

**Investigations Encompassing the Equations of
Motion and Proper and Accurate Treatment of
Algorithmic Variables: Computational Structural
Dynamics and Stiff Systems**

A THESIS

SUBMITTED TO THE FACULTY OF THE GRADUATE SCHOOL
OF THE UNIVERSITY OF MINNESOTA

BY

Andrew J Hoitink

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
MASTER OF SCIENCE

Dr. K. Tamma, Adviser

November 2009

© Andrew J Hoytink November 2009

Abstract

Different from the way we have been looking in the past at developments encompassing linear and nonlinear dynamics applications via time stepping methods, and significantly different from the manner in which past developments have attempted to explain and/or justify the algorithms and their design, this is the first time we provide clear and concise advances to the field. The research and subsequent developments address: 1) How to properly integrate the equations of motion and the accurate treatment of the underlying time levels at which the computations should occur, 2) How to precisely conduct acceleration computations for general nonlinear dynamic situations and the significance of failure to follow the present propositions, and 3) Proof of concept strategies and studies to enable long term nonlinear dynamics simulations for conservative systems such that the approaches foster completion of analysis to the desired time duration of interest. All these are described in the context of illustrations to the class of LMS methods as an example to simply demonstrate the basic ideas. The proposed results help achieve required order of time accuracy of computational algorithms, required order of accuracy of acceleration fields, and foster long term dynamics of stiff nonlinear dynamics applications. Numerous numerical illustrations demonstrate the proposed developments, therein provide significant advances to the field of computational structural dynamics.

Contents

List of Figures	v
List of Tables	xi
Chapter 1 Introduction	1
1.1 Objectives and Contributions of this Research	6
Chapter 2 Integrating the dynamic equation of motion and determination of the time level	8
2.1 Introduction	8
2.2 The Framework of the GSSSS Family of Algorithms Encompassing LMS Methods	14
2.3 The Hidden Point Collocation Within the Framework of LMS Methods and the Equation of Motion Time Level	19
2.4 The Framework for the U0 Family and the Corresponding Time Level	23
2.5 Time Level Consistency for \tilde{a} , \tilde{v} , and \tilde{u}	28
2.6 EOM Time Level: A General Derivation for the U0 and V0 Family of Algorithms	30

2.7 Overview of Existing Methods 34

2.8 Importance and Significance of the EOM Time Level 35

2.9 Numerical Example: SDOF Duffing Equation 38

 2.9.1 ”Classical” Approach 39

 2.9.2 ”Symplectic-Momentum Based” Representations 46

 2.9.3 ”Energy-Momentum Based” Representations 50

2.10 Example: Non-linear Tetrahedral Spring-Mass System 53

 2.10.1 Classical Approach 54

 2.10.2 ”Symplectic-Momentum Based” Representations 54

 2.10.3 ”Energy-Momentum Based” Representations 54

2.11 Conclusion 55

Chapter 3 Precise computation of algorithmic accelerations for non-dissipative and dissipative methods under the class of LMS methods with single step and single solve 66

3.1 Introduction 66

3.2 U0 family acceleration time level 72

3.3 V0 family acceleration time level 73

3.4 Implications of a shifted acceleration time level 73

3.5 Interpretation and description of a consistent convergence plot 75

3.6 Single degree of freedom (SDOF) example 78

3.7	Example: Non-linear Tetrahedral Spring-Mass System	80
3.8	Conclusion	83
Chapter 4 Observations on the long term stability of algorithms for numerically stiff non-linear problems		97
4.1	Introduction	97
4.2	History	98
4.3	Motivation	101
4.4	Numerical example: Rigid-flexible beam system	102
4.5	Results	103
4.6	A preliminary investigation on accuracy	105
4.7	Conclusion	106
Chapter 5 Concluding Remarks		114
Bibliography		117
Appendix A Acceleration Alignment: Further Numerical Examples		121
A.1	Four Spring-mass system	121
A.2	The Simple Pendulum	121
Appendix B Programming Codes and Input Data		138

List of Figures

2.1	A basic linear interpolation of external force at t_{n+W_1}	25
2.2	Overview of the GSSSS algorithm in a-form	27
2.3	Resulting view of a single time step for a shifted acceleration time level	29
2.4	Time level of the equation of motion for the U0 and V0 families of algorithms	35
2.5	U0/V0(1,1,1) Classical method results for Duffing equation	44
2.6	U0(0.6,0.6,0.6) Classical method results for Duffing equation	45
2.7	U0(0.25,1.0,0.25) Classical method results for Duffing equation	45
2.8	U0(0.9,0.9,0.0728) Classical method results for Duffing equation	46
2.9	U0(0.25,0.25,0) Classical method results for Duffing equation	46
2.10	V0 family classical approach results for Duffing equation	57
2.11	U0 family symplectic based method results for Duffing equation	58
2.12	V0 family symplectic based method results for Duffing equation	59
2.13	U0/V0(1,1,1) Energy and momentum conserving method results for Duffing equation	59

2.14	U0(0.6,0.6,0.6) Energy and momentum conserving method results for Duffing equation	60
2.15	U0(0.25,1.0,0.25) Energy and momentum conserving method results for Duffing equation	60
2.16	U0(0.9,0.9,0.0728) Energy and momentum conserving method results for Duffing equation	60
2.17	U0(0.25,0.25,0) Energy and momentum conserving method results for Duffing equation	61
2.18	V0 family Energy and momentum conserving method results for Duffing equation	61
2.19	Dynamic system response	62
2.20	U0/V0(0.25,1,0.25) Classical method results for spring-mass system .	62
2.21	U0(0.25,1,0.25) Symplectic based approach results for spring-mass system	63
2.22	U0/V0(0.25,1,0.25) Energy and momentum conserving approach results for spring-mass system	63
2.23	Conservation and accuracy using the correct time level , U0/V0(1,1,1) or V0(1,1, any $\rho_{3\infty}$)	64
2.24	Conservation and accuracy using the incorrect time level, U0/V0(1,1,1) or V0(1,1, any $\rho_{3\infty}$)	65
3.1	Traditional convergence of algorithms without dissipation	70
3.2	Fig. from Ref [1] showing u and v second order accurate, but a is only cited to be first order accurate using the generalized- α method . . .	71

3.3	Time level of the resulting acceleration over time step t_n to t_{n+1}	74
3.4	Example of the final time steps used in creating a convergence plot when $\phi = 0$	76
3.5	Example of the final time steps used in creating a convergence plot when $\phi \neq 0$	77
3.6	Velocity based scheme - acceleration aligned convergence plot	79
3.7	Symplectic-momentum based Duffing equation U0 family traditional convergence plots	81
3.8	Symplectic-momentum based Duffing equation U0 family acceleration aligned convergence plots	82
3.9	Symplectic-momentum based Duffing equation V0 family traditional convergence plots	83
3.10	Symplectic-momentum based Duffing equation V0 family acceleration aligned convergence plots	84
3.11	Energy-momentum based Duffing equation U0 family traditional con- vergence plots	85
3.12	Energy-momentum based Duffing equation U0 family acceleration aligned convergence plots	86
3.13	Energy-momentum based Duffing equation V0 family traditional con- vergence plots	87
3.14	Energy-momentum based Duffing equation V0 family acceleration aligned convergence plots	88
3.15	Tetrahedral spring-mass	88

3.16	Energy-momentum based tetrahedral spring-mass U0 family traditional convergence plots	89
3.17	Energy-momentum based tetrahedral spring-mass U0 family acceleration aligned convergence plots	90
3.18	Energy-momentum based tetrahedral spring-mass V0 family traditional convergence plots	91
3.19	Energy-momentum based tetrahedral spring-mass V0 family acceleration aligned convergence plots	92
3.20	Symplectic-momentum based tetrahedral spring-mass U0 family traditional convergence plots	93
3.21	Symplectic-momentum based tetrahedral spring-mass U0 family acceleration aligned convergence plots	94
3.22	Symplectic-momentum based tetrahedral spring-mass V0 family traditional convergence plots	95
3.23	Symplectic-momentum based tetrahedral spring-mass V0 family acceleration aligned convergence plots	96
4.1	Sample spectral radius plot from [2]	107
4.2	A rigid-flexible beam system from [3]	107
4.3	Central difference energy composition ($\Delta t = 0.000001$)	108
4.4	U0V0 $\rho_\infty = 0.5$ energy composition ($\Delta t = 0.05$)	108
4.5	U0V0 $\rho_\infty = 0.9$ energy composition ($\Delta t = 0.05$)	108
4.6	Midpoint rule energy composition ($\Delta t = 0.05$)	109

4.7	EMM energy composition ($\Delta t = 0.05$)	109
4.8	Adaptive timestep history	110
4.9	1e6 stiffness central difference 5 percent error accuracy limit ($\Delta t = 0.001$)	110
4.10	1e6 stiffness EMM 5 percent error accuracy limit ($\Delta t = 0.001$)	111
4.11	1e6 stiffness midpoint rule 5 percent error accuracy limit ($\Delta t = 0.001$)	111
4.12	1e6 stiffness U0V0 $\rho_\infty = 0.5$ 5 percent error accuracy limit ($\Delta t = 0.001$)	111
4.13	1e3 stiffness central difference 5 percent error accuracy limit ($\Delta t = 0.002$)	112
4.14	1e3 stiffness EMM 5 percent error accuracy limit ($\Delta t = 0.003$)	112
4.15	1e3 stiffness midpoint rule 5 percent error accuracy limit ($\Delta t = 0.003$)	112
4.16	1e3 stiffness U0V0 $\rho_\infty = 0.5$ 5 percent error accuracy limit ($\Delta t = 0.003$)	113
A.1	Energy-momentum based four spring-mass U0 family traditional convergence plots	122
A.2	Energy-momentum based four spring-mass U0 family acceleration aligned convergence plots	123
A.3	Energy-momentum based four spring-mass V0 family traditional convergence plots	124
A.4	Energy-momentum based four spring-mass V0 family acceleration aligned convergence plots	125
A.5	Symplectic-momentum based four spring-mass U0 family traditional convergence plots	126

A.6	Symplectic-momentum based four spring-mass U0 family acceleration aligned convergence plots	127
A.7	Symplectic-momentum based four spring-mass V0 family traditional convergence plots	128
A.8	Symplectic-momentum based four spring-mass V0 family acceleration aligned convergence plots	129
A.9	Energy-momentum based simple pendulum U0 family traditional convergence plots	130
A.10	Energy-momentum based simple pendulum U0 family acceleration aligned convergence plots	131
A.11	Energy-momentum based simple pendulum V0 family traditional convergence plots	132
A.12	Energy-momentum based simple pendulum V0 family acceleration aligned convergence plots	133
A.13	Symplectic-momentum based simple pendulums U0 family traditional convergence plots	134
A.14	Symplectic-momentum based simple pendulum U0 family acceleration aligned convergence plots	135
A.15	Symplectic-momentum based simple pendulum V0 family traditional convergence plots	136
A.16	Symplectic-momentum based simple pendulum V0 family acceleration aligned convergence plots	137

List of Tables

2.1	Commonly known algorithms	34
2.2	Predictor multi-corrector coefficients for the incremental a-, v- and d- form representations	42
2.3	Selected U0 Algorithms used in examples and their corresponding EOM time level	44
2.4	V0 Algorithms used in examples and their corresponding EOM time level	47

Chapter 1

Introduction

The equations of motion arise in many fields of science and engineering, and accurately integrating the equations of motion for general linear and nonlinear dynamics is of fundamental importance to the field. The areas related to computational structural dynamics, which are of interest here, are of the focus of this study. In particular, the interest is in the class of the so-called inertial dynamic problems or structural vibrations. In this class of problems, mostly the low frequency modes dominate. However, the spatially induced unwanted high frequency modes that arise from the space discretization such as the finite element method, also participate in the solution and cause a host of numerical problems including non physical solutions.

