A UNIFIED LIBRARY OF NONLINEAR SOLUTION SCHEMES: AN EXCURSION INTO NONLINEAR COMPUTATIONAL MECHANICS

BY

SOFIA LEON

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Civil Engineering
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2010

Urbana, Illinois

Advisor:

Professor Glaucio H. Paulino

Abstract

A library of nonlinear solution schemes including load, displacement, arc-length, work, generalized displacement, and orthogonal residual control are cast into a unified framework for solving nonlinear finite element systems. Each of these solution schemes differs in the use of a constraint equation for the incremental-iterative procedure. The governing finite element equations and constraint equation for each solution scheme are combined into a single matrix equation, which characterizes the unified approach. This theoretical model leads naturally to an effective object-oriented implementation and potential for integration into a finite element analysis code. Using this framework, the strengths and weaknesses of the various solution schemes are examined through several numerical examples.

To my family

Acknowledgments

First and foremost I would like to express my gratitude to my advisor, Professor Glaucio H. Paulino, without whom this work would not have been possible. His enthusiasm, support and guidance have had a profound impact on my intellectual development and academic career.

This thesis would not have been possible without the contribution of Dr. Ivan Menezes and Dr. Anderson Pereira who prepared the latest version of the code for the Unified Library of Nonlinear Solution Schemes. Special thanks are due to Dr. Ivan Menezes whose guidance and suggestions were instrumental in this work.

I also thank my friends and colleagues: Professor Eshan Dave, Dr. Kyoungsoo Park, Dr. Tam Nguyen, Rodrigo Espinha, Cameron Talischi, Lauren Stromberg, Arun Lal Gain, Tomas Zegard, Daniel Spring, and Junho Chun for their help and support throughout my graduate student career. I would also like to thank Wylie Ahmed; words cannot express my gratitude for his continued support and encouragement.

I am grateful for the financial support of the Graduate Research Fellowship provided by the National Science Foundation and of the Structural Engineering Fellowship provided by the Civil and Environmental Engineering Department at the University of Illinois at Urbana-Champaign.

Finally, I wish to thank my parents, Wendy Prothro and Fritz Leon, and my siblings Jake and Carly Leon. They are a constant source of inspiration for me and it is with great pleasure that I dedicate this thesis to them.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

Nonlinear problems are prevalent in structural and continuum mechanics, and there is high demand for computation tools to solve these problems. Many methods and algorithms have been developed to solve such problems over the past forty years. Despite these efforts, no single algorithm is capable of solving any and all nonlinear problems; depending on the system and the degree of nonlinearity, one solution scheme may be preferred over another. In this Chapter a brief review of nonlinear systems is presented including sources of nonlinearity in structural systems and typical characteristics of nonlinear behavior. Next is a discussion of existing nonlinear solution schemes, some of which are explored in more detail later in this thesis and others are provided as a reference for the reader. Afterwards is a motivation for the Unified Library of Nonlinear Solvers. Finally, the organization of this thesis is outlined.

1.1 Nonlinear systems

Nonlinear behavior can arise from either material or geometric nonlinearity. In the former, the constitutive relation describing the material is itself nonlinear and the structural response associated with physical phenomena such as plasticity or strain-softening must be captured. In the latter, nonlinearity is due to changes in geometry, arising from large strains and/or rotations, which enter the formulation from a nonlinear strain-displacement relationship, and may occur even if the constitutive relation in linear [76]. Furthermore, in geometric nonlinearity the applied loads will either have an effect on the deformed configuration, or the configuration will have an effect on the load, (e.g. follower loads [100]).

Nonlinear behavior in structural systems is seen, for example, in the load versus displacement curve. A linear system implies that the load is linearly related to the displacement, and clearly this is not the case for nonlinear systems. The load versus displacement curve is also referred to as the solution or equilibrium path, as it is comprised of points which satisfy equilibrium conditions throughout the loading history.

Nonlinear problems arising from either geometric or material nonlinearity feature critical points along the solution path. Critical points or stability points, shown in Figure 1.1, are points on the solution path where the structure loses stability (e.g. buckling) or where

bifurcation occurs (i.e. solution switches to two or more branches). Load limit points occur when a local maximum or minimum load is reached on the load versus displacement curve, as shown at points A and D in Figure 1.1. A horizontal tangent is present at load limit points. Displacement limit points, shown at points B and C in Figure 1.1, occur at vertical tangents on the solution curve. Displacement limit points are also commonly referred to as snap-back points or turning points in the literature. Methods capable of passing displacement limit points are said to capture snap-back behavior.

Other important characteristics of an equilibrium path are stiffening and softening, loading and unloading, and stable and unstable regions. Softening occurs from points O to A in Figure 1.1, while stiffening occurs beyond point D. Loading and unloading is straightforward and corresponds to an increase or decrease in the load, respectively. In Figure 1.1, loading occurs from points O to A and again from D to E, while unloading occurs from points A to D.

Stability is directly related to load limit points, as shown in Figure 1.1, where the region between the load limit points is unstable. The unstable region following a load limit point corresponds to a physical instability in the structural system, such as buckling. Using the theory of stability in a conservative system, a critical point occurs when the stiffness matrix in singular [80]. One class of methods (load control methods, discussed in Section 1.2.1.1) for tracing nonlinear load versus displacement curves is not capable of capturing behavior beyond a load limit point, and instead these methods yield snap-through behavior. As shown in Figure 1.2, the dashed part of the equilibrium path is not traced; the curve snaps through and only the portions with increasing loads are captured. Hence, these methods miss unstable regions of the equilibrium curve (dashed line in Figure 1.2).

Tracing an equilibrium path beyond the simple linear region and into a nonlinear region is a complicated task in structural analysis. In fact, in many cases, it may seem unnecessary to trace a path beyond the first load limit point. However, the full equilibrium path, including critical points and regions of instability, gives more information about the structural behavior than a simpler analysis [27]. Once a structure passes a load limit point, the nature of the unloading may be of importance to the analyst, rather than just the loading behavior. Additionally, information about the structural response past a displacement limit point may be of importance. For instance, if snap-back behavior was not captured in Figure 1.1, the structure would appear to have a sharp drop in the load at point B and the nature of the unloading would be lost.

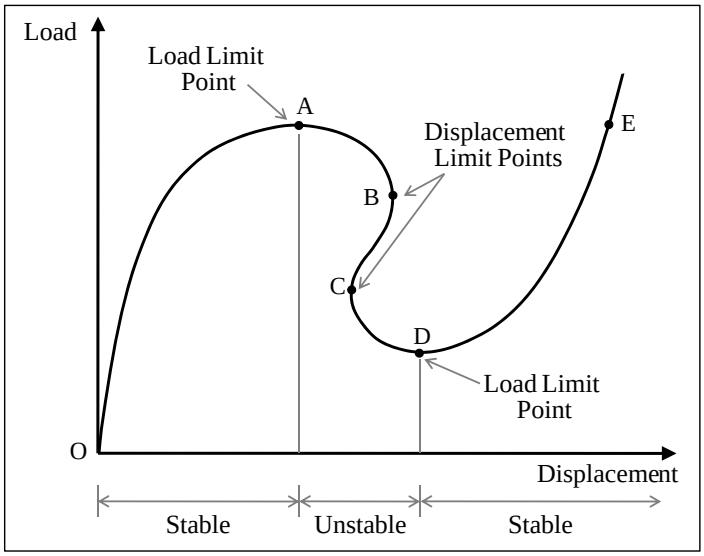

Figure 1.1: Limit points in nonlinear equilibrium paths

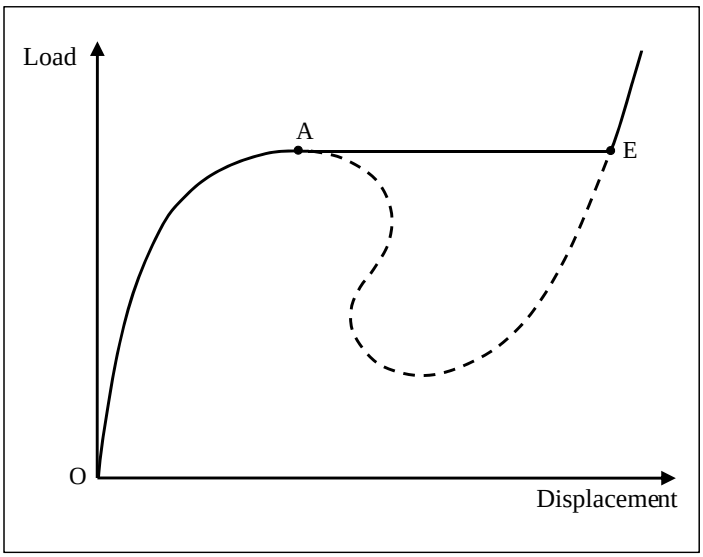

Figure 1.2: Snap through behavior

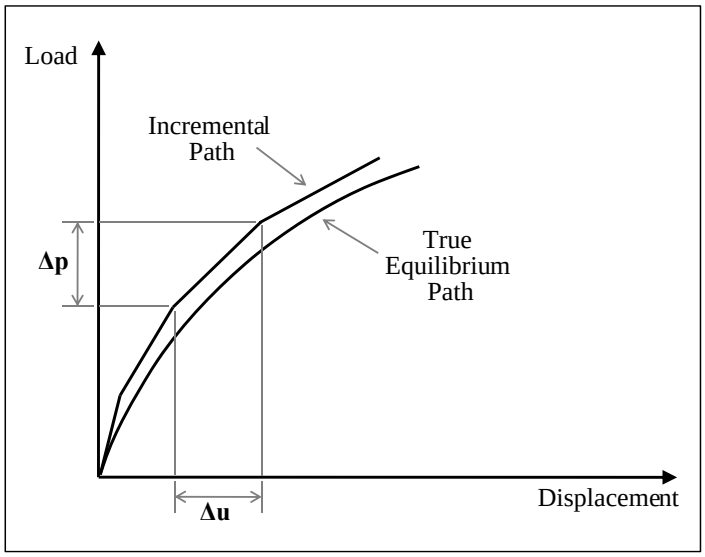

Figure 1.3: Purely incremental procedure

1.2 Review of nonlinear solution schemes

A nonlinear equilibrium path can be traced by means of either a purely incremental procedure or an incremental-iterative procedure. In an incremental procedure, shown in Figure 1.3, the load is applied at relatively small load steps and the structure is assumed to respond linearly within each step. This method is simple to implement and is computationally inexpensive, however as the solution progresses, it diverges considerably from the actual equilibrium path because equilibrium is not guaranteed at every step [64, 95]. Conversely, in an incremental-iterative procedure, a series of iterations or corrections are performed at each incremental step until a specified convergence criterion is satisfied, Figure 1.4. If convergence, which is typically a tolerance on the unbalanced forces, is achieved before a maximum number of iterations is reached then equilibrium is satisfied for that step. The incremental-iterative approach is accurate, but it comes with a higher computational cost. Incremental-iterative procedures are used for highly nonlinear behavior, as purely incremental methods are inadequate. The incremental-iterative scheme will be discussed in further detail in Section 2.1.

Within the context of incremental-iterative procedures, the stiffness of the structure can either follow a standard or modified update. The standard method, also referred to as the Standard Newton-Raphson method, is to update the stiffness matrix at each iteration, Figure 1.4(a). In the modified method, the stiffness is calculated once at the beginning of the incremental step and held constant for subsequent iterations, Figure 1.4(b). The modified method requires fewer computations, but convergence is generally slower than with the standard method [64]. For an extended discussion on nonlinear systems and overview of

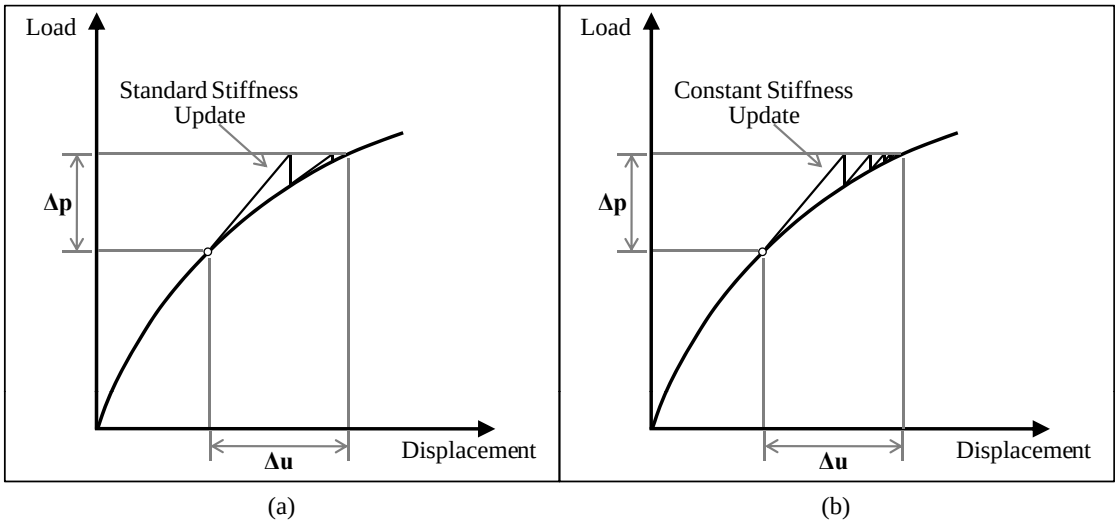

Figure 1.4: Incremental-iterative procedures with (a) standard and (b) constant stiffness updates

nonlinear solution schemes, the reader is directed to following text books [14, 9, 18, 29, 30, 76]

1.2.1 Development of nonlinear solution schemes

Six nonlinear solution schemes are examined in this work: load control method, displacement control method, work control method, arc-length control method, generalized displacement control method, and the orthogonal residual procedure. The development of these methods will be briefly discussed in the next subsection, followed by a review of other nonlinear solution schemes not implemented in this work. Detailed formulations of the six aforementioned schemes will be presented in the next chapters.

1.2.1.1 Supported methods

Common and widely used methods for solving nonlinear systems of equations in structural mechanics are load control, or Newton-Raphson, type algorithms, and displacement control algorithms. Both methods are effective for solving problems with limited nonlinearity, but break down at load and displacement limit points, respectively. A detailed discussion of the formulation and implementation of these methods into the $(N+1)$ space is given in Section 2.3. Many advances in nonlinear solvers consist of variations of these basic methods. Powell and Simons [74] extended the load control method to capture load and displacement limit points by decomposing the displacement vector into separate increments each weighted by various criteria, thus making the method flexible for use with different types of nonlinear problems. Simons and Bergan [89] and Fujii et al. [43] developed extensions of the traditional

displacement control method capable of capturing snap-back behavior, which was a weakness of the original formulation.

Further advances of nonlinear solution schemes lead to arc-length type methods where both the load and displacement are incremented simultaneously, thus allowing for recovery of both load and displacement limit points. Extensive review on the development and progress of this method is available in the literature, so only a brief overview will be given here. Wempner [92] and Riks [79] applied the method to geometrically nonlinear structures. The incremental load is varied through an additional constraint equation that describes the surface over which the next point on the equilibrium path lies. This additional constraint equation destroys the symmetry and increases the bandwidth of the original system. Subsequent modifications by Crisfield [27] used the decomposition proposed by Batoz and Dhatt [11] to preserve the symmetry and bandwidth of the original system, thus making the method appropriate for standard finite element codes.

The arc-length method is generally successful at capturing complex nonlinear behavior, however, weakness have been identified and subsequent improvements have been made by several authors [81, 89, 56]. Carrera [20] documented the failure of several versions of the arc-length method related to factors including the constraint equation, linearization, and computer precision. Some improvements and extensions of the arc-length methods are described in Section 2.3, and a few others are briefly reviewed here. Forde and Stiemer [42] identified orthogonality relationships among various versions of the arc-length method and generalized the formulation in terms of them. Al-Rasby [1] developed an arc-length method using diagonal scaling matrices to remove inconsistencies associated with mixed units (i.e. displacements, rotations, forces and moments). Mallardo and Alessandri [62] improved the nonlinear capabilities of the Boundary Element Method (BEM) from only load or displacement type control to arc-length control, thus allowing for complex nonlinear problems to be solved by the BEM. Similarly, Mukherjee and Chandra [66] and Chandra and Mukherjee [22, 23] presented boundary element formulations for large strain deformations of plasticity and viscoplasticity, thus incorporating both geometric and material nonlinearity. Paulino and Liu [73] also use a BEM formulation to model nonlinear materials with applications to nonlinear fracture mechanics and J integral calculations. Meshless methods, based on boundary integral equations, have also been used to analyze nonlinear elastoplastic problems [63]. Ritto-Corrêa and Camotim [82] developed techniques to use in conjunction with the arc-length method to ensure (i) the proper root of the quadratic equation resulting in the load parameter is chosen and (ii) the detection of convergence to wrong solutions.

A new method, called the work control method, was developed in the mid 1980's to alleviate the issue of inconsistent physical units associated with the arc-length method. The method, which can be found in the papers by Yang and McGuire [96] and Bathe and Dvorkin [10],

will be discussed in detail in Section 2.3. Improvements were made to this method as it suffered difficulties near displacement limit points for certain problems. Lin et al. [59] developed a method to address the inconsistent units in the arc-length method and weakness at snap-back of the work-control method using the work weighted state vector to control the incremental length throughout the solution tracing process. Chen and Blandford [24] developed a work-increment control algorithm, which was an improvement over other work increment type algorithms in that it is quadratically convergent. A stabilized form of the work control method was presented by Kouhia [52] who reformulated the load parameter to be well defined even in areas of snap-back.

The generalized displacement control method by Yang and Sheih [97] overcomes the problems previously mentioned for other algorithms. The formulation and justification of the method are presented in Section 2.3. The method has been used with success by the authors when studying large deflection of trusses [99, 91], and ultimate load carrying capacity of structures considering both member and structure instability [58], to name a few areas of application.

Krenk [53] presented algorithm called the orthogonal residual procedure which uses an orthogonality condition to determine to optimal sign and direction of the next load step. The method was later modified [54], then presented in a stabilized form [52]. While the method is not as widely used as other methods (i.e arc-length type methods), it has been recently been utilized by authors to solve nonlinear equations associated with the extended finite element method (FEM) applied to cohesive crack growth [2, 3, 65]. A detailed discussion and several modifications and improvements will be discussed in Chapter 3.

1.2.1.2 Other methods

In the area of nonlinear solution schemes, most methods are extensions and improvements to those already established, the most common being traditional load control methods and more powerful arc-length methods. However, other families of methods exist which are not directly related to those discussed above. For instance, accurate and quick converging iterative solvers have been presented by Golbabai and Javidi [45] and Golbabai and Javidi [45] using the homotopy perturbation method by He [46]. Additional quick converging methods have been shown to achieve cubic and super cubic convergence [32, 5, 68]. Quadrature formulas have also been used to develop cubically-convergent incremental-iterative procedures [67, 69]. Also to improve convergence over typical Newton-Raphson type methods, families of higher order methods have been developed, including a fourth-order method by King [50], a third order method by Darvishi and Barati [31], and an eighth-order method by Bi et al. [17], to name a few. In addition to higher order Newton-type methods, Babajee et al. [4] have studied third order Chebyshev-type methods for solving systems of nonlinear equations. Recently, Shin et al. [88] compared higher order methods with Newton-Krylov methods and found the

Newton-Krylov methods perform better, if the system is sparse, as the size of the problem increases.

Many methods are available for solving nonlinear problems, and several are even successful at tracing beyond load and displacement limit points. However, these methods typically do not directly compute the location of such stability points. Many methods rely on the inspection of the determinant of the stiffness matrix for the location of stability points, but a more rigorous approach was needed. Wriggers and Simo [93] appended a constraint equation that characterizes load and displacement limit points to the system of nonlinear equations; thus the solution of the extended system not only yields the load parameter and displacement field, but also information about stability points. Fujii and Okazawa [44] developed a technique to pinpoint stability points both locally and globally that performed better than previous bisection or bracketing techniques. Rezaiee-Pajand and Vejdani-Noghreiyan [78] presented a method to locate multiple bifurcation points in structures using eigenvalue perturbation of the tangent stiffness matrix. Lopez [60] increased the range of validity over traditional methods for computing singular points with a robust algorithm using asymptotic extrapolation in the predictor phase of the continuation method, resulting in a more accurate location of limit points. Similarly, Korelc [51] computed the location of critical points with the extended system formulation and automatic differentiation, which resulted in significantly increased convergence.

Nonlinear solution methods have been applied extensively for nonlinear analysis of structural systems. For example, Yang et al. [98] present a simple formulation for nonlinear elastic structural systems that utilizes different components of the tangent stiffness matrix depending on the degree of nonlinearity of the system. Hrinda [49] analyzed highly geometrically nonlinear 3D truss systems to determine their actual load carrying capacity, which often occurs beyond the first limit point. Both linear and nonlinear solution methods were utilized by Saffari et. al. [83] to capture the response of space trusses beyond limit points. Also, Wang et al. conducted post-buckling analysis of structural systems using a combination of a genetic algorithm and a quasi-Newton method to determine the multiple equilibrium states of the nonlinear system.

In addition to individual solution schemes, libraries and tool kits for solving systems modeled by partial differential equations have also been developed. The Portable, Extensible Toolkit for Scientific Computation (PETSc) [8, 6, 7], for instance, is a suite of routines designed to solve large-scale applications through utilization of parallel linear and nonlinear solvers, including a parallel Newton-based solver. Similarly, the Library of Continuation Algorithms, LOCA [84], was developed at Sandia National Laboratories in Albuquerque, New Mexico to perform stability analysis on large scale problems by tracking multiple solution branches and bifurcation points. The software, LOCA, supports a variety of algorithms as one algorithm

is not sufficient to solve all problems.

1.2.2 Motivation for a library of nonlinear solution schemes

It is clear from the review in the previous section that it is nearly impossible to develop one single method capable of solving any general nonlinear problem. Depending on the problem and the severity of the nonlinearities, modifications to solution algorithms are necessary to recover the entire equilibrium path. Bergan et al. [16] stated

> "a computer program for non-linear analysis should possess several alternative algorithms for the solution of the non-linear system. These procedures should also allow for the possibility of an extensive control over the solution process by parameters that are input to the program. Such a scheme would lead to increased flexibility, and the experienced user has the possibility of obtaining improved reliability and efficiency for the solution of a particular problem."

Many authors have developed families of nonlinear solution schemes, which can be adjusted by the user depending on the problem. In the early days of development of nonlinear solution schemes Mondkar and Powell [64] developed a library of algorithms based on the standard and modified Newton-Raphson method. Seven solution schemes were formulated from 11 control parameters (stiffness update type and frequency, convergence tolerance, etc.) and tested on several nonlinear structural systems. Clark and Hancock [25] used the concept of load increment from the standard or modified Newton-Raphson method to unify several nonlinear solution schemes through a single load factor. The specific incremental-iterative procedure depends on the chosen constraint equation, which is used to calculate the unifying load factor. The constraint equations are based on iterations at constant load, displacement, work, arc-length, or minimum residual. Yang and Sheih [97] and Yang and Kuo [95] presented a similar library of nonlinear solvers unified through a single load parameter, and included the generalized displacement control method. More recently Rezaiee-Pajand et al. [77] unified five nonlinear solution schemes through a single general constraint equation. The schemes were identified by five different constraints, including minimizing error by means of its length, area, or perimeter, and then the strengths and weaknesses of each algorithm were evaluated.

The library of nonlinear solution schemes explored in this work is similar to its predecessors in that several solution schemes, defined by a constraint equation, are unified into a single space by means of a load parameter. The methods include load control, displacement control, work control, arc-length control, generalized displacement control, and the orthogonal residual procedure, which until now have not been incorporated into a collection of unified schemes.

The unified schemes are formulated and implemented such that (i) additional nonlinear solution schemes are readily incorporated and (ii) integration into a finite element analysis code is straightforward.

1.3 Thesis organization

The remainder of this thesis is organized as follows: the unified approach and the $(N+1)$ space formulation are discussed in Chapter 2. Also in this chapter, five of the six nonlinear solution schemes are discussed and cast into the $(N+1)$ space. The sixth nonlinear solution scheme, ORP, and its variations are discussed in more detail in Chapter 3. Next, in Chapter 4, the object-oriented implementation of the unified library, called NLS++, and complete class structure is presented. Five nonlinear systems are analyzed with NLS++ and their results discussed in Chapter 5. Finally, a summary of this work is presented with suggestions for improvements and future extensions.

Chapter 2

Unified approach to nonlinear solution schemes

Nonlinear solution schemes have been studied extensively for use with the finite element method to solve complex material and geometrically nonlinear problems. The following solution schemes are presented in the context of the unified approach: load control, displacement control, arc-length control, work control, generalized displacement control, and orthogonal residual. The solution schemes are inherently different in their formulations and therefore feature unique constraint equations for the incremental-iterative procedure. The governing finite element equation and constraint equations are combined into a single matrix equation, which will be used to characterize the unified approach. In this Chapter, incremental-iterative schemes for solving nonlinear systems of finite element equations are first reviewed. Next, the formulation of the $(N+1)$ dimensional space is presented. Finally, various algorithms, including load control, displacement control, arc-length method, work control method, and generalized displacement control are discussed using this approach. The orthogonal residual procedure will be discussed in more detail in the next Chapter.

2.1 Overview of incremental-iterative schemes

Numerical methods for solving nonlinear finite element systems generally adopt an incremental-iterative procedure, where at each incremental step a series of iterations is performed until a specified convergence criterion is reached. Figure 2.1 illustrates the basic idea, adopted by most incremental-iterative methods, using a single degree of freedom system. Here the increment is denoted with the superscript i, iterations within the i^{th} incremental step are denoted with the subscript j, and ξ is a load scaling factor.

As shown in Figure 2.1, for a certain step i, the method for solving nonlinear systems starts from a load increment vector, denoted by $\overline{\Delta \mathbf{p}}^{\,i}$. The first estimate of the incremental displacement vector, $\Delta \mathbf{u}^i$, can be calculated using the linearized tangent stiffness, $\mathbf{K}_0^i$, which is calculated at the beginning of the increment.

$$\mathbf{K}_0^i \Delta \mathbf{u}^i = \overline{\Delta \mathbf{p}}^{\,i} \tag{2.1}$$

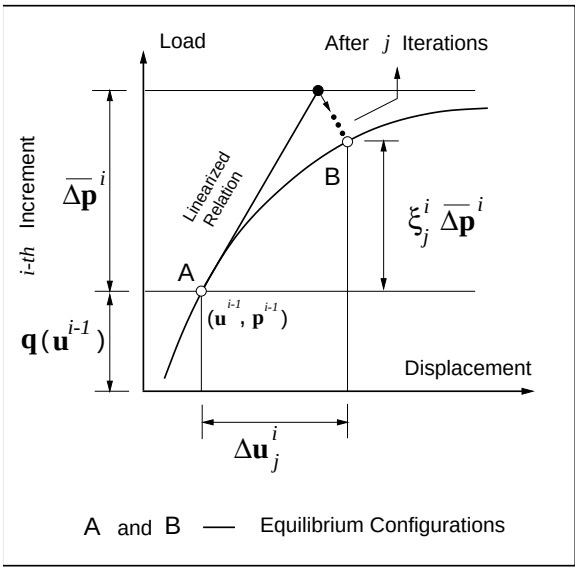

Figure 2.1: Incremental-iterative scheme for solving nonlinear problems

Due to the nonlinear nature of the problem, the internal forces $\mathbf{q} \equiv \mathbf{q}(\mathbf{u})$, evaluated at

$$\mathbf{u} = \mathbf{u}^{i-1} + \Delta\mathbf{u}^i \tag{2.2}$$

are not in equilibrium with the externally applied forces $\mathbf{p}$, given by

$$\mathbf{p} = \mathbf{p}^{i-1} + \overline{\Delta\mathbf{p}}^{\,i} \tag{2.3}$$

Thus, an unbalanced force or a residual is generated, which is expressed by

$$\mathbf{r} = \mathbf{p} - \mathbf{q}(\mathbf{u}) \tag{2.4}$$

To establish equilibrium, the iterative steps inside each increment generates a series of load and displacement updates, given by $\delta\mathbf{p}_j^i$ and $\delta\mathbf{u}_j^i$, respectively. Hence, the total load and displacement at the j^{th} iteration of the i^{th} incremental step are given by

$$\Delta\mathbf{u}_j^i = \Delta\mathbf{u}_{j-1}^i + \delta\mathbf{u}_j^i \tag{2.5}$$

$$\Delta\mathbf{p}_j^i = \Delta\mathbf{p}_{j-1}^i + \delta\mathbf{p}_j^i \tag{2.6}$$

where $\Delta\mathbf{u}_j^i$ is the incremental displacement vector in iteration j of step i, $\delta\mathbf{u}_j^i$ is the iterative displacement vector in iteration j of step i, $\Delta\mathbf{p}_j^i$ is the incremental force vector in iteration j of step i, and $\delta\mathbf{p}_j^i$ is the iterative force vector in iteration j of step i. The iterative process continues until, in a certain iteration j, the residual $\mathbf{r}_j^i$, obtained by

$$\mathbf{r}_j^i = \mathbf{p}^{i-1} + \Delta\mathbf{p}_j^i - \mathbf{q}(\mathbf{u}^{i-1} + \Delta\mathbf{u}_j^i) \tag{2.7}$$

is sufficiently small in a certain norm (e.g. Euclidean) with respect to a reference value. If

the convergence criterion is not satisfied, a new iteration ($j \leftarrow j+1$) is processed and a new iterative displacement vector $\delta\mathbf{u}_j^i$ is calculated using a linearized stiffness relation [95]. Essentially, an iteration consists of the solution of a system of equations linearized about the current state of the nonlinear problem.

2.2 N+1 dimensional space formulation

In nonlinear finite element analysis [29, 9, 14, 76, 95]the governing system of nonlinear equations to be solved at the j^{th} iteration of the i^{th} incremental step is given by

$$\mathbf{K}_{j-1}^i \delta\mathbf{u}_j^i = \mathbf{p}_j^i - \mathbf{q}_{j-1}^i \tag{2.8}$$

where $\mathbf{K}_{j-1}^i$ is the tangent matrix, $\delta\mathbf{u}_j^i$ is the displacement vector, $\mathbf{p}_j^i$ is the external force vector, and $\mathbf{q}_{j-1}^i$ is the internal load vector. The unified approach decomposes this system and reformulates the nonlinear solution process into an $(N+1)$ dimensional space, which includes N displacement components ($\delta\mathbf{u}_j^i$) and one load parameter ($\delta\lambda_j^i$) [97]. The load parameter is first introduced through the force relation, where $\delta\mathbf{p}_j^i$ is replaced with $\delta\lambda_j^i\overline{\mathbf{p}}$

$$\mathbf{p}_j^i = \mathbf{p}^{i-1} + \Delta\mathbf{p}_{j-1}^i + \delta\mathbf{p}_j^i \tag{2.9}$$

$$\mathbf{p}_j^i = \mathbf{p}^{i-1} + \Delta\mathbf{p}_{j-1}^i + \delta\lambda_j^i\overline{\mathbf{p}} \tag{2.10}$$

Equations 2.7, 2.8, and 2.10, are combined to give the following system of equations

$$\mathbf{K}_{j-1}^i \delta\mathbf{u}_j^i = \mathbf{r}_{j-1}^i + \delta\lambda_j^i\overline{\mathbf{p}} \tag{2.11}$$

Equation 2.11 is a system of N equations with $(N+1)$ unknowns, therefore an additional constraint equation of the general form

$$\Phi(\delta\mathbf{u}, \delta\lambda) = 0 \tag{2.12}$$

must be added to the system. The particular constraint equation, as proposed by Yang and Kuo [95], has the form

$$\mathbf{a}_j^i \cdot \delta\mathbf{u}_j^i + b_j^i \delta\lambda_j^i = c_j^i \tag{2.13}$$

Equations 2.11 and 2.13 yield a system of $(N+1)$ equations and $(N+1)$ unknowns, shown in matrix form

$$\begin{bmatrix} \mathbf{K}_{j-1}^i & -\overline{\mathbf{p}} \\ (\mathbf{a}_j^i)^T & b_j^i \end{bmatrix} \begin{Bmatrix} \delta\mathbf{u}_j \\ \delta\lambda_j^i \end{Bmatrix} = \begin{Bmatrix} \mathbf{r}_{j-1}^i \\ c_j^i \end{Bmatrix} \tag{2.14}$$

The augmented system matrix is no longer symmetric and has a significantly increased bandwidth due to the added load parameter. The solution of this system with a traditional method would be computationally undesirable with respect to both storage and efficiency. However, Batoz and Dhatt presented a technique to overcome this problem [11]. It consists of decomposing the iterative displacement vector into two parts

$$\delta \mathbf{u}_j^i = \delta\lambda_j^i \delta {\mathbf{u}_I}_j^i + \delta {\mathbf{u}_{II}}_j^i \tag{2.15}$$

then Equation 2.11 becomes

$$\begin{aligned} \mathbf{K}_{j-1}^i \delta {\mathbf{u}_I}_j^i &= \overline{\mathbf{p}} \\ \mathbf{K}_{j-1}^i \delta {\mathbf{u}_{II}}_j^i &= \mathbf{r}_{j-1}^i \end{aligned} \tag{2.16}$$

It is clear that Equations 2.16 and 2.11 are mathematically equivalent by means of Equation 2.15. The components of the total iterative displacement, $\delta {\mathbf{u}_I}_j^i$ and $\delta {\mathbf{u}_{II}}_j^i$ are computed using the original system matrix, $\mathbf{K}_{j-1}^i$. Thus, the banded and symmetric properties of the original system remain intact [55]. Finally, the load parameter is needed to compute the total displacement for the j^{th} iteration of the i^{th} incremental step. Solving Equation 2.13 for the load parameter and combining with Equation 2.15 yields

$$\delta\lambda_j^i = \frac{c_j^i - \mathbf{a}_j^i \cdot \delta {\mathbf{u}_{II}}_j^i}{\mathbf{a}_j^i \cdot \delta {\mathbf{u}_I}_j^i + b_j^i} \tag{2.17}$$

The constraint equation will be directly associated with a particular nonlinear solution scheme. Or, in other words, the formulation of each nonlinear solution scheme will give rise to the constraint parameters, $\mathbf{a}_j^i$, b_j^i, and c_j^i.

2.3 Nonlinear solution schemes

A number of nonlinear solution procedures have been proposed to trace equilibrium paths, such as load control, displacement control, work control, arc-length and generalized displacement control. The general aspects and modifications for the N+1 dimensional space formulation of the aforementioned methods are discussed in this section. One additional nonlinear solution scheme, the orthogonal residual procedure, will be discussed in detail in Chapter 3.

2.3.1 Load control method (LCM)

Traditionally, load control or Newton-type methods have been the most popular ones to solve nonlinear system of equations [79, 64]. Furthermore, many advances in nonlinear

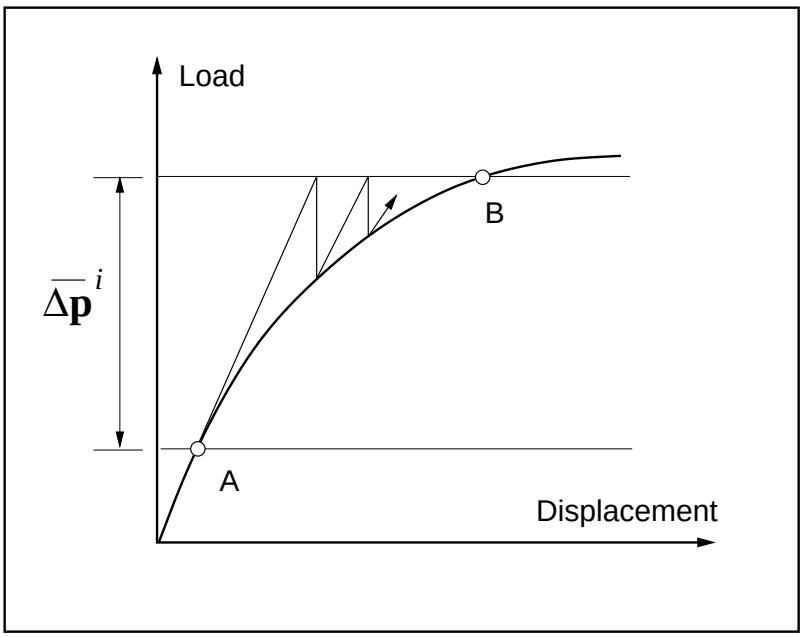

Figure 2.2: Load control method

solvers consist of variations of the basic Newton-Raphson method [74, 70]. This method is the simplest to cast into the $(N+1)$ dimensional space because, by definition of the algorithm, the external loads are computed at the first iteration of each incremental step and held constant throughout the remaining iterations in the step. Hence, this method is also referred to a the load control method, as illustrated in Figure 2.2.

$$\delta\lambda_j^i = \begin{cases} \text{prescribed value} & \text{for } j = 1 \\ 0 & \text{for } j \geq 2 \end{cases} \tag{2.18}$$

Comparing Equation 2.18 and Equation 2.17, one obtains the constraint parameters are clearly:

$$\mathbf{a}_j^i = 0 \tag{2.19}$$

$$b_j^i = 1 \tag{2.20}$$

$$c_j^i = \begin{cases} \overline{\Delta\lambda} & \text{for } j = 1 \\ 0 & \text{for } j \geq 2 \end{cases} \tag{2.21}$$

where $\overline{\Delta\lambda}$ is a prescribed initial load factor. Because the Newton-Raphson method imposes the load factor, the system has only N unknowns and the decomposition of Equation 2.11

is not needed. Instead, the system can be solved from Equation 2.11 directly

$$\begin{aligned} \mathbf{K}_{j-1}^{i}\delta\mathbf{u}_{1}^{i} &= \overline{\Delta\lambda}\overline{\mathbf{p}} \quad \text{for } j=1 \\ \mathbf{K}_{j-1}^{i}\delta\mathbf{u}_{j}^{i} &= \mathbf{r}_{j-1}^{i} \quad \text{for } j\geq 2 \end{aligned} \tag{2.22}$$

The tangent matrix, $\mathbf{K}_{j-1}^{i}$, may either be computed in accordance with the *standard* Newton-Raphson method or with the *modified* Newton-Raphson method. In the former method, the tangent matrix is calculated at the beginning of each iteration, while in the latter the tangent matrix is only computed at the beginning of each incremental step and held constant for each iteration (i.e. $\mathbf{K}_{j-1}^{i} = \mathbf{K}_{0}^{i}$ for $j \geq 2$). The modified Newton-Raphson method has a lower computational cost at each iteration than the standard version, but convergence will be slower.

