del;
                else
                    k = 2*labindex - 2;
                    J(:,i - index(k)) = ( fu_p - fu_i )/del;
                end
                uperturb(i) = u_i(i); %uperturb(i) - del;
            end
        end
    end
end

J = [J{1:NP}]; % convert from composite to double
```

APPENDIX F

COPYRIGHTS

VITA

Geoffrey Kenneth Rose has spent most of his life living in the Hampton Roads region of Virginia. He began his career as a mechanical engineer with the Department of Navy in 1999, became licensed as a Professional Engineer in 2004, and completed a Masters of Engineering degree in 2005. He transferred to NASA Langley Research Center in 2008 and was accepted into a Ph.D. program in the Mechanical and Aerospace Engineering Department at Old Dominion University in 2010. The address of the department is 238 Kaufman Hall, Norfolk, VA 23529. Most of his career at NASA has been spent working on the design and analysis of deployable structures for space.