Finite element formulations are traditionally conducted in the Cartesian coordinate description and are governed by Newtons second law. Alternately, other descriptions such as Lagrangian and Hamiltonian also exist, although they are not as popular in the engineering community. Although all three descriptions yield the same equations of motion with no new physics arising from the alternate forms of framework, literature is often filled with claims that some are more suitable and/or serve better for certain classes of applications. We defer this argument to another time and the focus is

strictly on the Newtons second law and the corresponding Cauchy equations of motion where finite element computations are mainly conducted. Once the partial differential equations are discretized in space by the finite element method, it leads to a system of ordinary differential equations in time subjected to given initial conditions. The objective next is to find solutions to the semi-discretized equation system such that it accurately predicts the true dynamic behavior.

The semi-discretized ordinary differential equation system is discretized in time using a host of strategies giving rise to the notion of the so-called time stepping methods. Here, the solutions are marched out in time starting from the initial conditions with a given time step until the total simulation time of interest. These resulting computational algorithms or time integrators are usually classified as explicit methods and implicit methods. The most important aspect of an algorithm is the consistency, which implies that there exist a certain stability condition and order of accuracy for convergence to exist for a given dynamic situation. That is, as the time step is reduced, pertaining to accuracy, the associated error goes to zero and the rate of convergence yields the order of time accuracy of the underlying computational algorithm. With regards to stability, computational algorithms are either conditionally stable, that is there exists a critical time step value above which the solution fails to converge, or there exist unconditionally stable algorithms. Here, there is no imposed condition on the time step selected; however, it is at the expense of accuracy of the resulting solution. Whereas explicit methods are mostly employed to solve wave propagation problems where all the frequencies arising from the finite element mesh participate in the problem, the drawbacks with these methods is that the resulting algorithms are only conditionally stable. However, they have the advantages that one does not need to solve an equation system, are relatively simple to implement, and no nonlinear iterations are required in the time marching process. On the other hand, implicit methods are mostly preferred for the class of inertial or structural dynamics or vi-

brations problems where mostly the problem is low frequency dominated. However, due to the spatially induced unwanted high frequencies that arise from finite element discretization and participate in the solution, there exist a whole slew of numerical issues that need to be resolved.

In this regard, the latter class of applications, namely the class of inertial problems which are of primary interest and focus here, are integrated in time using those termed as non-dissipative methods or dissipative methods using implicit algorithms. The advantages are that implicit algorithms can afford to take a larger time step value for integrating the equation of motion, but the disadvantages are that a system of equations need to be solved at each time step; and for nonlinear dynamic situations, these methods require iterative Newton type strategies within each time step and convergence failure of these nonlinear iterations is often a bottleneck for completing the analysis to the desired simulation time. The notion of stability mostly holds for linear dynamic situations and is often conducted via various approaches with the most common being the spectral analysis approach to determining stability of an algorithm. On the other hand, for nonlinear dynamic situations, it is extremely hard to prove the notion of stability; in particular to guarantee unconditional stability of an algorithm. Heuristically, this is mostly theorized through the ability of an algorithm to have a constant energy maintained at each time step and/or there is a gradual decay of energy in the physical dynamic system as the algorithmic solution evolves through time. The terms energy conserving or energy consistency are often associated with those algorithms that are claimed to be unconditionally stable even for nonlinear dynamics situations. In this regard, literature often describes the implicit algorithms as non-dissipative or dissipative.

The class of non-dissipative methods are those which often seek to preserve the physics of the dynamic problem as much as possible. That is, the conservation of proper-

ties such as energy, linear and angular momentum in the solution process of time stepping. Although they are regarded as unconditionally stable, due to the presence of unwanted high frequencies that participate in the solution and which are a result of the spatial discretization, for stiff nonlinear dynamic problems, these methods often fail in convergence of nonlinear Newton iterations at certain time levels during the simulation time period and thereby cause the analysis to stop. Furthermore, even though such representations are cited to foster long term simulations, for the class of stiff problems, this is often the bottleneck. Alternatively, to circumvent such situations, computational algorithms have been proposed with controllable numerical dissipative features which seek to eliminate or filter out the participation of unwanted high frequencies so as to enable a longer simulation time window. These are the class of the so-called dissipative algorithms. However, these algorithms also do not ensure completion of the analysis for long term, certainly cause decay of conservation properties related to the underlying physics of the problem such as those associated with energy and angular momentum, and require a tuning parameter such as that termed as the value of the spectral radius at infinity, namely, ρ_∞ , that are associated with the so-called spurious root of the characteristic algorithmic eigenvalues. Based on the aforementioned discussions, it is clear that for the general case, there exists a dilemma for long term dynamic simulations. This is the state-of-the-art, and later in this exposition we seek to address some of the issues to foster long term dynamic simulations, in particular, stiff nonlinear dynamic systems.

Turning attention to time stepping approaches in the field of computational structural dynamics, mostly Linear Multi-Step (LMS) methods are employed and are the most popular in research and commercial software. In this class of methods (the other classes of methods are not included here in identifying the wish list as the overall and overarching issues will complicate the matters and we wish to stay focused), the desirable attributes that we cite include:

1. Unconditional stability 2. Single step single solve representation 3. Second order time accuracy 4. Zero order algorithmic overshoot either in the displacement or velocity field is desired, since the overshoot amount affects the quality and physics of the dynamic response in the initial time period after starting the solution process, and subsequently cause other numerical problems such as convergence of nonlinear Newton iterations at certain time levels during the transient duration. It also affects the amount of numerical energy dissipation of the algorithms, that is, the algorithms are more prone to show significant loss of energy in the system even for conservative systems. The higher the overshoot amount, the more significant is the loss or decay. 5. Finally, the ultimate challenge and desirable goal is to have algorithmic designs and strategies that are readily applicable for the general case, namely conservation of energy, linear and angular momentum for conservative stiff dynamic systems, or at least maintain as close as possible to the physics and enable long term simulations to be completed in reasonable time.

The challenges associated with the above wish list are daunting, and it is not trivial. Over five decades of research has been underway and only recently, we are able to see the big picture a little more clearly.

Whereas for linear dynamic systems, most of the challenges appear to be within reach in a few years from now, for nonlinear dynamic situations, in particular, for stiff nonlinear dynamic applications and consideration of general material models, we are still far away toward providing a resolution to the underlying problem at hand. More importantly, the challenge is to provide guarantee of unconditional stability at least in an energy sense, and/or some sort of energy consistency whilst ensuring the satisfaction of stability in the general case. A few noteworthy points are in order at this juncture and the focus in this exposition is on the class of LMS methods for illustration. While it happens to be, and obviously fortunate that the common

non-dissipative methods (Newmark, mid-point rule and the so-called velocity based scheme) can be integrated in time at known time values without any doubt, this is not true in general for the class of dissipative methods. How and at what time levels the equations of motion need to be integrated is not clear from the literature. In fact, it is to the contrary. As a consequence, there needs to be a resolution provided for the general LMS framework for general linear and nonlinear dynamics applications. This, and several other issues are addressed in this research to enable an in depth understanding of how and precisely at what time levels one has to integrate the equations of motion in the dynamic process, while still attempting to meet the desired goals of the wish list identified above.

1.1 Objectives and Contributions of this Research

The present research accomplishes the following specific objectives and the various contributions are highlighted in the text to follow:

1. The proper treatment and precise evaluations for integrating the equations of motion for linear and nonlinear dynamic situations. The focus is on the class of the so-called LMS methods simply for illustration of the basic ideas and since they are the most popular in research and commercial codes to-date. The significance of the present developments is noteworthy since failure to enact proper treatment for the respective non-dissipative and dissipative algorithms will lead to inaccurate analysis and inferences that will be misleading. Furthermore, this is the first time that such a proper and rigorous treatment exists to help analysts to carefully implement computational algorithms for general dynamics applications.
2. Precise determination of acceleration time levels is yet another important contribution. To-date, the literature is filled with misconceptions and lacks an in depth understanding of the underlying problem. As a consequence, inaccurate interpreta-

tions and incorrect analysis results for general structural dynamics applications. The relevant issues are completely resolved once and for all with an in depth understanding of how to perform the respective computations accurately and precisely.

3. The final contribution addresses the notorious issue of long term dynamics and stiff nonlinear dynamics applications with a proof of concept suggested strategies to foster such simulations. This is one of the most challenging of the issues to-date in computational structural dynamics and the present contributions explore new avenues not available currently.

The manuscript is arranged as follows. In Chapter 1 we described an introduction to the field and some challenges and issues and unresolved topics and a wish list of desirable attributes. In Chapter 2 we focus on the equation of motion and proper treatment and evaluations for conducting the time integration procedure with focus on LMS methods. In Chapter 3, we turn attention to precise and accurate treatment of acceleration computations for general linear and nonlinear dynamic applications. In Chapter 4 we take upon the challenges of practical problems with a fairly general case of stiff nonlinear structural dynamics and long term simulations for conservative dynamics systems. Some proof of concept strategies are proposed to foster these simulations for computational structural dynamics. Finally, Chapter 5 describes concluding remarks. Relevant references and appendices are also included which are used advance the present research efforts.