While this method is very widely used, it is not inherently robust. Since the externally applied loads are kept constant, this method has difficulties near load limit points. Yang and Sheih [97] further showed that the constraint parameters imposed by this method will yield unbounded displacements near load limit points when the tangent matrix is nearly singular.

2.3.2 Displacement control method (DCM)

Analogous to the Newton-Raphson method with a fixed load parameter, the displacement control method uses a fixed displacement component as the control parameter to trace the equilibrium path. In multi-degree of freedom system, one displacement component is selected as the control component, denoted $\delta u_{j_{\text{CTRL}}}^{i}$.

$$\delta u_{j_{\text{CTRL}}}^{i} = \begin{cases} \text{prescribed value} & \text{for } j=1 \\ 0 & \text{for } j\geq 2 \end{cases} \tag{2.23}$$

Solving Equation 2.15 for the control parameter with respect to the control component gives

$$\delta\lambda_{j}^{i} = \frac{\delta u_{j_{\text{CTRL}}}^{i} - \delta {u_{II}}_{j_{\text{CTRL}}}^{i}}{\delta {u_{I}}_{j_{\text{CTRL}}}^{i}} \tag{2.24}$$

On the first iteration the residual will be zero, so $\delta {\mathbf{u}_{II}}_{j}^{i}$ must also be zero, as evident from Equation 2.16. Then the expression for $\delta\lambda_{j}^{i}$ reduces to the following

$$\delta\lambda_j^i = \begin{cases} \frac{\delta u^i_{j_{\text{CTRL}}}}{\delta u_{I^i_{j_{\text{CTRL}}}}} & \text{for } j = 1 \\ \\ \frac{-\delta u_{II^i_{j_{\text{CTRL}}}}}{\delta u_{I^i_{j_{\text{CTRL}}}}} & \text{for } j \geq 2 \end{cases} \tag{2.25}$$

Now comparing Equation 2.25 and Equation 2.17, the constraint parameters are

$$\mathbf{a}_j^i = \underbrace{[0, 0, ...1, ..., 0]}_{\text{control displacement}} \tag{2.26}$$

$$b_j^i = 0 \tag{2.27}$$

$$c_j^i = \begin{cases} \overline{\Delta u} & \text{for } j = 1 \\ 0 & \text{for } j \geq 2 \end{cases} \tag{2.28}$$

The final expression for the load factor is given by

$$\delta\lambda_j^i = \begin{cases} \frac{\overline{\Delta u}}{\delta u_{I^i_{j_{\text{CTRL}}}}} & \text{for } j = 1 \\ \\ \frac{-\delta u_{II^i_{j_{\text{CTRL}}}}}{\delta u_{I^i_{j_{\text{CTRL}}}}} & \text{for } j \geq 2 \end{cases} \tag{2.29}$$

where $\overline{\Delta u}$ is the prescribed initial displacement. For this reason, the displacement control method works well for load limit points, however it may fail near displacement limit points. Figure 2.3 illustrates the behavior of the displacement control method. Yang and Sheih [97] further showed that at displacement limit points the control displacement, $\delta\mathbf{u}_{I^i_{j_{\text{CTRL}}}}$, approaches zero, and the load parameter approaches infinity, hence numerical instability occurs at displacement limit points.

The main drawback of this method is that the control parameter is selected intuitively or empirically, and remains fixed during the whole path-tracing process. Fujii et al. [43] devised a technique to systematically select the best control parameter. A decreasing component of the displacement vector typically suggests that a displacement limit point may be approaching in that component. However, the largest component of the displacement vector is the least likely component to experience a displacement limit point. The best candidate for the control component is therefore the one with the maximum absolute value of displacement. The sign of this component should also be retained to ensure the correct direction of equilibrium tracing. With this minor modification, the displacement control method can potentially capture displacement limit points.

The previous two solution methods keep either the external load or displacement constant through iterations, giving difficulties in load and displacement limit points, respectively. To

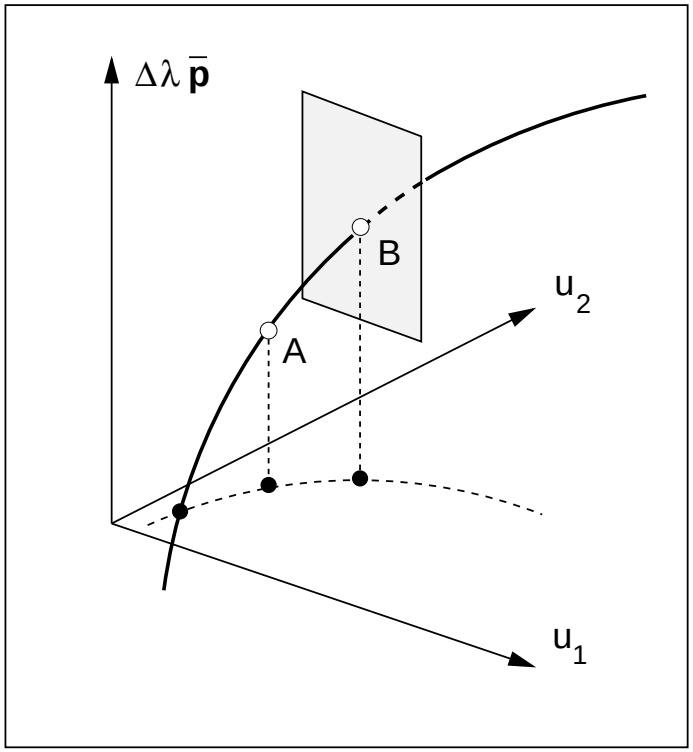

Figure 2.3: Displacement control method

circumvent those problems, methods which allow simultaneous changes of both the load and displacement levels, along the incremental-iterative process, have been extensively explored in the technical literature. These methods treat the load level as an additional variable, so that the equilibrium configuration can be followed beyond limit points.

2.3.3 Arc-length control method (ALCM)

The most typical example of a nonlinear solution scheme which considers simultaneous iteration on both the load and displacement variables is the arc-length method. The premise of the arc length method is to impose a constraint where the norm (i.e. Euclidean norm) of the increment, $(\Delta \mathbf{u}_j^i, \Delta \mathbf{p}_j^i)$, is prescribed for the first iteration and held constant at subsequent iterations. An arc-length, Δs_j^i, is calculated at the beginning of each increment and held constant throughout the iterations.

$$\Delta s_j^i = \begin{cases} \text{prescribed value} & j = 1 \\ 0 & j \geq 2 \end{cases} \tag{2.30}$$

There are several versions of the arc-length method, including cylindrical, spherical, elliptical and linearized. The general constraint equation, which can represent all of these variations

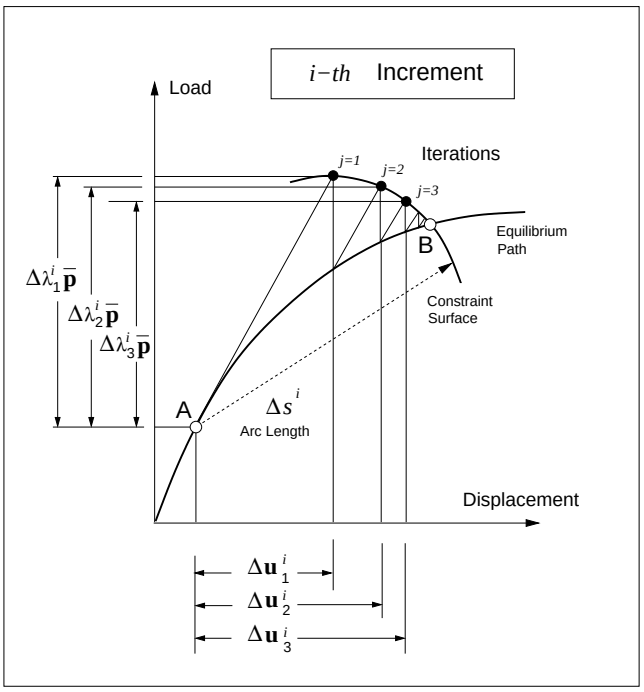

Figure 2.4: Arc-length control method

is of the form

$$\Delta\mathbf{u}_j^i \cdot \Delta\mathbf{u}_j^i + \eta\Delta\mathbf{p}_j^i \cdot \Delta\mathbf{p}_j^i = (\Delta s^i)^2$$

Replacing $\Delta\mathbf{p}_j^i$ with $\Delta\lambda_j^i\overline{\mathbf{p}}$, as before, the constraint equation now becomes

$$\Delta\mathbf{u}_j^i \cdot \Delta\mathbf{u}_j^i + \eta(\Delta\lambda_j^i)^2(\overline{\mathbf{p}} \cdot \overline{\mathbf{p}}) = (\Delta s_j^i)^2 \tag{2.31}$$

where η is a non-negative real parameter, which is unique for each version of the arc-length method. Figure 2.4 illustrates the incremental-iterative procedure of the arc-length method for a one dimensional problem. An initial arc-length is determined in accordance with Equation 2.31, then subsequent iterations lie on the constraint surface created by the arc. Iterations eventually converge, which is shown in the Figure as the intersection of the arc and the equilibrium path. Notice that the constraint equation is applied to the entire incremental step, rather than to the particular iteration, i.e. $\Delta\mathbf{u}_j^i$ and $\Delta\lambda_j^i$ are used rather than $\delta\mathbf{u}_j^i$ and $\delta\lambda_j^i$, respectively.

Spherical arc-length control method

The spherical arc-length method, [27, 75], sets the scaling parameter to one. The constraint equation becomes

$$\Delta\mathbf{u}_j^i \cdot \Delta\mathbf{u}_j^i + (\delta\lambda_j^i)^2(\overline{\mathbf{p}} \cdot \overline{\mathbf{p}}) = (\Delta s_j^i)^2 \tag{2.32}$$

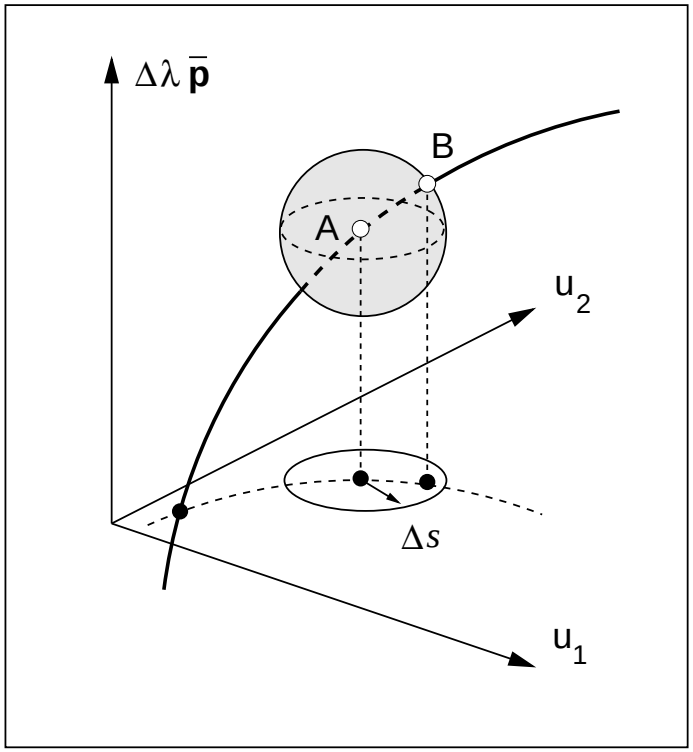

Figure 2.5: Spherical arc-length method

The constraint equation now represents a sphere in three dimensional space, as shown in Figure 2.5. Through several benchmark problems, Bellini and Chulya [13] demonstrated that in areas of very high slope this method may have difficulties. The initial arc-length may be adjusted such that is its very small, however this increases the number of incremental steps to trace the entire equilibrium path, thereby increasing computational time.

Cylindrical arc-length control method

As indicated by Crisfield [27], the cylindrical arc-length method consists of setting η to zero. Bellini and Chulya [13] illustrated the effectiveness of this version in capturing sharp turns at load limit points. The constraint equation represents a cylinder in three dimensional space,

$$\Delta \mathbf{u}_j^i \cdot \Delta \mathbf{u}_j^i = (\Delta s_j^i)^2 \tag{2.33}$$

as it is illustrated by Figure 2.6.

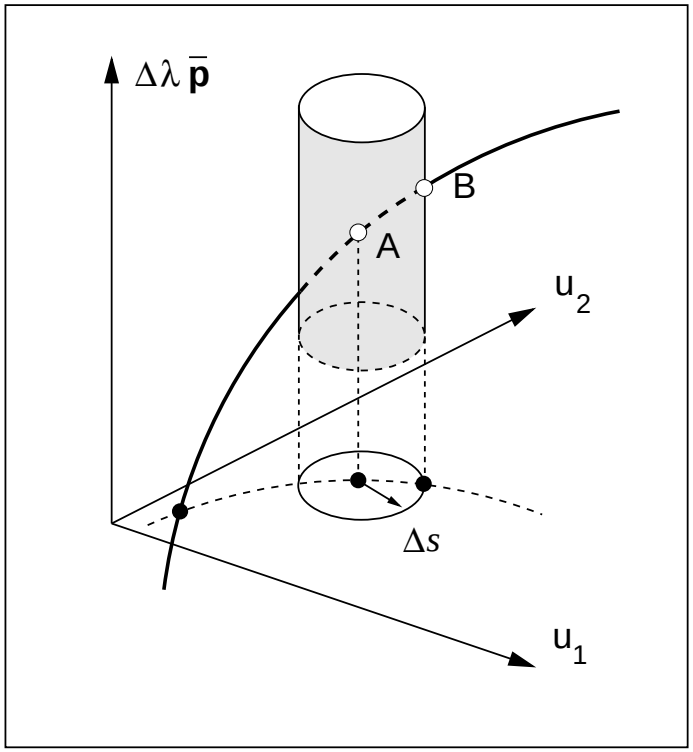

Figure 2.6: Cylindrical arc-length method

Elliptical arc-length control method

The most general form is the elliptical arc-length method, where $\eta > 0$ and $\eta \neq 1$. Figure 2.7 shows that the constraint equation represents an ellipsoid in three dimensional space. Park [72] has proposed a method where the scaling parameter, η, is the current stiffness parameter, introduced by Bergan [16]. Bellini and Chulya [13] documented the success of this particular version of the elliptical arc-length method.

Linearized arc-length control method

The linearized arc-length method was studied by Wempner [92] and Riks [79], and investigated further by Riks [80] and Crisfield [27]. These methods are also referred to as orthogonal arc-length methods because, in general, a norm constraint is imposed at the first iterative step, then an orthogonality condition is met in subsequent iterations of that increment. The orthogonality condition determines the type of linearized arc-length method. For example, in the Fixed Normal Plane version, the iterative vectors $(\delta\mathbf{u}_j^i, \delta\mathbf{p}_j^i)$ are orthogonal to the initial incremental vector $(\delta\mathbf{u}_j^i, \delta\mathbf{p}_j^i)$. In the Updated Normal Plane version, on the other hand, the iterative vectors $(\delta\mathbf{u}_j^i, \delta\mathbf{p}_j^i)$ are orthogonal to the previous incremental vector$(\delta\mathbf{u}_j^i, \delta\mathbf{p}_j^i)$. The linearized versions are shown in Figure 2.9.

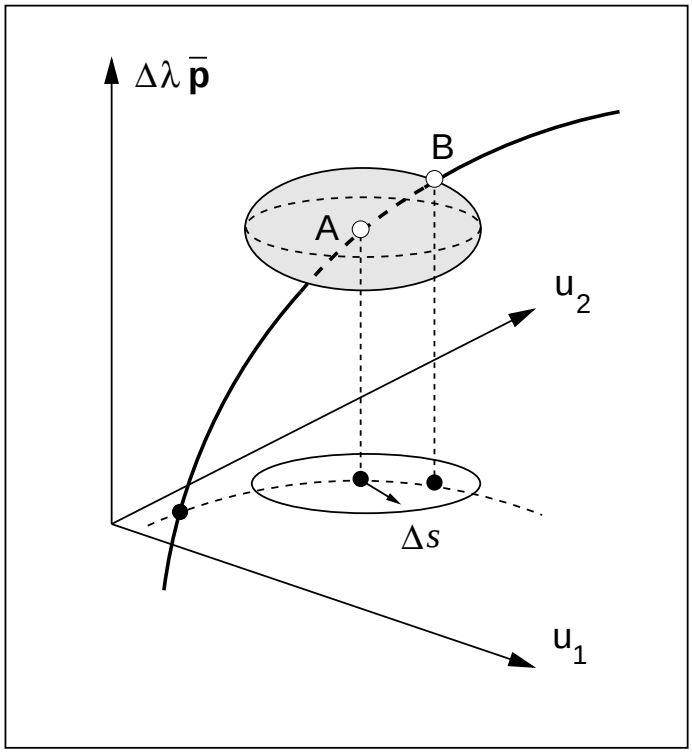

Figure 2.7: Elliptical arc-length method

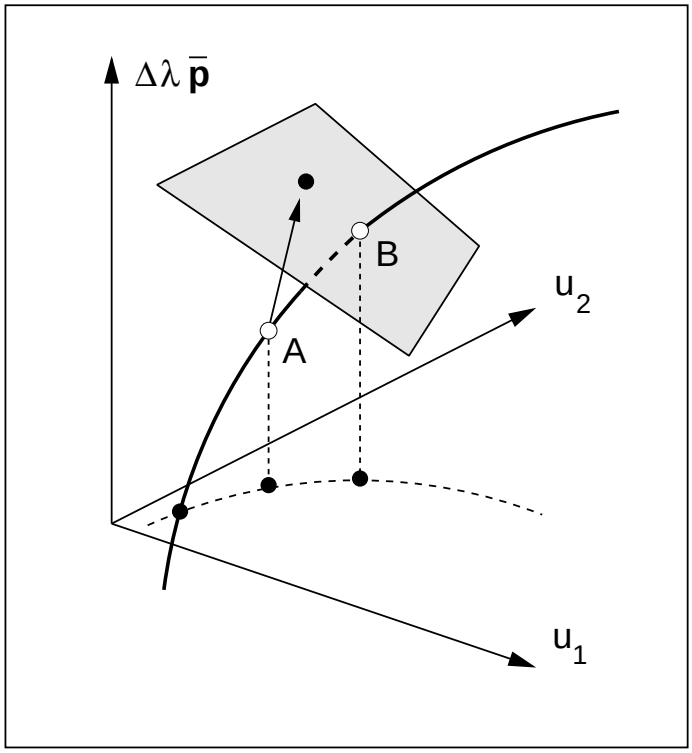

Figure 2.8: Linearized arc-length method

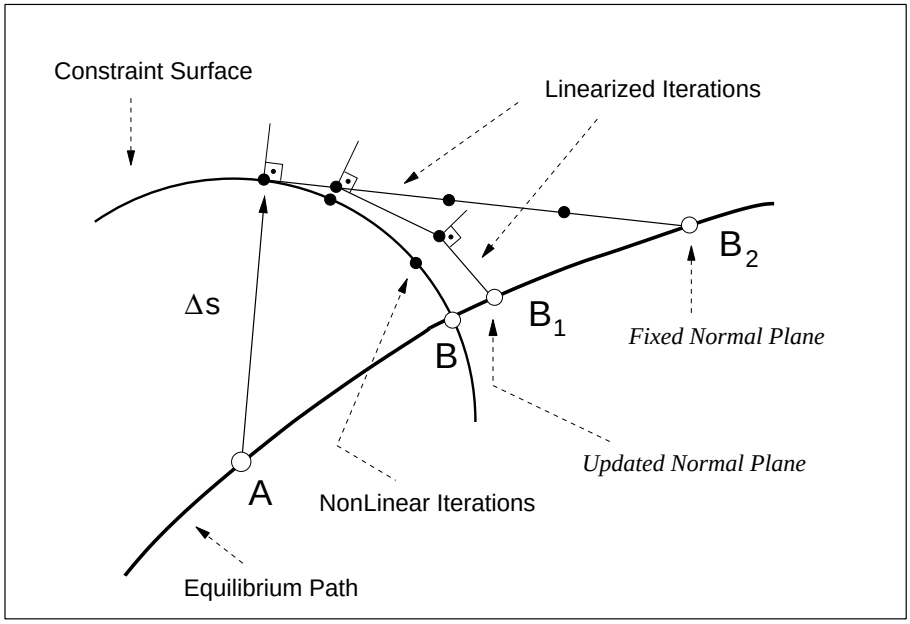

Figure 2.9: Updated normal plane and fixed normal plane versions of the linearized arc-length method

When the general nonlinear constraint equation, 2.31, of the arc-length method is adopted, a quadratic equation in terms of $\delta\lambda_j^i$ is obtained. It is necessary to adopt a set of rules to treat either real or complex roots obtained from the quadratic equation. For real roots, in general, the selected root is the one that corresponds to the smallest change in the direction of the iterative displacement vector compared to the previous displacement vector [13, 28]. Lam and Morley [56] have presented a methodology for treating the complex roots that can arise in the above mentioned quadratic equation.

The constraint equation given in Equation 2.31 can be written with respect to the iterations rather than the increments. The inner product of $\overline{\mathbf{p}}$ with itself will be taken as unity, since the reference load vector can always be expressed as a unit vector. Then the constraint equation becomes

$$\delta\mathbf{u}_1^i \cdot \delta\mathbf{u}_j^i + \eta\delta\lambda_1^i\delta\lambda_j^i = (\Delta s_j^i)^2 \tag{2.34}$$

Now combining Equations 2.34 and 2.30 and comparing with Equation 2.13, one obtains the

constraint parameters below

$$\mathbf{a}_j^i = \delta\mathbf{u}_1^i = \delta\lambda_1^i \delta\mathbf{u}_{I1}^i \tag{2.35}$$

$$b_j^i = \eta\delta\lambda_1^i \tag{2.36}$$

$$c_j^i = \begin{cases} (\overline{\Delta S})^2 & \text{for } j = 1 \\ 0 & \text{for } j \geq 2 \end{cases} \tag{2.37}$$

where $\overline{\Delta S}$ is the prescribed arc-length to be assigned at the first iteration and held constant for subsequent iterations. The load factor is then given by

$$\delta\lambda_j^i = \begin{cases} \pm\frac{\overline{\Delta S}}{\sqrt{\delta\mathbf{u}_{I1}^i \cdot \delta\mathbf{u}_{I1}^i + \eta}} & \text{for } j = 1 \\ \\ -\frac{\delta\mathbf{u}_1^i \cdot \delta\mathbf{u}_{Ij}^i}{\delta\mathbf{u}_1^i \cdot \delta\mathbf{u}_{Ij}^i + \eta\delta\lambda_1^i} & \text{for } j \geq 2 \end{cases} \tag{2.38}$$

The sign of the load factor in the previous equation is based on whether the system is softening, in which case the sign should be negative, or stiffening, in which case the sign should be positive. However, this method does not specify this criteria explicitly. The arc-length method is widely used for highly nonlinear problems because it can capture nonlinearities at load limit points and has the potential to capture behavior even at displacement limit points.

A potential shortcoming of this method, however, is that the terms in the expression for $\delta\lambda_j^i$ are of different units. For example, the load parameter, $\delta\lambda_1^i$, is a scalar, while the displacements $\delta\mathbf{u}$ contain both translations and rotations, which are of different orders of magnitude [96]. In areas near displacement limit points with very high gradient it is possible that $\delta\lambda_1^i$ is so large that the sign of $\delta\lambda_j^i$ depends only on the angle between $\delta\mathbf{u}_1^i$ and $\delta\mathbf{u}_{Ij}^i$. It follows that the sign of $\delta\lambda_j^i$ may change incorrectly, causing numerical divergence near displacement limit points [97]. It should be noted that the load factor should only change sign at areas of load limit points, not at displacement limit points.

2.3.4 Work control method (WCM)

The work control method was studied by Bathe and Dvorkin [10] and Yang and McGuire [96]. Yang and McGuire identified the motivation for the method, which was to overcome the issue of inconsistent physical units, as discussed for the arc-length method. This method

uses a constant work increment, δW_j^i, through the iterations of an incremental step

$$\delta W_j^i = \begin{cases} \text{prescribed value} & \text{for } j = 1 \\ 0 & \text{for } j \geq 2 \end{cases} \tag{2.39}$$

The constraint equation is given by

$$\delta W_j^i = \delta\lambda_j^i \overline{\mathbf{p}} \cdot \delta\mathbf{u}_j^i \tag{2.40}$$

The constraint parameters can be determined directly from Equation 2.40, i.e.

$$\mathbf{a}_j^i = \delta\lambda_j^i \overline{\mathbf{p}} \tag{2.41}$$

$$b_j^i = 0 \tag{2.42}$$

$$c_j^i = \begin{cases} \overline{\Delta W} & \text{for } j = 1 \\ 0 & \text{for } j \geq 2 \end{cases} \tag{2.43}$$

where $\overline{\Delta W}$ is the prescribed work increment. Substituting Equation 2.15 into $\delta\mathbf{u}_j^i$ in Equation 2.40 and noting that $\delta\mathbf{u}_{II}{}_j^i = 0$ on the first iteration, one obtains

$$\delta\lambda_j^i = \begin{cases} \pm\sqrt{\left|\frac{\overline{\Delta W}}{\overline{\mathbf{p}} \cdot \delta\mathbf{u}_{I}{}_1^i}\right|} & \text{for } j = 1 \\ -\frac{\overline{\mathbf{p}} \cdot \delta\mathbf{u}_{II}{}_1^i}{\overline{\mathbf{p}} \cdot \delta\mathbf{u}_{I}{}_1^i} & \text{for } j \geq 2 \end{cases} \tag{2.44}$$

Unlike the arc-length method, the sign of the load parameter is easily determined. The term inside the square root, called the current stiffness parameter [96], indicates whether the system is stiffening or softening. The sign of this term should be applied to the load parameter: if the term is positive the system is stiffening and the load parameter should increase, and if it is negative the stiffness is decreasing and the load parameter should also decrease.

Some weaknesses of the work control method have been examined by Yang and Sheih [97]. A potentially problematic situation occurs when there are a small number of degrees of freedom and the displacement associated with the major forcing direction tends to snap back (i.e. at a displacement limit point). The quantity $\overline{\mathbf{p}} \cdot \delta\mathbf{u}_{I}{}_1^i$ will tend to zero forcing the load parameter to infinity. Thus this method only has limited success near displacement limit points. Additionally, the presence of the reference load vector in the expression for $\delta\lambda_j^i$ may have adverse effects because it is somewhat arbitrary and does not represent the structural system.

2.3.5 Generalized displacement control method (GDCM)

The generalized displacement control method was investigated by Yang and Sheih [97] as a result of the limitations discussed for other methods. Noticing that numerical stability depends on the selection of the constraint parameters, the following values were adopted:

$$\mathbf{a}_j^i = \delta\lambda_1^i \delta\mathbf{u}_{I_1}^{i-1} \tag{2.45}$$

$$b_j^i = 0 \tag{2.46}$$

$$c_j^i = \begin{cases} \text{generalized displacement} & \text{for } j = 1 \\ 0 & \text{for } j \geq 2 \end{cases} \tag{2.47}$$

Plugging the constraint parameters into Equation 2.17 one obtains an expression for $\delta\lambda_j^i$

$$\delta\lambda_j^i = \frac{c_j^i - \delta\lambda_1^i \left(\delta\mathbf{u}_{I_1}^{i-1} \cdot \delta\mathbf{u}_{II_j}^{i}\right)}{\delta\lambda_1^i \left(\delta\mathbf{u}_{I_1}^{i-1} \cdot \delta\mathbf{u}_{I_j}^{i}\right)} \tag{2.48}$$

which, when recalling that $\delta\mathbf{u}_{II_1}^i = 0$ when $j = 1$, simplifies to

$$\delta\lambda_1^i = \sqrt{\frac{c_1^i}{\delta\mathbf{u}_{I_1}^{i-1} \cdot \delta\mathbf{u}_{I_1}^{i}}} \tag{2.49}$$

$$\delta\lambda_j^i = -\frac{\delta\mathbf{u}_{I_1}^{i-1} \cdot \delta\mathbf{u}_{II_j}^{i}}{\delta\mathbf{u}_{I_1}^{i-1} \cdot \delta\mathbf{u}_{I_j}^{i}} \tag{2.50}$$

Let $\delta\mathbf{u}_{I_1}^1 = \delta\mathbf{u}_{I_1}^0$ and solve Equation 2.49 for c_j^i, the generalized displacement,

$$c_j^i = \begin{cases} (\delta\lambda_1^1)^2 \left(\delta\mathbf{u}_{I_1}^1 \cdot \delta\mathbf{u}_{I_1}^1\right) & \text{for } j = 1 \\ 0 & \text{for } j \geq 2 \end{cases} \tag{2.51}$$

Now the expression for $\delta\lambda_1^i$ becomes

$$\delta\lambda_1^i = \pm\delta\lambda_1^1 \left(\left| \frac{\delta\mathbf{u}_{I_1}^1 \cdot \delta\mathbf{u}_{I_1}^1}{\delta\mathbf{u}_{I_1}^{i-1} \cdot \delta\mathbf{u}_{I_1}^{i}} \right|\right)^{\frac{1}{2}} \tag{2.52}$$

Similar to the work control method, the generalized displacement control method adjusts the sign of the load parameter based on the stiffness of the system. The generalized stiffness parameter (GSP), defined below, will be positive for stiffening systems and negative for softening systems. The behavior of the GSP, including sign changes of load parameter, is illustrated in Figure 2.10.

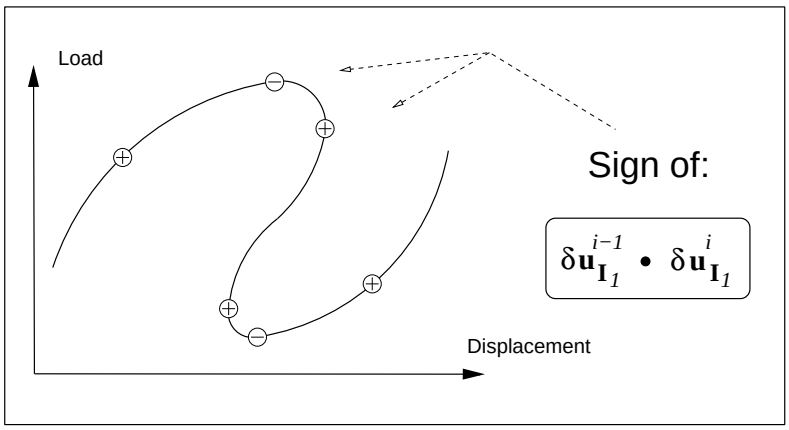

Figure 2.10: Generalized stiffness parameter used in the generalized displacement control method

$$GSP = \frac{\delta \mathbf{u}_{I_1}^{1} \cdot \delta \mathbf{u}_{I_1}^{1}}{\delta \mathbf{u}_{I_1}^{i-1} \cdot \delta \mathbf{u}_{I_1}^{i}} \tag{2.53}$$

The load parameter at the first iteration of each incremental step is simply

$$\delta \lambda_1^i = \pm \delta \lambda_1^1 \, |GSP|^{\frac{1}{2}} \tag{2.54}$$

Use of this physical quantity to represent the stiffness of the system makes this method computationally effective. The stiffness of the structure is measured with respect to the first incremental step, so stiffening and softening behavior are readily identified. Furthermore, the GSP changes sign only immediately after load limit points, meaning the direction of the load will only change at load limit points, not at displacement limit points, as with other methods. It can also be shown that the GSP remains bounded while tracing the equilibrium path.

While the generalized displacement control algorithm is very successful at capturing complex nonlinear behavior at both load and displacement limit points, it is not as widely used as the various versions of the arc-length method. Cardoso and Fonseca [19] identified that this method can actually be seen as an orthogonal arc-length method. The constraint equation posed by the generalized displacement method can be written as an orthogonal arc length constraint where adjustments in the radius of the arc are dependent on the value of the GSP. Furthermore, by viewing the generalized displacement control method as a variation of the arc-length method, many extensions and improvements made to arc-length methods could also apply to the generalized displacement control method.

Chapter 3

Orthogonal residual procedure

The orthogonal residual procedure is the sixth nonlinear solution algorithm included in the unified scheme formulation. It is more complex, in terms of formulating the procedure into the $(N+1)$ dimensional space, than the other solvers discussed in the previous Chapter. In this Chapter the original orthogonal residual procedure is presented followed by a few modifications and stabilizations. Then the formulation of the original algorithm into the N+1 dimensional space is discussed. Finally the implementation of the orthogonal residual load factor in the context of the N+1 dimensional space is presented.

3.1 Basic formulation

The orthogonal residual procedure (ORP), investigated by Krenk in 1995 [53], adjusts the load increment at each iterative step such that the current displacement increment is orthogonal to the current residual. The orthogonality constraint is based on the following arguments: the direction of the current displacement increment, $\Delta\mathbf{u}_j^i$, is taken as the best estimate of the direction of the actual displacement increment. The magnitude, however, will increase or decrease based on the projection of the residual force on the current displacement increment. The magnitude of the current displacement increment should not change; therefore the orthogonality between the residual and the current displacement increment should be enforced

$$\mathbf{u}_j^i \cdot \Delta\mathbf{r}_j^i = 0 \tag{3.1}$$

as illustrated by Figure 3.1 (a).

To justify why the residual should not change the magnitude of the current displacement, consider the case where the residual and current displacement increment are not orthogonal, as shown in Figure 3.1 (b). First, note the order in which computations occur in the j^{th} iteration. The incremental displacement is calculated using the tangent matrix and residual vector from the previous iteration

$$\delta\mathbf{u}_j^i = (\mathbf{K}_{j-1}^i)^{-1}\mathbf{r}_{j-1}^i \tag{3.2}$$

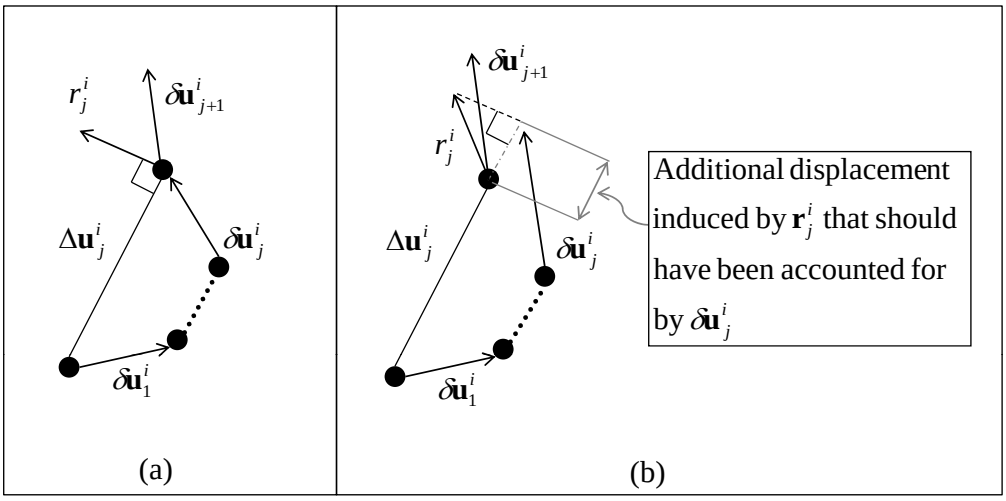

Figure 3.1: Orthogonality constraint of the orthogonal residual procedure

then the increment displacement is updated

$$\Delta\mathbf{u}_j^i = \Delta\mathbf{u}_{j-1}^i + \delta\mathbf{u}_j^i \tag{3.3}$$

and finally the residual is calculated. Again, the direction of the current displacement increment should not change, as it was calculated in the previous iteration and is taken as the best estimate of the actual displacement direction. If the residual is not orthogonal to the current displacement increment, then the iterative displacement increment, $\delta\mathbf{u}_j^i$, should have accounted for the induced displacement by the residual when it was calculated in the previous iteration. Changing the iterative displacement will also change the current displacement increment through Equation 3.3. If the magnitude of the current displacement increment changes due to the residual, then the calculation of the current displacement increment in the previous iterative step was not optimal.

The load factor, ξ_j^i, is calculated from the orthogonality constraint and is applied to the current load increment.

$$\mathbf{p}^i = \mathbf{p}^{i-1} + \xi_j^i \overline{\Delta\mathbf{p}}^i \tag{3.4}$$

The internal forces, $\mathbf{q}$, are calculated from the previous values of the displacement increment. Similarly to Equation 2.7, the residual is

$$\mathbf{r}_j^i = \mathbf{p}^{i-1} + \xi_j^i \overline{\Delta\mathbf{p}}^i - \mathbf{q}(\mathbf{u}^{i-1} + \Delta\mathbf{u}_j^i) \tag{3.5}$$

The optimal load increment factor is obtained by inserting Equation 3.5 into the constraint

Equation, 3.1.

$$\xi_j^i = -\frac{\left[\mathbf{p}^{i-1} - \mathbf{q}(\mathbf{u}^{i-1} + \Delta\mathbf{u}_j^i)\right] \cdot \Delta\mathbf{u}_j^i}{\overline{\Delta\mathbf{p}}^{\,i} \cdot \Delta\mathbf{u}_j^i} \tag{3.6}$$

A flowchart of the ORP code is shown in Figure 3.2. To improve the robustness of the ORP algorithm a few conditions are checked throughout the code, as indicated by the shaded "A", "B", "C" circles and first conditional statement in Figure 3.2. These conditions aim to address three important concerns when solving nonlinear problems: (1) a maximum displacement increment, (2) a displacement direction criterion, and (3) a procedure to modify the equilibrium iterations if convergence is not met.