Chapter 2

Integrating the dynamic equation of motion and determination of the time level

2.1 Introduction

With particular attention to the class of LMS methods with a single solve which are implicit, second-order time accurate, and are the predominant methods in most research and commercial software for the analysis of inertial dynamic problems, we restrict the focus in Part I of this exposition to providing the theoretical developments and basis, by capitalizing upon the well known generalized single-step-single-solve (GSSSS) framework of algorithms (see Ref [4,5]) which encompasses this class of LMS methods for applications to general dynamic problems. Amongst a variety of approaches in time integration algorithms for structural dynamics problems, the designs of algorithms resulting from the LMS framework possess a unique and optimal balance between the computational complexity and the convergence properties (see [6] for de-

tailed analysis). Hence, their notoriety for use in practical real world applications in research and commercial software is evident (for example, the Newmark average acceleration method [7]). Although it is a non-dissipative method, other predominant non-dissipative methods exist such as the mid point rule and the so-called velocity based scheme [8]. However, these non-dissipative methods have limitations for the class of inertial dynamic problems, in particular for resolving high frequency modes. As such, although this trio serve as the pillars and backbone for LMS methods, subsequent modifications can be embedded using these as the basis to further designing those termed as controllable numerical dissipative methods that serve the purposes of tackling high frequency participation of modes encountered in practical large scale applications. Nonetheless, although the controllable numerical dissipative algorithms within the LMS framework have the advantage of dissipating the un-resolved high frequency modes due to the spatial discretization(see [9–11], and [4] and references therein, only a select few are mostly being utilized in the research codes, and this number is still very scant in commercial codes primarily due to a lack of confidence and a detailed understanding of the limitations, pros/cons, and bottlenecks that exist to-date. There exists two barriers associated with the application of the controllable numerical dissipative methods for the nonlinear structural dynamic problems: (i) there does not exist a fundamental basis for how to precisely evaluate, for example, the internal force for the general cases of algorithms with controllable numerical dissipation in general nonlinear structural dynamic problems within a time step interval; and (ii) the resulting accelerations from these numerical dissipative algorithms are only reported to be first-order time accurate [1,4]. The fundamental question that is to be posed is whether these are true barriers or simply misconceptions giving rise to a lack of a fundamental understanding of the basics? Indeed, we confirm the latter to be true, and in Part I of this exposition we describe the fundamental theoretical developments wherein, these perceptions can be readily overcome. As a consequence,

we thereby provide confidence to the community at large to readily employ the class of LMS methods for general nonlinear dynamic applications (however, this does not imply that unconditional stability as in linear dynamic situations can be guaranteed for the nonlinear dynamic situations, and research is still underway to foster such developments). A few key points are of noteworthy mention at this juncture. The aforementioned barriers do not exist for the Newmark average acceleration method, since the Newmark method is derivable from the point collocation approach at the end of the time step of t_{n+1} . The implication is that at the time level of t_{n+1} , the variables, namely, acceleration, velocity, and the internal forces indeed truly satisfy the equation of motion strongly. As a consequence, one only needs to evaluate the internal force for the Newmark method at the time level of t_{n+1} , and the acceleration resulting from the Newmark method is indeed second-order time accurate. Following this line of thought, the question now is whether there exists a point collocation for the framework in general and consequently within all numerical non-dissipative and dissipative LMS methods as well? The answer is in the affirmative, and the remedy to the aforementioned questions now readily circumvents the two barriers, which can be removed accordingly. The time level corresponding to the point collocation within the numerical non-dissipative and dissipative LMS methods can provide a physical argument regarding where to precisely evaluate the internal forces within the time interval. And, it has been only recently demonstrated in [4, 12] that the acceleration evaluations from this critical collocation point time level yield second order time accuracy for general non-dissipative and numerical dissipative LMS methods.

The present chapter attempts to shed light on the way that all designs of algorithms contained within the realm of second order accurate linear multi-step methods (LMS), satisfy the semi-discrete equation of motion (EOM) resulting from discretization due to spatial finite elements or the like. For the first time we are able to identify the specific time level point at which a particular algorithm within the LMS framework

satisfies the EOM. As the title of this exposition suggests, this can be thought of as a point collocation of the primary variables at some discrete time point over the range of a single time step.

The importance of understanding such a time level has numerous serious consequences when the linear dynamic algorithms are applied as extensions to solve nonlinear dynamic problems. Firstly, without consistent evaluation of all the variables at a single time point there will be a loss of the designed order of accuracy, and consequently the convergence rate. Secondly, our discovery of this time level has given us a much more rigorous and thorough understanding of the acceleration term; a fundamental understanding which led to the discovery of how to correctly interpret acceleration results. With this proper interpretation, we are now able to show for the first time that all non-dissipative, and even algorithms with controllable numerical dissipation are indeed second order time accurate in acceleration [12]. This is in complete contrast to all past efforts to-date which consistently describe the results for algorithms with controllable numerical dissipation to be only first order time accurate regardless of the fact that the algorithms are originally designed with second order time accuracy as a condition. Lastly, given this specific time level we are now able for the first time to describe how to correctly extend the set of parent linear dynamic algorithms to the nonlinear dynamics realm with great confidence, such that all terms in the equation of motion for dynamics applications are being evaluated consistently and correctly. This is the bottom line.

Some background and some of our recent developments that have been accomplished are highlighted next to provide context to the present developments. Recently, Zhou and Tamma [4–6, 13, 14] described a unified theory underlying computational algorithms for designing computational algorithms for time dependent phenomena. In particular, focusing attention on the class of LMS methods involving a single solve

within a time step, implicit, and second-order time accurate for general structural dynamics applications which are the predominant methods in most commercial software, Zhou and Tamma described a general framework under the umbrella of generalized single step single solve (GSSSS) family of algorithms. This novel framework encompasses all of the LMS methods that have been developed over the past fifty years, and also includes new methods that have not been available to-date, which are optimal (this is in the sense of least amount of numerical dissipation, dispersion, and overshoot behavior) for various applications in structural dynamics. The GSSSS family of algorithms can be viewed as the fully discretized time integration framework for the equation of motion which inherently contains all past approaches, and also new computational developments. The beauty of this framework is that a single precise time integration module is all that is needed to be implemented in the next generation of structural dynamics codes which provides all possible algorithms as available choices and options for the analyst to use. As a research tool this unified framework allows us to study a very generic algorithmic structure, making it possible to study all of the algorithms contained within the GSSSS family from a neutral and unbiased viewpoint. It was this broad view, in depth insight, and an open mind that has led to a deeper understanding of how and under what conditions time integration algorithms satisfy the spatially discrete equations of motion.

To convey the significance of this work first we describe how the generic GSSSS framework can be examined to show the time level at which any given algorithm satisfies the equation of motion. Specifically, the term 'time level' is used to describe the discrete point in time at which the continuous equation of motion, often generalized as $M\ddot{a}(t) + C\dot{v}(t) + K\tilde{u}(t) = \tilde{F}(t)$ is satisfied exactly. A somewhat physical interpretation of this time level can be thought of as follows: the initial values of u , v , and a are known at the time point n , and we desire the value of these variables at a new time point $n+1$. To move forward, the EOM is satisfied at some intermediate time point

between n and $n+1$, and these values are then used to compute the desired quantities at the time point $n+1$. For a select few well known algorithms, this time level happens to be correctly interpreted, but this is not true in general. For example, the precise time level for the well known Newmark algorithm is at the time point $n+1$. That is, the discrete time point at which the algorithm provides a solution to the EOM is at the time point $n+1$. Similarly, the commonly used midpoint rule satisfies the EOM specifically at the time point $n+1/2$, as the name suggests. For these two common algorithms it may be relatively easy to understand, but when we consider the other non-dissipative velocity based scheme (see Ref [8]), and the general class of algorithms which contain controllable numerical dissipation this time level is much less easy to pinpoint. This is the issue and bottleneck. As such, in the past there has indeed been some degree of misunderstanding and also a lack of clarity regarding how to accurately extend such algorithms (in particular, the HHT- α method [15], WBZ method [10], the three-parameters optimal scheme [11] cited in [16] and which was recently made aware to the present authors or equivalently that referred to as the Generalized- α method [17], and others as in the GSSSS framework [4, 5] for all possible designs of algorithms within the class of LMS methods) to nonlinear dynamics applications. The main source of a clear lack of clarity stems primarily from the precise evaluation of the internal force term. With a sound theoretical basis, and an in depth understanding of the underlying time level for any given algorithm within this general GSSSS framework, with full confidence we are now able to also describe how to extend any parent linear dynamic algorithm to nonlinear dynamic problems while systematically maintaining the desired order of time accuracy in all three primary variables; namely, displacement, velocity and acceleration.

The outline of this exposition is as follows. After a detailed description of the underlying concepts, we show numerical examples which verify the concepts regarding the significance of the algorithmic time level, as well as demonstrate its importance

in accurately extending the parent linear dynamic algorithms to nonlinear dynamics problems. We illustrate via the well known single degree of freedom nonlinear Duffing oscillator, and then extend the applicability to a multi-DOF geometrically nonlinear dynamic system of springs and masses. These relatively simplistic systems are used solely to demonstrate the concepts; the basic ideas however, may be applied without modification to problems of any level of complexity. The results of primary concern and of interest here are preservation of the order of time accuracy of the primary variables, as well as concepts underlying the conditions for conservation of fundamental physical quantities such as energy and momentum.

2.2 The Framework of the GSSSS Family of Algorithms Encompassing LMS Methods

A brief background of the important and subtle issues and underlying mechanics of the GSSSS unified framework which encompasses LMS methods is highlighted next. Recently, a novel concept that provides new avenues to design time integration operators for linear dynamic problems via a unified framework, which unifies all of the existing algorithms known to the authors and provides new designs of time operators encompassing the class of LMS methods involving a single solve within a given time step, and second order time accuracy was introduced in a series of papers by Zhou et al. [4–6,13,14]; a notion called algorithms by design. By choosing certain key algorithmic parameters called the algorithmic DNA markers, the unified framework under the umbrella of the family of generalized single solve single step (GSSSS) time operators was uniquely designed. By simply and properly selecting only the two principal roots and the spurious root, the underlying design enabled one to not only recover all of the existing time operators in the literature in the sense of LMS methods including non-dissipative algorithms such as the Midpoint rule, velocity based scheme [8] and

the Newmark average acceleration method [7], but also additionally provided new avenues towards new algorithmic designs with optimal features. This, is in the sense of the least amount of algorithmic dissipation, dispersion, and overshoot behavior. In general, for the single field form of representation involving a single solve within each time step, in the sense of LMS methods, there exist three roots participating in the algorithm, namely, the two principal roots $\rho_{1\infty}$, $\rho_{2\infty}$ and the spurious root $\rho_{3\infty}$. It has been shown in [4,5] that the second-order time accurate, unconditionally stable LMS framework is basically comprised of two distinct algorithmic structures, termed as constrained U (displacement overshooting aspect) and constrained V (velocity overshooting aspect) family of algorithms for linear structural dynamic systems. Most of the developments to-date for both numerically non-dissipative and dissipative algorithms, pertain to the constrained U-family of algorithms within the LMS class of methods for finite element computations. Also as reported in the literature [5], only those with either zero order displacement overshooting behavior in U (U0 family) or zero order velocity in V (V0 family) are competitive amongst the class of LMS methods, and hence only these are considered. This is because for a class of nonlinear dynamics applications, the others pose problematic issues as related to convergence of nonlinear iterations in the step by step time marching process, and are hence undesirable. Most of the traditionally developed time integration operators appearing in the literature to-date for linear dynamic problems, belong only to the U0 family such as the non-dissipative Newmark method [7] and mid-point rule, while the controllable dissipative algorithms such as the *WBZ* [10] and *HHT* - α [15], and the like exist (see various references contained in [4,5] (but are not of focus here), and all others with the exception of the so-called U0/V0 have the drawbacks of overshooting behavior present in them. More recent developments address avenues to circumvent these drawbacks with optimal features [4,5].

Some details of the U0 family and the V0 family of algorithms follow next. The

U0 family of algorithms contains all the algorithms with zero-order displacement overshooting behavior, and the V0 family of algorithms contains all the algorithms with zero-order velocity overshooting behavior. These two families of algorithms have been derived and described in [5]. However, for the purpose of easy reference in this chapter, the U0 family and the V0 family are highlighted next.

Consider the semi-discretized system of equations of linear structural dynamic problems by space discretization of the single field form as:

$$\begin{aligned} \mathbf{M}\ddot{\mathbf{u}}(t) + \mathbf{C}\dot{\mathbf{u}}(t) + \mathbf{K}\mathbf{u}(t) &= \mathbf{f}(t) \\ \mathbf{u}(0) &= \mathbf{u}_0 \quad , \quad \dot{\mathbf{u}}(0) = \dot{\mathbf{u}}_0 \end{aligned} \tag{2.1}$$

where \mathbf{M} is the mass matrix, \mathbf{C} is the damping matrix, and \mathbf{K} is the stiffness matrix. The U0 family of algorithms is given as follows.

Algorithm 1 (U0 Family of Algorithms)

Given \mathbf{u}_n , $\dot{\mathbf{u}}_n$, and $\ddot{\mathbf{u}}_n$, find \mathbf{u}_{n+1} , $\dot{\mathbf{u}}_{n+1}$, and $\ddot{\mathbf{u}}_{n+1}$ from

$$\begin{aligned} &(\Lambda_6 W_1 \mathbf{M} + \Lambda_5 W_2 \mathbf{C} \Delta t + \Lambda_3 W_3 \mathbf{K} \Delta t^2) \Delta \mathbf{a} \\ = &-\mathbf{M} \ddot{\mathbf{u}}_n - \mathbf{C} (\dot{\mathbf{u}}_n + \Lambda_4 W_1 \ddot{\mathbf{u}}_n \Delta t) \\ &-\mathbf{K} (\mathbf{u}_n + \Lambda_1 W_1 \dot{\mathbf{u}}_n \Delta t + \Lambda_2 W_2 \ddot{\mathbf{u}}_n \Delta t^2) \\ &+(1 - W_1) \mathbf{f}_n + W_1 \mathbf{f}_{n+1} \end{aligned}$$

$$\begin{aligned} \mathbf{u}_{n+1} &= \mathbf{u}_n + \lambda_1 \dot{\mathbf{u}}_n \Delta t + \lambda_2 \ddot{\mathbf{u}}_n \Delta t^2 + \lambda_3 \Delta \mathbf{a} \Delta t^2 \\ \dot{\mathbf{u}}_{n+1} &= \dot{\mathbf{u}}_n + \lambda_4 \ddot{\mathbf{u}}_n \Delta t + \lambda_5 \Delta \mathbf{a} \Delta t \\ \ddot{\mathbf{u}}_{n+1} &= \ddot{\mathbf{u}}_n + \Delta \mathbf{a} \end{aligned}$$

where

$$\begin{aligned}
W_1\Lambda_1 &= \frac{1}{1 + \rho_{3\infty}} , & \lambda_1 &= 1 \\
W_2\Lambda_2 &= \frac{1}{2(1 + \rho_{3\infty})} , & \lambda_2 &= \frac{1}{2} \\
W_3\Lambda_3 &= \frac{1}{(1 + \rho_{1\infty})(1 + \rho_{2\infty})(1 + \rho_{3\infty})} , & \lambda_3 &= \frac{1}{(1 + \rho_{1\infty})(1 + \rho_{2\infty})} \\
W_1\Lambda_4 &= \frac{1}{1 + \rho_{3\infty}} , & \lambda_4 &= 1 \\
W_2\Lambda_5 &= \frac{3 + \rho_{1\infty} + \rho_{2\infty} - \rho_{1\infty}\rho_{2\infty}}{2(1 + \rho_{1\infty})(1 + \rho_{2\infty})(1 + \rho_{3\infty})} , & \lambda_5 &= \frac{3 + \rho_{1\infty} + \rho_{2\infty} - \rho_{1\infty}\rho_{2\infty}}{2(1 + \rho_{1\infty})(1 + \rho_{2\infty})} \\
W_1\Lambda_6 &= \frac{2 + \rho_{1\infty} + \rho_{2\infty} + \rho_{3\infty} - \rho_{1\infty}\rho_{2\infty}\rho_{3\infty}}{(1 + \rho_{1\infty})(1 + \rho_{2\infty})(1 + \rho_{3\infty})}
\end{aligned}$$

The weighted time field is suggested to be,

$$W = 1 - \frac{15(1 - 2\rho_{3\infty})}{1 - 4\rho_{3\infty}} \frac{\tau}{\Delta t} + \frac{15(3 - 4\rho_{3\infty})}{1 - 4\rho_{3\infty}} \left(\frac{\tau}{\Delta t}\right)^2 - \frac{35(1 - \rho_{3\infty})}{1 - 4\rho_{3\infty}} \left(\frac{\tau}{\Delta t}\right)^3 ; \quad \tau \in [0, \Delta t]$$

and

$$W_i = \frac{\sum_{j=0}^3 \frac{w_j}{1+i+j}}{\sum_{j=0}^3 \frac{w_j}{1+j}} ; \quad i = 1, 2, 3$$

$\rho_{1\infty}$, $\rho_{2\infty}$, and $\rho_{3\infty}$ are the first principal root, the second principal root, and the spurious root at the high-frequency limit, respectively, and they satisfy the following relation:

$$0 \leq \rho_{3\infty} \leq \rho_{1\infty} \leq \rho_{2\infty} \leq 1$$

Additionally, the V0 family of algorithms is given as follows.

Algorithm 2 (V0 Family of Algorithms)

Given \mathbf{u}_n , $\dot{\mathbf{u}}_n$, and $\ddot{\mathbf{u}}_n$, find \mathbf{u}_{n+1} , $\dot{\mathbf{u}}_{n+1}$, and $\ddot{\mathbf{u}}_{n+1}$ from

$$\begin{aligned}
& (\Lambda_6 W_1 \mathbf{M} + \Lambda_5 W_2 \mathbf{C} \Delta t + \Lambda_3 W_3 \mathbf{K} \Delta t^2) \Delta \mathbf{a} \\
= & -\mathbf{M} \ddot{\mathbf{u}}_n - \mathbf{C} (\dot{\mathbf{u}}_n + \Lambda_4 W_1 \ddot{\mathbf{u}}_n \Delta t) \\
& -\mathbf{K} (\mathbf{u}_n + \Lambda_1 W_1 \dot{\mathbf{u}}_n \Delta t + \Lambda_2 W_2 \ddot{\mathbf{u}}_n \Delta t^2) \\
& +(1 - W_1) \mathbf{f}_n + W_1 \mathbf{f}_{n+1}
\end{aligned}$$

$$\begin{aligned}
\mathbf{u}_{n+1} &= \mathbf{u}_n + \lambda_1 \dot{\mathbf{u}}_n \Delta t + \lambda_2 \ddot{\mathbf{u}}_n \Delta t^2 + \lambda_3 \Delta \mathbf{a} \Delta t^2 \\
\dot{\mathbf{u}}_{n+1} &= \dot{\mathbf{u}}_n + \lambda_4 \ddot{\mathbf{u}}_n \Delta t + \lambda_5 \Delta \mathbf{a} \Delta t \\
\ddot{\mathbf{u}}_{n+1} &= \ddot{\mathbf{u}}_n + \Delta \mathbf{a}
\end{aligned}$$

where

$$\begin{aligned}
W_1 \Lambda_1 &= \frac{3 + \rho_{1\infty} + \rho_{2\infty} - \rho_{1\infty} \rho_{2\infty}}{2(1 + \rho_{1\infty})(1 + \rho_{2\infty})}, \quad \lambda_1 = 1 \\
W_2 \Lambda_2 &= \frac{1}{(1 + \rho_{1\infty})(1 + \rho_{2\infty})}, \quad \lambda_2 = \frac{1}{2} \\
W_3 \Lambda_3 &= \frac{1}{(1 + \rho_{1\infty})(1 + \rho_{2\infty})(1 + \rho_{3\infty})}, \quad \lambda_3 = \frac{1}{2(1 + \rho_{3\infty})} \\
W_1 \Lambda_4 &= \frac{3 + \rho_{1\infty} + \rho_{2\infty} - \rho_{1\infty} \rho_{2\infty}}{2(1 + \rho_{1\infty})(1 + \rho_{2\infty})}, \quad \lambda_4 = 1 \\
W_2 \Lambda_5 &= \frac{2}{(1 + \rho_{1\infty})(1 + \rho_{2\infty})(1 + \rho_{3\infty})}, \quad \lambda_5 = \frac{1}{1 + \rho_{3\infty}} \\
W_1 \Lambda_6 &= \frac{2 + \rho_{1\infty} + \rho_{2\infty} + \rho_{3\infty} - \rho_{1\infty} \rho_{2\infty} \rho_{3\infty}}{(1 + \rho_{1\infty})(1 + \rho_{2\infty})(1 + \rho_{3\infty})}
\end{aligned}$$

The weighted time field is suggested to be,

$$\begin{aligned}
W = & 1 - \frac{30(3 - 4\rho_{1\infty} - 4\rho_{2\infty} + 6\rho_{1\infty}\rho_{2\infty})}{9 - 11\rho_{1\infty} - 11\rho_{2\infty} + 19\rho_{1\infty}\rho_{2\infty}} \frac{\tau}{\Delta t} \\
& + \frac{15(25 - 37\rho_{1\infty} - 37\rho_{2\infty} + 53\rho_{1\infty}\rho_{2\infty})}{2(9 - 11\rho_{1\infty} - 11\rho_{2\infty} + 19\rho_{1\infty}\rho_{2\infty})} \left(\frac{\tau}{\Delta t}\right)^2 \\
& - \frac{35(3 - 5\rho_{1\infty} - 5\rho_{2\infty} + 7\rho_{1\infty}\rho_{2\infty})}{9 - 11\rho_{1\infty} - 11\rho_{2\infty} + 19\rho_{1\infty}\rho_{2\infty}} \left(\frac{\tau}{\Delta t}\right)^3 ; \\
\tau \in & [0, \Delta t]
\end{aligned}$$

and

$$W_i = \frac{\sum_{j=0}^3 \frac{w_j}{1+i+j}}{\sum_{j=0}^3 \frac{w_j}{1+j}} ; \quad i = 1, 2, 3$$