(1) Maximum displacement increment

Near load limit points the stiffness is very small, which will result in very large incremental and iterative displacements after solving $\Delta\mathbf{u}_1^i = (\mathbf{K}_0^i)^{-1}\overline{\Delta\mathbf{p}}^{\,i}$ and $\delta\mathbf{u}_i^i = (\mathbf{K}_{j-1}^i)^{-1}\mathbf{r}_{j-1}^i$, respectively. Therefore it is necessary to impose a maximum incremental and iterative displacement. The maximum incremental displacement is calculated at the first shaded circle "A" in Figure 3.2

$$\Delta\mathbf{u}_1^i = (\mathbf{K}_0^i)^{-1}\overline{\Delta\mathbf{p}}^{\,i} \tag{3.7}$$

$$\text{If } \left\|\Delta\mathbf{u}_1^i\right\| > U_{\max} \Longrightarrow \Delta\mathbf{u}_1^i = \frac{U_{\max}}{\|\Delta\mathbf{u}_1^i\|}\Delta\mathbf{u}_1^i \tag{3.8}$$

The maximum iterative displacement is calculated at the second shaded circle "A" in Figure 3.2, i.e.

$$\delta\mathbf{u}_i^i = (\mathbf{K}_{j-1}^i)^{-1}\mathbf{r}_{j-1}^i \tag{3.9}$$

$$\text{If } \left\|\delta\mathbf{u}_j^i\right\| > U_{\max} \Longrightarrow \delta\mathbf{u}_j^i = \frac{U_{\max}}{\left\|\delta\mathbf{u}_j^i\right\|}\delta\mathbf{u}_j^i \tag{3.10}$$

(2) Displacement direction criterion

The first conditional statement in Figure 3.2 accounts for changes of direction in the displacement. If the direction of the displacement increment is reversed relative to the previously converged displacement increment then the sign of the load and displacement increments are changed. Such a reversal of direction will occur at load limit points.

(3a) Modification for convergence problems

The maximum number of iterations is specified by the user when solving nonlinear problems. If convergence is not met within those iterations, the iterative procedure can be restarted

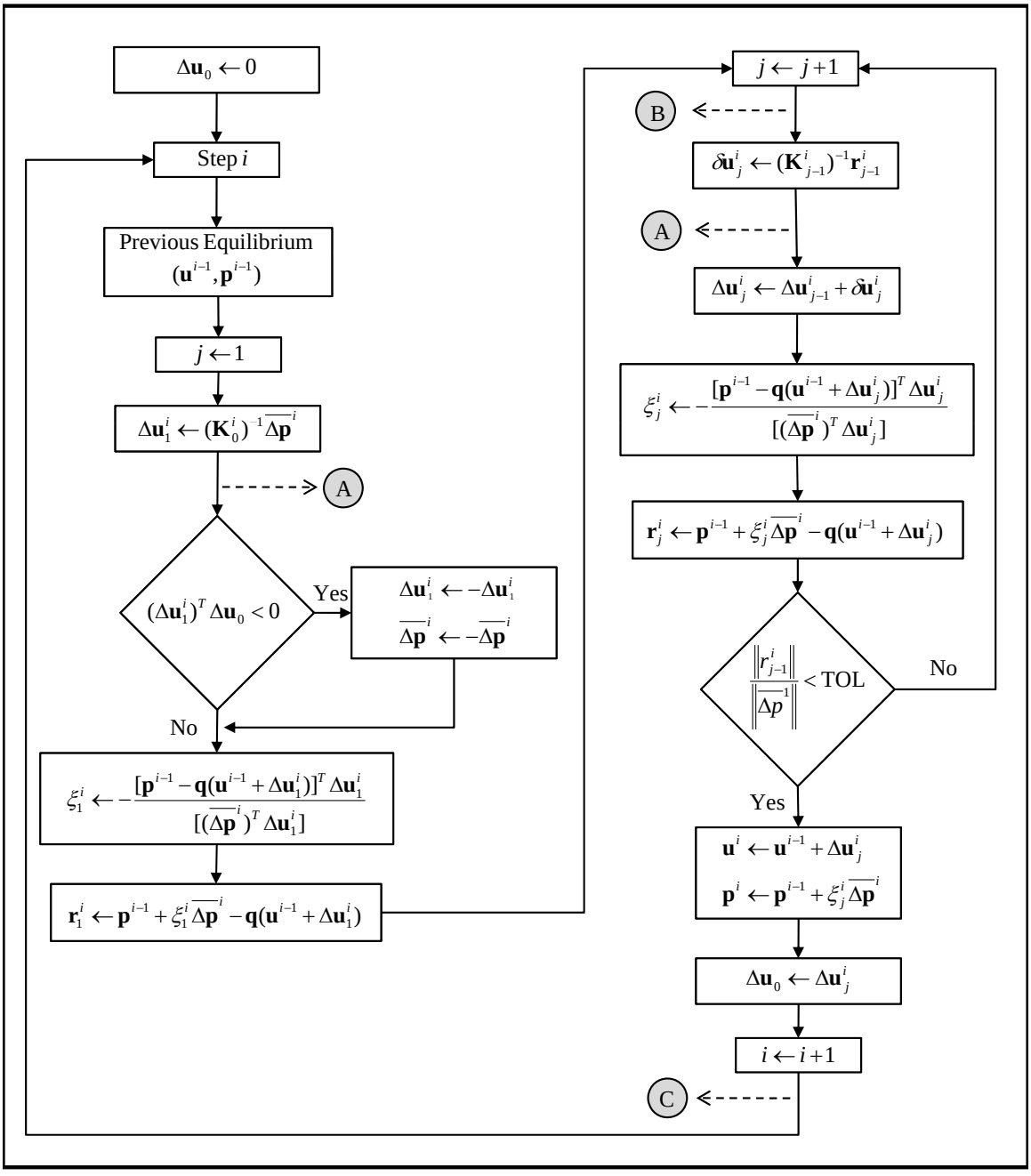

Figure 3.2: Original orthogonal residual procedure

with half the initial displacement and load increments at the shaded circle "B" in Figure 3.2. Then the maximum displacement increment imposed by shaded circle "A" would also be reduced

$$\overline{\Delta \mathbf{p}}^{\,i} \leftarrow \frac{1}{2}\overline{\Delta \mathbf{p}}^{\,i} \tag{3.11}$$

$$\Delta \mathbf{u}_1^i \leftarrow \frac{1}{2}\Delta \mathbf{u}_1^i \tag{3.12}$$

$$U_{\max} \leftarrow \frac{1}{2}U_{\max} \tag{3.13}$$

(3b) Modification for convergence problems

Information from the previous load increment may be used to overcome convergence problems in the current increment. The load increment can be scaled by a ratio of the desired number of iterations, denoted Ite_D, and the actual number of iterations used to reach equilibrium in the last incremental step, denoted Ite_A [28, 13]

$$\overline{\Delta \mathbf{p}}^{\,i} = \left(\frac{\text{Ite}_\text{D}}{\text{Ite}_\text{A}}\right)^{\alpha} \overline{\Delta \mathbf{p}}^{\,i-1} \tag{3.14}$$

as illustrated by the shaded circle "C" in Figure 3.2.

The exponent α is typically chosen to be 0.5-2.0. Of course, this procedure is dependent on the behavior of the previous step, thus it may be ineffective for problems with very sudden changes.

3.1.1 Modifications of the orthogonal residual procedure

The original orthogonal residual procedure has the potential to capture complex nonlinearities at load and displacement limit points associated with nonlinear finite element systems of equations. Krenk [53] compares the algorithm to arc-length type methods claiming that it can be more effective. Unlike some arc-length methods, the ORP iterations do not lock on a hyperplane. Locking on a hyperplane is problematic, particularly if that hyperplane does not intersect the actual equilibrium path. Since the iteration surfaces are curved, it is implied that the ORP can handle both load and displacement limit points. However, further investigation by Krenk and Hededal [54] and later by Kouhia [52] indicated that the original formulation has weaknesses around displacement limit points.

Krenk and Hededal [54] modified the procedure by imposing a second orthogonality constraint that adjusts the displacement iteration in addition to the first constraint that adjusts the load increment. The dual orthogonality procedure is derived using the original orthogonality constraint, $\mathbf{r}_j^i \cdot \Delta \mathbf{u}_j^i = 0$, and a modified Newton-Raphson approach with quasi-Newton

modifications of the stiffness matrix. When a BFGS type update with the Sherman-Morrison formula is used, the following expression for the inverse of the stiffness matrix is obtained

$$(\mathbf{K}_j^i)^{-1} = \left[\mathbf{I} - \frac{\Delta\mathbf{u}_j^i(\Delta\mathbf{q}_j^i)^T}{\Delta\mathbf{q}_j^i \cdot \Delta\mathbf{u}_j^i}\right](\mathbf{K}_0^i)^{-1}\left[\mathbf{I} - \frac{\Delta\mathbf{q}_j^i(\Delta\mathbf{u}_j^i)^T}{\Delta\mathbf{q}_j^i \cdot \Delta\mathbf{u}_j^i}\right] + \frac{\Delta\mathbf{u}_j^i(\Delta\mathbf{u}_j^i)^T}{\Delta\mathbf{q}_j^i \cdot \Delta\mathbf{u}_j^i} \tag{3.15}$$

The iterative displacement is obtained through

$$\delta\mathbf{u}_j^i = (\mathbf{K}_j^i)^{-1}\mathbf{r}_j^i \tag{3.16}$$

$$\delta\mathbf{u}_j^i = \left\{\left[\mathbf{I} - \frac{\Delta\mathbf{u}_j^i(\Delta\mathbf{q}_j^i)^T}{\Delta\mathbf{q}_j^i \cdot \Delta\mathbf{u}_j^i}\right](\mathbf{K}_0^i)^{-1}\left[\mathbf{I} - \frac{\Delta\mathbf{q}_j^i(\Delta\mathbf{u}_j^i)^T}{\Delta\mathbf{q}_j^i \cdot \Delta\mathbf{u}_j^i}\right] + \frac{\Delta\mathbf{u}_j^i(\Delta\mathbf{u}_j^i)^T}{\Delta\mathbf{q}_j^i \cdot \Delta\mathbf{u}_j^i}\right\}\mathbf{r}_j^i \tag{3.17}$$

Applying the orthogonality condition, the iterative displacement is simply

$$\delta\mathbf{u}_j^i = \left\{\left[\mathbf{I} - \frac{\Delta\mathbf{u}_j^i(\Delta\mathbf{q}_j^i)^T}{\Delta\mathbf{q}_j^i \cdot \Delta\mathbf{u}_j^i}\right](\mathbf{K}_0^i)^{-1}\right\}\mathbf{r}_j^i \tag{3.18}$$

It can be shown that this simple expression for the iterative displacement is the result of an orthogonality condition between the current displacement iteration and the internal force increment. First, note that in Quasi-Newton methods, such as the one used in this procedure, the current stiffness matrix satisfies a linear stiffness relation for an already converged set of displacements and internal force increments

$$\mathbf{K}_j^i\Delta\mathbf{u}_j^i = \Delta\mathbf{q}_j^i \tag{3.19}$$

Starting from the original orthogonality constraint, and substituting relations 3.16 and 3.19, one obtains the second orthogonality condition between the current displacement iteration and internal force increment.

$$\Delta\mathbf{u}_j^i \cdot \mathbf{r}_j^i = 0 \tag{3.20}$$

$$(\Delta\mathbf{u}_j^i)^T\mathbf{K}_j^i\delta\mathbf{u}_j^i = 0 \tag{3.21}$$

$$\Delta\mathbf{q}_j^i \cdot \delta\mathbf{u}_j^i = 0 \tag{3.22}$$

Like the ORP, the dual orthogonality procedure also requires the additional checks to improve robustness, which are listed in the previous section.

Kouhia [52] presented a stabilized version of the original ORP by relaxing the orthogonality constraint near displacement limit points. First, the load increment factors, ξ_j^i, used in the original ORP "are modified by the procedure, and thus do not add up to the total increase of load" [53]. Kouhia modified the load increment factors such that they would be additive

within an incremental step. Using the new load factor, $\delta\lambda_j^i$, one obtains the orthogonality condition

$$\Delta\mathbf{u}_j^i \cdot \left(\mathbf{r}_{j-1}^i - \delta\lambda_j^i \overline{\mathbf{p}}\right) = 0 \tag{3.23}$$

Furthermore, the orthogonality condition is relaxed

$$\Delta\mathbf{u}_j^i \cdot \left(\mathbf{r}_{j-1}^i - \delta\lambda_j^i \overline{\mathbf{p}}\right) = \tau \text{sign}(\mathbf{u}_{j-1}^i \cdot \overline{\mathbf{p}}) \left\|\mathbf{u}_{j-1}^i\right\| \|\overline{\mathbf{p}}\| \, \delta\lambda_j^i \tag{3.24}$$

where τ is a dimensionless non-negative stabilization parameter that can be related to the current stiffness parameter proposed by Bergan et. al [16], hence τ should be large at displacement limit points and small at load limit points. The norms in Equation 3.24 should be scaled with positive definite diagonal matrices to make the quantities dimensionally homogeneous. The following definition is used for the inner product

$$\cos(\mathbf{a}, \mathbf{b}) = \frac{\mathbf{a} \cdot \mathbf{b}}{\|\mathbf{a}\| \, \|\mathbf{b}\|} \tag{3.25}$$

Equation 3.24 is solved for the load factor

$$\delta\lambda_j^i = \frac{(\Delta\mathbf{u}_j^i)^T(\mathbf{r}_{j-1}^i)}{\text{sign}((\mathbf{u}_{j-1}^i)^T\overline{\mathbf{p}}) \left\|\mathbf{u}_{j-1}^i\right\| \|\overline{\mathbf{p}}\| \left(\left|\cos(\Delta\mathbf{u}_j^i, \overline{\mathbf{p}})\right| + \tau\right)}$$

Kouhia demonstrated the improvement of the stabilized version over the original ORP through numerical examples.

3.2 Formulation into N+1 dimensional space

The $N+1$ dimensional space unifies multiple nonlinear solution schemes through the load increment factor, $\Delta\lambda_j^i$. However, the load increment factor, ξ_j^i, used in ORP is not used in the same context. In the ORP theory the load increment factors are calculated at each iteration and are not dependent on any previous load increment factor. In the N+1 dimensional space each load increment factor is dependent on the previous, similarly to the modification made by Kouhia, i.e.

$$\Delta\lambda_j^i = \Delta\lambda_{j-1}^i + \delta\lambda_j^i \tag{3.26}$$

Figure 3.3 shows the dependence of $\Delta\lambda_j^i$ on previous iterations, while ξ_j^i is independent at each iteration.

To unify the ORP into the $N+1$ dimension space, the load factor, $\Delta\lambda_j^i$, must be included in the constraint equation. From Figure 3.3 the equivalence between the load factors can be

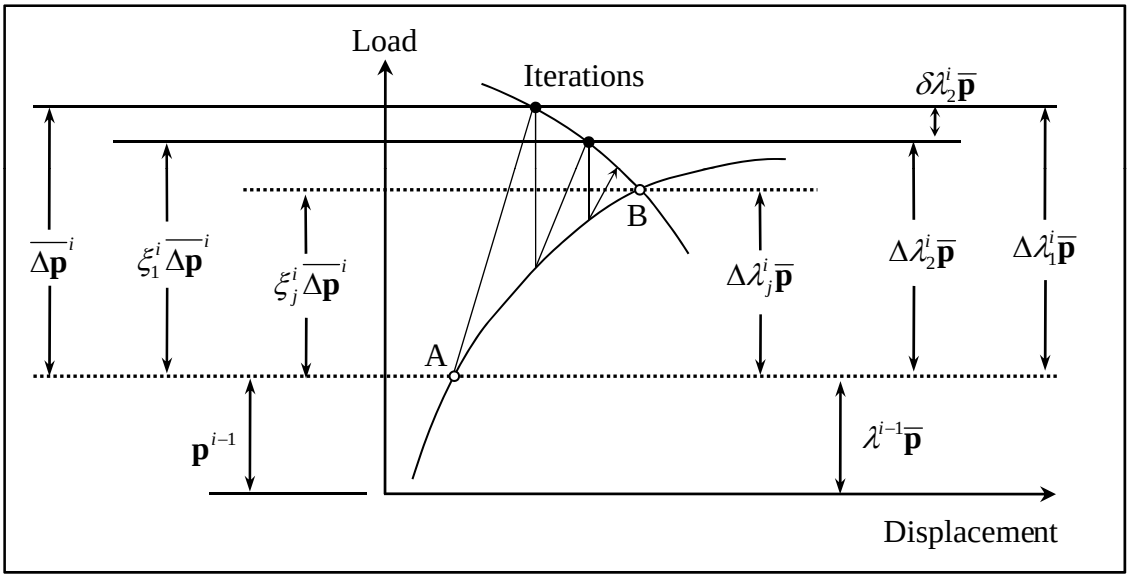

Figure 3.3: Comparison of orthogonal residual procedure load factor and unified schemes load factor

expressed as

$$\mathbf{p}^{i-1} + \xi_j^i \overline{\Delta\mathbf{p}}^{\,i} = \lambda^{i-1}\overline{\mathbf{p}} + \Delta\lambda_{j-1}^i\overline{\mathbf{p}} + \delta\lambda_j^i\overline{\mathbf{p}} \tag{3.27}$$

Then this is inserted into Equation 3.5, and the residual becomes

$$\mathbf{r}_j^i = (\lambda^{i-1} + \Delta\lambda_{j-1}^i + \delta\lambda_j^i)\overline{\mathbf{p}} - \mathbf{q}(\mathbf{u}^{i-1} + \Delta\mathbf{u}_j^i) \tag{3.28}$$

Finally, one can solve for the load factor, $\delta\lambda_j^i$, to obtain

$$\delta\lambda_j^i = \frac{\left[\mathbf{q}(\mathbf{u}^{i-1} + \Delta\mathbf{u}_j^i)\right]\cdot\Delta\mathbf{u}_j^i}{(\overline{\mathbf{p}}\cdot\Delta\mathbf{u}_j^i)} - (\lambda^{i-1} + \Delta\lambda_j^{i-1}) \tag{3.29}$$

Rearranging terms, Equation 3.29 can be written in the form of the $N+1$ dimensional space constraint, Equation 2.13,

$$\left[\mathbf{q}(\mathbf{u}^{i-1} + \Delta\mathbf{u}_j^i)\right]\cdot\delta\mathbf{u}_j^i - \overline{\mathbf{p}}\cdot\Delta\mathbf{u}_j^i\delta\lambda_j^i = \lambda^{i-1} + \Delta\lambda_j^{i-1} - \left[\mathbf{q}(\mathbf{u}^{i-1} + \Delta\mathbf{u}_j^i)\right]\cdot\Delta\mathbf{u}_j^{i-1} \tag{3.30}$$

Thus, the constraint parameters are

$$\mathbf{a}_j^i = \begin{cases} 0 & \text{for } j = 1 \\ \mathbf{q}(\mathbf{u}^{i-1} + \Delta\mathbf{u}_j^i) & \text{for } j \geq 2 \end{cases} \tag{3.31}$$

$$b_j^i = \begin{cases} 1 & \text{for } j = 1 \\ -\overline{\mathbf{p}} \cdot \Delta\mathbf{u}_j^i & \text{for } j \geq 2 \end{cases} \tag{3.32}$$

$$c_j^i = \begin{cases} \overline{\delta\lambda} & \text{for } j = 1 \\ \lambda^{i-1} + \Delta\lambda_j^{i-1} - \left[\mathbf{q}(\mathbf{u}^{i-1} + \Delta\mathbf{u}_j^i)\right] \cdot \Delta\mathbf{u}_j^{i-1} & \text{for } j \geq 2 \end{cases} \tag{3.33}$$

where $\overline{\delta\lambda}$ is the prescribed initial load parameter. A flowchart of the steps to compute the ORP load factor in the context of the unified schemes is shown in Figure 3.4. After the load factor is calculated, the function computes the displacement increment to be used in the next calculation of the load factor. The maximum displacement increment is calculated at the first iteration of the first step using the initial load parameter and an initial scale factor, β, both prescribed by the user. The load parameter and displacement increments are then scaled by a ratio of the magnitude of the new displacement increment to the maximum displacement increment.

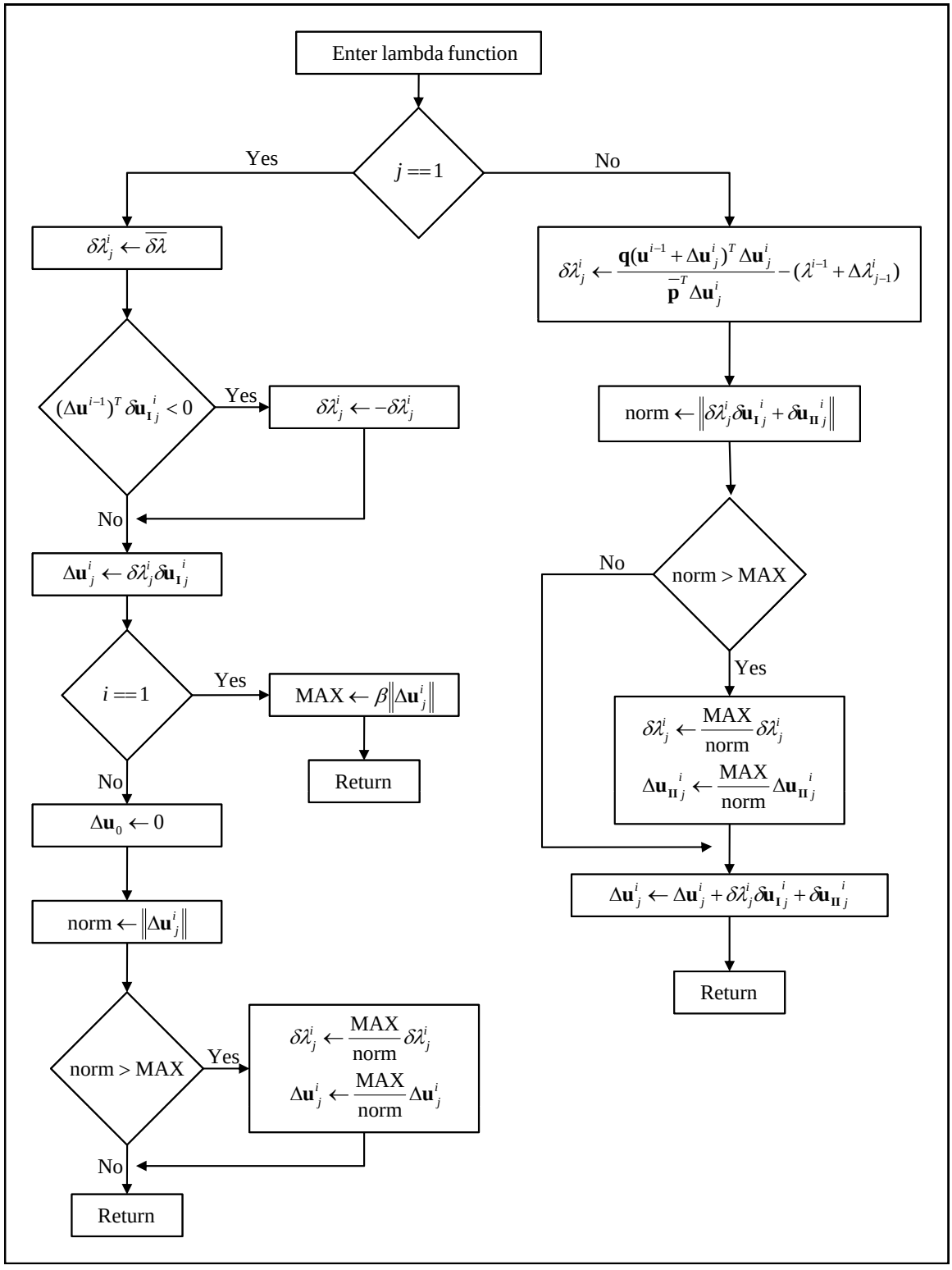

Figure 3.4: Orthogonal residual procedure load factor in the unified schemes

Chapter 4

Computational implementation

The unified approach to nonlinear solution schemes for the finite element method lends itself well to an object-oriented implementation. This is due in part to the fact that the incremental-iterative process is generalized in such a way that many nonlinear solution schemes can be formulated into the N+1 dimensional space. The purpose of this Chapter is to outline the computational implementation of the unified schemes and demonstrate its usability in solving nonlinear finite element problems.

4.1 Object-oriented programming

Object oriented programming for the finite element method was originally introduced to overcome limitations of conventional finite element analysis software [41]. Two series of articles by Zimmermann and co-authors discuss the concepts of object oriented programming and the implementation of a finite element code in object-oriented programming languages [101, 35, 34, 102, 37, 38, 39]. Mackie [61] demonstrated the benefits and advantages of object oriented programming for the finite element method, with emphasis on inheritance and virtual methods. Scholz [86] presented a finite element program with vector and matrix classes to represent and manipulate quantities symbolically. Donescu [33] generalized the idea of object-oriented finite element codes for a large range of initial/boundary value problems, rather than for specialized problems.

In more recent years object-oriented programming techniques have been used to improve the data storage associated with finite element codes. Celes et al. [21] developed a topological data structure, TopS, for efficient representation of finite element meshes. In fact, TopS was further improved using parallel computing with an object-oriented framework [36]. Recently, Heng [48] reviewed object-oriented programming over the past 15 years and introduced a method for finite element programs using software design patterns to improve software quality and reduce development time.

The present discussion of object-orient finite element programing is not meant to be a comprehensive review. For a more detailed discussion the reader is directed to the technical

literature. In the remainder of this section the features of object-oriented programming that are pertinent to the implementation of the unified schemes will be discussed [85].

Object

Objects consist of data and functions, which performs actions on the data. The object encapsulates the data, called instance variables, and the functions in order to preserve each from outside interference. The unified schemes were implemented such that three main objects are utilized: a model, a linear solver and a nonlinear algorithm. The model object contains variables to describe the model, such as the number of equations, and functions to operate on the model, such as one to compute the internal force vector. Similarly, the linear solver object contains all necessary functions to solve the linear system $\mathbf{Ax} = \mathbf{b}$. Finally, the nonlinear algorithm object holds the parameters input by the user and a function to compute the load factor.

Class

The *class* is the foundation of object-oriented programming. It defines a new data type through functions that operate on its data. Objects are instances of classes, so a class is essentially a plan describing how to build an object. The model, linear solver, and nonlinear algorithm objects described in the previous section are instances of the model, linear solver, and nonlinear algorithm classes, respectively. Each class contains the definitions of what the object will be.

Functions

Functions are the building blocks of object-oriented programs. They are simply subroutines that contain programming statements to perform tasks on data. The unified schemes are built of a single executable function, called "main", and several internal functions. The "main" function collects the inputs from the user, then builds the model, linear solver, and nonlinear algorithm objects from their respective class definitions, and finally calls the "solver" function, which performs analysis. The nonlinear computations are performed through the "solver" function, rather than the "main" function, which allows for a clean, and easily readable code.

Inheritance

Inheritance is the process whereby a new object is derived from an existing one by acquiring properties from the existing object. The new derived (child) class inherits data and functions

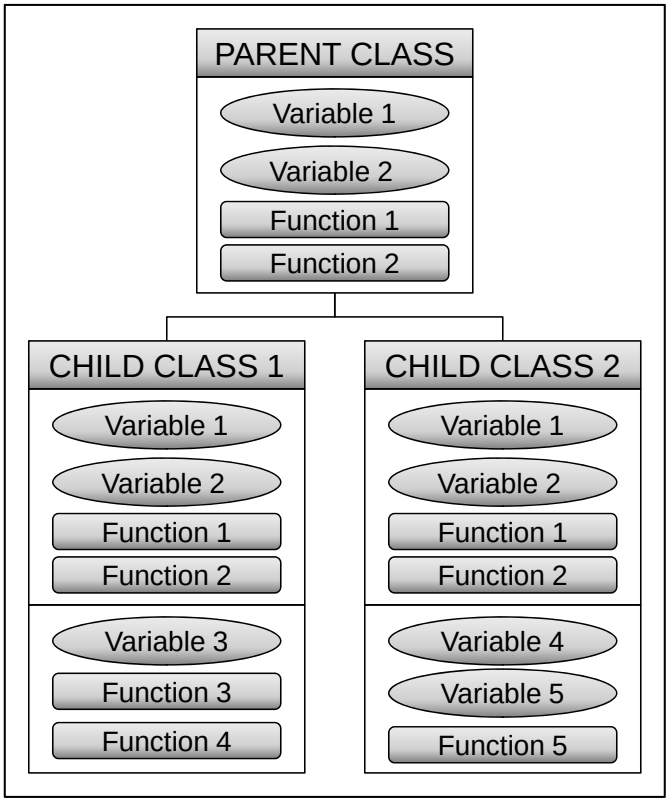

Figure 4.1: Class hierarchy through inheritance

from its parent class. In this way, inheritance supports a hierarchical program design, as shown in Figure 4.1. The unified schemes naturally feature a hierarchical design - the nonlinear algorithm classes are children of a parent nonlinear algorithm class. Rather than each nonlinear algorithm class defining its own number of equations, it simply refers to its parent class, thus the number of equations only needs to be stored in one place. Conversely, the function to compute the load factor must be implemented in each nonlinear algorithm class because it is computed differently for each.

4.2 The NLS++

A simple, robust, and object-oriented (C++) computer program, called NLS++ (NonLinear Solver), implements the nonlinear solution schemes discussed in previous Chapters. Through the unified approach, the solvers share a common interface and only vary in the computation of the load parameter, which is determined by each solver's unique constraint Equation [55]. This module is used to thoroughly test the solution schemes and characterize their performance in capturing various nonlinearities. In this section, the structure and usage of the NLS++ code are discussed.

4.2.1 Class hierarchy

The NLS++ code is organized into three distinct components, each of which consists of a family of classes: Model, Control, and Linear Solver. A separate application class creates instances of these components to execute the nonlinear analysis.

Model class

All model classes inherit properties from the parent model class, whose purpose is to represent the finite element model of the system to be solved. The primary function of the model class is to compute and store information associated with the system, including the tangent matrix, and internal and external load vectors. Because the model class stores the system tangent matrix, it also stores information needed to solve the linear system (e.g. profile and sparsity pattern).

Three element types currently supported in NLS++ are bar, two-dimensional beam and three-dimensional beam elements. Rather than using structural elements, the user can also define the nonlinear problem directly in terms of a nonlinear function, in which case the function class would be used. New elements are easily added by creating a new child class that inherits the instance variables and functions from the parent class. The benefit of the object-oriented approach is evident here because inheritance and polymorphism are heavily relied upon. Consider the following example: when the application class needs the tangent matrix to solve $\mathbf{Ku} = \mathbf{f}$, it calls the tangent matrix function of the parent class. However, the actual computation of the tangent matrix is done in the particular child class. The parent class serves as the interface, and the application knows nothing of the child class. Therefore a single application class can run any number of elements, provided they are children of the parent model class. Figure 4.2 illustrates the model class hierarchy with the supported elements as children. As shown in Figure 4.2, a function or new elements is easily represented in NLS++ through the variables and functions inherited from the parent model class.

Linear solver class

The purpose of the linear solver class is simply to solve the linear system, i.e. $\mathbf{Ku} = \mathbf{f}$. Child classes of the parent linear system class are different implementations of linear solvers and include Crout and conjugate gradient solvers. For the purpose of this work, the Crout solver was used as the primary linear solver for symmetric systems. One small non-symmetric example is presented in Section 5.2; a simple 2×2 solver, in which the stiffness matrix is inverted (tenable for 2×2 systems), was utilized. Please refer to Section 6.1 for a discussion on future improvements to handle general non-symmetric systems.

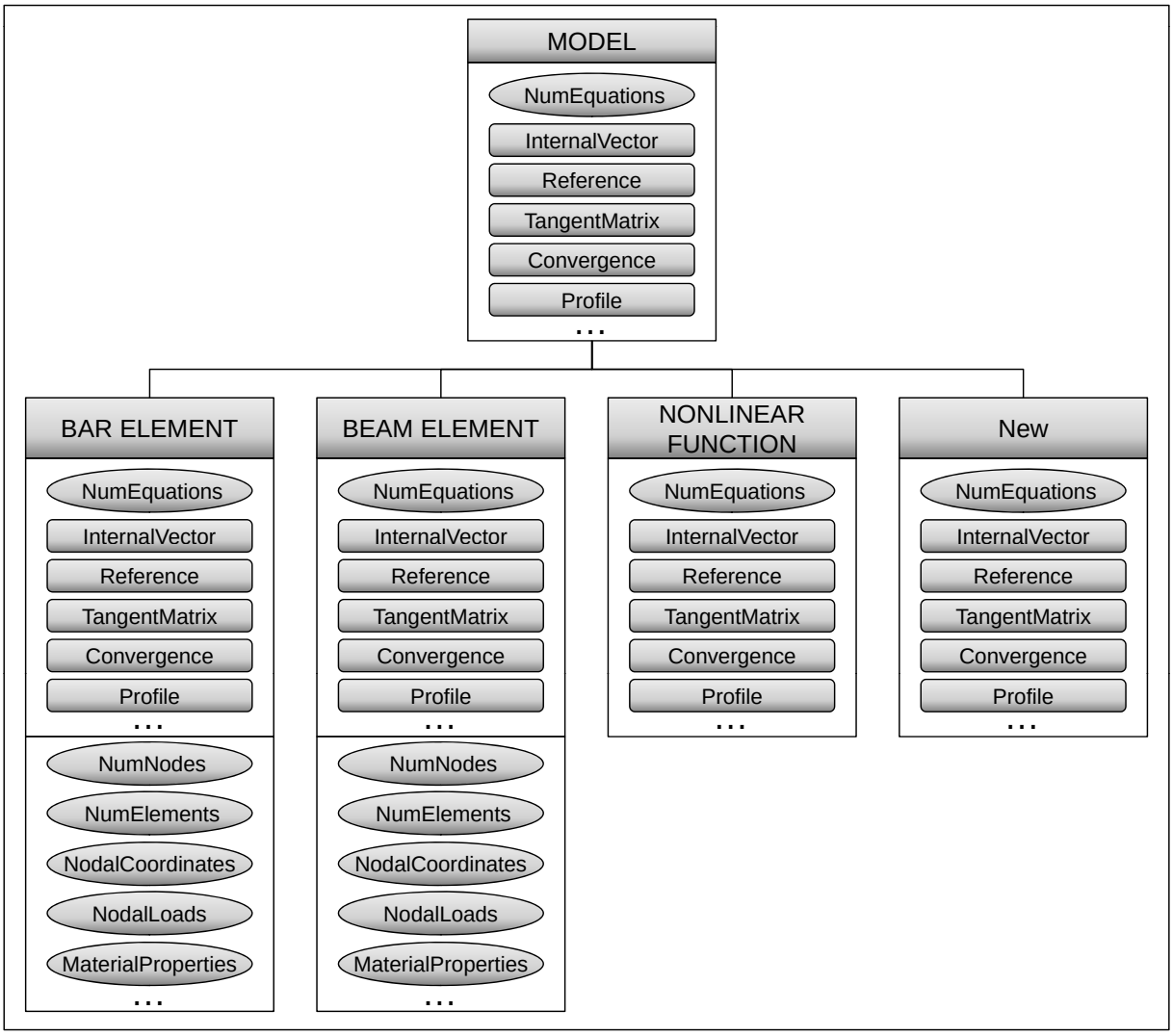

Figure 4.2: Model class hierarchy

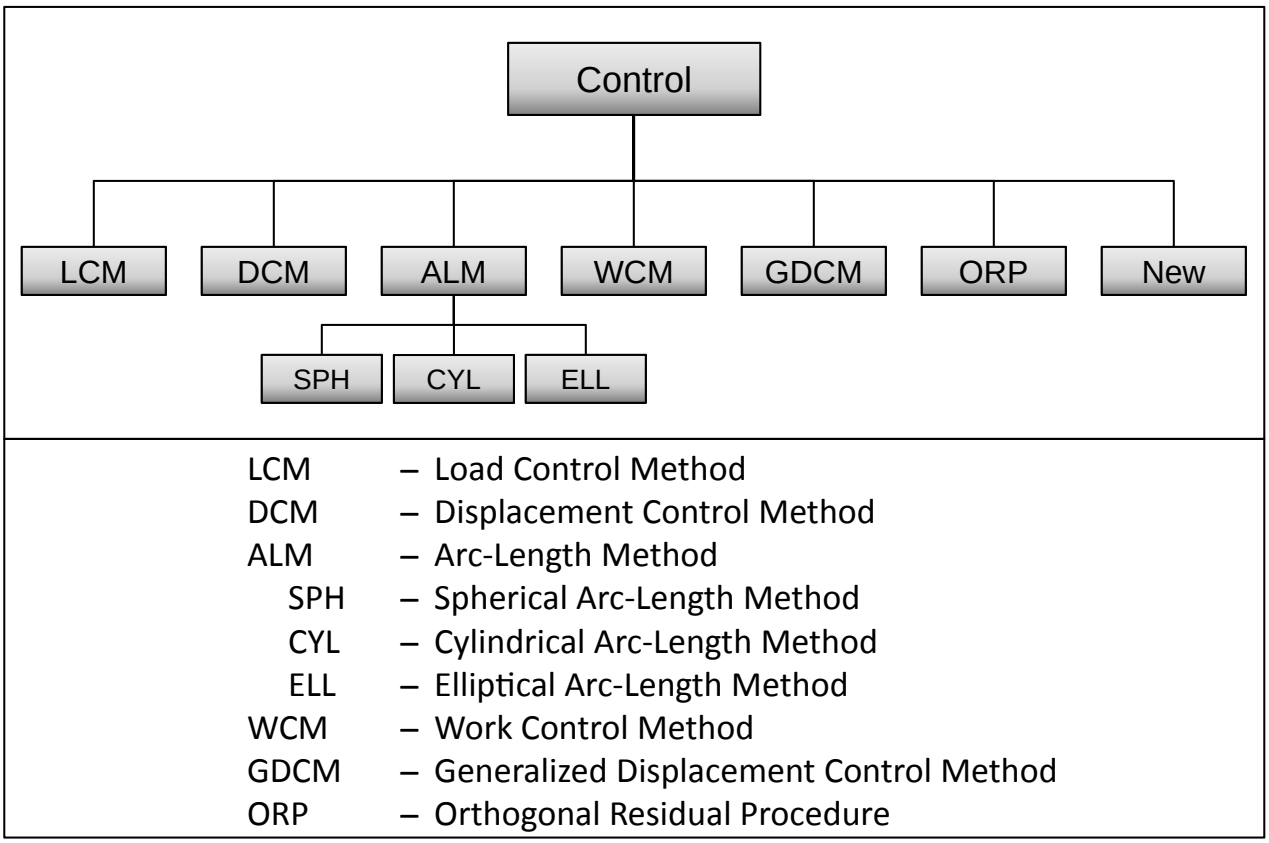

Figure 4.3: Control class hierarchy

Control class

The parent control class is the engine of the NLS++ code, in the sense that it has the function that traces the equilibrium path via the incremental-iterative procedure. Children of the parent control class are the nonlinear solution schemes discussed in the previous sections, as shown in Figure 4.3. The child classes have one function that is not implemented in the parent class, called `Lambda`, which computes the load factor for that nonlinear scheme. The incremental-iterative procedure is implemented in the parent control class and is shared among all the nonlinear solution schemes. As shown in Figure 4.3, a new nonlinear solution scheme is easily implemented, provided the constraint condition can be written in the form of Equation 2.13.

The incremental-iterative procedure implemented in the parent control class is illustrated in Figure 4.4. The flow chart represents the operations performed on the j^{th} iteration of the i^{th} incremental step. The diamond-shaped boxes are conditional statements, and the shaded elliptical box is the only part that changes for each algorithm.

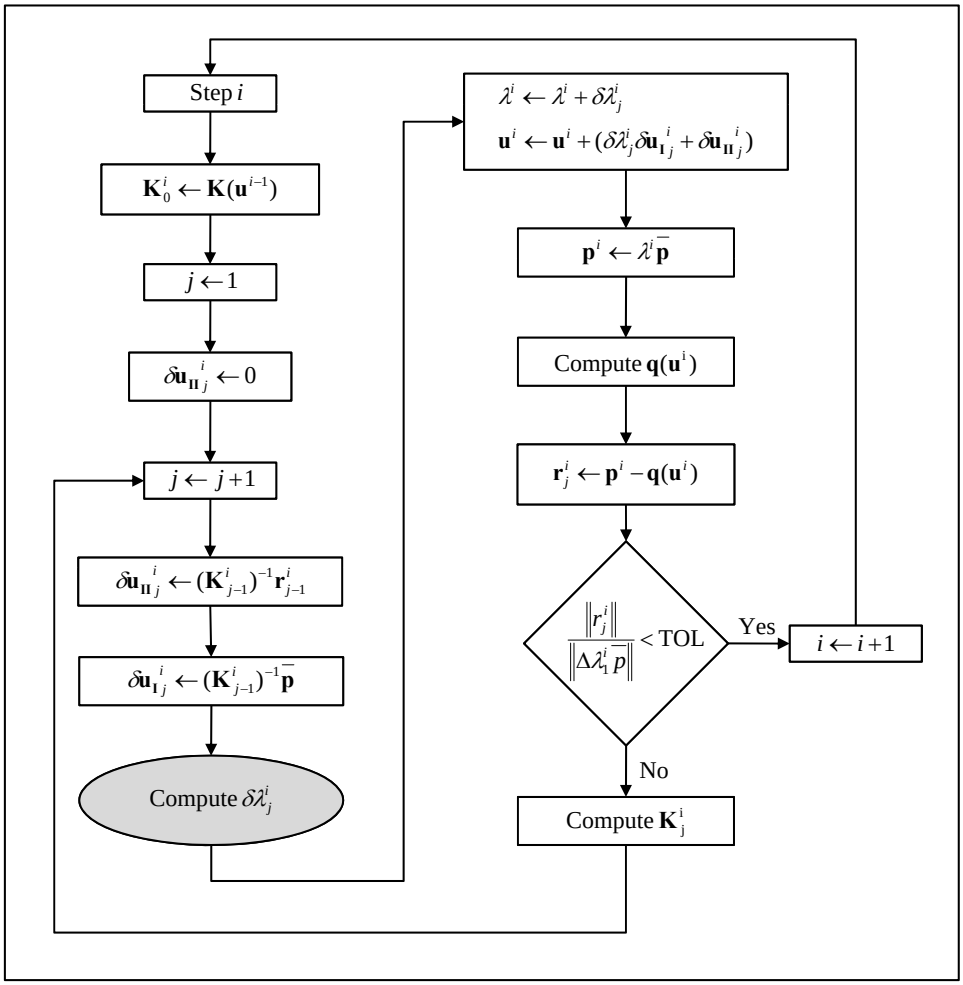

Figure 4.4: Incremental-iterative procedure of the unified scheme

Table 4.1: Application class

1. Read input files
2. Create a new model
3. Initialize model
4. Create a new linear solver
5. Create a new control for nonlinear solver
6. Call solver function of control

4.2.2 Implementation into finite element analysis software

The object oriented nature of NLS++ makes it a good candidate for implementation into the object oriented finite element analysis software, TopFEM (see section 6.1). The integration will require a child of the existing analysis class in TopFEM be developed. Rather than utilize the limited TopFEM nonlinear solver, it would instead call the solver functions of NLS++. Therefore, the model would be represented by TopFEM, but the nonlinear solution computations would take place in NLS++. No modification to the original codes are required, and only an additional class is needed for the two code to communicate. This is the desirable solution because each code could still be run independently.

4.2.3 NLS++ usage

The simple class hierarchy of NLS++ was designed as such so that the code would be executed by a simple application class, requiring minimal work from the end user. Essentially, new instances of the three classes discussed in the previous section are created, the incremental-iterative procedure is called, and the analysis is executed. Table 4.1 lists the steps of the application class.

Data is passed into NLS++ via various input files: a model file and algorithm file. The model file contains information about the finite element model including the element type (i.e., bar or beam), finite element mesh (i.e. nodes, elements, connectivity), boundary conditions, applied loads and displacements, and material properties. The algorithm file contains information about the nonlinear solution scheme including type, initial control factor, maximum number of steps and iterations, convergence tolerance, and type of linear solver. Table 4.2 shows initial control factor required for each nonlinear solution scheme.

Additional inputs required for a few of the nonlinear solution schemes are given in table 4.3.

Table 4.2: Nonlinear solution scheme inputs

Algorithm type	**Initial control factor**
Load control method	Load increment, $\overline{\Delta\lambda}^{\,i}$
Displacement control method	Displacement increment, $\overline{\Delta u}^{\,i}$
Arc-length control method	Arc-length increment, $\overline{\Delta S}^{\,i}$
Work control method	Work increment, $\overline{\Delta W}^{\,i}$
Generalized displacement control method	Load parameter, $\delta\lambda_1^1$
Orthogonal residual procedure	Iterative load increment, $\overline{\delta\lambda}$

Table 4.3: Additional inputs for selected nonlinear solution schemes

Algorithm type	**Additional parameter 1**	**Additional parameter 2**
Displacement control method	Control degree of freedom	Constant or variable displacement
Arc-length control method	Constant or variable arc-length	n/a
Orthogonal residual procedure	Initial incremental scale factor, β	n/a

Chapter 5

Applications and examples

The ability of the solution schemes to capture nonlinear behavior is tested in this chapter with five examples. The examples include two functions: one uni-dimensional and one multi-dimensional, and three structural systems: the Von Mises Truss, 12 Bar Truss, and Lee Frame. These examples were chosen because they feature a host of nonlinearities, including load and displacement limit points and large fluctuations in stiffness. The structural systems exhibit geometrically nonlinear behavior by means of very large displacements. It should be noted that material nonlinearity is not explicitly explored in the following examples; please refer to Section 6.1 for suggestions on examples solving material nonlinear problems with NLS++.

Unless otherwise stated, all computation results were obtained with a maximum of 40 iterations per step and a convergence tolerance (see Figure 4.4) of 10^{-4} assigned to each algorithm. In general, all algorithms used a standard, rather than modified, update to the stiffness matrix.

5.1 Uni-dimensional function

The first example to test the nonlinear solution schemes is a single-degree-of-freedom problem. The function was used by Chen and Blandford [24] to test their work increment control method. While the function is very simple to incorporate into NLS++, it features some complex nonlinearities such as two load limit points and an infinite slope, thus making it a good candidate to evaluate the nonlinear solution schemes.

The uni-dimensional function is given by

$$f(u) = -3\text{sign}(u)|u|^{\frac{1}{3}} + 4u + 1 \tag{5.1}$$

where the sign function is defined as

$$\text{sign}(x) = \begin{cases} -1 & \text{for } x < 0 \\ 0 & \text{for } x = 0 \\ 1 & \text{for } x > 0 \end{cases} \tag{5.2}$$

The internal load is simply given by Equation 5.1. Derivation of Equation 5.1 with respect to the degree of freedom, u, gives the tangent stiffness.

$$\frac{df}{du} = -\frac{1}{|u|^{\frac{2}{3}}} + 4 \tag{5.3}$$

From Equation 5.3, it is clear that load limit points (horizontal tangents) will occur at $u = \pm\frac{1}{8}$, and that the slope will be infinite at $u = 0$. The exact solution is plotted in Figure 5.1.

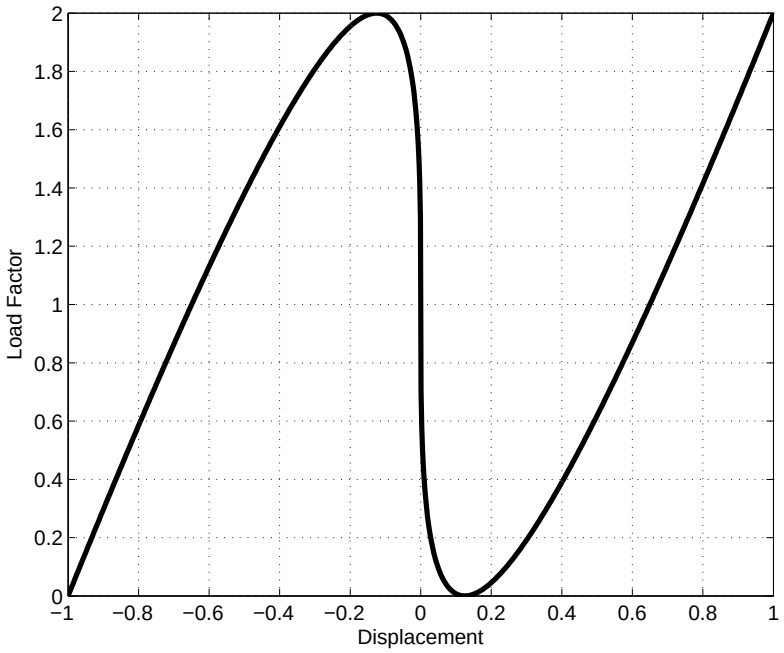

Figure 5.1: Exact solution to uni-dimensional function

5.1.1 Computational results

Each solution scheme was applied to the uni-dimensional function; the results are shown in Figures 5.2-5.4. As expected, the load control method could not capture the full behavior

at the load limit points. When the load factor reaches the first local maximum of 2, the method continues to increase load, therefore snapping through the softening behavior shown in Figure 5.1. Since the load can only increase with the load control method, only stiffening behavior can be captured, as shown in Figure 5.2.

The remainder of the nonlinear solution schemes fully captured the behavior, as shown in Figures 5.3 and 5.4. Again, this behavior is expected because all of the methods are capable of capturing load limit points. It should be noted that methods that have difficulty near displacement limit points (i.e. displacement control method and work control method) were able to capture the full behavior of this example, because even though there is a vertical tangent at $u = 0$, there is no snap-back.

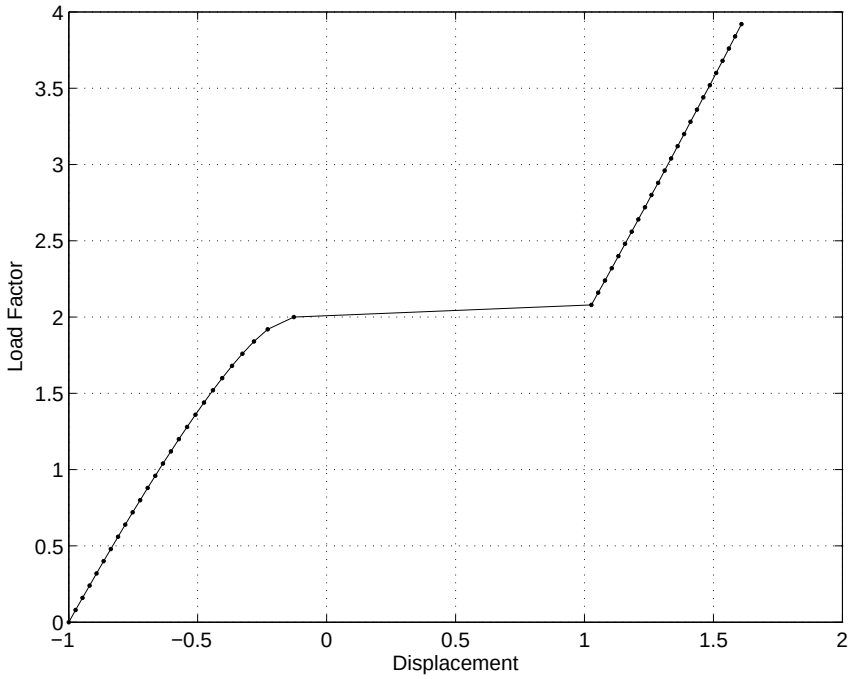

Figure 5.2: Solution to the uni-dimensional function example using the LCM

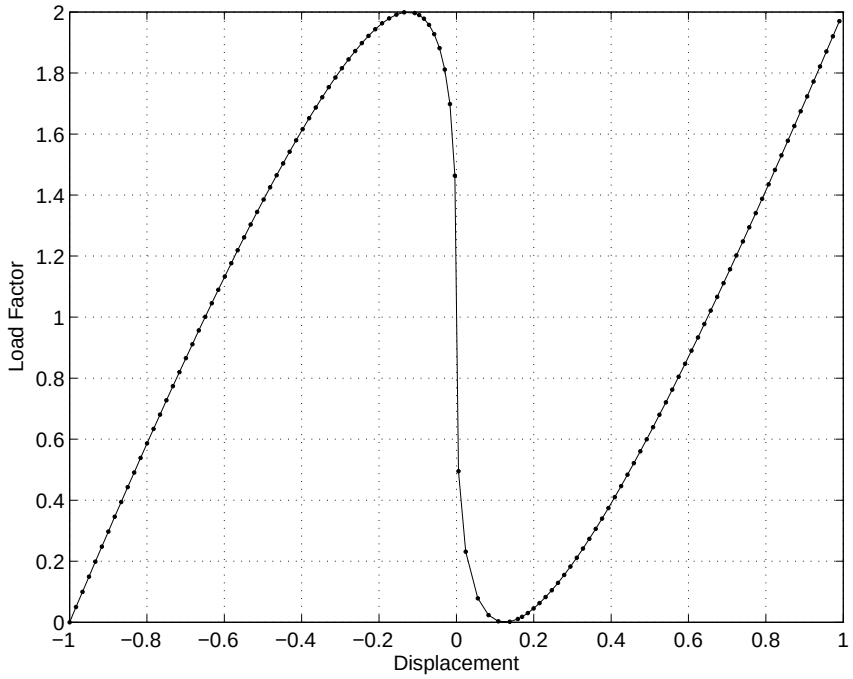

Figure 5.3: Solution to the uni-dimensional function example using the DCM, WCM, GDCM, and ORP (GDCM solution shown)

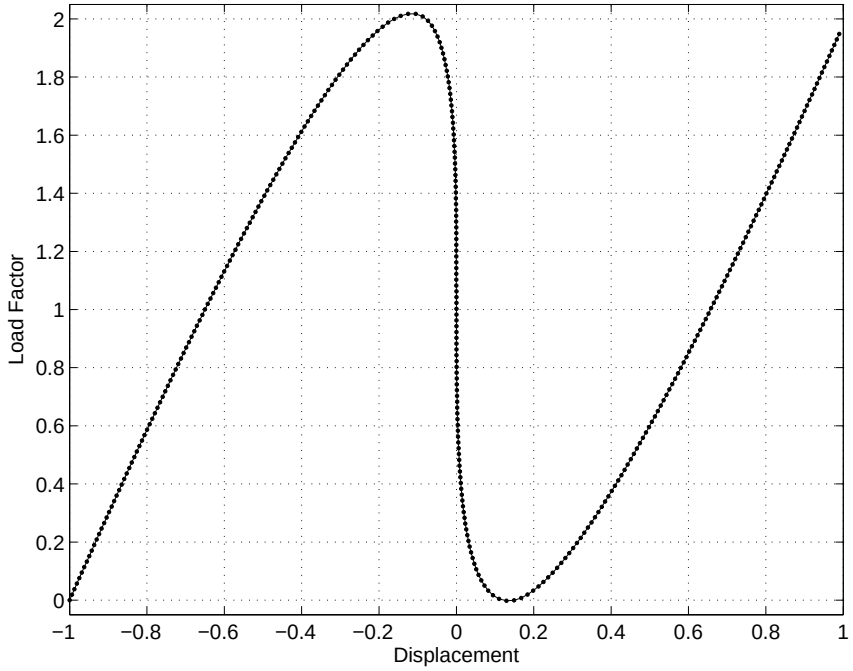

Figure 5.4: Solution to the uni-dimensional function example using the ALCM

The parameters adopted in each algorithm and resulting behavior are given in Table 5.1. The

spherical version of the arc-length method (i.e. $\eta = 1$) was employed in this example and all subsequent examples. It should be noted that a fairly small control factor and large number of steps were required for the arc-length control method. There is very small change in the displacement corresponding to a very large change in the load near the vertical tangent. Other methods that increment the load and displacement independently can recover the curve between the maximum and minimum load with very few steps, as shown in Figure 5.3. However, the step size in the constant update arc-length control method is defined by an arc of the same radius at every step, so the method cannot make large jumps in either load or displacement in one step. Therefore, several steps at small increments are required, as shown in Figure 5.4. The tolerance was also adjusted to 10^{-1} to allow for the largest possible step size while still recovering the correct solution path.

Table 5.1: Summary of the uni-dimensional function example

Algorithm	**Max. steps**	**Control factor**	**Scale factor**	**Converged/diverged**
LCM	50	0.08	n/a	Snaps through
DCM	100	0.02	n/a	Fully converged
ALCM*	325	0.02	n/a	Fully converged
WCM	95	0.001	n/a	Fully converged
GDCM	60	0.1	n/a	Fully converged
ORP	55	1.0	0.1	Fully converged

* Convergence tolerance = 10^{-1}

5.2 Two-dimensional function

The next example to test the nonlinear solution schemes is a function of two variables, i.e. u_1 and u_2. It features complex nonlinearities including load and displacement limit points for both degrees of freedom.

The problem is given by a vector of external and internal forces [90], shown below

$$\mathbf{p} = \begin{pmatrix} 40 \\ 15 \end{pmatrix} \tag{5.4}$$

$$\mathbf{q}(\mathbf{u}) = \begin{pmatrix} 10u_1 + 0.4u_2^3 - 5u_2^2 \\ 0.4u_1^3 - 3u_1^2 + 10u_2 \end{pmatrix} \tag{5.5}$$

Derivation of the internal force vector in Equation 5.5 with respect to the degrees of freedom,

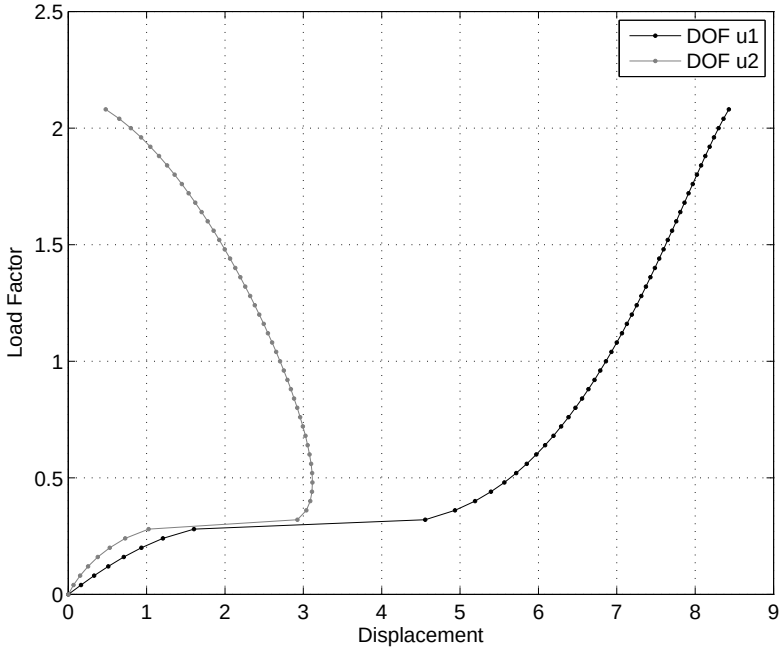

Figure 5.5: Solution to the two-dimensional function example using the LCM

u_1 and u_2, gives the components of the tangent stiffness matrix

$$K_{ij} = \frac{\partial q_i}{\partial u_j} \tag{5.6}$$

$$\mathbf{K}(\mathbf{u}) = \begin{bmatrix} 10 & 1.2u_2^2 - 10u_2 \\ 1.2u_1^2 - 6u_1 & 10 \end{bmatrix} \tag{5.7}$$

5.2.1 Computational results

The equilibrium curves traced by each of the nonlinear solution schemes are shown in Figures 5.5-5.11. The load control method snapped through at the first limit points on each curve and diverged at the next load limit point encountered, as shown in Figure 5.5. The snap through behavior is expected because the load continues to increase after the initial load limit points, hence the solution scheme was able to converge to the next points. Next, a displacement limit point occurs in the second degree of freedom, however since the load is still increasing, the load control method captured this behavior. Divergence occurred at the next load limit point in the second degree of freedom. Snap through behavior could not have occurred because the load only decreases after this point.

Figures 5.6 and 5.7 show the equilibrium paths obtained using the displacement control

method with the first and second degree of freedom as the control degree of freedom, respectively. In both cases, the first load limit points are captured by the method, which is expected because the displacement, not load, is incremented at each step. In general, the displacement control method captures behavior up to a displacement limit point in the control degree of freedom. In Figure 5.6, the equilibrium path is traced until the displacement limit point in the first degree of freedom is reached. Notice that the displacement limit point in the second degree of freedom is captured. This is due to the fact that the displacement is only incremented for the first degree of freedom. In Figure 5.7 however, the solution scheme diverges much earlier at the displacement limit point of the second degree of freedom.

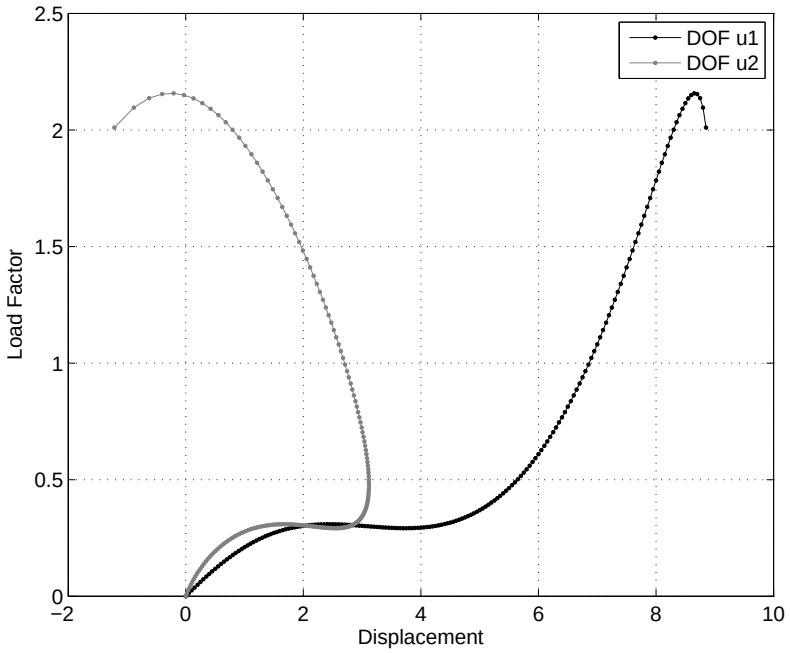

Figure 5.6: Solution to the two-dimensional Function example using the DCM with $\mathbf{u}_1$ as the control displacement

The variable displacement control method, discussed in Chapter 2, was also employed and the results are shown in Figure 5.8. The entire equilibrium path was traced with the method because the control degree of freedom changes automatically when a displacement limit point is approaching in that degree of freedom. The control displacements and snap back locations are listed in Table 5.2 and correspond to the labels in Figure 5.8. From steps 1-73 the control displacement is u_1, and a displacement limit points is encountered at step 58 in degree of freedom u_2 (Label 1 in 5.8). At step 74 the control displacement changes to u_2, and snap back is captured in degree of freedom u_1 at step 158 (Label 2 in 5.8). The final change is control degree of freedom occurs in step 217, after which point two snap back points are

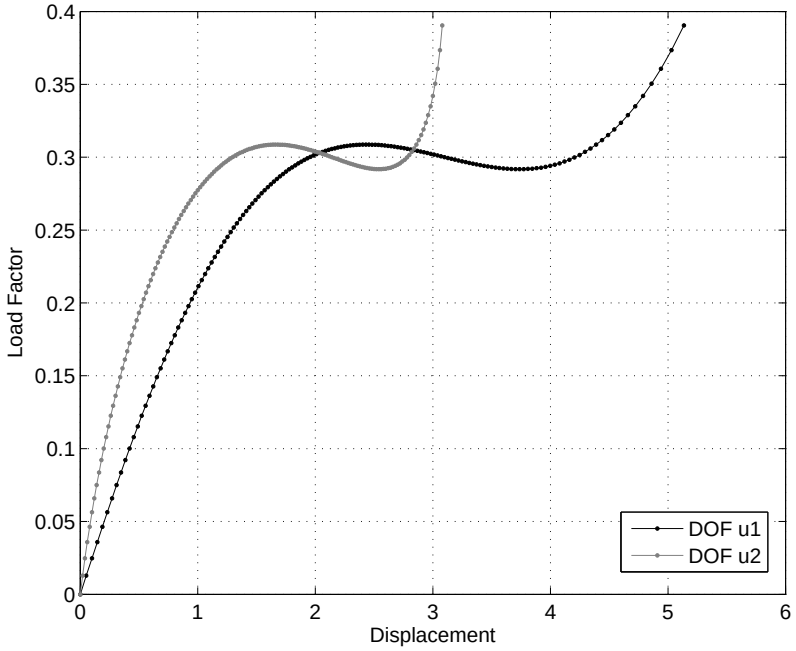

Figure 5.7: Solution to the two-dimensional function example using the DCM with $\mathbf{u}_2$ as the control displacement

passed in degree of freedom u_2 (Labels 3 and 4 in 5.8). These automatic changes in the control degree of freedom allow the method to recover the entire solution path.

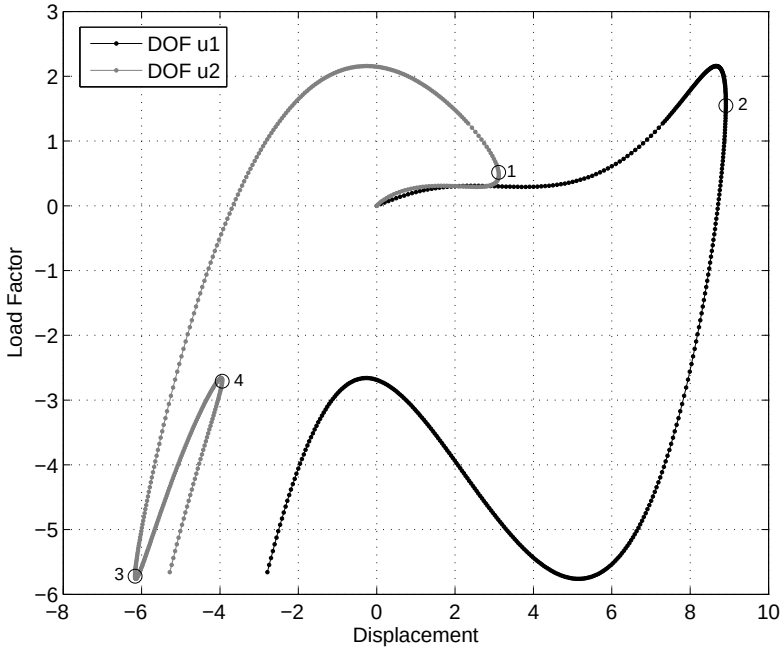

Figure 5.8: Solution to the two-dimensional function example using the variable DCM

Table 5.2: Summary of the variable DCM for the two-dimensional function

Step	**Control displacement**	**Snap back**
1-73	u_1	Step 58 in u_2 (Label 1 in Figure 5.8)
74-216	u_2	Step 158 in u_1 (Label 2 in Figure 5.8)
217-445	u_1	Step 266 in u_2 (Label 3 in Figure 5.8)
		Step 398 in u_2 (Label 4 in Figure 5.8)

Figures 5.9 and 5.10 show the results using the work control method and orthogonal residual procedure, respectively. Both methods fail to capture the equilibrium curve beyond the second set of load limit points. The work control method fails because the sign of the load increment factor, $\delta\lambda_j^i$, oscillates at every iteration (j) until the maximum number of iterations is reached. Similarly, the sign of the load increment factor in the orthogonal residual procedure artificially changes, and the method begins tracing the previously converged solution curve. This behavior in the orthogonal residual procedure requires more investigation into the computational implementation. Please refer to Chapter 6 for further discussion.

Finally, Figure 5.11 shows the full equilibrium path trace by both the arc-length control method and the generalized displacement control method. Table 5.3 shows the parameters used and behavior captured for this two degree of freedom example. Notice in the table that

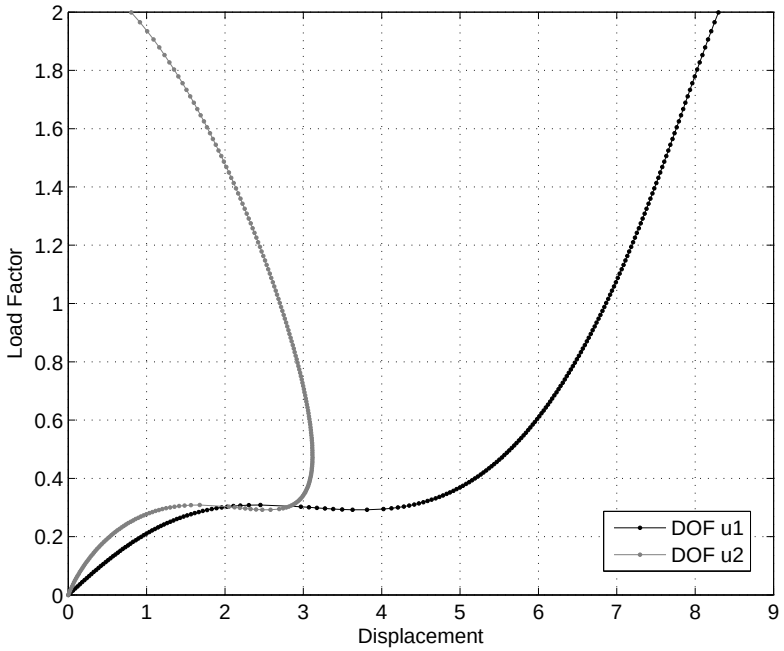

Figure 5.9: Solution to the two-dimensional function example using the WCM

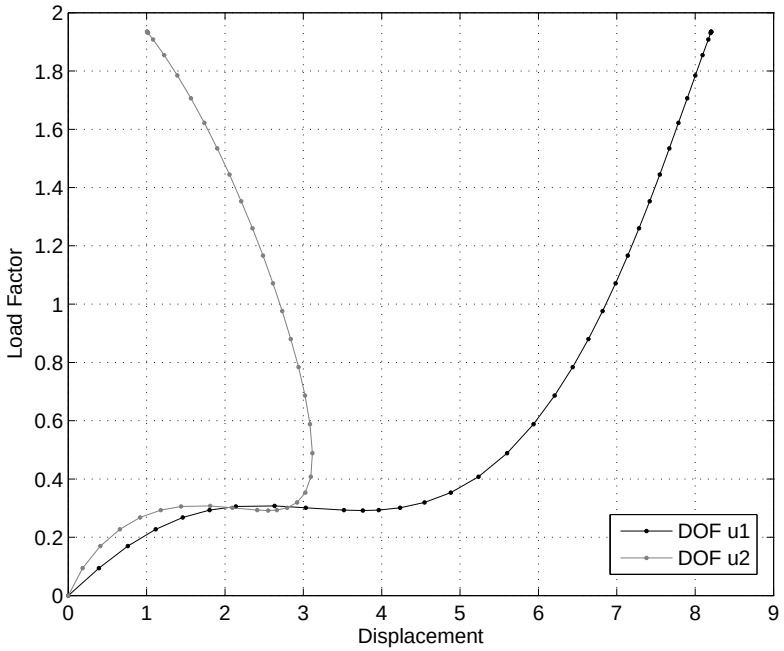

Figure 5.10: Solution to the two-dimensional function example using the ORP

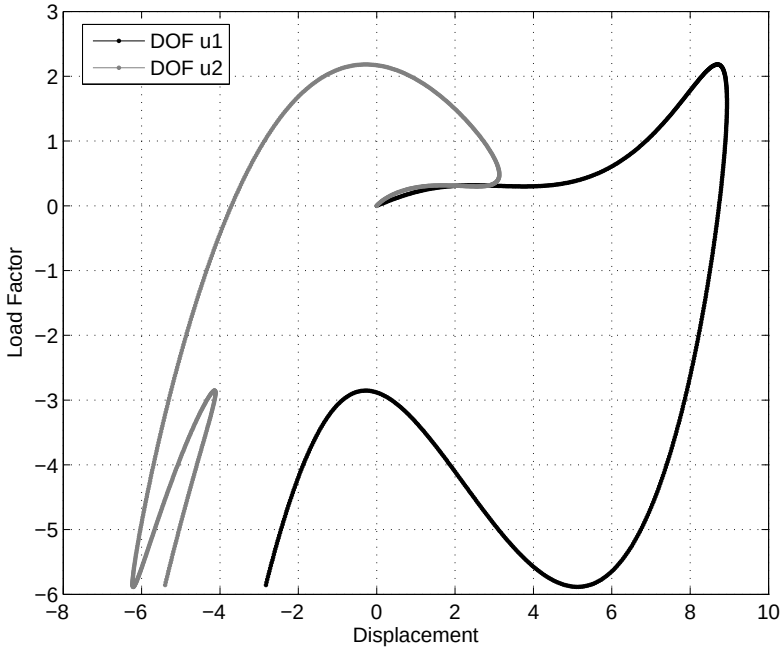

Figure 5.11: Solution to the two-dimensional function example using the ALCM and GDCM (ALCM solution shown)

the number of steps used for the arc-length control method and the generalized displacement control method are similar, but the control factor for the arc-length control method is five times greater. Recall from Table 4.2 though that the control factors represent different quantities for each scheme and cannot be directly compared. In the generalized displacement control method the control factor is only used once at the first iteration of the first step, and the load factor is adjusted by the algorithm for all subsequent steps and iterations. The control factor in the arc-length control method, however, is used once at every step.

Table 5.3: Summary of the two-dimensional function example

Algorithm	**Max. steps**	**Control factor**	**Scale factor**	**Converged/diverged**
LCM	100	0.04	n/a	Diverged at step 54
DCM^{s,1}	200	0.05	n/a	Diverged at step 179
DCM^{s,2}	200	0.02	n/a	Diverged at step 156
DCM^{v,1}	445	0.1	n/a	Fully converged
DCM^{v,2}	190	0.1	n/a	Fully converged
ALCM	700	0.05	n/a	Fully converged
WCM	300	0.01	n/a	Diverged at step 202
GDCM	650	0.01	n/a	Fully converged
ORP	100	0.1	1.0	Diverged at step 50

^{s} Standard ^{v} Variable

^{1} Fixed control coordinate: $\mathbf{u}_1$ ^{2} Fixed control coordinate: $\mathbf{u}_2$

5.3 Von Mises truss

The Von Mises Truss is a two-degree of freedom system consisting of two bar (truss) elements loaded indirectly through a spring of stiffness C, as shown in Figure 5.12. It has been studied by several authors, including Bergan [15], Bazant and Cedolin [12], and Yang and Sheih [97], among others. Although very simple, this example can be used to demonstrate the ability of the nonlinear solution algorithms to capture both load and displacement limit points, as well as sudden changes of direction in the equilibrium paths.