$\rho_{1\infty}$, $\rho_{2\infty}$, and $\rho_{3\infty}$ are the first principal root, the second principal root, and the spurious root at the high-frequency limit, respectively, and they satisfy the following relation:

$$0 \leq \rho_{3\infty} \leq \rho_{1\infty} \leq \rho_{2\infty} \leq 1$$

2.3 The Hidden Point Collocation Within the Framework of LMS Methods and the Equation of Motion Time Level

The time level at which the equation of motion needs to be computed is a very fundamental and important concept to understand, especially in the case of providing extensions of linear dynamic algorithms to nonlinear dynamic problems. Improper calculation of time dependent variables which are included in the equation of motion can lead to decreased accuracy of the algorithm, and also a failure to conserve key quantities in algorithm designs which seek to establish energy and momentum conservation. To understand its importance and significance, we must first define

the meaning and implication regarding the time level aspect for the equation of motion. Fundamentally, computational structural dynamics problems first comprise of the governing strong form of the partial differential equations of motion. The semi-discrete weak form of the equation of motion is obtained through spatial discretization. This yields a system of ordinary differential equations which can be solved computationally. The ordinary differential equations arising from the semi-discretized problem are then discretized in time leading to various forms of representation for the underlying algorithm. In the linear dynamic cases, this fully discretized equation of motion has the form:

$$M\tilde{a}(t) + C\tilde{v}(t) + K\tilde{u}(t) = \tilde{F}(t) \quad (2.2)$$

This equation must be valid at any time t . The goal of the framework of the GSSSS family of algorithms is to take any spatially discretized system of equations, and therein provide a means to march forward in time. All that is required are the set of given initial conditions $u(t_0)$, $v(t_0)$, and it is assumed the matrices M , C , K , and vector F are known from the spatially discretized mesh. Given these initial values, every algorithm within the entire framework of the GSSSS family of algorithms, then solves the equation of motion at some key time point, namely, \tilde{t} , where $t_n \leq \tilde{t} \leq t_{n+1.5}$, and subsequently uses an asymptotic series type expansion to approximate the values of u and v at time t_{n+1} . Specifically, it is this key time point, namely, \tilde{t} that is here referred to as the equation of motion time level; it is defined as the discrete time point precisely at which, the equation of motion must be satisfied.

Although knowledge of this time level is important for linear dynamic algorithmic designs, also for the case of nonlinear dynamics it is particularly important to understand this key time level point in question, so as to ensure an accurate evaluation and implementation of the dynamic application at hand in the process of time inte-

gration of the equations of motion. As an illustrative example, if a problem involves large displacements, a non-linear definition of strain must be employed in order to accurately describe the deformation, and Eq. 2.2 then typically takes the form:

$$M\tilde{a}(t) + C\tilde{v}(t) + \tilde{p}(u(t), \epsilon(t)) = \tilde{F}(t) \quad (2.3)$$

Masuri et. al [18, 19] have recently shown that if the strain and displacement are approximated independent of each another, we are able to design algorithms within the GSSSS framework that conserve energy and momentum exactly under given conditions of selection of the principle roots and the spurious root. The illustration to the Saint Venant Kirchoff model for demonstrating the basic ideas via a newly developed normalized time weighted residual approach was described. In the context of the equation of motion and the significance of the time level for precisely integrating the equation of motion, since the internal force p is now a function of $u(t)$ and $\epsilon(t)$, the time level \tilde{t} at which Eq. 2.3 is being evaluated is no longer trivial. Though it may seem obvious that all four terms (\tilde{a} , \tilde{v} , \tilde{u} , and \tilde{F}) in Eq. 2.3 should be evaluated at the same time \tilde{t} , to-date, a fundamental and general explanation and description for what that time point is exactly, has not been described in general in the literature, and also has not been understood clearly. In fact, very little is understood to-date for the general cases of algorithms typical of the framework of GSSSS family encompassing LMS methods. For a very limited and some commonly used algorithms, the issue has not arisen or stood out for obvious reasons as explained previously. For example, the Newmark algorithm satisfies the equation of motion at the collocation time t_{n+1} . In this case it is easy to see that the internal force p needs to be also calculated at precisely the time t_{n+1} as well. Also, the midpoint rule satisfies the equation of motion at time $t_{n+\frac{1}{2}}$. Again, it is not a very far stretch to understand at what time p should be evaluated for an algorithm referred to as the *midpoint* rule. However for others

such as the non-dissipative velocity based scheme ([8]), and all others dealing with a generalized family of algorithms such as the GSSSS framework [4,5] that encompass the class of LMS methods which include controllable numerical dissipation, that it is not the case is immediately obvious, and how to handle the internal force term poses added complexities in general. Especially for algorithms with controllable numerical dissipation, it is particularly difficult to pinpoint this exact time level that is required for integrating the equation of motion. The goal of this chapter is to present the theoretical details, derivations and proofs and demonstrate that we are indeed able to precisely define this time level for any algorithm contained within the framework of the GSSSS family of algorithms.

To understand fully the time level for the equation of motion, we first look at the general form of the second order accurate GSSSS family of linear dynamic algorithms [4].

$$\begin{aligned}
M[\ddot{u}_n + \Lambda_6 W_1(\ddot{u}_{n+1} - \ddot{u}_n)] + C(\dot{u}_n + \Lambda_4 W_1 \dot{u}_n \Delta t + \Lambda_5 W_2(\dot{u}_{n+1} - \dot{u}_n) \Delta t) \\
+ K[u_n + \Lambda_1 W_1 u_n \Delta t + \Lambda_2 W_2 \ddot{u}_n \Delta t^2 + \Lambda_3 W_3(\ddot{u}_{n+1} - \ddot{u}_n) \Delta t^2] \\
= (1 - W_1) f_n + W_1 f_{n+1} \quad (2.4)
\end{aligned}$$

The associated updates for advancing to the next time level for the above are:

$$u_{n+1} = u_n + \lambda_1 \dot{u} \Delta t + \lambda_2 \ddot{u} \Delta t^2 + \lambda_3 \Delta a \Delta t^2 \quad (2.5)$$

$$\dot{u}_{n+1} = \dot{u}_n + \lambda_4 \ddot{u} \Delta t + \lambda_5 \Delta a \Delta t \quad (2.6)$$

$$\ddot{u}_{n+1} = \ddot{u}_n + \Delta a \quad (2.7)$$

We see that given the general algorithm in Eq. 2.4, in order to take the form of 2.2 the values of \tilde{a} , \tilde{v} , \tilde{u} , and \tilde{F} are then defined as follows.

$$\tilde{a} = \ddot{u}_n + \Lambda_6 W_1 (\ddot{u}_{n+1} - \ddot{u}_n) \quad (2.8)$$

$$\tilde{v} = \dot{u}_n + \Lambda_4 W_1 \ddot{u}_n \Delta t + \Lambda_5 W_2 (\ddot{u}_{n+1} - \ddot{u}_n) \Delta t \quad (2.9)$$

$$\tilde{u} = u_n + \Lambda_1 W_1 \dot{u}_n \Delta t + \Lambda_2 W_2 \ddot{u}_n \Delta t^2 + \Lambda_3 W_3 (\ddot{u}_{n+1} - \ddot{u}_n) \Delta t^2 \quad (2.10)$$

$$\tilde{F} = (1 - W_1) f_n + W_1 f_{n+1} \quad (2.11)$$

As highlighted earlier and as described in our previous efforts, contained within the GSSSS family of algorithms are two distinct classes of algorithms termed as U0 algorithms and V0 algorithms. The U0 family is the set of algorithms which display zero overshoot in the displacement term, and the V0 family the set which display zero overshoot in the velocity term. Since the set of U0 and V0 algorithms have different parameters W_i , Λ_i , and λ_i it is first necessary that the time level be derived separately, and clearly identified for each of the two distinct families of algorithms.

2.4 The Framework for the U0 Family and the Corresponding Time Level

The framework of the U0 family of algorithms is the one which contains the majority of the commonly used algorithms appearing in the literature such as the Newmark method, the midpoint rule, three-parameters optimal schemes (generalized- α), HHT- α , WBZ, and U0-V0-optimal and the like. These commonly known algorithms are often those used in some research and in certain commercial structural dynamics codes. Therefore, most illustrative examples appearing in this chapter are given specific to them. The derivation and interpretation for the equation of motion time level in the U0 family can be shown in two different ways. The first is a very nice, straight-forward derivation based upon linear interpolation. The second looks closely

at the physical meaning of the series expansion to determine the correct time level. It is shown subsequently that in the latter, although it appears a bit more cumbersome, it is extremely useful and serves as a starting point that is necessary for providing a basis, and for describing the fundamental concepts underlying the V0 family of algorithms which have only recently been designed under the class of LMS methods. These do not appear in the common literature other than our recent developments, and have not been well understood to-date as they are not as straightforward as the U0 family of algorithms.

To start, the U0 family time level is derived by taking the general form of the GSSSS framework of algorithms and rearranging it into the a-form, v-form, and d-form representations. They all yield the same results and have been shown to be exactly equivalent, with the only difference being the variable in question that is being solved as the primary variable. In the a-form, the primary variable is acceleration, and as such the term Δa is the computed variable. Similarly, in v-form and d-form the variables Δv and Δd are the respective primary variables. Looking at the primary variable in each form leads to an easy understanding of the time level for each \tilde{a} , \tilde{v} , and \tilde{d} .

Before looking at the three unknowns, the way in which the GSSSS framework handles the (known) external force can help one to understand how the other unknowns are to be treated. In 2.4, we see that the load term appears in the form:

$$(1 - W_1)f_n + W_1f_{n+1} \tag{2.12}$$

This type of approximation between two known values is simply a linear interpolation of f at some distance W_1 forward from f_n . To visualize this concept see Figure 2.1. In the case of a time dependent load the algorithm uses a trapezoidal rule to approximate the value of the external force at the time t_{n+W_1} .

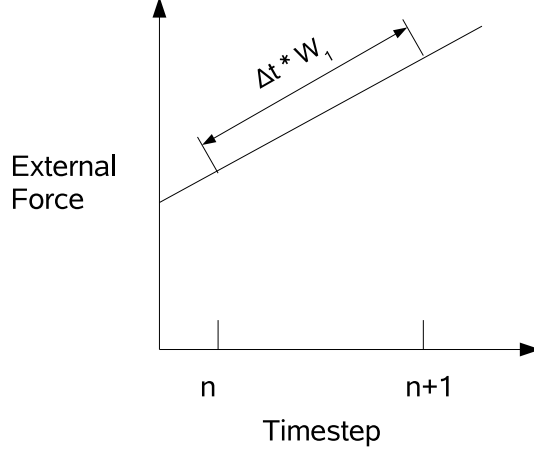


Figure 2.1: A basic linear interpolation of external force at t_{n+W_1}

The general form of GSSSS given above in Eq. 2.4, is called the a-form since the primary variable (the variable which the algorithm directly solves for) is the acceleration. From Eq. 2.8, it is clear that \tilde{a} is a linear interpolation between the values of a_{n+1} and a_n , located at the intermediate value $\Lambda_6 W_1 \Delta t$.