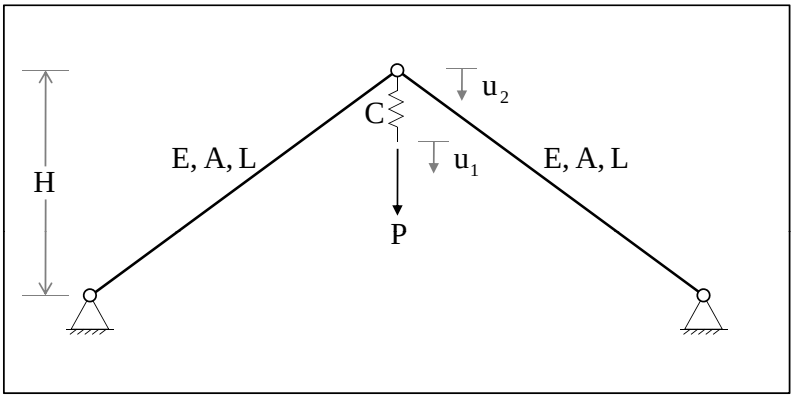

Figure 5.12: Von Mises truss schematic

The Von Mises Truss, shown in Figure 5.12 was assigned the following data (consistent units

are assumed):

- Reference Load Vector:$\{1.0,\ 0.0\}^T$
- $EA = 1.0$
- $H = 5.0$
- $L = 10.0$

A derivation of the finite element matrices (i.e. stiffness matrix and internal load vector) using the data above is shown in Appendix A. The behavior of this structure depends strongly on the stiffness of the spring, C, Figure 5.13. While snap-through behavior is always present, snap-back behavior will only result if the spring stiffness is below the critical value, $C_{\text{cr}} = 0.030940$ in this case (see Appendix A). In the examples in the next section C was chosen to be 0.02 to obtain snap-back behavior, and 0.04 for no snap-back behavior.

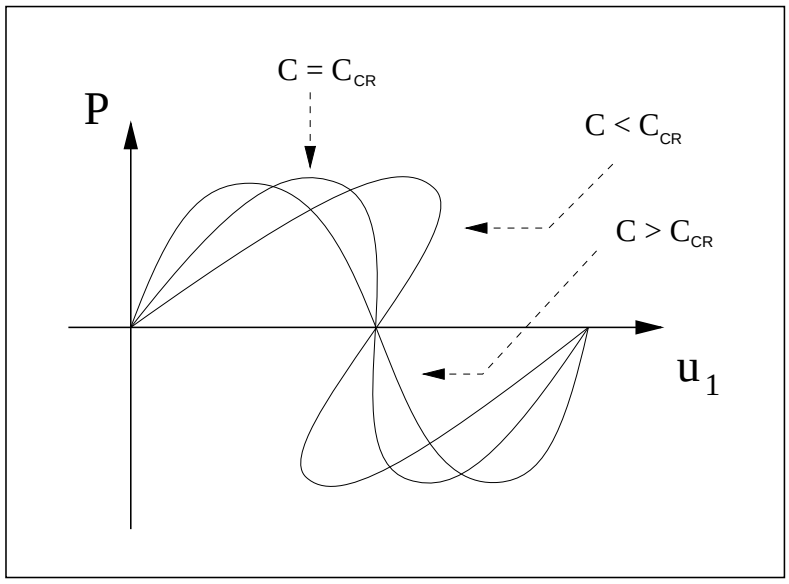

Figure 5.13: Equilibrium paths for the Von Mises truss with varying spring stiffness, C

5.3.1 Computational results

Figures 5.14 through 5.19 show the nonlinear behavior captured by each of the solution schemes. As expected, the load control method failed to trace the equilibrium path beyond the load limit point, as shown in Figure 5.14.

The variable displacement control method captured the full equilibrium path with snap back behavior when either of the degrees of freedom was chosen as the control, shown in Figure

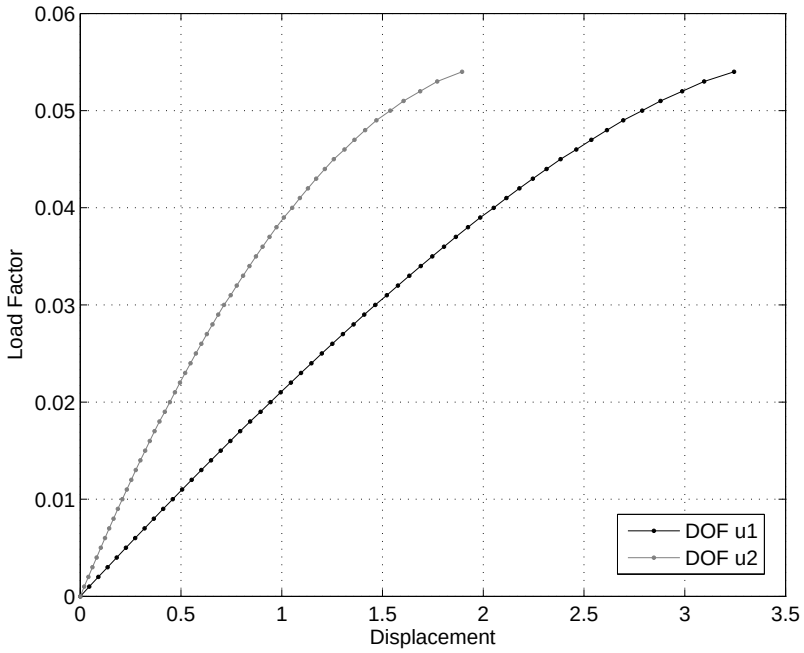

Figure 5.14: Solution to the Von Mises truss example using the LCM

5.16. However, the standard displacement control could only trace the full equilibrium curve when degree of freedom $\mathbf{u}_2$ was the control because there is no snap back in this degree of freedom. When $\mathbf{u}_1$ was chosen as the control displacement, the method snaps through at the displacement limit point and does capture the snap back behavior, as illustrated in Figure 5.15.

The arc-length control method, generalized displacement control method and orthogonal residual procedure also successfully captured the full equilibrium path for the system with snap-back behavior, shown in Figure 5.16.

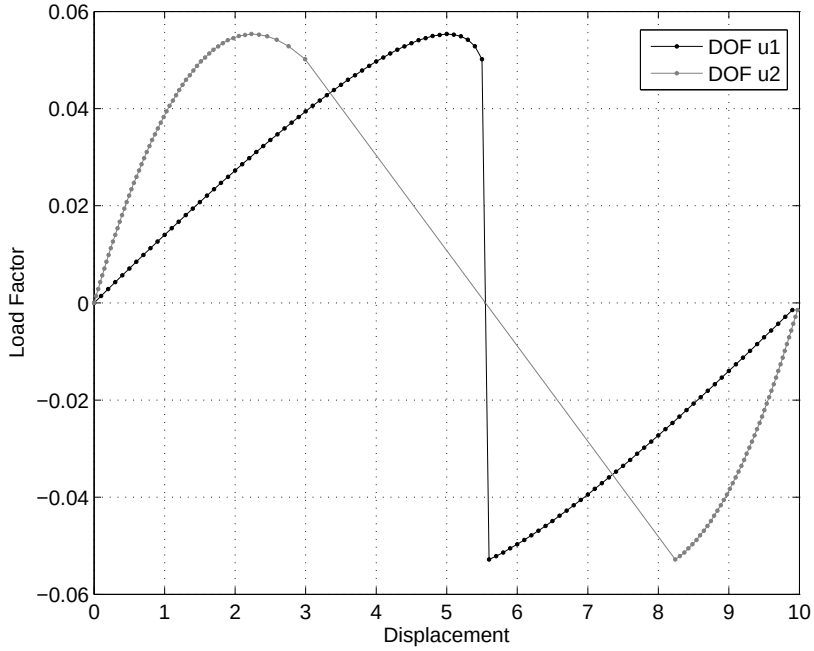

Figure 5.15: Solution to the Von Mises truss example with $C = 0.02$ using the DCM with $\mathbf{u}_1$ as the control displacement

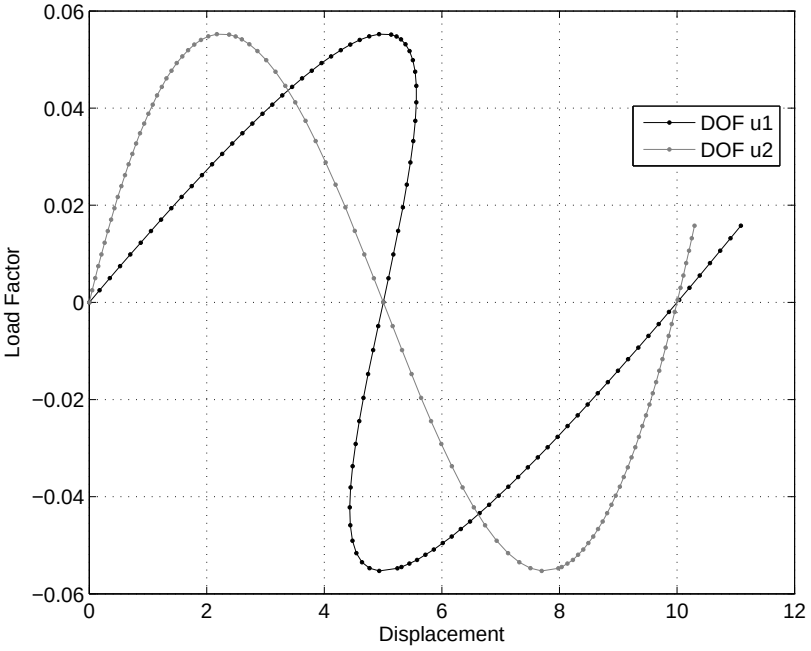

Figure 5.16: Solution to the Von Mises truss example with $C = 0.02$ using the DCM with $\mathbf{u}_2$ as the control displacement, variable DCM, ALCM, GDCM, and ORP (GDCM solution shown)

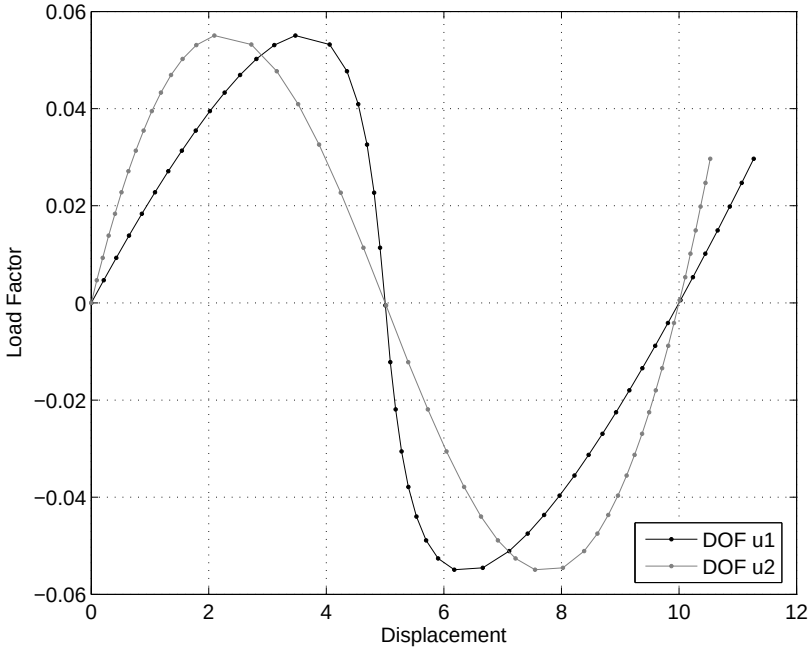

Figure 5.17: Solution to the Von Mises truss example with $C = 0.04$ using the WCM

The work control method was only successful at capturing the full load-displacement curve when no snap-back behavior is present, Figure 5.17. When the spring stiffness is chosen such that snap-back behavior should be present (i.e. $C = 0.02$), the method fails at the displacement limit point, Figure 5.18. This behavior is expected from the work control method because the snap back occurs in the major forcing direction. The external load vector contains only one load, and since snap back occurs in the loading direction, the displacement increment in that direction is zero. The denominator of the load parameter in Equation 2.44 becomes unbounded and the method diverges.

The orthogonal residual procedure captured the full equilibrium paths for both the snap-back and no snap-back scenarios, however the parameters are quite different between the two cases. A larger control factor, larger scale factor, and fewer steps could be used in the system without snap-back behavior. In order for the orthogonal residual procedure to capture snap back, the step size had to be smaller than the case with no snap back, as shown in Figures 5.20 and 5.19, respectively. Table 5.4 provides a summary of all parameters and the resulting behavior of each algorithm.

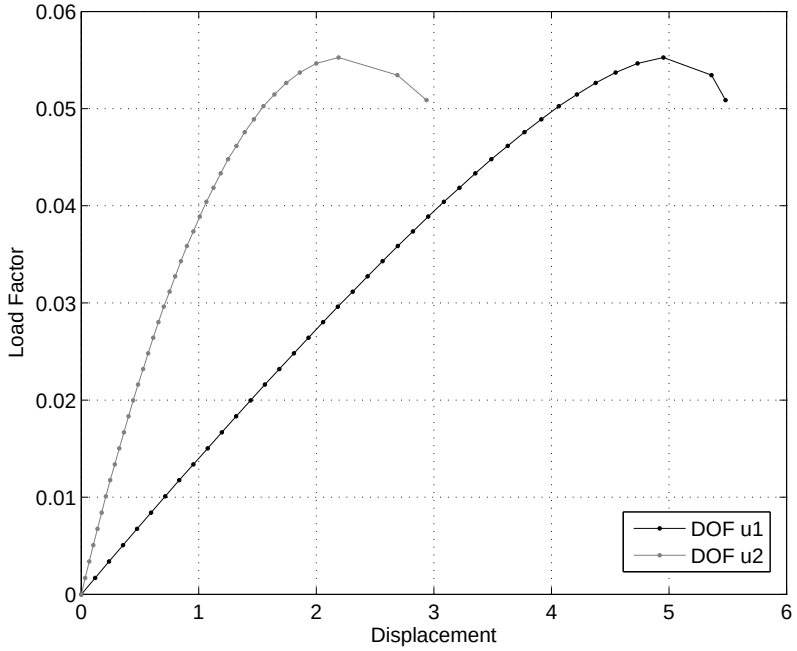

Figure 5.18: Solution to the Von Mises truss example with $C = 0.02$ using the WCM

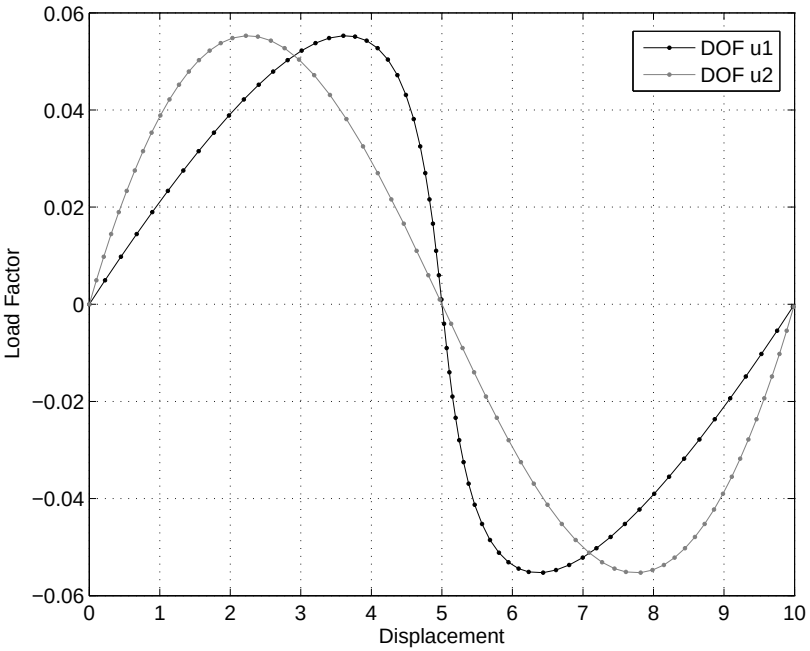

Figure 5.19: Solution to the Von Mises truss example with $C = 0.04$ using the ORP

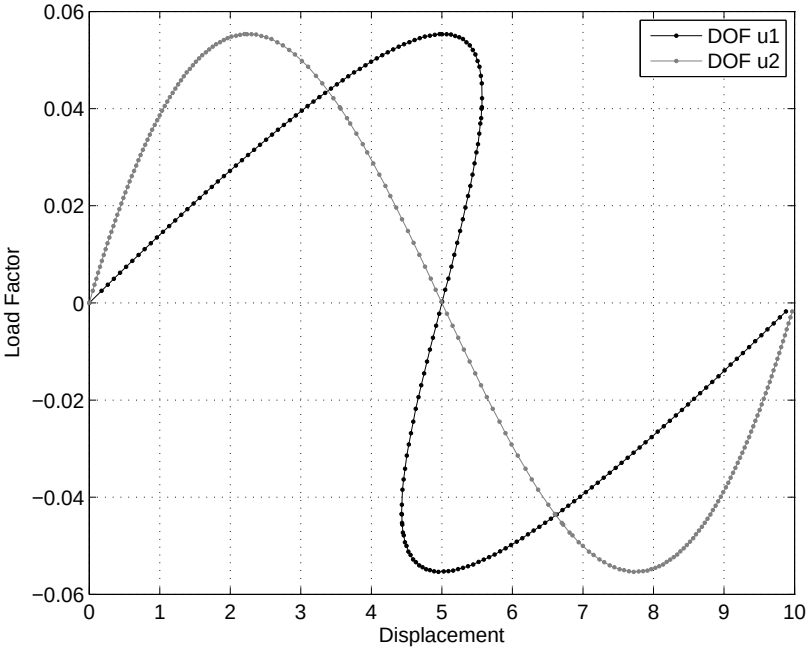

Figure 5.20: Solution to the Von Mises truss example with $C = 0.02$ using the ORP

Table 5.4: Summary of the Von Mises truss example

Algorithm	**C**	**Max. steps**	**Control factor**	**Scale factor**	**Converged/diverged**
LCM	0.02	100	0.001	n/a	Diverged at step 56
DCM^{s,1}	0.02	100	0.1	n/a	Snaps through
DCM^{s,2}	0.02	100	0.1	n/a	Fully converged
DCM^{v,1}	0.02	100	0.3	n/a	Fully converged
DCM^{v,2}	0.02	80	0.1	n/a	Fully converged
ALCM	0.02	100	0.17	n/a	Fully converged
WCM	0.02	100	0.0002	n/a	Diverged at step 41
WCM	0.04	50	0.0001	n/a	Fully converged
GDCM	0.02	100	0.0025	n/a	Fully converged
ORP	0.02	65	0.005	1	Fully converged
ORP	0.04	190	0.0025	0.5	Fully converged

^{s} Standard ^{v} Variable

^{1} Fixed control coordinate: $\mathbf{u}_1$ ^{2} Fixed control coordinate: $\mathbf{u}_2$

5.4 Twelve bar truss

This example consists of a 12-bar truss structure as illustrated in Figure 5.21a. It has been studied by Yang and Leu [94] and Krenk and Hededal [54], among others. This example features highly nonlinear behavior: the load changes direction eight times and the structure experiences several very large changes in stiffness through the load history.

The 12 Bar Truss, shown in Figure 5.21 was assigned the following data (constant units are assumed):

- Reference Load Vector:$\{0.0,\ 0.75,\ 0.25\}^T$
- $EA = 1.0$

The finite element matrices for the bar elements used in this problem are derived in Appendix A. For the present work, double symmetry has been considered and, therefore, the deformations of the structure can be described by three displacement components (u_1, u_2, and u_3) as shown in Figure 5.21b. The relative dimensions are shown in Figure 5.21.

For a complete explanation of the expected behavior of this structure, the reader is referred to Krenk and Hendal [54].

5.4.1 Computational results

The computational results are shown in Figures 5.22 through 5.25. All of the solution schemes captured the full nonlinear behavior except the load control method and work control method. As expected, the load control method failed at the first load limit point. Similarly, the work control method failed near snap back points in the loading directions. As shown in Figure 5.22, the method fails near snap back points in the u_2 and u_3 direction, which are also loading directions. The arc-length control method, generalized displacement control method, and the orthogonal residual procedure captured the full behavior shown in Figure 5.23.

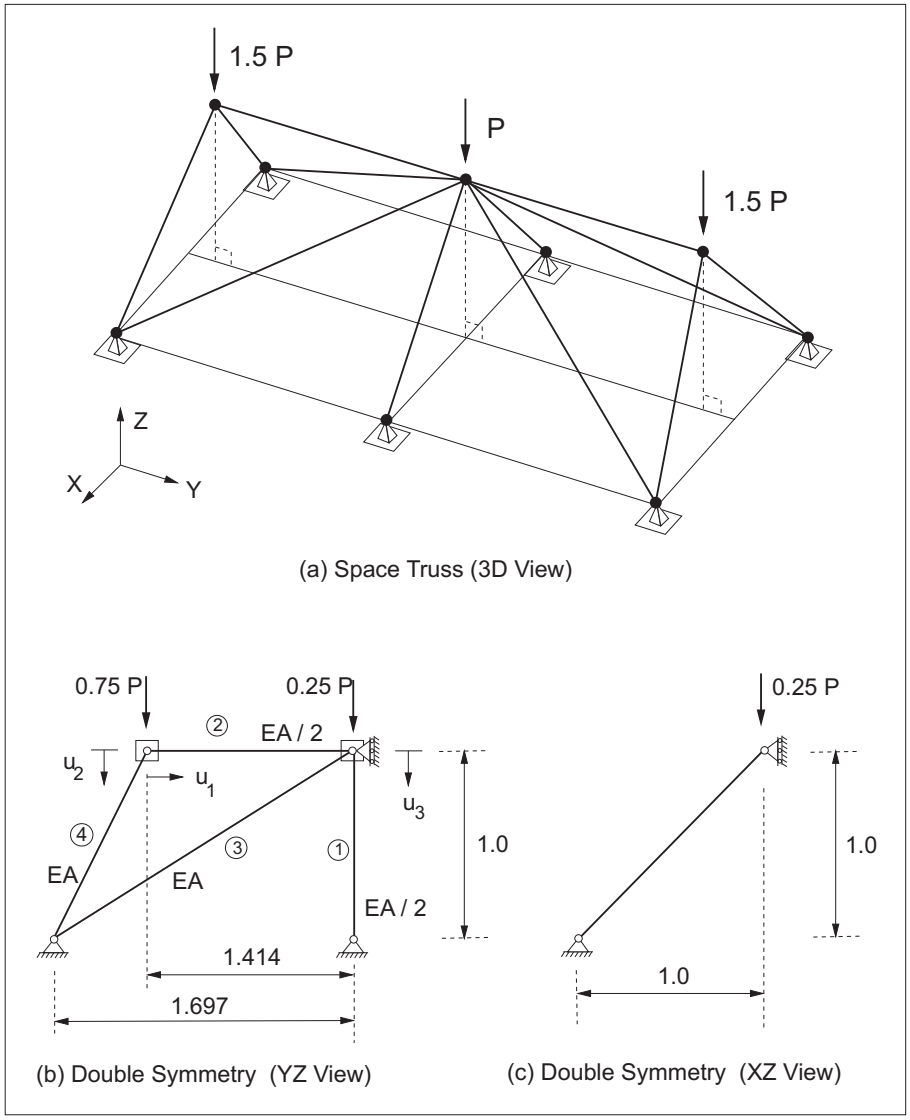

Figure 5.21: 12 bar truss schematic

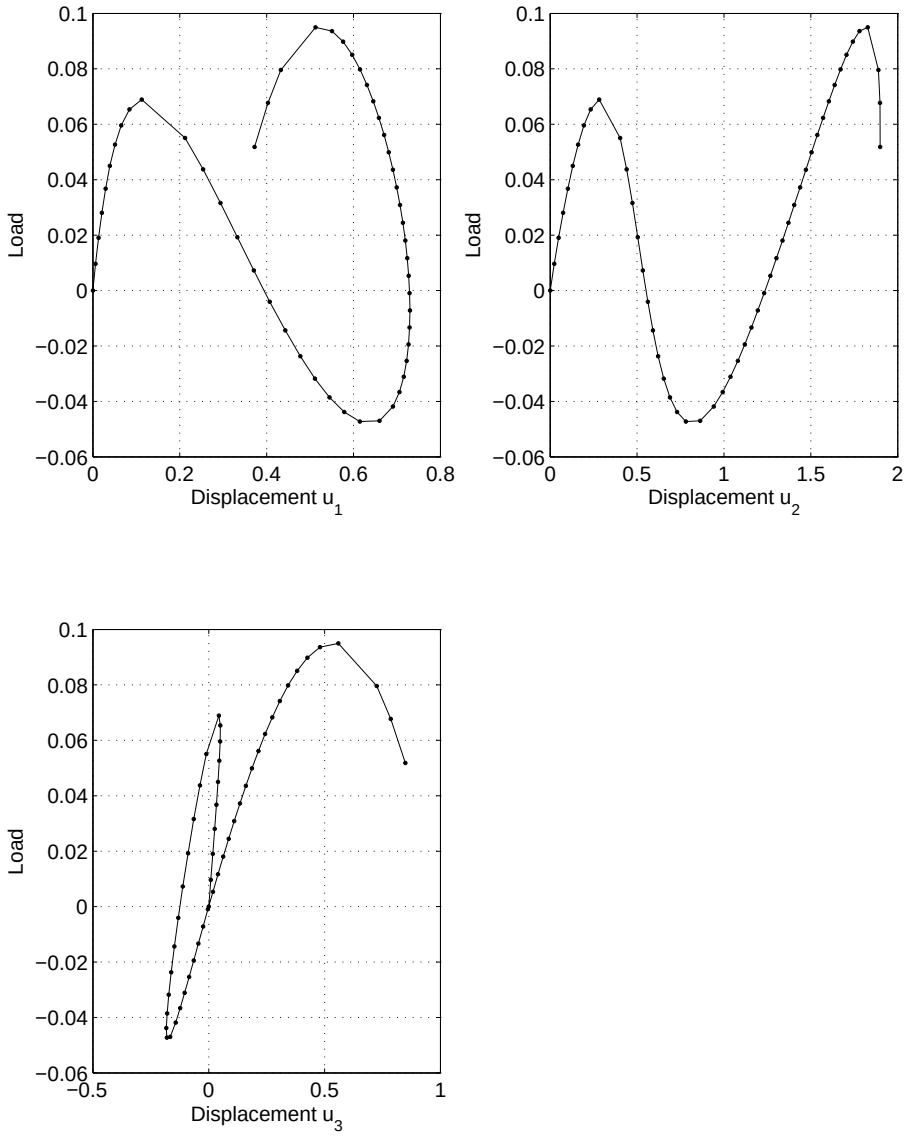

Figure 5.22: Solution to the 12 bar truss example using the WCM

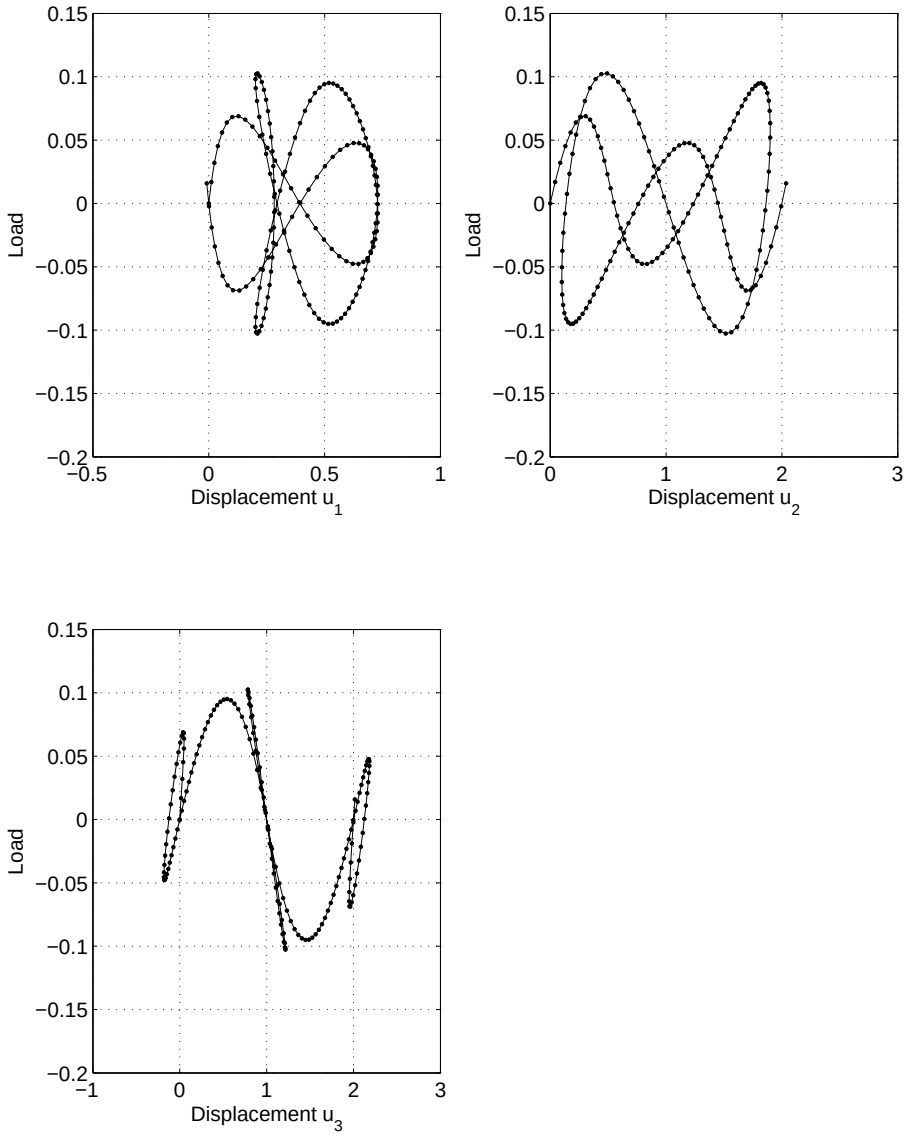

Figure 5.23: Solution to the 12 bar truss example using the ALCM, GDCM and ORP (ALCM solution shown)

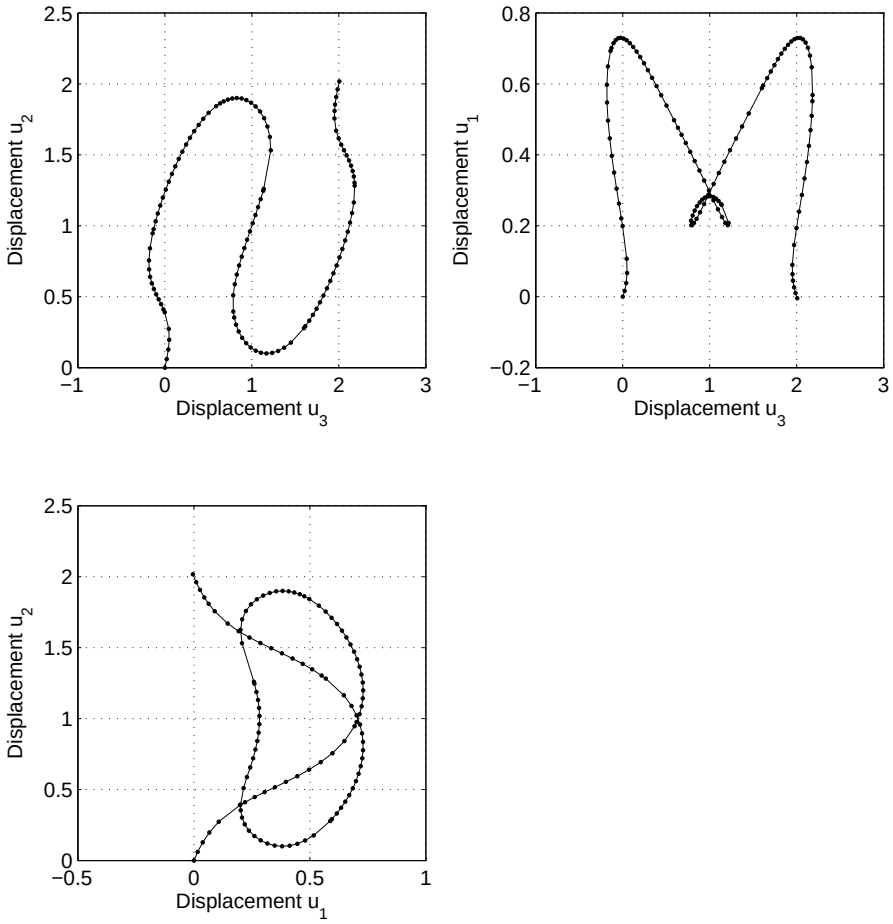

Figure 5.24: Displacement-displacement curves for the 12 bar truss

From the displacement-displacement curves shown in Figure 5.24, it is seen that no linear combination of the displacement components increases monotonically, therefore a traditional displacement control method could not capture the full behavior [54]. The variable displacement control method, however, does capture the full equilibrium path when variable displacement was employed, as shown in Figure 5.25. The method automatically changes the control displacement when a displacement limit point is likely approaching. The control displacements and snap back locations are listed in Table 5.5 and correspond to the labels in Figure 5.25. From steps 12-25 the control displacement is u_1, and no displacement limit points are encountered. At step 26 the control displacement changes to u_2, and snap back is captured in degree of freedom u_3 (label 2 in Figure 5.25c) and u_1 (label 3 in Figure 5.25a). Then again at step 63, the control displacement switches to u_3 and snap back is captured in degree of freedom u_2 (label 4 in Figure 5.25b). These automatic changes in the control

degree of freedom allow the method to recover the entire solution path.

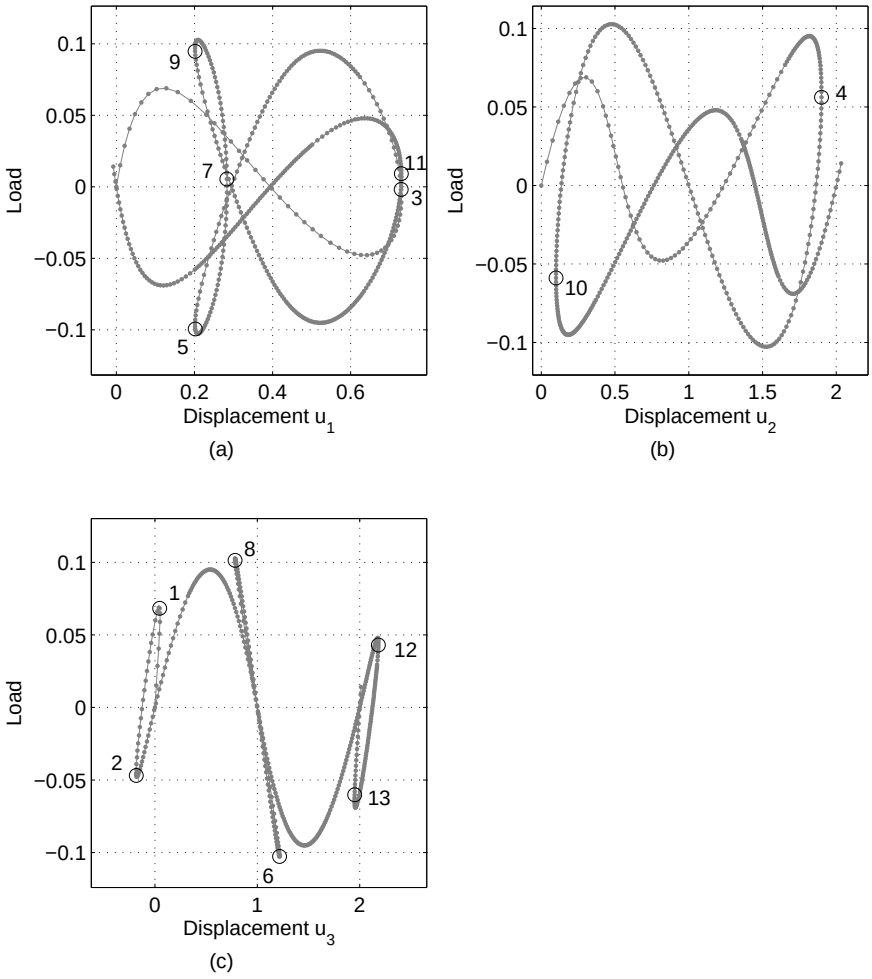

Figure 5.25: Solution to the 12 bar truss example using the variable DCM

Table 5.5: Summary of the variable DCM for the 12 bar truss

Step	**Control displacement**	**Snap back**
1	u_1	none
2-11	u_2	Step 8 in u_3 (Label 1 in 5.25c)
12-25	u_1	none
26-62	u_2	Step 30 in u_3 (Label 2 in 5.25c)
		Step 47 in u_1 (Label 3 in 5.25a)
63-125	u_3	Step 103 in u_2 (Label 4 in 5.25b)
126-222	u_2	Step 137 in u_1 (Label 5 in 5.25a)
		Step 141 in u_3 (Label 6 in 5.25c)
		Step 175 in u_1 (Label 7 in 5.25a)
		Step 209 in u_3 (Label 8 in 5.25c)
		Step 213 in u_1 (Label 9 in 5.25a)
223-302	u_3	Step 251 in u_2 (Label 10 in 5.25b)
303-377	u_2	Step 337 in u_1 (Label 11 in 5.25a)
		Step 372 in u_3 (Label 12 in 5.25c)
378-429	u_1	none
430-465	u_2	Step 446 in u_3 (Label 13 in 5.25c)

Table 5.6: Summary of the 12 bar truss example

Algorithm	**Max. steps**	**Control factor**	**Scale factor**	**Converged/diverged**
LCM	100	0.001	n/a	Diverged at step 70
DCM^{v,1}	465	0.01	n/a	Fully converged
ALCM	165	0.05	n/a	Fully converged
WCM	100	0.0002	n/a	Diverged at step 52
GDCM	115	0.025	n/a	Fully converged
ORP	650	0.0025	2	Fully converged

^{v} Variable ^{1} Fixed control coordinate: $\mathbf{u}_1$

5.5 Lee frame

The Lee frame is a well known example for evaluating nonlinear solvers, for which an analytical solution exists [57]. Schweizerhof and Wriggers [87] compared updated and spherical plane path following schemes using the Lee frame discretized with beam elements. Parente and Vaz [71] used this example discretized with quadratic isoparametric 8-node elements for shape design sensitivity analysis for nonlinear structures.

Deformation of the structure is characterized by large rigid body displacements and rotations resulting in instability. The behavior is highly nonlinear with two load limit points and snap-back behavior (i.e. displacement limit point).

The Lee Frame, shown in Figure 5.26, was discretized with 10 beam elements, each with the following properties (consistent units are assumed):

- EA = 4320
- GJ = 2160
- EI = 1440

The finite element matrices for the beam elements used in this problem are derived in Appendix A.

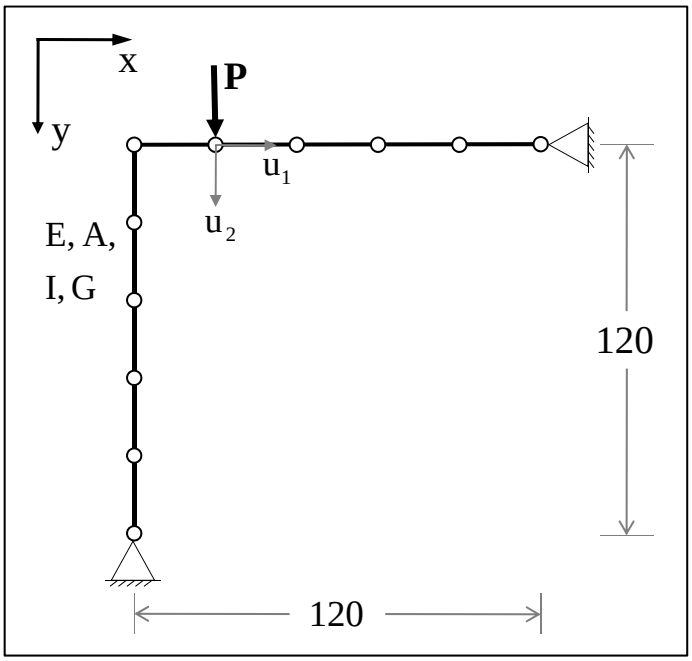

Figure 5.26: Lee frame schematic

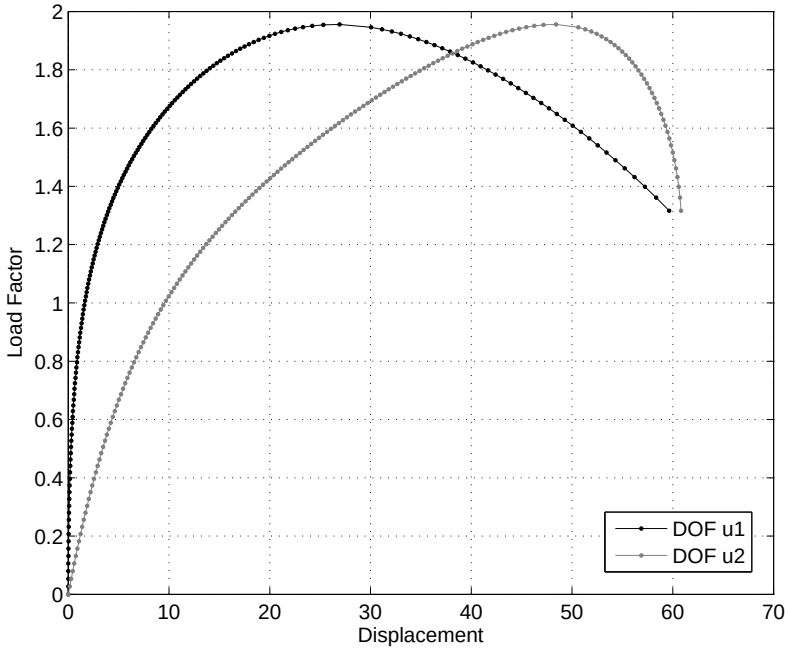

Figure 5.27: Solution to Lee frame example using the WCM

5.5.1 Computational results

The equilibrium paths for the Lee frame example are shown in Figures 5.27 to 5.29. The results are also summarized in Table 5.7. The nonlinear solution schemes generally behaved as expected and as they did in the previous examples. The load control method only captured behavior up to the first load limit point. The work control method failed at the snap back point in degree of freedom u_2, which is also the loading direction, shown in Figure 5.27. The variable displacement control method, arc-length control method and generalized displacement control method captured the full behavior shown in Figure 5.28.

The orthogonal residual procedure was not able to capture the entire solution path beyond the snap back points. As shown in Figure5.29, the method diverges at the first displacement limit point when the scale factor, β, is 0.01. The scale factor was adjusted to 0.02 and the method traced more of the equilibrium path, but diverged at the second displacement limit point, Figure 5.30. This suggests (1) the method is very sensitive to the scale factor input by the user, and (2) the method has difficulty near displacement limit points. Please refer to Section 6.1 for a detailed discussion of the implementation suggestions to improve these issues.

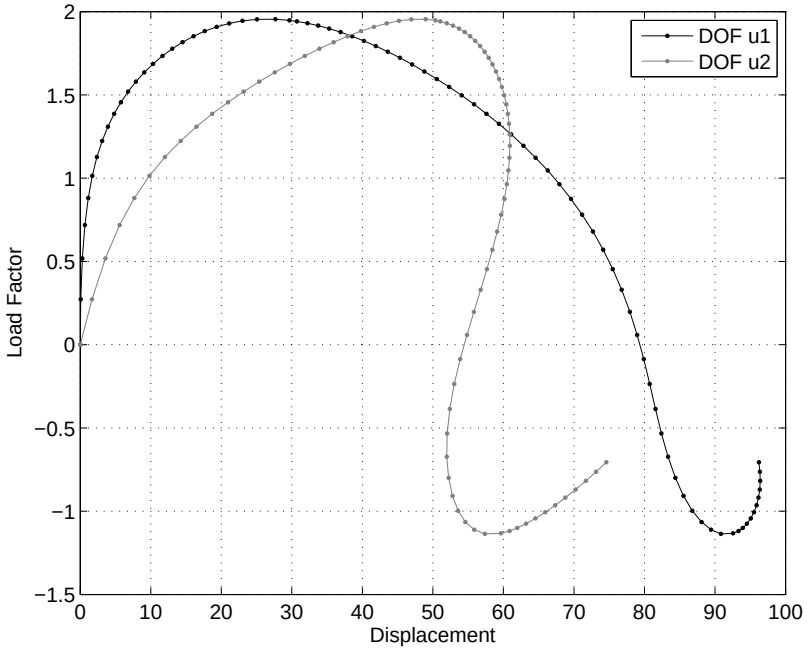

Figure 5.28: Solution to Lee Frame example using the variable DCM, ALCM, and GDCM (GDCM solution shown)

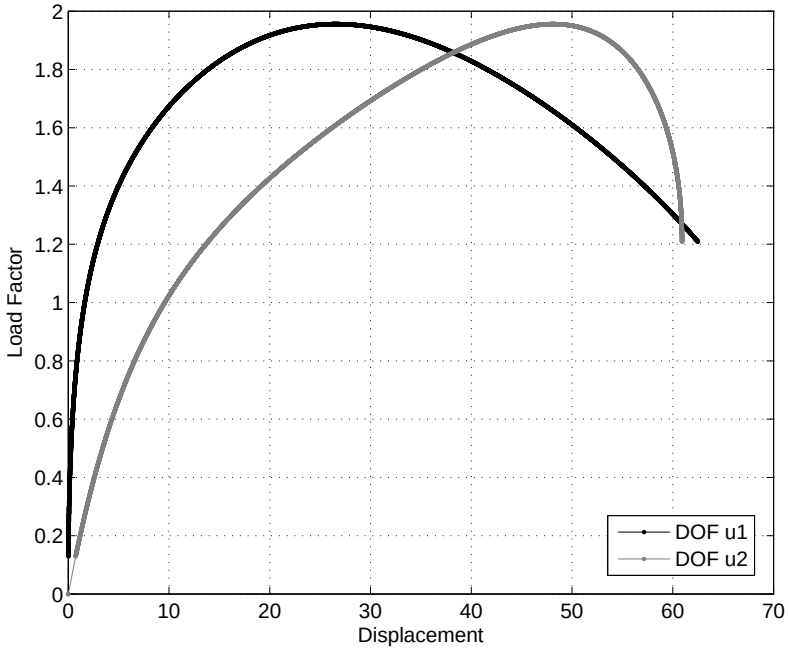

Figure 5.29: Solution to Lee frame example using the ORP with $\beta = 0.01$

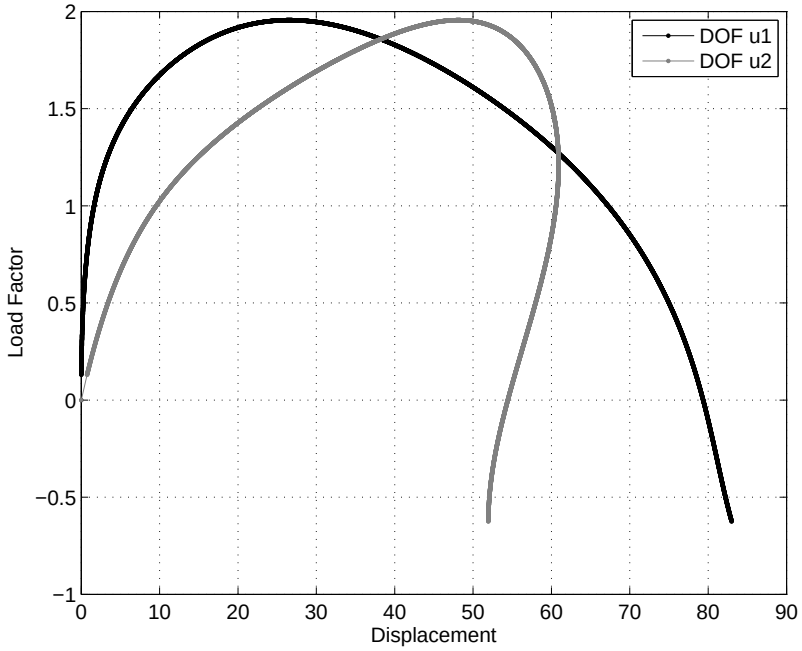

Figure 5.30: Solution to Lee frame example using the ORP with $\beta = 0.02$

Table 5.7: Summary of the Lee frame example

Algorithm	**Max. steps**	**Control factor**	**Scale factor**	**Converged/diverged**
LCM	100	0.05	n/a	Diverged at step 40
DCM^{v,1}	100	0.1	n/a	Fully converged
ALCM	80	5	n/a	Fully converged
WCM	300	0.004	n/a	Diverged at step 180
GDCM	80	0.3	n/a	Fully converged
ORP	23000	0.1	0.01	Diverged at step 9173
ORP	14000	0.1	0.02	Diverged at step 13676

^{v} Variable ^{1}Fixed control coordinate: $\mathbf{u}_1$

Chapter 6

Conclusions and future work

Geometric and material nonlinearity are prevalent in computational mechanics, and algorithms to capture these behaviors are critical to the analysis of such problems. The unified library of solution schemes is capable of solving complex nonlinear problems, including functions of multiple variables or structural systems comprised of bar or beam elements. Six nonlinear solution schemes are unified into a single code by augmenting the system of N unknowns and including one additional unknown, a load parameter $\delta\lambda$. An additional constraint equation is introduced for each algorithm, and the new system of $(N+1)$ unknowns is solved by decomposing the unknown field (i.e. iterative displacement vector).

The implementation of NLS++ into an object-oriented framework is a natural approach because the $(N+1)$ space formulation is generalized to include any number of algorithms. The straight-forward implementation makes usage of NLS++ very easy, thus giving users the ability to solve complex nonlinear problems quickly. The six algorithms were tested extensively with five simple examples featuring very complex nonlinear behavior. The strengths and weaknesses of the algorithms are evident from the results in Chapter 5. While most algorithms behaved as expected, the testing gave rise to some needed implementation improvements, particularly for the orthogonal residual procedure, which is discussed in Section 6.1 below.

6.1 Suggestions for future work

The unified library of nonlinear solution schemes is a comprehensive set of algorithms suitable for a range of nonlinear computational mechanics problems. Some improvements, however, are necessary to increase the efficiency and functionality of each algorithm. Furthermore, extensions of the current work will expand the capabilities of the library to an even larger class of nonlinear problems.

Improved implementation of the orthogonal residual procedure

The ORP required special attention to formulate into the $(N+1)$ dimensional space, which may have resulted in some of the unexpected behavior. First, the ORP has problems near displacement limit points. This problem was overcome in most of the examples in Chapter 5 by careful selection of the control factor and scale factor, however, divergence occurred at displacement limit points in Section 5.5. Therefore, a very beneficial improvement to the method would be automatic detection of displacement limit points. The current implementation of the ORP could be enhanced to include the relaxed orthogonality condition near displacement limit points presented by Kouhia [52]. Another approach, similar to that used in the variable displacement control method, is to enforce the orthogonality constraint only on those degrees of freedom which are not likely to experience snap back at a given step.

Another problem associated with the ORP is seen in Section 5.2 where the method artificially changes the sign of the load factor then bounces back on the curve and traces the previously converged solution path. The second recommended improvement to the ORP is to insert a condition to check if the method is capturing a previously converged solution path. Ritto-Correa and Camotim[82] developed a method to guarantee that arc-length type methods do not bounce back on previously converged solution paths. If the constraint equation of the ORP could be reformulated into an arc-length type constraint, then this method may be utilized.

Integration into finite element analysis software

Two structural elements were used to describe the structural systems studied in this work: the bar and beam element, which are implemented in the software. In order to represent a wider range of problem, NLS++ should be extended to support continuum finite elements (i.e. T3, Q4, T6, Q8, etc.). Furthermore, the full potential of NLS++ to capture highly nonlinear behavior can be achieved by integrating it into a general finite element analysis package. The software, TopFEM, is an object oriented finite element analysis code that utilizes TopS [21], a topological data structure for representing finite element meshes. Although TopFEM can represent large scale problems and can be used for complex analysis including topology optimization and dynamic fracture simulation, the present capability to capture highly nonlinear behavior is minimal. Integration of NLS++ into TopFEM would therefore greatly improve the capabilities of both object-oriented analysis codes.

Efficient linear solvers

The incremental-iterative procedure to solve nonlinear problems results in solving a linearized system at each step. Therefore an efficient linear solver is always necessary, even in nonlinear

analysis. The systems analyzed in this work were relatively small, so the efficiency of the linear solver was not an issue. However, if NLS++ is used to solve large systems of equations, a faster and more sophisticated linear solver will be necessary. A future investigation related to NLS++ is then to implement and test various linear system solvers. Linear solvers for both symmetric and non-symmetric matrices should be tested. The question of whether to use direct or iterative solvers should also be addressed. Direct solvers, such as Gaussian elimination or LU decomposition, will produce the exact solution assuming exact arithmetic is used and are applicable to any type of linear system (e.g. symmetry, positive definiteness, etc. are not required), thereby making them a good candidate for NLS++. These methods, however, are known to be computationally expensive, especially as the problem size increases [47]. Conversely, iterative solvers attempt to solve the problem through successive approximations beginning from an initial estimate. Optimization based iterative solvers seem to be better suited for NLS++ than stationary iterative solvers, such as the Jacobi method or Gauss-Seidel method, which have slow convergence rates. Iterative solvers, in general, require less storage and fewer operations than direct solvers, making them more effective for large systems. The conjugate gradient method for example, is very efficient for large problems, however, it is only applicable to symmetric, positive-definite systems [47]. Other optimization based solvers applicable to non-symmetric matrices, such as the bi-conjugate gradient method or the generalized minimum residual method, may be explored for NLS++ in order to represent a larger class of problems. Given the variability of the linear solvers, the current implementation of NLS++ would benefit from a library of linear solvers, in which the user could select the most appropriate one for the problem at hand.

Additional nonlinear solvers

The unified library of nonlinear solvers and its object-oriented implementation easily supports the addition of solution schemes. In addition to the six solvers examined in this work, the strain control and strain ratio control algorithms are also implemented in the library. These solvers are similar to the displacement control method discussed in this work, however, instead of incrementing the displacements of one of the degrees of freedom, the strain or rate of strain is incremented. Thorough testing of these schemes is necessary to evaluate their effectiveness in capturing nonlinear behavior.

Furthermore, several versions of the arc-length method were discussed in Chapter 2, however only the spherical version was tested in this work. A natural extension would therefore be to implement the remaining versions of the arc-length method and evaluate the effectiveness of each at capturing complex nonlinearity. Recommendations could then be made as to the most powerful and efficient version for various type of behavior.

6.2 Lessons learned

As a result of the computational tests performed in Chapter 5, the most effective algorithms for capturing highly nonlinear behavior, in the present implementation, are the variable displacement control method, arc-length method, and generalized displacement control method. These methods successfully traced the full equilibrium paths of in each of the examples in Chapter 5, and seem well suited for application to more complex systems. Of these three methods, the generalized displacement control method was the most consistent, requiring little variation in input parameter values from one example to the next. The $(N + 1)$ dimensional space formulation, variety of supported algorithms, and object-oriented design make the NLS++ a powerful tool for solving nonlinear systems of equations.

Appendix A

Nonlinear finite element formulation

One of two incremental formulations may be used to describe the elements of a nonlinear problem: Total Lagrangian (TL) formulation and Updated Lagrangian (UL) formulation. In the TL formulation all static and kinematic variables refer to the original undeformed configuration, while in the UL formulation all variables refer to the last known deformed configuration. Both formulations include nonlinear effects due to large displacements and rotations (geometric nonlinearity) and can include large strain behavior if such nonlinearity is modeled in the constitutive relationship (material nonlinearity) [29, 40]. The TL formulation for geometric nonlinearity was used to develop the governing linearized equations for the structural system examples in Chapter 5 (i.e. Von Mises Truss, 12 Bar Truss, Lee Frame). The structural elements used in the Von Mises and 12 bar truss are prismatic bar elements that resist longitudinal loads and can undergo large displacements and rotations. The Lee Frame is comprised of beam elements that can resist longitudinal and transverse loads applied between its supports as well as large rotations.

The focus of the appendix is the nonlinear finite element formulation of bar elements. First the principle of virtual work is reviewed in Section A.1, followed by the derivation of the finite element matrices for bar elements in section A.2. An additional approach for obtaining the equilibrium equations of the Von Mises Truss and derivation of its critical spring stiffness are presented in section A.3.

A.1 Principle of virtual work

The notation adopted in this study will first be reviewed [95, 9, 76]. Configurations are denoted C_i, where $i = 0, 1, 2$. C_0 refers to the original undeformed configuration, C_1 refers to the last known configuration, and C_2 refers to the current (unknown) configuration, as illustrated in Figure A.1. A left superscript denotes the configuration in which the quantity occurs. A left subscript denotes the configuration in which the quantity is measured. So ${}^i_jQ$ refers to the quantity Q that occurs in the C_i configuration, but is measured in the C_j configuration. A quantity with no left superscript and only a left subscript refers to an incremental quantity occurring between the C_1 and C_2 configurations.

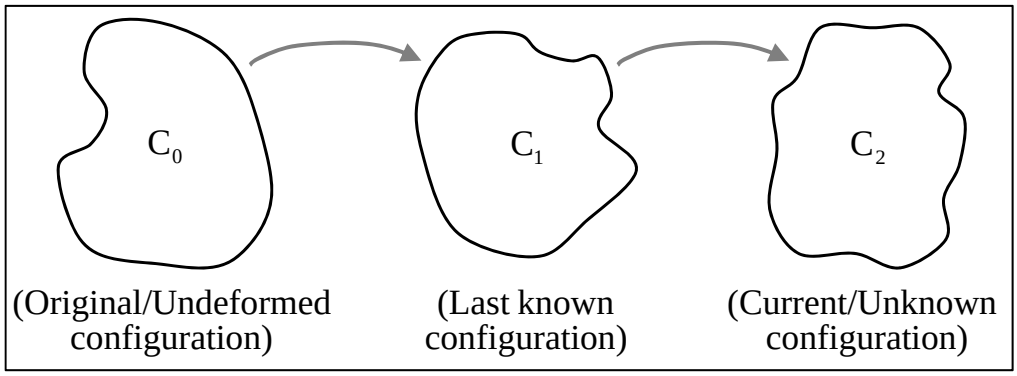

Figure A.1: C_0, C_1, and C_2 configurations

The principle of virtual work is given by

$$\delta W = \int_{{}_2V} \left({}^2_2\sigma_{ij}\right) \delta \left({}^2_2e_{ij}\right) d\,{}^2V - \int_{{}_2V} \left({}^2_2b_i\right) \delta \left(u_i\right) d\,{}^2V - \int_{{}_2S} \left({}^2_2t_i\right) \delta \left(u_i\right) d\,{}^2S = 0 \qquad \text{(A.1)}$$

where σ_{ij} is the Cauchy stress tensor, e_{ij} is the infinitesimal strain increment tensor, b_i is the body force vector (measured per unit volume), t_i is the boundary traction vector (measured per unit surface area), u_i in the unknown displacement field, and δ is the Variational symbol acting on δ.

In Equation A.1, all values are measured in the current configuration, C_2. However Equation A.1 cannot be solved directly because the current configuration is unknown in nonlinear analysis. To overcome this problem, the integrals in Equation A.1 will be transformed to integrals over a configuration which is known. In the case of the Total Lagrangian formulation the known configuration is the original undeformed configuration, C_0. The following identities will be used [9]

$$\int_{{}_2V} \left({}^2_2\sigma_{ij}\right) \delta \left({}^2_2e_{ij}\right) d\,{}^2V = \int_{{}_0V} \left({}^2_0S_{ij}\right) \delta \left({}^2_0E_{ij}\right) d\,{}^0V \qquad \text{(A.2)}$$

$$\int_{{}_2V} \left({}^2_2b_i\right) \delta \left(u_i\right) d\,{}^2V = \int_{{}_0V} \left({}^2_0b_i\right) \delta \left(u_i\right) d\,{}^0V \qquad \text{(A.3)}$$

$$\int_{{}_2S} \left({}^2_2t_i\right) \delta \left(u_i\right) d\,{}^2S = \int_{{}_0S} \left({}^2_0t_i\right) \delta \left(u_i\right) d\,{}^0S \qquad \text{(A.4)}$$

where S_{ij} is the Second Piola-Kirchhoff stress tensor and E_{ij} is the Green-Lagrange strain tensor.

Plugging Equations A.2-A.4 into Equation A.1 gives

$$\int_{{}_0V} \left({}^2_0S_{ij}\right) \delta \left({}^2_0E_{ij}\right) d\,{}^0V - \int_{{}_0V} \left({}^2_0b_i\right) \delta \left(u_i\right) d\,{}^0V - \int_{{}_0S} \left({}^2_0t_i\right) \delta \left(u_i\right) d\,{}^0S = 0 \quad \text{(A.5)}$$

$$\int_{{}_0V} \left({}^2_0S_{ij}\right) \delta \left({}^2_0E_{ij}\right) d\,{}^0V - \delta \left({}^2_0R\right) = 0 \quad \text{(A.6)}$$

where

$$\delta \left({}^2_0R\right) = \int_{{}_0V} \left({}^2_0b_i\right) \delta \left(u_i\right) d\,{}^0V + \int_{{}_0S} \left({}^2_0t_i\right) \delta \left(u_i\right) d\,{}^0S \quad \text{(A.7)}$$

The virtual Green-Lagrange strain tensor in the C_2 configuration is given by strain increment in the the C_1 configuration plus the incremental strain occurring between the C_1 to C_2 configurations, but measured in the C_0 configuration. Notice that $\delta \left({}^1_0E_{ij}\right)$ is zero because it is not a function of the unknown displacements.

$$\delta \left({}^2_0E_{ij}\right) = \delta \left({}^1_0E_{ij}\right) + \delta \left({}_0\varepsilon_{ij}\right) \quad \text{(A.8)}$$

$$= \delta \left({}_0\varepsilon_{ij}\right) \quad \text{(A.9)}$$

$$= \delta \left({}_0e_{ij}\right) + \delta \left({}_0\eta_{ij}\right) \quad \text{(A.10)}$$

where $\delta \left({}_0e_{ij}\right)$ and $\delta \left({}_0\eta_{ij}\right)$ are the linear and nonlinear components of the Green-Lagrange strain increment tensor.

Then equation A.6 becomes

$$\int_{{}_0V} \left({}^2_0S_{ij}\right) \delta \left({}_0\varepsilon_{ij}\right) d\,{}^0V - \delta \left({}^2_0R\right) = 0 \quad \text{(A.11)}$$

The components of the second Piola-Kirchhoff stress in the C_2 configuration are the stress in the C_1 configuration plus the stress increment occurring between the C_1 to C_2 configurations, but measured in the C_0 configuration.

$$\int_{{}_0V} \left({}^1_0S_{ij} + {}_0S_{ij}\right) \delta \left({}_0\varepsilon_{ij}\right) d\,{}^0V - \delta \left({}^2_0R\right) = 0 \quad \text{(A.12)}$$

$$\int_{{}_0V} \left({}_0S_{ij}\right) \delta \left({}_0\varepsilon_{ij}\right) + \left({}^1_0S_{ij}\right) \delta \left({}_0e_{ij} + {}_0\eta_{ij}\right) d\,{}^0V - \delta \left({}^2_0R\right) = 0 \quad \text{(A.13)}$$

Examining the statement of the principle of virtual work for the equilibrium configuration,

C_1, $\delta\left({}^1_0R\right)$ is identified as

$$\int_{{}_0V}\left({}^1_0S_{ij}\right)\delta\left({}_0e_{ij}\right)d\,{}^0V = \int_{{}_0V}\left({}^1_0b_i\right)\delta\left(u_i\right)d\,{}^0V + \int_{{}_0S}\left({}^1_0t_i\right)\delta\left(u_i\right)d\,{}^0S \quad \text{(A.14)}$$

$$\delta\left({}^1_0R\right) = \int_{{}_0V}\left({}^1_0S_{ij}\right)\delta\left({}_0e_{ij}\right)d\,{}^0V = \int_{{}_0V}\left({}^1_0b_i\right)\delta\left(u_i\right)d\,{}^0V + \int_{{}_0S}\left({}^1_0t_i\right)\delta\left(u_i\right)d\,{}^0S \quad \text{(A.15)}$$

Equation A.13 becomes

$$\int_{{}_0V}\left({}_0S_{ij}\right)\delta\left({}_0\varepsilon_{ij}\right)d\,{}^0V + \int_{{}_0V}\left({}^1_0S_{ij}\right)\delta\left({}_0\eta_{ij}\right) + \delta\left({}^1_0R\right) - \delta\left({}^2_0R\right) = 0 \quad \text{(A.16)}$$

Now use the constitutive relation to replace ${}_0S_{ij}$ in terms of strains, that is ${}_0S_{ij} = \left({}_0C_{ijkl}\right)\left({}_0\varepsilon_{kl}\right)$, and assume displacement increments between configurations are small so $\delta\left({}_0\varepsilon_{ij}\right) \approx \delta\left({}_0e_{ij}\right)$, thereby linearizing the equation.

$$\int_{{}_0V}\left({}_0C_{ijkl}\right)\left({}_0e_{kl}\right)\delta\left({}_0e_{ij}\right)d\,{}^0V + \int_{{}_0V}\left({}^1_0S_{ij}\right)\delta\left({}_0\eta_{ij}\right) = \delta\left({}^2_0R\right) - \delta\left({}^1_0R\right) \quad \text{(A.17)}$$

The above statement is the weak form of the finite element model based on the Total Lagrangian formulation, which can now be modified for different element types to determine the finite element matrices. In the next section the weak form will be used to derive the finite element matrices for the bar element.

A.2 Nonlinear finite element matrices for bar elements

Several simplifications of Equation A.17 may be made for bar elements. Only the axial component of the stress and strain tensors need be considered, and the constitutive tensor, C_{ijkl}, reduces to E, the modulus of elasticity.

$$\int_{{}_0V}E\left({}_0e_{xx}\right)\delta\left({}_0e_{xx}\right)d\,{}^0V + \int_{{}_0V}\left({}^1_0S_{xx}\right)\delta\left({}_0\eta_{xx}\right) = \delta\left({}^2_0R\right) - \delta\left({}^1_0R\right) \quad \text{(A.18)}$$

The linear and nonlinear components of the strain reduce to

$$e_{xx} = \frac{\partial u}{\partial x} = \frac{\Delta u}{L} \quad \text{(A.19)}$$

$$\eta_{xx} = \frac{1}{2}\left[\left(\frac{\partial u}{\partial x}\right)^2 + \left(\frac{\partial v}{\partial x}\right)^2 + \left(\frac{\partial w}{\partial x}\right)^2\right] = \frac{1}{2}\left[\left(\frac{\Delta u}{L}\right)^2 + \left(\frac{\Delta v}{L}\right)^2 + \left(\frac{\Delta w}{L}\right)^2\right] \quad \text{(A.20)}$$

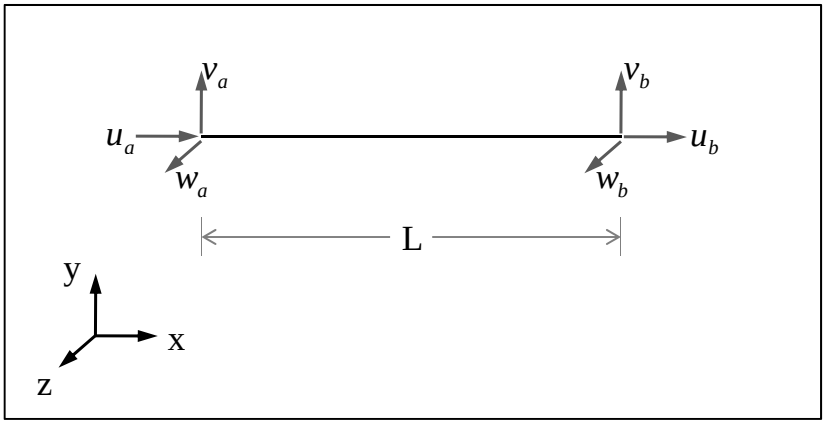

Figure A.2: Degrees of freedom for a bar element

where

$$\Delta u = u_a - u_b \tag{A.21}$$

$$\Delta v = v_a - v_b \tag{A.22}$$

$$\Delta w = w_a - w_b \tag{A.23}$$

The displacement vector consists of three degrees of freedom at each end, see Figure A.2

$$\mathbf{u}^T = \left\{ \begin{array}{cccccc} u_a & v_a & w_a & u_b & v_b & w_b \end{array} \right\}$$

Similarly, the force vectors in the C_1 and C_2 configurations contain three forces at each end

$$\left({}^1_0\mathbf{f}\right)^T = \left\{ \begin{array}{cccccc} {}^1_0F_{xa} & {}^1_0F_{ya} & {}^1_0F_{za} & {}^1_0F_{xb} & {}^1_0F_{yb} & {}^1_0F_{zb} \end{array} \right\}$$

$$\left({}^2_0\mathbf{f}\right)^T = \left\{ \begin{array}{cccccc} {}^2_0F_{xa} & {}^2_0F_{ya} & {}^2_0F_{za} & {}^2_0F_{xb} & {}^2_0F_{yb} & {}^2_0F_{zb} \end{array} \right\}$$

More simplifications can be made because a bar element can only resist axial force. The transverse shear forces vanish, that is $F_{ya} = F_{za} = F_{yb} = F_{zb} = 0$ and the axial forces are equal and opposite $F_{xa} = -F_{xb}$. Now each of the terms in Equation A.18 can be written

$$\int_{{}_0V} E\left({}_0e_{xx}\right)\delta\left({}_0e_{xx}\right)d\,{}^0V = \int_{{}_0V}\left(E\frac{\Delta u}{L}\right)\delta\left(\frac{\Delta u}{L}\right)d\,{}^0V = \delta\mathbf{u}^T\mathbf{K}_e\mathbf{u} \tag{A.24}$$

$$\begin{aligned}\int_{{}_0V}\left({}^1_0S_{xx}\right)\delta\left({}_0\eta_{xx}\right) &= \int_0^L\left({}^1_0F_x\right)\left[\frac{\Delta u}{L}\left(\frac{\delta\Delta u}{L}\right)+\frac{\Delta v}{L}\left(\frac{\delta\Delta v}{L}\right)+\frac{\Delta w}{L}\left(\frac{\delta\Delta w}{L}\right)\right]dx \\ &= \delta\mathbf{u}^T\mathbf{K}_g\mathbf{u}\end{aligned} \tag{A.25}$$

$$\delta\left({}^{1}_{0}R\right) = \int_{{}^{0}S} \left({}^{1}_{0}t_i\right) \delta\left(u_i\right) d\,{}^{0}S = \delta\mathbf{u}^{T}\,{}^{1}_{0}\mathbf{f} \tag{A.26}$$

$$\delta\left({}^{1}_{0}R\right) = \int_{{}^{0}S} \left({}^{2}_{0}t_i\right) \delta\left(u_i\right) d\,{}^{0}S = \delta\mathbf{u}^{T}\,{}^{2}_{0}\mathbf{f} \tag{A.27}$$

Assembling equations and admitting the arbitrary nature of the virtual displacements $\delta\mathbf{u}$, the incremental finite element equation is

$$\left(\mathbf{K}_e + \mathbf{K}_g\right)\mathbf{u} = {}^{2}_{0}\mathbf{f} - {}^{1}_{0}\mathbf{f} \tag{A.28}$$

where $\mathbf{K}_e$ is the elastic stiffness matrix and $\mathbf{K}_g$ is the geometric stiffness matrix, given below.

$$\mathbf{K}_e = \frac{EA}{L}\begin{bmatrix} 1 & 0 & 0 & -1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ -1 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix} \tag{A.29}$$

$$\mathbf{K}_g = \frac{{}^{1}_{0}F_x}{L}\begin{bmatrix} 1 & 0 & 0 & -1 & 0 & 0 \\ 0 & 1 & 0 & 0 & -1 & 0 \\ 0 & 0 & 1 & 0 & 0 & -1 \\ -1 & 0 & 0 & 1 & 0 & 0 \\ 0 & -1 & 0 & 0 & 1 & 0 \\ 0 & 0 & -1 & 0 & 0 & 1 \end{bmatrix} \tag{A.30}$$

A.3 Equilibrium equations for the Von Mises truss

An alternative approach to derive the nonlinear finite element matrices (i.e. tangent stiffness matrix and internal load vector) for the Von Mises Truss is the the principle of stationary potential energy. Because the Von Mises Truss is a relatively simple structure, the equilibrium equations can be derived for the entire system directly. The total potential energy of a system, Π, is comprised of the internal strain energy, U, and the potential due to externally applied loads, V.

$$\Pi = U + V \tag{A.31}$$

The principle is applicable to conservative systems, meaning that the work done by internal and external forces are independent of the path taken between the undeformed and deformed configurations. Furthermore, the load versus displacement relationship may be linear or nonlinear. The principle of stationary potential energy states that "Among all admissible

configurations of a conservative system, those that satisfy the equations of equilibrium make the potential energy stationary with respect to small admissible variations of displacement." [26]

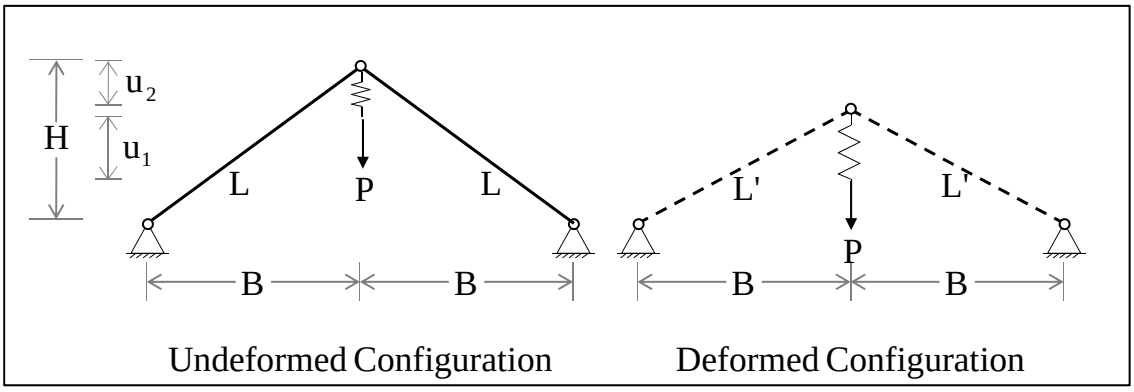

Figure A.3: Reference and deformed configurations of Von Mises truss. The first degree of freedom, u_1, is the displacement of the end of the spring. The second degree of freedom, u_2, is the displacement of the top node of the truss.