It is useful to note that in the U0 family of algorithms the following relations of parameters exist.

$$\begin{aligned}
 W_1 &= W_2 = W_3 \\
 \Lambda_1 &= \lambda_1 = 1 \\
 \Lambda_2 &= \lambda_2 \\
 \Lambda_3 &= \lambda_3 \\
 \Lambda_4 &= \lambda_4 = 1 \\
 \Lambda_5 &= \lambda_5
 \end{aligned} \tag{2.13}$$

Next, for \tilde{v} the v-form of framework of the GSSSS family of algorithms is constructed. To do so, the update for velocity in Eq. 2.6 is substituted into the general a-form in

Eq. 2.4. This yields an algorithm which solves first for velocity, and the updates to advance to the next time level are used to then find the corresponding values of the displacement and acceleration. Looking at the terms which C multiplies, the \tilde{v} can be readily found to be:

$$\tilde{v} = W_1 v_{n+1} + (1 - W_1) v_n \quad (2.14)$$

In this form \tilde{v} similarly is a linear interpolation of v_{n+1} and v_n , but instead is at the intermediate value W_1 , unlike acceleration. This difference and related aspects and consequences is the specific topic of a forthcoming paper [12], and it turns out to be a key fundamental concept to understand. In the past, this has been the source of great confusion, and the underlying issues concern the accuracy of acceleration computations which are not well understood to-date.

In an analogous manner, again following the steps above, the d-form of the framework of the GSSSS family of algorithms is next used to calculate the \tilde{u} term. As before, the update in Eq. 2.5 is substituted into Eq. 2.4. The resulting equation yields the following:

$$\tilde{u} = W_1 u_{n+1} + (1 - W_1) u_n \quad (2.15)$$

Similar to \tilde{v} , the value of \tilde{u} is simply a linear interpolation at time level t_{n+W_1} .

Looking back at Eq. 2.2, we now see specifically that the framework of the GSSSS family of algorithms satisfies the equation of motion at the time t_{n+W_1} for time dependent variables v , u , and F . The value of a appears to be inconsistently calculated not at time t_{n+W_1} but instead at time $t_{n+W_1\Delta t}$. A forthcoming paper [12] carefully describes this seemingly inconsistent time level evaluation, and shows that the acceleration is in fact also calculated at time t_{n+W_1} . Satisfying the equation of motion at a consistent time level is to be expected. Finally, this specific time level is now established as a known quantity for *any* algorithm that is designed from the second

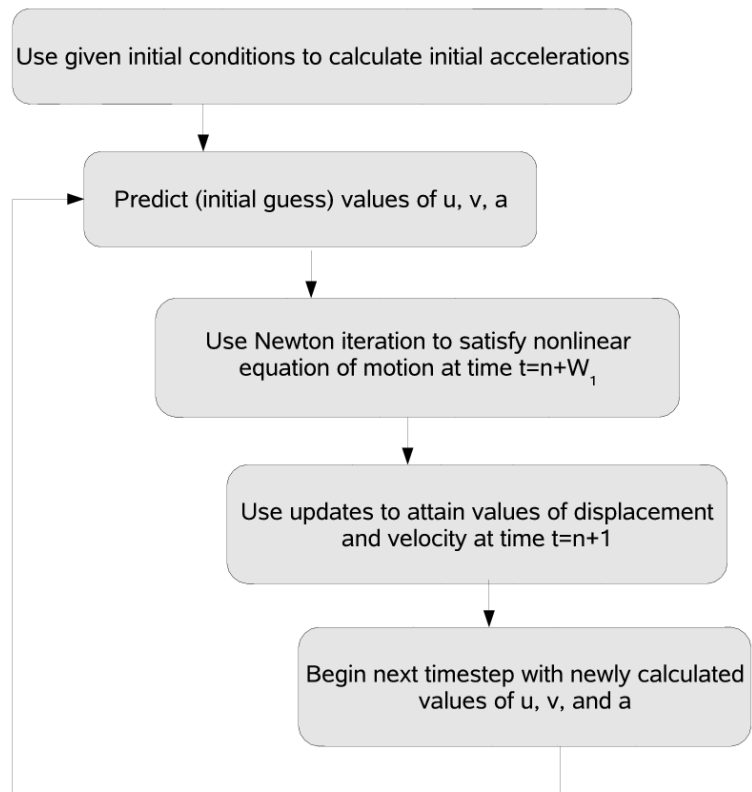


Figure 2.2: Overview of the GSSSS algorithm in a-form

order accurate framework of the GSSSS family of algorithms.

Based on the above findings, the very basic implementation aspects and computational steps of the framework of the GSSSS family of algorithms can be illustrated as shown in Figure 2.2.

2.5 Time Level Consistency for \tilde{a} , \tilde{v} , and \tilde{u}

Recall from the previous section that for the case of linear structural dynamics, all designs can be cast into the following form.

$$\begin{aligned}\tilde{a} &= \Lambda_6 W_1 a_{n+1} + (1 - \Lambda_6 W_1) a_n \\ \tilde{v} &= W_1 v_{n+1} + (1 - W_1) v_n \\ \tilde{u} &= W_1 u_{n+1} + (1 - W_1) u_n\end{aligned}\tag{2.16}$$

It was described earlier that for all of the variables in Eq. 2.2, with the exception of \tilde{a} , that the other variables were treated as a linear interpolation between time t_n and t_{n+1} at the specific time point t_{n+W_1} . Instead, \tilde{a} appeared to be interpolated at time level $t_{n+W_1\Lambda_6}$ from Eq. 2.8. Intuition tells us that Eq. 2.8 needs to be calculated at the same time level for all variables involved. That is, we do not evaluate the equation of motion with values of acceleration which are not at the same point in time as the values used in velocity and displacement. Doing so would not result in a solution of the original problem nor provide any physically meaningful results.

Combining the fact that algorithms appear to be calculating accelerations differently from all other variables, and the fact that it is reported in many places in the literature that accelerations in general, only appear to be first order time accurate (even though the algorithms are derived in such a way that all of the primary variables should ideally be second order accurate), it was conceived that the accelerations coming from the algorithm might not be aligned at the same time as the other variables. Physically, this would mean that the quantities a_n and a_{n+1} which come out of the algorithm are actually not at time t_n and t_{n+1} . To notate this shift, consider that the accelerations instead might be occurring at some time $t_{n-\phi}$ and $t_{n-\phi+1}$. To enforce the fact that when solving the equation of motion all of the variables need to be at the same time

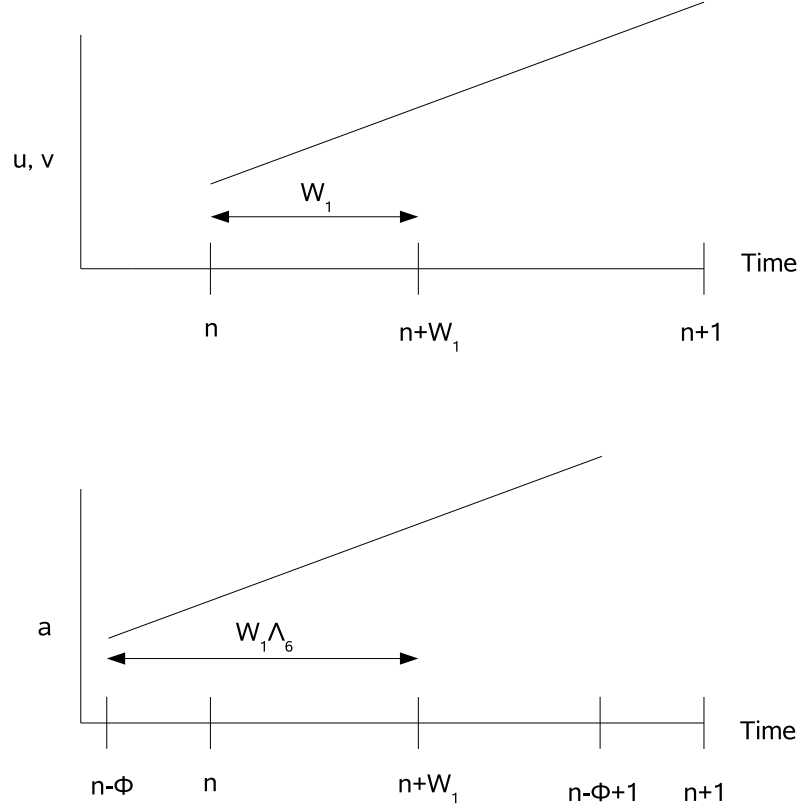


Figure 2.3: Resulting view of a single time step for a shifted acceleration time level

level, it is proposed that the interpolation between $a_{n-\phi}$ and $a_{n-\phi+1}$ at the time level $W_1\Lambda_6$ needs to result in the value of acceleration at the time level t_{n+W_1} . Visualizing such a result is not readily evident to the casual reader; hence, Fig. 2.3 is described to help visualize this concept.

To solve for the newly introduced variable ϕ we take advantage of the fact that we know we need $t_{n-\phi+W_1\Lambda_6}$ to be equal to time level t_{n+W_1} . We then see:

$$n\Delta t + W_1\Delta t = n\Delta t - \phi\Delta t + W_1\Lambda_6\Delta t \quad (2.17)$$

where ϕ can be solved from Eq. (2.17) as

$$\phi = W_1(\Lambda_6 - 1) \quad (2.18)$$

Given this assumed shift in the acceleration time level, the generic form of the framework of the GSSSS family of algorithms would now appear as follows.

$$\begin{aligned}
& M[\ddot{u}_{n-\phi} + \Lambda_6 W_1(\ddot{u}_{n+1-\phi} - \ddot{u}_{n-\phi})] \\
& + C(\dot{u}_n + \Lambda_4 W_1 \ddot{u}_{n-\phi} \Delta t + \Lambda_5 W_2(\ddot{u}_{n+1-\phi} - \ddot{u}_{n-\phi}) \Delta t) \\
& + K[u_n + \Lambda_1 W_1 \dot{u}_n \Delta t + \Lambda_2 W_2 \ddot{u}_{n-\phi} \Delta t^2 + \Lambda_3 W_3(\ddot{u}_{n+1-\phi} - \ddot{u}_{n-\phi}) \Delta t^2] \\
& = (1 - W_1) f_n + W_1 f_{n+1} \tag{2.19}
\end{aligned}$$

or

$$M\tilde{a}(t) + C\tilde{v}(t) + K\tilde{u}(t) = \tilde{F}(t) \tag{2.20}$$

Assuming this, the definition of variables \tilde{a} , \tilde{v} , \tilde{u} , now become:

$$\tilde{a} = \ddot{u}_{n-\phi} + \Lambda_6 W_1(\ddot{u}_{n+1-\phi} - \ddot{u}_{n-\phi}) \tag{2.21}$$

$$\tilde{v} = \dot{u}_n + \Lambda_4 W_1 \ddot{u}_{n-\phi} \Delta t + \Lambda_5 W_2(\ddot{u}_{n+1-\phi} - \ddot{u}_{n-\phi}) \Delta t \tag{2.22}$$

$$\tilde{u} = u_n + \Lambda_1 W_1 \dot{u}_n \Delta t + \Lambda_2 W_2 \ddot{u}_{n-\phi} \Delta t^2 + \Lambda_3 W_3(\ddot{u}_{n+1-\phi} - \ddot{u}_{n-\phi}) \Delta t^2 \tag{2.23}$$

$$\tilde{F} = (1 - W_1) f_n + W_1 f_{n+1} \tag{2.24}$$

In [12] a detailed description of how to interpret acceleration results described, and it is shown that this shifted acceleration time level ideally provides second order accuracy in acceleration as well as displacement and velocity.