The strain energy stored in the system is the sum of the strain energies in the bars and in the spring, moving from the undeformed (reference) configuration to the deformed configuration. As shown in Figure A.3, the bars have length L in undeformed configuration and are deformed to length $L^{'}$ in the deformed configuration, while the spring experiences a change in length given by $u_1 - u_2$. The strain energy is therefore

$$U = 2U_{\text{bar}} + U_{\text{spring}} \tag{A.32}$$

where

$$U_{\text{spring}} = \frac{1}{2}C(u_1 - u_2)^2, \quad U_{\text{bar}} = \frac{1}{2}\frac{EA}{L}(L' - L)^2 \tag{A.33}$$

Using the geometry of the system shown in Figure A.3, the strain energy stored in each bar can be written

$$U_{\text{bar}} = \frac{1}{2}\frac{EA}{L}\left[\sqrt{(H - u_2)^2 + B^2} - L\right]^2 = \frac{1}{2}\frac{EA}{L}\left[\sqrt{L^2 - 2Hu_2 + u_2^2} - L\right]^2 \tag{A.34}$$

Combining Equations A.32, A.33, and A.34, one obtains the total strain energy in the system

$$U = \frac{EA}{L}\left[\sqrt{L^2 - 2Hu_2 + u_2^2} - L\right]^2 + \frac{1}{2}C(u_1 - u_2)^2 \tag{A.35}$$

The potential energy due to externally applied load, P, is given by

$$V = -Pu_1 \tag{A.36}$$

Inserting Equations A.35 and A.36 into Equation A.31, the total potential energy of the system is

$$\Pi = \frac{EA}{L}\left[\sqrt{L^2 - 2Hu_2 + u_2^2} - L\right]^2 + \frac{1}{2}C(u_1 - u_2)^2 - Pu_1 \tag{A.37}$$

The equations of equilibrium, and therefore the finite element matrices, will be derived by applying the Principle of Stationary Potential Energy to the potential energy of the system given in Equation A.37. The potential energy is stationary when small admissible variations of displacement are zero, hence

$$\frac{\partial \Pi}{\partial \mathbf{u}} = \mathbf{0} \tag{A.38}$$

The equilibrium equations can then be written in general matrix form

$$\mathbf{Ku} = \mathbf{q(u)} = \mathbf{p} \tag{A.39}$$

where $\mathbf{K}$ is the tangent stiffness matrix, $\mathbf{q(u)}$ is the internal load vector, $\mathbf{p}$ is the external load vector, and $\mathbf{u}$ is the displacement vector. The finite element matrices and vectors are given by the following expressions

$$K_{ij} = \frac{\partial U}{\partial u_i \partial u_j} \tag{A.40}$$

$$q_i = \frac{\partial U}{\partial u_i} \tag{A.41}$$

$$p_i = \frac{\partial V}{\partial u_i} \tag{A.42}$$

and the displacement vector is simply

$$\mathbf{u} = \begin{Bmatrix} u_1 \\ u_2 \end{Bmatrix} \tag{A.43}$$

Using Equations A.40-A.42, the equations of equilibrium for the Von Mises Truss are:

$$\begin{bmatrix} C & -C \\ \text{SYMM} & 2\left(\frac{EA}{L}\right)\left(1 + \frac{L(H^2 - L^2)}{(L^2 - 2Hu_2 + u_2^2)^{\frac{3}{2}}}\right) + C \end{bmatrix} \begin{Bmatrix} u_1 \\ u_2 \end{Bmatrix} = \begin{Bmatrix} P \\ 0 \end{Bmatrix} \tag{A.44}$$

Critical spring stiffness for the Von Mises truss

The stiffness of the spring in the Von Mises Truss dictates the overall behavior of the system. Stiffness below a critical value result in snap back behavior, while value above do not. Snap back begins when the tangent line to the solution curve is vertical. Using the equations of equilibrium, $\partial\Pi/\partial u_1 = 0$ and $\partial\Pi/\partial u_2 = 0$, with the total potential energy in Equation A.37 gives

$$\frac{\partial \Pi}{\partial u_1} = C(u_1 - u_2) - P = 0 \tag{A.45}$$

$$\frac{\partial \Pi}{\partial u_2} = 2\frac{EA}{L}\left(u_2 - H\right)\left(1 - \frac{L}{\sqrt{L^2 - 2Hu_2 + u_2^2}}\right) - C\left(u_1 - u_2\right) = 0 \tag{A.46}$$

Combining equations A.45 and A.46 gives

$$P = 2\frac{EA}{L}\left(u_2 - H\right)\left(1 - \frac{L}{\sqrt{L^2 - 2Hu_2 + u_2^2}}\right) \tag{A.47}$$

Snap back occurs in the u_1 component, which mathematically corresponds to $\partial\Pi/\partial u_1 = 0$. Additionally the onset of snap back occurs at $u_1 = H$ and $P = 0$ (i.e. when the structure reaches a plane configuration). First express P in terms of u_1 using Equation A.45, then differentiate with respect to P, i.e.

$$P = 2\frac{EA}{L}\left(u_1 - \frac{P}{C} - H\right)\left(1 - \frac{L}{\sqrt{L^2 - 2H\left(u_1 - \frac{P}{C}\right) + \left(u_1 - \frac{P}{C}\right)^2}}\right) \tag{A.48}$$

$$\begin{aligned} 1 = {} & 2\frac{EA}{L}\left(\frac{\partial u_1}{\partial P} - \frac{1}{C}\right)\left(1 - \frac{L}{\sqrt{L^2 - 2H\left(u_1 - \frac{P}{C}\right) + \left(u_1 - \frac{P}{C}\right)^2}}\right) + \\ & 2\frac{EA}{L}\left(u_1 - \frac{P}{C} - H\right)\left(\frac{L}{2}\left[L^2 - 2H\left(u_1 - \frac{P}{C}\right) + \left(u_1 - \frac{P}{C}\right)^2\right]^{-\frac{3}{2}}\right) \\ & \left(2H\left(\frac{1}{C} - \frac{\partial u_1}{\partial P}\right) + 2\left(u_1 - \frac{P}{C}\right)\left(\frac{\partial u_1}{\partial P} - \frac{1}{C}\right)\right) \end{aligned} \tag{A.49}$$

The critical case occurs when $\partial u_1/\partial P = 0$, $u_1 = H$, and $P = 0$), which yields the following expression for the C_{cr}

$$C_{\text{cr}} = 2\frac{EA}{L}\left(\frac{L}{\sqrt{L^2 - H^2}} - 1\right) \tag{A.50}$$

Finally, the values from Section 5.3 give the critical spring stiffness, $C_{\text{cr}} = 0.03094$.

Appendix B

The NLS++ code and sample input files

B.1 Implementation of NLS++ in C++

B.1.1 Main

```
1  // ------------------------------------------------- //
2  // Main.cpp - Main program for testing the NLS library. //
3  // ------------------------------------------------- //
4
5  // Include Statements
6  #include <stdio.h>
7  #include <stdlib.h>
8  #include <string.h>
9
10 // NLS library
11 #include <nls.h>
12 #include "modbeam2.h"
13 #include "modbeam3.h"
14 #include "modfunc.h"
15 #include "modfunc_mt.h"
16 #include "modst.h"
17 #include "utl.h"
18
19 // Main Program:
20 int main(int argc, char* argv[]) {
21
22 /* Declare variables */
23 char     sMod[80];
24 int      iAlg;
25 int      iLinSys;
26 int      iLinSysMaxIter;
27 double   iLinSysTol;
28 cControl *pcCtrl;
29 sControl sCtrl;
30 cModel   *pcModel;
31 cLinSys  *pcLinSys;
32
33 /* Check arguments */
```

```
34 if (argc < 3) {
35 fprintf(stderr, "Usage:\n   n1s model_file algorithm_file\n");
36 exit(-1);
37 }
38
39 /* Get arguments from argv */
40 char *pcModFileName = new char[strlen(argv[1])+1];
41 char *pcAlgFileName = new char[strlen(argv[2])+1];
42 strcpy(pcModFileName,argv[1]); //model file name
43 strcpy(pcAlgFileName,argv[2]); //algorithm file
44
45 /* Open model file */
46 FILE *fpi = fopen( pcModFileName, "r" );
47 if( !fpi )  {
48         printf( "\n\nModel file %s not found...\n\n", pcModFileName );
49         exit( -1 );
50 }
51 read_string( fpi, sMod );
52
53 /* Close model file */
54 fclose(fpi);
55
56 /* Create a new model */
57 if(strcmp(sMod, "function" ) == 0) {
58   pcModel = new cModelFunction( pcModFileName );
59 }  else if(strcmp(sMod, "function_mt" ) == 0)  {
60   pcModel = new cModelFunction_MT( pcModFileName );
61 }  else if(strcmp(sMod, "space_truss") == 0)   {
62   printf("\n\ncreating a cModelSpaceTruss (%s)\n", pcModFileName );
63   pcModel = new cModelSpaceTruss( pcModFileName );
64 }  else if( strcmp ( sMod, "plane_frame" ) == 0 )   {
65   printf("\n\ncreating a cModelBeam2D ( %s )\n", pcModFileName );
66   pcModel = new cModelBeam2D( pcModFileName );
67 }   else if( strcmp ( sMod, "space_frame" ) == 0 )   {
68   printf("\n\ncreating a cModelBeam3D ( %s )\n", pcModFileName );
69   pcModel = new cModelBeam3D( pcModFileName );
70 }   else   {
71   printf( "\n\nModel not available...\n\n" );
72   exit( -1 );  }
73
74 /* Initialize model */
75 pcModel->Init( );
76
77 /* Open algorithm file */
78 fpi = fopen( pcAlgFileName, "r" );
79 if( !fpi )  {
80   printf( "\n\nAlgoritm file %s not found...\n\n", pcAlgFileName );
81   exit( -1 );
```

```
82 }
83
84 /* Read the linear solver */
85 fscanf( fpi, "%d", &iLinSys);
86 if( (iLinSys == 1) || (iLinSys == 2) )
87   fscanf( fpi, "%d%lf", &iLinSysMaxIter, &iLinSysTol );
88
89 /* Create a new LinSys (LinearSystems) */
90 switch( iLinSys )  {
91 case 0:
92   pcLinSys = new cCroutProfile( );
93   break;
94 case 1:
95   pcLinSys = new cPCGProfile( iLinSysMaxIter, iLinSysTol );
96   break;
97 default:
98   printf( "\n\nLinear solver not available...\n\n" );
99   exit( -1 );
100 break;
101 }
102
103 /* Read algorithm type */
104 fscanf( fpi, "%d%d%lf", &iAlg, &sCtrl.UpdateType, &sCtrl.CtrlFactor );
105 if( iAlg == 1 ) fscanf( fpi, "%d%d", &sCtrl.CtrlEq, &sCtrl.CtrlType );
106 if( iAlg == 2 ) fscanf( fpi, "%d",   &sCtrl.CtrlType );
107 if( iAlg == 5 ) fscanf( fpi, "%lf",  &sCtrl.CtrlIniFactor );
108 if( iAlg == 6 ) fscanf( fpi, "%lf",  &sCtrl.CtrlIniFactor );
109 if( iAlg == 7 ) fscanf( fpi, "%d%d", &sCtrl.CtrlEq, &sCtrl.CtrlType );
110 if( iAlg == 8 ) fscanf( fpi, "%lf",  &sCtrl.CtrlIniFactor );
111 fscanf(fpi,"%d%d%lf",&sCtrl.NumMaxStep,&sCtrl.NumMaxIte,&sCtrl.Tol );
112
113
114 /* Close algorithm file */
115 fclose( fpi );
116
117 /* Create new control */
118 switch( iAlg )  {
119 case 0:
120   pcCtrl = new cNewtonRaphson( pcModel, &sCtrl, pcLinSys );
121   break;
122 case 1:
123   pcCtrl = new cDisplacementControl( pcModel, &sCtrl, pcLinSys );
124   break;
125 case 2:
126   pcCtrl = new cArcLengthControl( pcModel, &sCtrl, pcLinSys );
127   break;
128 case 3:
129   pcCtrl = new cWorkControl( pcModel, &sCtrl, pcLinSys );
```

```
130   break;
131 case 4:
132   pcCtrl = new cGenDisplacementControl(pcModel, &sCtrl, pcLinSys);
133   break;
134 case 5:
135   pcCtrl = new cOrthResidualControl( pcModel, &sCtrl, pcLinSys );
136   break;
137 case 6:
138   pcCtrl = new cStrainRatioControl( pcModel, &sCtrl, pcLinSys );
139   break;
140 case 7:
141   pcCtrl = new cStrainControl( pcModel, &sCtrl, pcLinSys );
142   break;
143 case 8:
144   pcCtrl = new cOldOrthResidualControl(pcModel, &sCtrl, pcLinSys );
145   break;
146 default:
147   printf( "\n\nAlgorithm not available...\n\n" );
148   exit( -1 );
149   break;
150 }
151
152 /* Run */
153 pcCtrl->Solver( );
154
155 /* Free memory */
156 delete pcCtrl;
157 delete pcModel;
158 return 1;
159 }
```

B.1.2 Model classes: headers and definitions

```
1  // ------------------------------------------------- //
2  // Model.h - Model Class Header //
3  // ------------------------------------------------- //
4  #ifndef _MODEL_H_
5  #define _MODEL_H_
6  #include <stdio.h>
7
8  class cLinSys;
9
10 class cModel {
11 protected:
12 int      _iNumEpsEq;
13 int      _iNumEq;
14 int      _iNumGra;
15 FILE   **_afOut;
16
17 public:
18 cModel   ( void ) { };
19 virtual ~cModel   ( void );
20 int     NumEpsEq  ( void );
21 int     NumEq     ( void );
22 virtual int Profile   ( int * );
23 virtual int SparsityPattern  ( int ** );
24 virtual void Convergence ( double, double * );
25 virtual void Init ( void )= 0;
26 virtual void InternalVector ( double *, double  * ) = 0;
27 virtual void Reference ( double * ) = 0;
28 virtual void TangentMatrix ( double *, cLinSys * ) = 0;
29 virtual void StrainVector ( double *, double  * ) { }
30 virtual void DeltaStrainVector ( double *, double  *, double * ){ }
31
32 protected:
33 void InitFile ( void );
34 };
35 #endif
36
37 // ------------------------------------------------- //
38 // Model.cpp - Model Class Definition //
39 // ------------------------------------------------- //
40 #include <stdio.h>
41 #include "mempack/mempack.h"
42 #include "model/model.h"
43
44 cModel :: ~cModel ( void ) {
45   for( int i = 0; i < _iNumGra; i++ ) fclose( _afOut[i] );
46   MemFree( _afOut );
47   _iNumEq = 0;
```

```
48 }
49
50 int cModel :: NumEpsEq ( void ) {
51   return( _iNumEpsEq );
52 }
53
54 int cModel :: NumEq ( void ) {
55   return( _iNumEq );
56 }
57
58 int cModel :: Profile ( int *piProfile ) {
59   // Generate profile vector (full matrix)
60   if (piProfile==0) return( 0 );
61   for( int i = 0; i < _iNumEq; i++ ) piProfile[i] = 0;
62   return( 1 );
63 }
64
65 int cModel :: SparsityPattern ( int **piSparsity ) {
66   // Generate sparticity matrix (full matrix)
67   if (piSparsity==0) return( 0 );
68   for( int i = 0; i < _iNumEq; i++ ) {
69     for( int j = 0; j < _iNumEq; j++ )
70       piSparsity[i][j] = 1;
71   }
72   return( 1 );
73 }
74
75 void cModel :: Convergence ( double dFactor, double *pdSol ) {
76   for( int i = 0; i < _iNumGra; i++ )
77     fprintf( _afOut[i], "%f   %f\n", pdSol[i], dFactor );
78   }
79
80 void cModel :: InitFile ( void ) {
81   _afOut = (FILE **) MemAlloc( _iNumEq, sizeof(FILE *) );
82   for( int i = 0; i < _iNumEq; i++ )  {
83     char fn[50];
84     sprintf( fn, "nls%d.out", i );
85     _afOut[i] = fopen( fn, "w" );
86     fprintf( _afOut[i], "%f   %f\n", 0.0, 0.0 );
87   }
88 }
```

```
1  // ------------------------------------------------- //
2  // ModST.h - Space Truss Class Header //
3  // ------------------------------------------------- //
4  #ifndef _MODST_H_
5  #define _MODST_H_
6
7  #include <nls.h>
8
9  class cLinSys;
10
11 class cModelSpaceTruss : public cModel {
12
13 protected:
14 int _iNumNodes; // Number of nodes
15 int _iNumDofsOfNodes;
16 int _iNumElms;  // Number of elements
17 double **_paCoord; // List of coordinates (x, y and z)
18 double **_paLoad; // List of loads (px, py and pz)
19 int    **_paDof; // List of degree of freedom (u, v and w)
20 int    **_paInc; // List of incidences (initial and final)
21 double **_paProp; // List of properties (EA, N0 and L0)
22 int    **_paGra; // List of plots (node and direction)
23
24 public:
25 cModelSpaceTruss  ( char *filename );
26 ~cModelSpaceTruss  ( void );
27  void Init ( void );
28 void Convergence ( double,   double  * );
29 void InternalVector ( double *, double  * );
30 void Reference ( double * );
31 void TangentMatrix ( double *, cLinSys * );
32 void DeltaStrainVector ( double *, double  *, double * );
33 void StrainVector ( double *, double  * );
34
35 protected: double Ldef ( int, double * );
36
37 };
38 #endif
39
40 // ------------------------------------------------- //
41 // ModST.cpp - Space Truss Class Definition //
42 // ------------------------------------------------- //
43 #include <math.h>
44 #include <stdlib.h>
45 #include "modst.h"
46 #include "utl.h"
47
48 /* Constructor for space truss objects */
```

```
49 cModelSpaceTruss :: cModelSpaceTruss ( char *filename )
50 : _iNumDofsOfNodes(3), cModel ( ) {
51 int   i, j;
52 int   id;
53 int   num_load, num_sup;
54 char mod_type[80];
55 FILE *in = NULL;
56
57 // Read data file
58 in = fopen( filename, "r" );
59 if( !in )  {
60   printf( "\n\n ### %s not defined !!! ###\n\n", filename );
61   exit( -1 );
62 }
63
64 read_string( in, mod_type );
65 fscanf( in, "%d", &_iNumNodes );
66 _paCoord = (double**) MemAlloc( _iNumNodes, sizeof(double*) );
67 _paLoad  = (double**) MemAlloc( _iNumNodes, sizeof(double*) );
68 _paDof   = (int**)    MemAlloc( _iNumNodes, sizeof(int*) );
69
70 for( i = 0; i < _iNumNodes; i++ )  {
71   _paCoord[i] = (double*) MemAlloc( 3, sizeof(double) );
72   for( j = 0; j < 3; j++ )
73     fscanf( in, "%lf", &_paCoord[i][j] );
74
75   _paLoad[i] = (double*) MemAlloc( 3, sizeof(double) );
76   _paDof[i]  = (int*)    MemAlloc( 3, sizeof(int) );
77   for( j = 0; j < 3; j++ ) {
78     _paLoad[i][j] = 0.;
79     _paDof[i][j]  = 0;
80   }
81 }
82
83 fscanf( in, "%d", &num_sup );
84 for( i = 0; i < num_sup; i++ ) {
85   fscanf( in, "%d", &id );
86   for( j = 0; j < 3; j++ ) fscanf( in, "%d", &_paDof[id][j] );
87 }
88
89 fscanf( in, "%d", &num_load );
90 for( i = 0; i < num_load; i++ ) {
91   fscanf( in, "%d", &id );
92   for( j = 0; j < 3; j++ ) fscanf( in, "%lf", &_paLoad[id][j] );
93 }
94
95 fscanf( in, "%d", &_iNumElms );
96 _paInc  = (int**)    MemAlloc( _iNumElms, sizeof(int*) );
```

```
97  _paProp = (double**) MemAlloc( _iNumElms, sizeof(double*) );
98
99  for( i = 0; i < _iNumElms; i++ ) {
100   _paInc[i]  = (int*)    MemAlloc( 2, sizeof(int) );
101   _paProp[i] = (double*) MemAlloc( 3, sizeof(double) );
102   for( j = 0; j < 2; j++ ) fscanf( in, "%d",  &_paInc[i][j] );
103   for( j = 0; j < 2; j++ ) fscanf( in, "%lf", &_paProp[i][j] );
104 }
105
106 fscanf( in, "%d", &_iNumGra );
107 _paGra = (int**) MemAlloc( _iNumGra, sizeof(int*) );
108 for( i = 0; i < _iNumGra; i++ )  {
109   _paGra[i] = (int*) MemAlloc( 2, sizeof(int) );
110   for( j = 0; j < 2; j++ ) fscanf( in, "%d", &_paGra[i][j] );
111 }
112
113 // Close data file
114 fclose( in );
115
116 // Compute initial length of the bars
117 for( i = 0; i < _iNumElms; i++ )  {
118   _paProp[i][2] = 0.;
119   for( j = 0; j < 3; j++ )
120   _paProp[i][2]+=pow(_paCoord[_paInc[i][1]][j]-
121     _paCoord[_paInc[i][0]][j], 2.0 );
122   _paProp[i][2] = sqrt( _paProp[i][2] );
123 }
124
125 // Generate d.o.f.
126 _iNumEq = 0;
127 for( i = 0; i < _iNumNodes; i++ )
128   for( j = 0; j < 3; j++ )
129     _paDof[i][j] = ( _paDof[i][j] == 0 ) ? _iNumEq++ : -1;
130
131 // Define number of strain components
132 _iNumEpsEq = _iNumElms;
133 }
134
135 /* Destructor for bar element objects */
136 cModelSpaceTruss :: ~cModelSpaceTruss ( void ) {
137 int i;
138
139 //Free all memory
140 for( i = 0; i < _iNumNodes; i++ ) {
141   MemFree( _paCoord[i] );
142   MemFree( _paLoad[i] );
143   MemFree( _paDof[i] );
144 }
```

```
145
146 MemFree( _paCoord );
147 MemFree( _paLoad );
148 MemFree( _paDof );
149
150 for( i = 0; i < _iNumElms; i++ ) {
151   MemFree( _paInc[i] );
152   MemFree( _paProp[i] );
153 }
154
155 MemFree( _paInc );
156 MemFree( _paProp );
157
158 for( i = 0; i < _iNumGra; i++ )
159   MemFree( _paGra[i] );
160
161 MemFree( _paGra );
162 }
163
164 /* Initalize output file */
165 void cModelSpaceTruss :: Init ( void ) {
166
167 char  fn[50];
168 char *dir[3] = { "u", "v", "w" };
169 _afOut = (FILE **) MemAlloc( _iNumGra, sizeof(FILE *) );
170
171 for( int i = 0; i < _iNumGra; i++ ) {
172   sprintf( fn, "node%d%s.out", _paGra[i][0], dir[_paGra[i][1]] );
173   _afOut[i] = fopen( fn, "w" );
174   fprintf( _afOut[i], "%f   %f\n", 0.0, 0.0 );
175 }
176 }
177
178 /* Print convergence data */
179 void cModelSpaceTruss :: Convergence
180 ( double dFactor, double *pdSol ) {
181 for( int i = 0; i < _iNumGra; i++ )  {
182   int    dof = _paDof[_paGra[i][0]][_paGra[i][1]];
183   double u   = (dof >= 0) ? pdSol[dof] : 0.0;
184   fprintf( _afOut[i], "%f   %f\n", u, dFactor );
185 }
186 }
187
188 /* Computes strain vetcor */
189 void cModelSpaceTruss :: StrainVector ( double *u, double *e ) {
190 for( int i = 0; i < _iNumElms; i++ )
191 e[i] = (Ldef( i, u ) - _paProp[i][2]) / _paProp[i][2];
192 }
```

```
193
194 /* Computes incremental strain vector*/
195 void cModelSpaceTruss :: DeltaStrainVector
196 ( double *u, double *du, double *de ) {
197
198 int    i, j, k;
199 int    edof[6];
200 double cdef[2][3];
201 double duelm[2][3];
202
203 for( i = 0; i < _iNumElms; i++ )  {
204   double ldef = Ldef( i, u );
205   for( j = 0; j < 2; j++ )
206   for( k = 0; k < 3; k++ )    {
207     edof[3*j+k] = _paDof[_paInc[i][j]][k];
208     cdef[j][k]  = _paCoord[_paInc[i][j]][k];
209     duelm[j][k] = 0.0;
210     if( edof[3*j+k] >= 0 ) {
211       cdef[j][k] +=  u[edof[3*j+k]];
212       duelm[j][k] = du[edof[3*j+k]];
213     }
214   }
215   de[i] = 0.0;
216   for( j = 0; j < 3; j++ )
217     de[i] += (cdef[1][j] - cdef[0][j]) *
218       (duelm[1][j] - duelm[0][j]);
219   de[i] *= 1.0 / (ldef * _paProp[i][2]);
220 }
221 }
222
223 /* Computes internal force vector for a TL bar element */
224 void cModelSpaceTruss :: InternalVector
225 ( double *u, double *f ) {
226
227 int    i, j, k;
228 int    edof[6];
229 double fe[6];
230 double cdef[2][3];
231 MathVecZero( _iNumEq, f );
232
233 for( i = 0; i < _iNumElms; i++ ) {
234   double ldef = Ldef( i, u );
235   double n = _paProp[i][0] / _paProp[i][2] *
236     (ldef - _paProp[i][2]) + _paProp[i][1];
237   double coef = n / ldef;
238   for( j = 0; j < 2; j++ )
239     for( k = 0; k < 3; k++ ) {
240       edof[3*j+k] = _paDof[_paInc[i][j]][k];
```

```
241         cdef[j][k]  = _paCoord[_paInc[i][j]][k];
242         if( edof[3*j+k] >= 0 ) cdef[j][k] += u[edof[3*j+k]];
243       }
244     for( j = 0; j < 3; j++ ) {
245       fe[j]   = - coef * (cdef[1][j] - cdef[0][j]);
246       fe[j+3] =   coef * (cdef[1][j] - cdef[0][j]);
247     }
248     for( j = 0; j < 6; j++ )
249       if( edof[j] >= 0 ) f[edof[j]] += fe[j];
250   }
251   }
252
253   /* Computes reference vector for a bar element*/
254   void cModelSpaceTruss :: Reference ( double *pdReference ) {
255   int i,j;
256   for( i = 0; i < _iNumNodes; i++ )
257     for( j = 0; j < _iNumDofsOfNodes; j++ )
258       if( _paDof[i][j] >= 0 )
259         pdReference[_paDof[i][j]] = _paLoad[i][j];
260   }
261
262   /* Computes tangent stiffness matrix for a TL bar element */
263   void cModelSpaceTruss :: TangentMatrix
264   ( double *u, cLinSys *kt ) {
265
266   int    i, j, k;
267   int    edof[6];
268   double ke[6][6];
269   double cdef[2][3];
270
271   //initialize total matrix
272   kt->Zero();
273
274   for( i = 0; i < _iNumElms; i++ ) {
275     //updated length of the element
276     double ldef = Ldef( i, u );
277     //stress in element (Hooke's Law)
278     double n    = _paProp[i][0] / _paProp[i][2] *
279       (ldef - _paProp[i][2]) + _paProp[i][1];
280
281     //compute deformed length
282     for( j = 0; j < 2; j++ ) {
283       for( k = 0; k < 3; k++ ) {
284         edof[3*j+k] = _paDof[_paInc[i][j]][k];
285         cdef[j][k]  = _paCoord[_paInc[i][j]][k];
286         if( edof[3*j+k] >= 0 )
287          cdef[j][k] += u[edof[3*j+k]];
288       }
```

```
289    }
290
291    //element tangent matrix
292    for( j = 0; j < 3; j++ ) {
293      for( k = 0; k < 3; k++ ) {
294        double coef = (_paProp[i][0] - _paProp[i][1]) /
295         (ldef * ldef * ldef) * (cdef[1][j] - cdef[0][j]) *
296         (cdef[1][k] - cdef[0][k]);
297
298        ke[j][k] = coef;
299        if( j == k ) ke[j][k] += n / ldef;
300
301        ke[j+3][k] = - coef;
302        if( j == k ) ke[j+3][k] -= n / ldef;
303
304        ke[j][k+3] = - coef;
305        if( j == k ) ke[j][k+3] -= n / ldef;
306
307        ke[j+3][k+3] = coef;
308        if( j == k ) ke[j+3][k+3] += n / ldef;
309      }
310    }
311
312    //add any new contribution to total tangent matrix
313    for( j = 0; j < 6; j++ ) {
314      if( edof[j] >= 0 ) {
315        for( k = 0; k < 6; k++ ) {
316          if( edof[k] >= 0 && edof[j] >= edof[k] )
317            kt->AddA(edof[j],edof[k],ke[j][k]);
318        }
319      }
320    }
321  }
322  }
323
324  /* Computes deformed length of a bar element */
325  double cModelSpaceTruss :: Ldef ( int elm, double *u ) {
326  int    i, j;
327  int    edof[6];
328  double cdef[2][3];
329
330  for( i = 0; i < 2; i++ ) {
331    for( j = 0; j < 3; j++ ) {
332
333    //gets element dofs for this element
334    edof[3*i+j] = _paDof[_paInc[elm][i]][j];
335
336    //gets node locations for this element
```

```
337    cdef[i][j]  = _paCoord[_paInc[elm][i]][j];
338
339    //if its a free dof then edof[3*i+j] != 0, add the
340      //previous displacement associated with that dof
341    if( edof[3*i+j] >= 0 ) cdef[i][j] += u[edof[3*i+j]];
342    }
343  }
344
345  double ldef = 0.0;
346  for( i = 0; i < 3; i++ )
347    ldef += pow( cdef[1][i] - cdef[0][i], 2.0 );
348
349  //returns new length of the element - deformed length
350  return( sqrt( ldef ) );
351  }
```

```
1  // ------------------------------------------------- //
2  // Modfunc.h - Unidimensional Function Class Header //
3  // ------------------------------------------------- //
4  #ifndef _MODFUNC_H_
5  #define _MODFUNC_H_
6
7  #include <nls.h>
8
9  class cLinSys;
10
11 class cModelFunction : public cModel {
12
13 public:
14 cModelFunction ( char *filename );
15 ~cModelFunction ( void );
16 void Init ( void );
17 void InternalVector ( double *, double  * );
18 void Reference ( double * );
19 void TangentMatrix ( double *, cLinSys * );
20 };
21 #endif
22
23 // ------------------------------------------------- //
24 // Modfunc.cpp - Unidimensional Function Class Definition //
25 // ------------------------------------------------- //
26 #include <math.h>
27 #include "modfunc.h"
28
29 /* Constructor for unidimensional function object */
30 cModelFunction :: cModelFunction ( char *filename ) :
31 cModel ( ) {
32 _iNumEq = _iNumGra = 1;
33 }
34
35 /* Destructor for unidimensional function object */
36 cModelFunction :: ~cModelFunction ( void ) { }
37
38 /* Initialize output file */
39 void cModelFunction :: Init ( void ) {
40 _afOut = (FILE **) MemAlloc( _iNumEq, sizeof(FILE *) );
41 for( int i = 0; i < _iNumEq; i++ )  {
42 char fn[50];
43 sprintf( fn, "function_%d.out", i );
44 _afOut[i] = fopen( fn, "w" );
45 fprintf( _afOut[i], "%f   %f\n", 0.0, 0.0 );  }
46 }
47
48 /* Compute internal force vector */
```

```
49 void cModelFunction :: InternalVector ( double *u, double *f ) {
50 double x = u[0] - 1.0;
51 f[0] = (x < 0 ? 3 : -3) * pow( fabs(x), 1.0/3.0 ) + 4 * x + 1;
52 }
53
54 /* Compute reference load vector*/
55 void cModelFunction :: Reference ( double *f ) {
56 f[0] = 1; //unit load
57 }
58
59 /* Compute tangent matrix (derivative of internal force */
60 void cModelFunction :: TangentMatrix ( double *u, cLinSys *kt ) {
61 double x = u[0] - 1.0; //shift curve
62 double k = x == 0 ? -1e32 : -1 / pow( fabs(x), 2/3.0 ) + 4;
63 kt->Zero(); //zero tangent matrix
64 kt->AddA(0,0,k); //add component to total matrix
65 }
```

```
1  // ------------------------------------------------------ //
2  // Modfunc_mt.h - Two-dimensional Function Class Header //
3  // ------------------------------------------------------ //
4  #include <nls.h>
5
6  class cLinSys;
7
8  class cModelFunction_MT : public cModel {
9  public:
10 cModelFunction_MT ( char *filename );
11 ~cModelFunction_MT ( void );
12
13 void Init           ( void );
14 void InternalVector ( double *, double  * );
15 void Reference      ( double * );
16 void TangentMatrix  ( double *, cLinSys * );
17 };
18
19 // ------------------------------------------------------ //
20 // Modfunc_mt.h - Two-dimensional Function Class Definition //
21 // ------------------------------------------------------ //
22 #include <math.h>
23 #include "modfunc_mt.h"
24
25 /* Constructor for two-dimensional function */
26 cModelFunction_MT :: cModelFunction_MT ( char *filename ) : cModel ( ) {
27 _iNumEq = _iNumGra = 2;
28 }
29 /* Destructor for two-dimensional function */
30 cModelFunction_MT :: ~cModelFunction_MT ( void ) { }
31
32 /* Initialize output file */
33 void cModelFunction_MT :: Init ( void ) {
34 InitFile( );
35 }
36
37 /* Compute internal load vector */
38 void cModelFunction_MT :: InternalVector ( double *u, double *f ) {
39 f[0] = 10.0 * u[0] + 0.4 * pow(u[1], 3) - 5.0 * pow(u[1], 2);
40 f[1] = 0.4 * pow(u[0], 3) - 3.0 * pow(u[0],2) + 10 * u[1];
41 }
42
43 /* Compute external load vector */
44 void cModelFunction_MT :: Reference ( double *f ) {
45 f[0] = 40.0;
46 f[1] = 15.0;
47 }
48
```

```
49 /* Compute tangent stiffness matrix */
50 void cModelFunction_MT :: TangentMatrix ( double *u, cLinSys *kt ) {
51 double k11 = 10.0;
52 double k12 = 1.2 * u[1] * u[1] - 10.0 * u[1];
53 double k21 = 1.2 * u[0] * u[0] -  6.0 * u[0];
54 double k22 = 10.0;
55 kt->FillA(0, 0, k11);
56 kt->FillA(0, 1, k12);
57 kt->FillA(1, 0, k21);
58 kt->FillA(1, 1, k22);
59 }
```

B.1.3 Linear system class: header and definition

```
1  // ------------------------------------------------------ //
2  // Crout.h - This header file contains the public data structure
3  //          definitions for the crout linear system class.
4  // ------------------------------------------------------ //
5  #ifndef _CROUT_LINSYS_H_
6  #define _CROUT_LINSYS_H_
7  #include "linearsystem.h"
8
9  class cModel;
10 // ------------------------------------------------------
11 // Crout Linear System class:
12
13 class cCroutProfile : public cLinSys {
14 protected:
15 int      *_piProfile;
16 double   **_paA;
17 int      _isFactorized;
18
19 public:
20 cCroutProfile    ( void );
21 virtual ~cCroutProfile    ( void );
22 virtual  void Init  ( cModel * );
23 virtual  void Zero  ( void );
24 virtual  void AddA  (int, int, double );
25 virtual  void FillA ( int, int, double);
26 virtual  void Solve  ( double *x );
27 virtual  void Solve  ( double *b, double *x );
28 virtual  void Solve2x2NonSym( double *b, double *x );
29 };
30 #endif
31
32 // ------------------------------------------------------ //
33 // Crout.cpp - Crout linear system class definition
34 // ------------------------------------------------------ //
35 #include "linsys/crout.h"
36 #include "mempack/mempack.h"
37 #include "mathpack/mathpack.h"
38 #include "model/model.h"
39 #include <math.h>
40 #include <linsys.h>
41
42 // Public functions:
43
44 cCroutProfile :: cCroutProfile ( )
45 : _piProfile(0),  _paA(0), _isFactorized(0) { }
46
47 cCroutProfile :: ~cCroutProfile ( void ) {
```

```
48 if (_paA!=0) MemProfileFree( _paA, _iNumEq, _piProfile );
49 if (_piProfile!=0) MemFree( _piProfile );
50 }
51
52 /* Initialize */
53 void cCroutProfile :: Init ( cModel *pcModel ) {
54 // Release previously allocated values
55 if (_paA!=0) MemProfileFree( _paA, _iNumEq, _piProfile );
56 if (_piProfile!=0) MemFree( _piProfile );
57
58 //assign daat
59 _iNumEq = pcModel->NumEq( );
60 _piProfile = (int *) MemAlloc( _iNumEq, sizeof(int) );
61 pcModel->Profile( _piProfile );
62 _paA = MemProfileAlloc( _iNumEq, _piProfile );
63 }
64
65 /* Zero matrix components */
66 void cCroutProfile :: Zero ( ) {
67 int i,j;
68 for(i=0;i<_iNumEq;i++) {
69   for(j=_piProfile[i];j<=i;j++) {
70     _paA[i][j] = 0.;
71   }
72 }
73 _isFactorized = 0;
74 }
75
76 /* Adds component to stiffness matrix*/
77 void cCroutProfile :: AddA ( int i, int j, double k ) {
78 if (i <= j) {      // in upper triangle or on main diagonal
79   if (i >= _piProfile[j]) {
80     _paA[j][i] += k;
81   } else {
82     // element not defined;
83   }
84 } else {      // in lower triangle
85   if (j >= _piProfile[i]) {
86     _paA[i][j] += k;
87   } else {
88     // element not defined;
89   }
90 }
91 _isFactorized = 0;
92 }
93
94 /* Inserts component in stiffness matrix */
95 void cCroutProfile :: FillA ( int i, int j, double k ) {
```

```
_paA[i][j] = k;
_isFactorized = 0;
}

/* Solves Ax=b */
void cCroutProfile :: Solve ( double *_pdDx ) {
if( _isFactorized ) {
  Crout_Profile( 3, _paA, _pdDx, _iNumEq, _piProfile );
} else {
  if (_pdDx==0)
    Crout_Profile( 2, _paA, _pdDx, _iNumEq, _piProfile );
  else
    Crout_Profile( 1, _paA, _pdDx, _iNumEq, _piProfile );
  _isFactorized = 1;
  }
}

/* Solves Ax=b */
void cCroutProfile :: Solve ( double *b, double *_pdDx ) {
 //copy b to _pdDx, so now _pdDx = b
MathVecAssign( _iNumEq, _pdDx, b );
Solve( _pdDx );
}

/* Solves 2x2 non symmetric system by inverting stiffness matrix*/
void cCroutProfile :: Solve2x2NonSym( double *b, double *_pdDx ) {
double x, y, det;
det = _paA[0][0] * _paA[1][1] - _paA[0][1] * _paA[1][0];

if ( abs ( det ) < 1.0e-15 ) {
  b[0] = b[1] = 0.0;
} else {
  x = ( b[0] * _paA[1][1] - b[1] * _paA[0][1] ) / det;
  y = ( b[1] * _paA[0][0] - b[0] * _paA[1][0] ) / det;
  _pdDx[0] = x;       _
  pdDx[1] = y;
}
}
```