2.6 EOM Time Level: A General Derivation for the U0 and V0 Family of Algorithms

Though the previous derivation for the U0 family time level computation of the equation of motion was rather straight-forward, it does not readily extend itself to

the V0 family of algorithms. The U0 family of algorithms are rather physically easy to interpret, unlike the relatively newly designed counterparts to the U0 family of algorithms, namely, the V0 family of algorithms as described by Zhou and Tamma [4]. As such, this section demonstrates an equivalent derivation for both the U0 family and the V0 family of algorithms within the GSSSS framework encompassing LMS methods.

The basis of this alternative derivation is in first defining the generic trapezoidal rule representation, and then applying a Taylor series expansion of the terms in the approximation. In general the trapezoidal rule for approximating a function f between t_n and t_{n+1} is:

$$f(t_{n+C}) = f(t_{n+1})C + (1 - C)f(t_n) \quad (2.25)$$

The constant C is the location of the approximation in time, beginning at t_n , along the straight line connecting $f(t_{n+1})$ and $f(t_n)$. The Taylor series expansion of a function $f(t)$ for $f(n + C)$ around a known value of $f(n)$ up to the second order term is:

$$f(C) = f(n) + f'(n)(C - n) + \frac{f''(n)(C - n)^2}{2!} \quad (2.26)$$

If now Eq. 2.26 is applied to Eq. 2.25 with the unknown $C = W_1$, we obtain an expected expansion of the terms \tilde{v} and \tilde{u} to be of the form:

$$f(W_1) = f(n) + f'(n)W_1\Delta t + \frac{1}{2}f''(n)W_1\Delta t^2 \quad (2.27)$$

To determine the time level of the equation of motion based upon Eq. 2.19, we first look at the acceleration term \tilde{a} . Recall that the above was defined as highlighted

below.

$$\tilde{a} = \ddot{u}_{n-\phi} + \Lambda_6 W_1 (\ddot{u}_{n+1-\phi} - \ddot{u}_{n-\phi}) \quad (2.28)$$

As noted earlier, for both the U0 family and V0 family of algorithms this is already of the form of a linear interpolation of a_n and a_{n+1} at some intermediate value $a_{n+W_1\Lambda_6}$, therein producing an acceleration at t_{n+W_1} . Therefore, we have

$$\begin{aligned} \tilde{a} &= \ddot{u}_{n-\phi} + \Lambda_6 W_1 (\ddot{u}_{n+1-\phi} - \ddot{u}_{n-\phi}) \\ &= a(t_{n+W_1}) \end{aligned} \quad (2.29)$$

For \tilde{v} to resemble the expansion shown in Eq. 2.27 from v_n to v_{n+W_1} , it needs to contain only values of u , v , and a at time t_n . Recall that currently it is defined to be:

$$\tilde{v} = \dot{u}_n + \Lambda_4 W_1 \ddot{u}_{n-\phi} \Delta t + \Lambda_5 W_2 (\ddot{u}_{n+1-\phi} - \ddot{u}_{n-\phi}) \Delta t \quad (2.30)$$

Knowing that the acceleration is approximated in a linear fashion, we can now construct $a_{n+1-\phi}$ and $a_{n-\phi}$ in terms of a_n and a_{n+1} as follows.

$$a_{n+1-\phi} = \phi a_n + (1 - \phi) a_{n+1} \quad (2.31)$$

$$a_{n-\phi} = (1 + \phi) a_n - \phi a_{n+1} \quad (2.32)$$

Substituting Eqs. 2.31 and 2.32 into the definition of \tilde{v} , and utilizing the relation

$\Lambda_4 = 1$ (applies to both U0 and V0 family)

$$\begin{aligned}
\tilde{v} &= v_n + a_n W_1 \Delta t + a_n \Delta t (\phi \Lambda_4 W_1 - \Lambda_5 W_2) + a_{n+1} \Delta t (\Lambda_5 W_2 - \phi \Lambda_4 W_1) \\
&= v_n + a_n W_1 \Delta t + \dot{a}_{n+\frac{1}{2}} \Delta t^2 (\Lambda_5 W_2 - \phi \Lambda_4 W_1) \\
&= v_n + a_n W_1 \Delta t + H.O.T. \\
&= v(t_{n+W_1})
\end{aligned} \tag{2.33}$$

In the sense of the expansion in Eq. 2.27, it can then be seen from Eq. 2.33 that \tilde{v} is precisely an expansion of v_n to v_{n+W_1} out to the necessary first order term. This applies to both the U0 family and the V0 family of algorithms, since no terms specific to either family are necessary in the formulation.

To show that \tilde{u} also can be represented by a truncated Taylor series, we follow the same procedure as for \tilde{v} above. For \tilde{u} to resemble a Taylor expansion of u_n to u_{n+W_1} , it needs to contain only values of u , v , and a at time t_n . Making use of Eqs. (2.31)-(2.32), we have the following which applies to both the U0 and V0 families,

$$\begin{aligned}
\tilde{u} &= u_n + \Lambda_1 W_1 \dot{u}_n \Delta t + \Lambda_2 W_2 \ddot{u}_{n-\phi} \Delta t^2 + \Lambda_3 W_3 (\ddot{u}_{n+1-\phi} - \ddot{u}_{n-\phi}) \Delta t^2 \\
&= u_n + \Lambda_1 W_1 \dot{u}_n \Delta t + \Lambda_2 W_1 \ddot{u}_{n-\phi} \Delta t^2 \\
&\quad + (\Lambda_3 W_3 - \phi \Lambda_2 W_2 - \frac{1}{2} \Lambda_2 (W_1 - W_2)) \dot{a}_{n+\frac{1}{2}} \Delta t^3 + \Lambda_2 (W_2 - W_1) \dot{a}_{n+\frac{1}{2}} \Delta t^3 \\
&= u_n + \Lambda_1 W_1 \dot{u}_n \Delta t + \Lambda_2 W_1 \ddot{u}_{n-\phi} \Delta t^2 \\
&\quad + (\Lambda_3 W_3 - \phi \Lambda_2 W_2 + \frac{1}{2} \Lambda_2 (W_1 - W_2)) \dot{a}_{n+\frac{1}{2}} \Delta t^3 \\
&= u_n + \Lambda_1 W_1 \dot{u}_n \Delta t + \Lambda_2 W_1 \ddot{u}_{n-\phi} \Delta t^2 + H.O.T. \\
&= u(t_{n+W_1})
\end{aligned} \tag{2.34}$$

For both the U0 algorithms and V0 family of algorithms, it is shown that \tilde{u} is precisely a Taylor series expansion of u_n to u_{n+W_1} up to second-order accuracy.

It is now clearly evident that the equation of motion which typical algorithms solve

for within the U0 family and V0 family of algorithms, lies at a time level t_{n+W_1} . An understanding of the importance of this, and its significance is described in depth in the following sections.

2.7 Overview of Existing Methods

Under the framework of the GSSSS family of algorithms, all of the currently existing and more recent new developments of optimal algorithms that are second order time accurate encompassing LMS methods can be constructed by first selecting the family (either the so-called U0 family or V0 family), and then the three parameters $\rho_{1\infty}$, $\rho_{2\infty}$, and $\rho_{3\infty}$ specify the particular algorithm that is selected. Simply for illustration, Table 2.1 highlights some of the more commonly known algorithms, and how to implement them under the GSSSS framework.

Table 2.1: Commonly known algorithms

Algorithms Family($\rho_{1\infty}, \rho_{2\infty}, \rho_{3\infty}$)	Common name	Conditions
U0(1,1,0)	Newmark	-
U0/V0(1,1,1)	Mid-point Rule	-
U0($\rho_{1\infty}, \rho_{2\infty}, \rho_{3\infty}$)	Three-parameter optimal scheme (Generalized- α)	$\rho_{1\infty} = \rho_{2\infty} = \rho_{3\infty} = \rho_{\infty}, 0 \leq \rho_{\infty} \leq 1$
U0($\rho_{1\infty}, \rho_{2\infty}, 0$)	WBZ	$\rho_{1\infty} = \rho_{2\infty} = \rho_{\infty}, 0 \leq \rho_{\infty} \leq 1$
U0($\rho_{1\infty}, \rho_{1\infty}, \rho_{3\infty}$)	HHT- α	$\rho_{1\infty} = \rho_{2\infty}, \rho_{3\infty} = \frac{1 - \rho_{1\infty}\rho_{2\infty}}{\rho_{1\infty} + \rho_{2\infty} + 2\rho_{1\infty}\rho_{2\infty}}$
U0/V0($\rho_{1\infty}, 1, \rho_{3\infty}$)	U0V0 Optimal	$\rho_{1\infty} = \rho_{3\infty} = \rho_{\infty}, 0 \leq \rho_{\infty} \leq 1$
V0(1,1,0)	Velocity-based Scheme	-

Parameter W_1 is defined differently for the U0 and V0 family of algorithms:

$$\text{U0 Family: } W_1 = \frac{1}{1 + \rho_{3\infty}} \quad (2.35)$$

$$\text{V0 Family: } W_1 = \frac{3 + \rho_{1\infty} + \rho_{2\infty} - \rho_{1\infty}\rho_{2\infty}}{2(1 + \rho_{1\infty})(1 + \rho_{2\infty})} \quad (2.36)$$

Using these relations, it is easy to see that the time level for which the equation of motion is being satisfied for the non-dissipative algorithms, for example, for the Newmark algorithm it is the time point t_{n+1} , for the Midpoint Rule it is the time point $t_{n+\frac{1}{2}}$, and for the Velocity-based scheme it is $t_{n+\frac{1}{2}}$. The algorithms which have controllable numerical dissipation satisfy the equation of motion at a time which is dependent on the selection of the parameters $\rho_{1\infty}, \rho_{2\infty}$, and $\rho_{3\infty}$. For example, for the WBZ it is t_n . Since W_1 is only a function of $\rho_{3\infty}$ the time level for any U0 algorithm can be visualized from Figure 2.7. On the other hand, since the V0 family of algorithms have a value of W_1 which is dependent on two variables, $\rho_{1\infty}$ and $\rho_{2\infty}$, a 3D surface best describes how to visualize the time level as seen in Fig. 2.7. It is worthy to note that the range of time which a typical algorithm solves the equation of motion is between $t_{n+\frac{1}{2}}$ and t_{n+1} for the U0 family of algorithms, and it is between $t_{n+\frac{1}{2}}$ and $t_{n+\frac{3}{2}}$ for the V0 family of algorithms.