B.1.4 Nonlinear solution schemes classes: headers and definitions

```
1  // ------------------------------------------------------ //
2  // Ctrl.h - This header file contains the public data structure
3  //          definitions for the control class.
4  // ------------------------------------------------------ //
5  #ifndef _CTRL_H_
6  #define _CTRL_H_
7
8  class cModel; class cLinSys;
9
10 typedef enum _updatetype {
11 STANDARD,
12 MODIFIED
13 } eUpdate;
14
15 typedef enum _ctrltype {
16 CONSTANT,
17 VARIABLE
18 } eCtrlType;
19
20 typedef struct _control {
21 int       CtrlEq;
22 double    CtrlFactor;
23 double    CtrlIniFactor;
24 eCtrlType CtrlType;
25 int       NumMaxIte;
26 int       NumMaxStep;
27 double    Tol;
28 eUpdate   UpdateType;
29 } sControl;
30
31 //Control Class:
32 class cControl {
33 protected:
34 sControl  _sControl;
35 int       _iConvFlag;
36 int       _iCurrIte;
37 int       _iCurrStep;
38 cModel   *_pcModel;
39 int       _iNumEq;
40 cLinSys  *_pcLinSys;
41 double   *_pdReference;
42 double    _dTotFactor;
43
44 public:
45 cControl    ( cModel *, sControl *, cLinSys * );
46 virtual ~cControl    ( void );
47 virtual  void Solver ( void ) = 0;
```

```
48 };
49 #endif
50
51 // ------------------------------------------------------- //
52 // Ctrl.cpp - Control class definition
53 // ------------------------------------------------------- //
54 #include "ctrl.h"
55 #include "mempack/mempack.h"
56 #include "model/model.h"
57 #include "linsys/linearsystem.h"
58
59 /* Constructor for control object */
60 cControl :: cControl ( cModel *pcModel, sControl *psControl, cLinSys *pcLinSys ) {
61 _pcModel  = pcModel;
62 _sControl = *psControl;
63 _pcLinSys = pcLinSys;
64 _iNumEq   = pcModel->NumEq( );
65 _pdReference = (double *) MemAlloc( _iNumEq, sizeof(double) );
66 pcModel->Reference( _pdReference );  _pcLinSys->Init(pcModel);
67 }
68
69 /* Destructor for control object */
70 cControl :: ~cControl ( void ) {
71 MemFree( _pdReference );
72 }
```

```
1  // ------------------------------------------------------ //
2  // UniSch.h - This header file contains the public data structure
3  //            definitions for the control problem class.
4  // ------------------------------------------------------ //
5  #ifndef _UNISCH_H_
6  #define _UNISCH_H_
7  #include "ctrl/ctrl.h"
8
9  class cLinSys;
10
11 class cUnifiedSchemes : public cControl {
12 protected:
13 // Incremental state vector due to reference vector
14 double  *_pdDx1;
15 // Incremental state vector due to unbalance vector
16 double  *_pdDx2;
17 // Incremental load factor vector
18 double   _dLambda;
19 // Euclidean norm of reference vector
20 double   _dNormReference;
21 // External contribution for unbalance vector
22 double  *_pdUe;
23 // Internal contribution for unbalance vector
24 double  *_pdUi;
25 // Total state variable vector - total displacement
26 double  *_pdX;
27 // Total state variable vector- residual
28 double  *_pdR;
29
30 public:
31 cUnifiedSchemes ( cModel *, sControl *, cLinSys * );
32 virtual ~cUnifiedSchemes ( void );
33 void Solver ( void );
34
35 protected:
36 int  CheckConvergence ( void );
37 virtual void Lambda   ( void ) = 0;
38 };
39 #endif
40
41 // ------------------------------------------------------ //
42 // UniSch.cpp - This file contains base class routines
43 // which are common among a number of different control
44 // algorithm solution classes.
45 // ------------------------------------------------------ //
46 #include <math.h>
47 #include <stdlib.h>
48 #include <stdio.h>
```

```
49 #include <time.h>
50 #include "ctrl/unisch/unisch.h"
51 #include "mathpack/mathpack.h"
52 #include "mempack/mempack.h"
53 #include "linsys/linearsystem.h"
54 #include "model/model.h"
55 #define TIMING
56
57 /* Constructor for Unified Schemes object */
58 cUnifiedSchemes :: cUnifiedSchemes
59 ( cModel *pcModel, sControl *psControl,cLinSys *pcLinSys ) :
60 cControl( pcModel, psControl, pcLinSys ) {
61 //allocate space
62 _pdDx1 = (double *) MemAlloc( _iNumEq, sizeof(double) );
63 _pdDx2 = (double *) MemAlloc( _iNumEq, sizeof(double) );
64 _pdUe = (double *) MemAlloc( _iNumEq, sizeof(double) );
65 _pdUi = (double *) MemAlloc( _iNumEq, sizeof(double) );
66 _pdR = (double *) MemAlloc( _iNumEq, sizeof(double) );
67 _pdX = (double *) MemAlloc( _iNumEq, sizeof(double) );
68 _dTotFactor = 0.0;
69 _dNormReference = MathVecNorm( _iNumEq, _pdReference );
70 }
71
72 /* Destructor for Unified Schemes object */
73 cUnifiedSchemes :: ~cUnifiedSchemes ( void ) {
74 //free memory
75 MemFree( _pdDx1 );
76 MemFree( _pdDx2 );
77 MemFree( _pdUe );
78 MemFree( _pdUi );
79 MemFree( _pdR );
80 MemFree( _pdX );
81 }
82
83 /* Solve nonlinear system */
84 void cUnifiedSchemes :: Solver ( void ) {
85
86 #ifdef TIMING
87 double tic = clock();
88 #endif
89
90 int  ite;
91
92 //initialize current step
93 _iCurrStep = 1;
94
95 //Outer Loop - Continues until max number of steps
96 //is reached or until solution diverges
```

```
97  do {
98  //initialize current iteration and set convergence to false
99  _iConvFlag = 0;
100 _iCurrIte  = 1;
101
102   //Inner loop - Continues until convergence achieved
103   //or max permitted iterations is exceeded
104   do {
105   //Compute tangent matrix if first iteration or
106   //if this is a standard update
107   //If this is modified then only compute tangent
108   //matrix on first iteration
109   if(_iCurrIte == 1 || _sControl.UpdateType == STANDARD) {
110         //model computes tangent matrix
111     _pcModel->TangentMatrix(_pdX, _pcLinSys);
112     //Linear system solves Ax=b
113     _pcLinSys->Solve(_pdReference, _pdDx1);
114   }
115
116 if(_iCurrIte == 1) {
117 //zero out _pdDx2
118 MathVecZero(_iNumEq, _pdDx2);
119 } else {
120 _pcLinSys->Solve( _pdR, _pdDx2 );
121 }
122
123 //compute lambda for current iteration, stores value in _dLambda
124 Lambda( );
125
126 //update total load factor for this step (outer loop)
127 _dTotFactor += _dLambda;
128
129 //compute _pdUe = _dLambda*_pdReference
130 //external load vector for this iteration
131 MathVecAdd1( _iNumEq, _pdUe, _dLambda, _pdReference );
132
133 //compute _pdX = _dLambda*_pdDx1 + _pdDx2
134 MathVecAdd2( _iNumEq, _pdX,  _dLambda, _pdDx1, _pdDx2 );
135
136 //sets up internal vector, gets stored in _pdUi
137 _pcModel->InternalVector( _pdX, _pdUi );
138 //calculate residual
139 } while( !CheckConvergence( ) && ++_iCurrIte <= _sControl.NumMaxIte );
140
141 //print results of convergering or not
142 if( _iConvFlag ) {
143 _pcModel->Convergence( _dTotFactor, _pdX );
144 //printf( "\n" );
```

```
} else {
printf( "\n\n\t ### Convergence not achieved (Step %d) !!! ###\n\n", _iCurrStep );
break;
}
//next step
} while( ++_iCurrStep <= _sControl.NumMaxStep );

#ifdef TIMING
double toc = clock();
printf("Solution time=%f\n", (toc-tic)/CLOCKS_PER_SEC);
#endif
}

/* Checks if iterative procedure has converged */
int cUnifiedSchemes :: CheckConvergence ( void ) {

//assign external load vector to residual, external load vector
//is all of the previous external loads plus the
//lambda*referenceLoad for this iteration
MathVecAssign( _iNumEq, _pdR, _pdUe );

//right hand side of governing equation, ie K_(j-1)*deltaU_j = p_j - f_(j-1)
MathVecSub( _iNumEq, _pdR, _pdUi );

double Error = MathVecNorm( _iNumEq, _pdR );
Error /= _dNormReference;

//check if error is less than tolerance
_iConvFlag = Error <= _sControl.Tol;

return( _iConvFlag );
}
```

B.1.4.1 Load factor functions

```
/* Load Control Method */
void cNewtonRaphson :: Lambda ( void ) {
_dLambda = _iCurrIte == 1 ? _sControl.CtrlFactor : 0.0;
}

/* Displacement Control Method (including variable displacement) */
void cDisplacementControl :: Lambda ( void ) {
int i;

if( _iCurrIte == 1 ) {
  if( _iCurrStep != 1 && _sControl.CtrlType == VARIABLE ) {
    for( i = 0; i < _iNumEq; i++ ) {
      _pdL[i]   += fabs( _pdX[i] - _pdXPrv[i] );
      _pdXPrv[i] = _pdX[i];
    }

    int EqMax = 0;

    //implementation of Fujii et al 1992 for best control parameter
    //find degree of freedom with the largest displacement
    for( i = 1; i < _iNumEq; i++ ) {
      if( fabs( _pdDx1[i] ) > fabs( _pdDx1[EqMax] ) )
        EqMax = i;
    }

    //if the control DOF is not the one with
    //the max displacement, switch the control dof
    //keep the sign of the max dof
    if( EqMax != _iCtrlEq ) {
      int Dir = (_pdDx1[EqMax] * _pdDx1[_iCtrlEq] > 0 ? 1 : -1 );
      _dCtrlFactor *= Dir * _pdL[EqMax] / _pdL[_iCtrlEq];
      _iCtrlEq = EqMax;
    }
  }
  _dLambda = _dCtrlFactor / _pdDx1[_iCtrlEq];
} else {
  _dLambda = - _pdDx2[_iCtrlEq] / _pdDx1[_iCtrlEq];
}
}

/* Arc-length method */
void cArcLengthControl :: Lambda ( void ) {

if( _iCurrIte == 1 ) {
  MathVecAssign( _iNumEq, _pdDx1_i_11, _pdDx1_i1 );
  _dLambda = sqrt( pow( _sControl.CtrlFactor, 2.0 ) /
    (1.0 + MathVecDot( _iNumEq, _pdDx1, _pdDx1 )) );
```

```
48    MathVecAssign( _iNumEq, _pdDx1_i1, _pdDx1 );
49
50    if( _iCurrStep != 1 )
51      if( MathVecDot( _iNumEq, _pdDx1_i_11, _pdDx1_i1 ) < 0 )
52        _iDir *= -1;
53
54    _dLambda *= _iDir;
55
56    if( _eCtrlType == VARIABLE ) {
57      _dSumLambda = _dLambda;
58      MathVecAssign1( _iNumEq, _pdSumDx, _dLambda, _pdDx1 );
59    }
60  } else {
61    if( _eCtrlType == CONSTANT ) {
62      _dLambda = - MathVecDot( _iNumEq, _pdDx1_i1, _pdDx2 ) /
63        (MathVecDot( _iNumEq, _pdDx1_i1, _pdDx1 ) + 1);
64    } else {
65      _dLambda = - MathVecDot( _iNumEq, _pdSumDx, _pdDx2 ) /
66        (MathVecDot( _iNumEq, _pdSumDx, _pdDx1 ) + _dSumLambda );
67      _dSumLambda += _dLambda;
68      MathVecAdd2( _iNumEq, _pdSumDx, _dLambda, _pdDx1, _pdDx2 );
69    }
70  }
71  }
72
73  /* Work Control Method */
74  void cWorkControl :: Lambda ( void ) {
75  double Csp;
76  double sign;
77
78  if( _iCurrIte == 1 ) {
79    Csp = _sControl.CtrlFactor / MathVecDot( _iNumEq, _pdReference, _pdDx1 );
80    _dLambda = sqrt( fabs( Csp ) );
81
82    if( Csp < 0.0 )
83      _dLambda *= -1;
84  } else {
85    _dLambda = - MathVecDot( _iNumEq, _pdReference, _pdDx2 )
86      /MathVecDot( _iNumEq, _pdReference, _pdDx1 );
87  }
88  }
89
90  /* Generalized Displacement Control Method */
91  void cGenDisplacementControl :: Lambda ( void ) {
92
93  if( _iCurrIte == 1 ) {
94    if( _iCurrStep == 1 ) {
95      MathVecZero( _iNumEq, _pdSumDx );
```

```
96
97      MathVecAssign( _iNumEq, _pdDx1_i1,   _pdDx1 );
98      MathVecAssign( _iNumEq, _pdDx1_i_11, _pdDx1 );
99
100     _dN2_Dx1_11 = MathVecDot( _iNumEq, _pdDx1, _pdDx1 );
101
102     _dLambda = _sControl.CtrlFactor;
103   } else {
104     MathVecAssign( _iNumEq, _pdDx1_i_11, _pdDx1_i1 );
105     MathVecAssign( _iNumEq, _pdDx1_i1, _pdDx1 );
106
107     double Gsp = _dN2_Dx1_11 /
108       MathVecDot( _iNumEq, _pdDx1_i_11, _pdDx1 );
109
110     _dLambda = sqrt( fabs( Gsp ) ) * _sControl.CtrlFactor;
111
112     if( MathVecDot( _iNumEq, _pdSumDx, _pdDx1 ) < 0 )
113       _dLambda *= -1;
114
115     MathVecZero( _iNumEq, _pdSumDx );
116   }
117 } else {
118   _dLambda = - MathVecDot( _iNumEq, _pdDx1_i_11, _pdDx2 ) /
119     MathVecDot( _iNumEq, _pdDx1_i_11, _pdDx1 );
120 }
121 MathVecAdd2( _iNumEq, _pdSumDx, _dLambda, _pdDx1, _pdDx2 );
122 }
123
124 /* Orthogonal Residual Procedure */
125 void cOrthResidualControl :: Lambda ( void ) {
126 if( _iCurrIte == 1 ) {
127   _dLambda = _sControl.CtrlFactor;
128
129   if( MathVecDot( _iNumEq, _pdSumDx, _pdDx1 ) < 0 )
130     _dLambda *= -1;
131
132   MathVecAssign1( _iNumEq, _pdSumDx, _dLambda, _pdDx1 );
133
134   if( _iCurrStep == 1 )
135     _dMaxDx = _sControl.CtrlIniFactor * MathVecNorm( _iNumEq, _pdSumDx );
136   else {
137     _dCurrDx = MathVecNorm( _iNumEq, _pdSumDx );
138     if( _dCurrDx > _dMaxDx ) {
139       _dLambda *= _dMaxDx / _dCurrDx;
140       MathVecScale( _iNumEq, _dMaxDx / _dCurrDx, _pdSumDx );
141     }
142   }
143 } else {
```

```
144     _dLambda = MathVecDot( _iNumEq, _pdUi, _pdSumDx ) /
145       MathVecDot( _iNumEq, _pdReference, _pdSumDx ) - _dTotFactor;
146
147     MathVecAssign2( _iNumEq, _pdDx, _dLambda, _pdDx1, _pdDx2 );
148     _dCurrDx = MathVecNorm( _iNumEq, _pdDx );
149     if( _dCurrDx > _dMaxDx ) {
150       _dLambda *= _dMaxDx / _dCurrDx;
151       MathVecScale( _iNumEq, _dMaxDx / _dCurrDx, _pdDx2 );
152     }
153     MathVecAdd2( _iNumEq, _pdSumDx, _dLambda, _pdDx1, _pdDx2 );
154   }
155   }
```

B.2 Sample input files

B.2.1 Model input file example: twelve bar truss

```
'space_truss' // Model type
4 // Number of nodes
0    0       0 // Node positions
-1   0.283   1
-1   1.697   1
0    1.697   0
4 // Number of boundary conditions
0   1   1   1 //Node number, d_x, d_y, d_z
1   1   0   0
2   1   1   0
3   1   1   1
2 // Number of loads
1   0   0   -0.75 //Node number, f_x, f_y, f_z
2   0   0   -0.25
4 // Number of elements
0   1   1     0 //node1, node2, EA, initial stress
1   2   0.5   0
0   2   1     0
2   3   0.5   0
3 // Number of displacement curves
1   1 //node, d.o.f
1   2
2   2
```

B.2.2 Algorithm input file examples

Linear solver		Algorithm		Update type	
0:	Crout solver	0:	Load control	0:	Standard
		1:	Displacement control	1:	Modified
		2:	Arc-length control		
		3:	Work control		
		4:	Generalized displacement control		
		5	Orthogonal residual		

B.2.2.1 Load control method

```
0                    //linear solver
0      0     0.001  //algorithm, update type, initial control factor
100    40    0.0001 //max steps, max iterations, convergence tolerance
```

B.2.2.2 Displacement control method

```
0                          //linear solver
1    0    0.01    0     1  //algorithm, update type, initial control factor,
                           //control d.o.f., constant (0) or variable (1)
465 40    0.0001           //max steps, max iterations, convergence tolerance
```

B.2.2.3 Arc-length control method

```
0                     //linear solver
2    0    0.05  0  //algorithm, update type, initial control factor,
                     //constant (0) or variable (1)
163  20   0.0001     //max steps, max iterations, convergence tolerance
```

B.2.2.4 Work control method

```
0                     //linear solver
3      0     0.0002  //algorithm, update type, initial control factor
100    40    0.0001  //max steps, max iterations, convergence tolerance
```

B.2.2.5 Generalized displacement control method

```
0                   //linear solver
4    0      0.025   //algorithm, update type, initial control factor
112  20    0.0001   //max steps, max iterations, convergence tolerance
```

B.2.2.6 Orthogonal residual procedure

```
0                      //linear solver
5   0      0.0025  2   //algorithm, update type, initial control factor,
                       //initial incremental scale factor
650 40    0.0001       //max steps, max iterations, convergence tolerance
```

Bibliography

[1] S. N. Al-Rasby. Solution techniques in nonlinear structural analysis. *Computers & Structures*, 40(4):985–993, 1991.

[2] J. L. Asferg, P. N. Poulsen, and L. O. Nielsen. A consistent partly cracked XFEM element for cohesive crack growth. *International Journal for Numerical Methods in Engineering*, 72(4):464–485, October 2007.

[3] J. L. Asferg, P. N. Poulsen, and L. O. Nielsen. A direct XFEM formulation for modeling of cohesive crack growth in concrete. *Computers and Concrete*, 4(2):83–100, 2007.

[4] D K R Babajee, M Z Dauhoo, M T Darvishi, A Karami, and A Barati. Journal of Computational and Applied Analysis of two Chebyshev-like third order methods free from second derivatives for solving systems of nonlinear equations. *Journal of Computational and Applied Mathematics*, 233(8):2002–2012, 2010.

[5] M. Bahrami-Nikkhah and R. Oftadeh. An effective iterative method for computing real and complex roots of systems of nonlinear equations. *Applied Mathematics and Computation*, 215(5):1813–1820, 2009.

[6] Satish Balay, Kris Buschelman, Victor Eijkhout, William D. Gropp, Dinesh Kaushik, Matthew G. Knepley, Lois Curfman McInnes, Barry F. Smith, and Hong Zhang. PETSc users manual. Technical Report ANL-95/11 - Revision 3.0.0, Argonne National Laboratory, 2008.

[7] Satish Balay, Kris Buschelman, William D. Gropp, Dinesh Kaushik, Matthew G. Knepley, Lois Curfman McInnes, Barry F. Smith, and Hong Zhang. PETSc Web page, 2009. http://www.mcs.anl.gov/petsc.

[8] Satish Balay, William D. Gropp, Lois Curfman McInnes, and Barry F. Smith. Efficient management of parallelism in object oriented numerical software libraries. In E. Arge, A. M. Bruaset, and H. P. Langtangen, editors, *Modern Software Tools in Scientific Computing*, pages 163–202. Birkhäuser Press, 1997.

[9] K. J. Bathe. *Finite Element Procedures*. Prentice Hall, Upper Saddle River, New Jersey, 1996.

[10] K. J. Bathe and E. N. Dvorkin. On the automatic solution of nonlinear finite element equations. *Computers & Structures*, 17(5-6):871–879, 1983.

[11] J. J. Batoz and G. Dhatt. Incremental Displacement Algorithms for nonlinear problems. *International Journal for Numerical Methods in Engineering*, 14(8):1262–1267, 1979.

[12] Z. P. Bazant and L. Cedolin. *Stability of Structures - Elastic, Inelastic, Fracture and Damage Theories*. Oxford University Press, New York, 1991.

[13] P. Bellini and A. Chuyla. An improved automatic incremental algorithm for the efficient solution of nonlinear finite element equations. *Computers & Structures*, 26(1-2):99–110, 1987.

[14] T. Belytschko, W.K. Liu, and B. Moran. *Nonlinear Finite Elements for Continua and Structures*. John Wiley & Sons, Inc., West Sussex, England, 2000.

[15] P. G. Bergan. Solution algorithms for nonlinear structural problems. *Computers & Structures*, 12(4):497–509, 1980.

[16] P. G. Bergan, G. Horrigmoe, B. Krakeland, and T. H. Soreide. Solution techniques for non-linear finite element problems. *International Journal for Numerical Methods in Engineering*, 12(11):1677–1696, 1978.

[17] W. Bi, Q. Wu, and H. Ren. A new family of eighth-order iterative methods for solving nonlinear equations. *Applied Mathematics and Computation*, 214(1):236–245, 2009.

[18] J. Bonet. *Nonlinear Continuum Mechanics for Finite Element Analysis*. Cambridge University Press, New York, USA, 1997.

[19] E. L. Cardoso and J. S. O. Fonseca. The GDC method as an orthogonal arc-length method. *Communications in Numerical Methods in Engineering*, 23(4):263–272, 2007.

[20] E. Carrera. A study on arc-length-type methods and their operation failures illustrated by a simple model. *Computers & Structures*, 50(2):217–229, 1994.

[21] W. Celes, G. H. Paulino, and R. Espinha. A compact adjacency-based topological data structure for finite element mesh representation. *International Journal for Numerical Methods in Engineering*, 64(11):1529–1556, 2005.

[22] A. Chandra and S. Mukherjee. Applications of the boundary element method to large strain large deformation problems of viscoplasticity. *Journal of Strain Analysis for Engineering Design*, 18(4), 1983.

[23] A. Chandra and S. Mukherjee. Boundary element formulations for large strain-large deformation problems of viscoplasticity. *International Journal of Solids and Structures*, 20(1):41–53, 1984.

[24] H. Chen and G. E. Blandford. Work-increment-control method for non-linear analysis. *International Journal for Numerical Methods in Engineering*, 36(6):909–930, 1993.

[25] M. J. Clarke and G. J. Hancock. A study of incremental-iterative strategies for non-linear analyses. *International Journal for Numerical Methods in Engineering*, 29(7):1365–1391, 1990.

[26] R. Cook, D. Malkus, M. Plesha, and R. Witt. *Concepts and Applications of Finite Element Analysis*. John Wiley & Sons, Inc., Hoboken, New Jersey, fourth edition, 2002.

[27] M. A. Crisfield. A fast incremental/iterative solution procedure that handles snap-through. *Computers and Structures*, 13(1-3):55–62, 1981.

[28] M. A. Crisfield. Arc-length method incluing line searches and accelerations. *International journal for numerical methods in engineering*, 19(9):1269–1289, 1983.

[29] M. A. Crisfield. *Non-linear Finite Element Analysis of Solids and Structures. Volume 1: Essentials*. John Wiley & Sons, Inc., West Sussex, England, 1991.

[30] M. A. Crisfield. *Non-linear Finite Element Analysis of Solids and Structures. Volume 2: Advanced Topics*. Wiley, John & Sons, Inc., West Sussex, England, 1997.

[31] M.T. Darvishi and A. Barati. A third-order Newton-type method to solve systems of nonlinear equations. *Applied Mathematics and Computation*, 187(2):630–635, 2007.

[32] M.T. Darvishi and A. Barati. Super cubic iterative methods to solve systems of nonlinear equations. *Applied Mathematics and Computation*, 188(2):1678–1685, 2007.

[33] P. Donescu and T. Laursen. A generalized object-oriented approach to solving ordinary and partial differential equations using finite elements. *Finite Elements in Analysis and Design*, 22(1):93–107, 1996.

[34] Y. Dubois-Pelerin and T. Zimmermann. Object-oriented finite element programming: III. An efficient implementation in C++. *Computer Methods in Applied Mechanics and Engineering*, 108(1-2):165–183, 1993.

[35] Y. Dubois-Pelerin, T. Zimmermann, and P. Bomme. Object-oriented finite element programming: II. A prototype program in Smalltalk. *Computer Methods in Applied Mechanics and Engineering*, 98(3):361–397, 1992.

[36] R. Espinha, W. Celes, N. Rodriguez, and G. H. Paulino. ParTopS: compact topological framework for parallel fragmentation simulations. *Engineering with Computers*, 25(4):345–365, June 2009.

[37] D. Eyheramendy and T. Zimmermann. Object-oriented finite elements II. A symbolic environment for automatic programming. *Computer Methods in Applied Mechanics and Engineering*, 132(3-4):277–304, 1996.

[38] D. Eyheramendy and T. Zimmermann. Object-oriented finite elements III. Theory and application of automatic programming. *Computer Methods in Applied Mechanics and Engineering*, 154(1-2):41–68, 1998.

[39] D. Eyheramendy and T. Zimmermann. Object-oriented finite elements IV. Symbolic derivations and automatic programming of nonlinear formulations. *Computer Methods in Applied Mechanics and Engineering*, 190(22-23):2729–2751, 2001.

[40] C. A. Felippa. Nonlinear Finite Element Methods (ASEN 6107) [Course Notes]. University of Colorado at Boulder: Department of Aerospace Engineering Sciences.

[41] B Forde, R FoschiI, and S Stiemer. Object-oriented finite element analysis. *Computers & Structures*, 34(3):355–374, 1990.

[42] W. R. Forde and S. F. Stiemer. Improved arc length orthogonality methods for nonlinear finite element analysis. *Computers & Structures*, 27(5):625–630, 1987.

[43] F. Fujii, K. Choong, and S. Gong. Variable displacement control to overcome turning points of nonlinear elastic frames. *Computers & Structures*, 44(1-2):133–136, July 1992.

[44] F. Fujii and S. Okazawa. Pinpointing bifurcation points and branch-switching. *Journal of Engineering Mechanics*, 123(3):179–189, 1997.

[45] A Golbabai and M Javidi. A new family of iterative methods for solving system of nonlinear algebric equations. *Applied Mathematics and Computation*, 190(2):1717–1722, July 2007.

[46] J He. A coupling method of a homotopy technique and a perturbation technique for non-linear problems. *International Journal of Non-Linear Mechanics*, 35(1):37–43, January 2000.

[47] M. T. Heath. *Scientific Computing: An Introductory Survey*. The McGraw-Hill Companies, New York, 2002.

[48] B.C.P. Heng and R.I. Mackie. Using design patterns in object-oriented finite element programming. *Computers & Structures*, 87(15-16):952–961, August 2009.

[49] G. A. Hrinda. Geometrically nonlinear static analysis of 3D trusses using the arc-length method. In *13th International Conference on Computational Methods and Experimental Measurements*, pages 243–252, 2007.

[50] R. King. A family of fourth order methods for nonlinear equations. *SIAM Journal on Numerical Analysis*, 10:873–879, 1973.

[51] Jože Korelc. Direct computation of critical points based on Crout's elimination and diagonal subset test function. *Computers & Structures*, 88(3-4):189–197, February 2010.

[52] R. Kouhia. Stabilized forms of orthogonal residual and constant incremental work control path following methods. *Computer Methods in Applied Mechanics and Engineering*, 197:1389–1396, 2008.

[53] S. Krenk. An orthogonal residual procedure for non-linear finite element equations. *International Journal for Numerical Methods in Engineering*, 38(5):823–840, 1995.

[54] S. Krenk and O. Hededal. A dual orthogonality procedure for non-linear equations. *Computer Methods in Applied Mechanics and Engineering*, 7825(1):95–107, 1995.

[55] E. N. Lages, G. H. Paulino, I. F. M. Menezes, and R. R. Silva. Nonlinear Finite Element Analysis using an Object-Oriented Philosophy - Application to Beam Elements and to the Cosserat Continuum. *Engineering with Computers*, 15(1):73–89, April 1999.

[56] W. F. Lam and C. T. Morley. Arc-length method for passing limit points in structural calculation. *Journal of Structural Engineering*, 118(1):169–185, 1992.

[57] S. L. Lee, F. S. Manuel, and E. C. Rossow. Large deflection and stability of elastic frames. *ASCE Journal of Engineering Mechanics*, 94:521–533, 1968.

[58] J. Y. R. Liew, N. M. Punniyakotty, and N. E. Shanmugam. Advanced analysis and design of spatial structures. *Journal of Constructional Steel Research*, 42(1):21–48, 1997.

[59] T. W. Lin, Y. B. Yang, and H. T. Shiau. A work weighted state vector control method for geometrically nonlinear analysis. *Computers & Structures*, 46(4):689–694, 1993.

[60] S. Lopez. Detection of bifurcation points along a curve traced by a continuation method. *International Journal for Numerical Methods in Engineering*, 53(4):983–1004, 2002.

[61] R. I. Mackie. Object oriented programming of the finite element method. *International Journal for Numerical Methods in Engineering*, 35(2):425–436, August 1992.

[62] V. Mallardo and C. Allesandri. Arc-length procedures with BEM in physically nonlinear problems. *Engineering Analysis with Boundary Elements*, 28(6):547–559, June 2004.

[63] L. S. Miers and J. C. F. Telles. The boundary element-free method for elastoplastic implicit analysis. *International Journal for Numerical Methods in Engineering*, 76(7):1090–1107, 2008.

[64] D. P. Mondkar and G. H. Powell. Evaluation of solution schemes for nonlinear structures. *Computers & Structures*, 9(3):223–236, September 1978.

[65] J. F. Mougaard, P. N. Poulsen, and L. O. Nielsen. An enhanced cohesive crack element for XFEM using a double enriched displacement field. In *6th International Conference on Fracture Mechanics of Concrete and Concrete Structures*, pages 139–146, 2007.

[66] S. Mukherjee and A. Chandra. *Boundary element formulations for large strain-large deformation problems of plasticity and viscoplasticity*, chapter 2, pages 27–58. Elsevier, Barking, Essex, UK, 1984.

[67] M. A. Noor and M. Waseem. Some iterative methods for solving a system of nonlinear equations. *Computers & Mathematics with Applications*, 57(1):101–106, 2009.

[68] R Oftadeh and A Najafi. A novel cubically convergent iterative method for computing complex roots of nonlinear equations. *Applied Mathematics and Computation*, 217(6):2608–2618, 2010.

[69] M. Ozel. A new decomposition method for solving system of nonlinear equations. *Mathematical and Computational Applications*, 15(1):89–95, 2010.

[70] J. Padovan and S. Tovichakchaikul. Self-adaptive predictor-corrector algorithms for static nonlinear structural analysis. *Computers & Structures*, 15(4):12, 1982.

[71] E. Parente and L. E. Vaz. Improvement of semi-analytical design sensitivities of non-linear structures using equilibrium relations. *International Journal for Numerical Methods in Engineering*, 50(9):2127–2142, 2001.

[72] K. C. Park. A family of solution algorithms for nonlinear structural analysis based on relaxation equations. *International Journal for Numerical Methods in Engineering*, 18(9):1337–1347, 1982.

[73] G. H. Paulino and Y. Liu. Implicit consistent and continuum tangent operators in elastoplastic boundary element formulations. *Computer Methods in Applied Mechanics and Engineering*, 190(15-17):2157–2179, 2001.

[74] G. Powell and J. Simons. Improved iteration strategy for nonlinear structures. *International Journal for Numerical Methods in Engineering*, 17(10):1455–1467, 1981.

[75] E. Ramm. *Strategies for tracing nonlinear responses near limit points*, page 68. Spring-Verlag, New York, 1981.

[76] J. N. Reddy. *An Introduction to Nonlinear Finite Element Analysis*. Oxford University Press, New York, 2004.

[77] M. Rezaiee-Pajand, M. Tatar, and B. Moghaddasie. Some geometrical bases for incremental-iterative methods. *International Journal of Engineering, Transactions B: Applications*, 22(3):245–256, 2009.

[78] M. Rezaiee-Pajand and H.R. Vejdani-Noghreiyan. Computation of multiple bifurcation point. *Engineering Computations*, 23(5):552–565, 2006.

[79] E. Riks. The application of Newton's method to the problem of elastic stability. *Transactions of the ASME: Journal of Applied Mechanics*, 39:1060–1066, 1972.

[80] E. Riks. An incremental approach to the solution of snapping and buckling problems. *International Journal of Solids and Structures*, 15(7):529–551, 1979.

[81] E. Riks. Some computational aspects of the stability analysis of nonlinear structures. *Applied Mechanics and Engineering*, 47(3):219–259, 1984.

[82] M. Ritto-Correa and D. Camotim. On the arc-length and other quadratic control methods: Established, less known and new implementation procedures. *Computers & Structures*, 86(11-12):1353–1368, June 2008.

[83] H. Saffari, M.J. Fadaee, and R. Tabatabaei. Nonlinear Analysis of Space Trusses Using Modified Normal Flow Algorithm. *ASCE Journal of Structural Engineering*, 134(6):998–1005, 2008.

[84] A. G. Salinger, N. M. Bou-Rabee, R. P. Pawlowski, E. D. Wilkes, E. A. Burroughs, R. B. Lehoucq, and L. A. Romero. LOCA 1.0 Library of continuation algorithms: Theory and implementation manual, 2002.

[85] H. Schildt. *C++ from the Ground Up*. McGraw-Hill, Berkeley, California, third edition, 2003.

[86] S.-P. Scholz. Elements of an object-oriented FEM++ program in C++. *Computers & Structures*, 43(3):517–529, 1992.

[87] K. H. Schweizerhof and P. Wriggers. Consistent linearization for path following methods in nonlinear FE analysis. *Computer Methods in Applied Mechanics and Engineering*, 59(3):261–279, 1986.

[88] B.-C. Shin, M. T. Darvishi, and C. H. Kim. A comparison of the Newton-Krylov method with high order Newton-like methods to solve nonlinear systems. *Applied Mathematics and Computation*, 217(7):3190–3198, 2010.

[89] J. Simons and P. G. Bergan. Hyperplane displacement control methods in nonlinear analysis. In W. K. Liu, T. Belytschko, and K. C. Park, editors, *Innovative Methods for Nonlinear Problems*, pages 345–364, Swansea, U.K, 1984. Pineridge Press.

[90] I. Stanciulescu. Assistant Professor of Civil Engineering, Rice University, personal communication, March 2, 2009.

[91] H.-T. Thai and S.-E. Kim. Large deflection inelastic analysis of space trusses using generalized displacement control method. *Journal of Constructional Steel Research*, 65(10-11):1987–1994, October 2009.

[92] G. A. Wempner. Discrete Approximation Related to Nonlinear Theories of Solids. *International Journal of Solids and Structures*, 1:1581–1599, 1971.

[93] P. Wriggers and J. C. Simo. A general procedure for the direct computation of turning and bifurcation points. *International Journal for Numerical Methods in Engineering*, 30(1):155–176, July 1990.

[94] Y. Yang and L. J. Leu. Constitutive laws and force recovery procedures in nonlinear analysis of trusses. *Computer Methods in Applied Mechanics and Engineering*, 92(1):121–131, November 1991.

[95] Y.-B. Yang and S.-R. Kuo. *Theory and Analysis of Nonlinear Framed Structures*. Prentice Hall PTR, 1994.

[96] Y.-B. Yang and W. McGuire. A work control method for geometrically nonlinear analysis. In J. Middleton and G. N. Pande, editors, *Proceedings of the 1985 International Conference on Numerical Methods in Engineering: Theory and Applications*, pages 913–921, 1985.

[97] Y.-B. Yang and M.-S. Shieh. Solution Method for Nonlinear Problems with Multiple Critical Points. *AIAA Journal*, 28(12):2110–2116, 1990.

[98] Y.B. Yang, S.P. Lin, and L.J. Leu. Solution strategy and rigid element for nonlinear analysis of elastically structures based on updated Lagrangian formulation. *Engineering Structures*, 29(6):1189–1200, 2007.

[99] Y.B. Yang, T.J. Lin, L.J. Leu, and C.W. Huang. Inelastic postbuckling response of steel trusses under thermal loadings. *Journal of Constructional Steel Research*, 64(12):1394–1407, December 2008.

[100] Yu. V. Zakharov, K. G. Okhotkin, and A. D. Skorobogatov. Bending of Bars under a Follower Load. *Journal of Applied Mechanics and Technical Physics*, 45(5):756–763, 2004.

[101] T. Zimmermann, Y. Dubois-Pelerin, and P. Bomme. Object-oriented finite element programming: I. Governing principles. *Computer Methods in Applied Mechanics and Engineering*, 98(2):291–303, 1992.

[102] T. Zimmermann and D. Eyheramendy. Object-oriented finite elements I. principles of symbolic derivations and automatic programming. *Computer Methods in Applied Mechanics and Engineering*, 132(3-4):259–276, 1996.