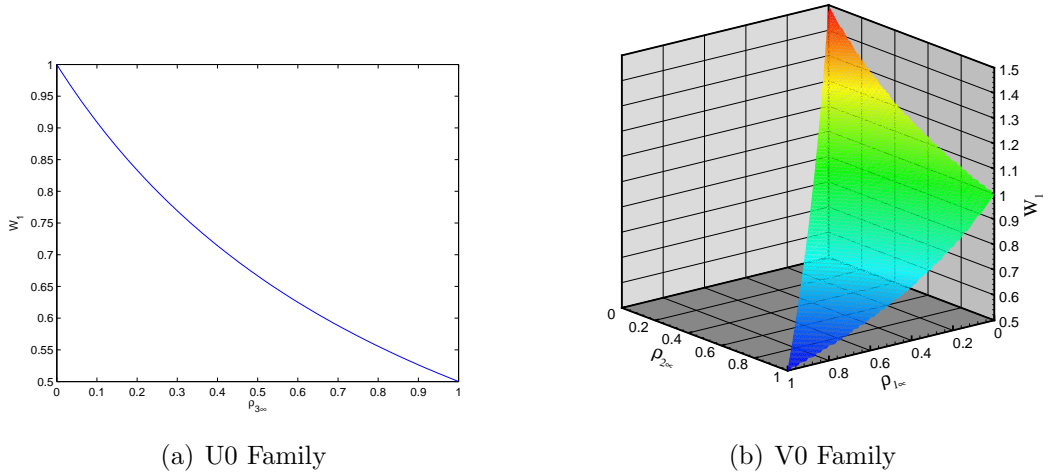


Figure 2.4: Time level of the equation of motion for the U0 and V0 families of algorithms

2.8 Importance and Significance of the EOM Time Level

Having described that both the U0 and V0 families of algorithms satisfy the equation of motion at a consistent time level t_{n+W_1} , the importance of this time level is consid-

ered next and demonstrated via numerical examples. For the linear dynamic cases, if the proper time level at which the equation of motion needs to be satisfied for a given algorithm under consideration is not taken into account, it will clearly lead to degradation in the order of accuracy of the accelerations in the framework of the GSSSS family of algorithms. Nonlinear dynamic applications are somewhat tricky and a brief highlight of the subtle issues follows next. We argue that the algorithmic designs and developments in linear dynamic situations are very valuable, and indeed these parent linear dynamic algorithms form the basis for extensions to the case of non-linear dynamic problems. How to extend such developments to nonlinear dynamics applications needs careful attention, and literature has not very careful or consistent in drawing the subtle, but important distinctions in enacting these extensions. In this regard, for nonlinear dynamic applications, we have described developments following three possible options of implementation as extensions of the parent linear dynamic algorithms: 1) the Classical time weighted residual based algorithmic designs (see Part II and III for details), 2) symplectic-momentum based algorithms (see Part II and III for details), and 3) energy-momentum based algorithms [18, 19] described in detail elsewhere. The classical time weighted residual approach fails to accurately represent the solution of nonlinear dynamic applications. As such, we have described a new normalized time weighted residual approach to more appropriately treat the nonlinear terms in the dynamic equation of motion. This provides an in depth insight to drawing clear distinctions and noting the subtle, yet important consequences in providing extensions to nonlinear dynamics applications. Interested readers should look into the details described in the references provided, and hence only highlights of the numerical illustrations are presented here in Part I of this exposition.

Recall, that for the general case of non-linear dynamic problems the equation of

motion takes the following form.

$$M\tilde{a}(t) + C\tilde{v}(t) + \tilde{p}(t) = \tilde{F}(t)$$

For the symplectic-momentum based algorithmic designs which focus on approximations of the primary variable only, incorrect evaluation of the equation of motion time level for the respective algorithm under consideration leads to degradation in the order of accuracy of the accelerations. On the other hand, for the energy-momentum based algorithmic designs which focus on approximating the strain and displacement independently, these approximations need to be constructed carefully. To be consistent with displacements, velocities, and accelerations (shown to be at time level t_{n+W_1}), the strain would need to be also approximated at the same time level. Until now, in the general case of the GSSSS framework of algorithms, in particular for the controllable dissipative algorithms, there does not exist a viable justification on how to correctly implement the approximation for strain. However, an accurate knowledge of the time level of the equation of motion now provides a means to enable extensions to nonlinear dynamics applications. Incorrect evaluation of the equation of motion time level for the respective algorithm under consideration, leads to degradation not only in the order of accuracy of the accelerations, but also in the displacement and velocity.

The importance of a consistent evaluation the equation of motion time level is first shown for the case of a single degree of freedom (SDOF) model problem. It is shown that disrupting this time level will result in degradation in accuracy of the algorithm under consideration, and in the case of energy-momentum based algorithms, the energy is longer be conserved for the numerical non-dissipative methods. Secondly, for the example of a multi-degree non-linear system of springs and masses, it is shown that an inconsistent evaluation of the equation of motion time level again results in

degradation in the order of accuracy of the algorithm under consideration.

2.9 Numerical Example: SDOF Duffing Equation

We first consider the fairly well known single-degree-of-freedom Duffing equation to demonstrate the fundamental concepts in a manner which permits easily replicable results. The undamped system considered is described by:

$$\ddot{u} + S_1 u + S_2 u^3 = 0 \quad (2.37)$$

or

$$\ddot{u} + p(u) = 0, p(u) = S_1 u + S_2 u^3 \quad (2.38)$$

To demonstrate that the equation of motion needs to be consistently evaluated at the time level introduced above for each respective algorithm under consideration, this problem was formulated using the three different formulations described earlier. The first is referred to as the "classical" approach, the second "symplectic-momentum based", and the third "energy-momentum based". The symplectic-momentum based and energy-momentum based representations are such that when numerical dissipation is turned off, they readily retrieve the original symplectic-momentum or energy-momentum conserving features, respectively. For the above Duffing equation in particular, the classical approach approximates \ddot{u} , u , and p . The symplectic-momentum based representations approximate \ddot{u} and u and simply evaluates the function $p(u)$, resulting in a generalized midpoint rule representation for $p(u)$. The energy-momentum based representation is a hybrid strain/displacement procedure. Here, both strain and displacement are approximated independently. For this particular problem we consider strain as the nonlinear quantity u^2 , which is synonymous to the nonlinear Green strain often used in spring-mass type examples.

2.9.1 "Classical" Approach

Recalling the generic form of the framework of the GSSSS family of algorithms in Eq. 2.4, and rearranging and substituting the update Eq. 2.5 into Eq. 2.4 allows us to formulate the so-called d-form representation where the variable being initially solved is the displacement. This form is somewhat more physically intuitive for this application and is shown simply for illustration. The generic form of the framework of the GSSSS family of algorithms in d-form is shown in Eq. 2.39 along with the corresponding updates needed to advance to the next time level.

$$\begin{aligned}
& M[\ddot{u}_n(1 - \frac{\lambda_2\Lambda_6W_1}{\lambda_3}) - \dot{u}_n \frac{\lambda_1\Lambda_6W_1}{\Delta t\lambda_3} + \Delta u \frac{\Lambda_6W_1}{\Delta t^2\lambda_3}] \\
& + C[\ddot{u}_n\Delta t(\Lambda_4W_1 - \frac{\lambda_2\Lambda_5W_2}{\lambda_3}) + \dot{u}_n(1 - \frac{\lambda_1\Lambda_5W_2}{\lambda_3}) + \Delta u \frac{\Lambda_5W_2}{\Delta t\lambda_3}] \\
& + K[\ddot{u}_n\Delta t^2(\Lambda_2W_2 - \frac{\lambda_2\Lambda_3W_3}{\lambda_3}) + \dot{u}_n\Delta t(\Lambda_1W_1 - \frac{\lambda_1\Lambda_3W_3}{\lambda_3}) + u_n + \Delta u \frac{\Lambda_3W_3}{\lambda_3}] \\
& = (1 - W_1)f_n + W_1f_{n+1}
\end{aligned} \tag{2.39}$$

Associated updates for the above are:

$$u_{n+1} = u_n + \Delta u \tag{2.40}$$

$$\dot{u}_{n+1} = \ddot{u}_n\Delta t(\lambda_4 - \frac{\lambda_2\lambda_5}{\lambda_3}) + \dot{u}_n(1 - \frac{\lambda_1\lambda_5}{\lambda_3}) + \Delta u \frac{\lambda_5}{\Delta t\lambda_3} \tag{2.41}$$

$$\ddot{u}_{n+1} = \ddot{u}_n(1 - \frac{\lambda_2}{\lambda_3}) - \dot{u}_n \frac{\lambda_1}{\lambda_3\Delta t} + \Delta u \frac{1}{\Delta t^2\lambda_3} \tag{2.42}$$

In the "classical" approach the approximations for \ddot{u} and u come from the parent linear dynamic algorithm, but with approximating the variable p (internal force) for the nonlinear dynamic case and not the u . Without the concept of the EOM time level and the details described in Part II and III [20, 21], we have no basis on how and

why to choose this approximation. This leads to the approximation for the internal force as:

$$\tilde{p} = \ddot{p}_n \Delta t^2 \left(\Lambda_2 W_2 - \frac{\lambda_2 \Lambda_3 W_3}{\lambda_3} \right) + \dot{p}_n \Delta t \left(\Lambda_1 W_1 - \frac{\lambda_1 \Lambda_3 W_3}{\lambda_3} \right) + p_n + \Delta p \frac{\Lambda_3 W_3}{\lambda_3} \quad (2.43)$$

Classical Time Weighted Residual Approach: Dissipative Schemes in Unified Predictor Multi-Corrector Representation

For structural dynamic problems for the illustrative truss element with general internal force of the form

$$\mathbf{P} = \mathbf{K}_1 \mathbf{u} + \mathbf{K}_2 \epsilon + \mathbf{K}_3 \mathbf{u} \epsilon \quad (2.44)$$

the effective structural equation employing the classical weighted residual approach is given by

$$\tilde{\mathbf{M}} \ddot{\mathbf{u}} + \tilde{\mathbf{C}} \dot{\mathbf{u}} + \tilde{\mathbf{P}} = \tilde{\mathbf{F}} \quad (2.45)$$

Employ the Newton-Raphson method to iteratively solve for the nonlinear effective structural equation above:

At the beginning of time step, predict the state vectors

$$\tilde{\mathbf{u}}_{n+1}^k = \Xi_{Pa}^{(1)} \mathbf{u}_n + \Xi_{Pa}^{(2)} \dot{\mathbf{u}}_n + \Xi_{Pa}^{(3)} \ddot{\mathbf{u}}_n \quad (2.46)$$

$$\tilde{\dot{\mathbf{u}}}_{n+1}^k = \Xi_{Pv}^{(1)} \mathbf{u}_n + \Xi_{Pv}^{(2)} \dot{\mathbf{u}}_n + \Xi_{Pv}^{(3)} \ddot{\mathbf{u}}_n \quad (2.47)$$

$$\mathbf{u}_{n+1}^k = \Xi_{Pd}^{(1)} \mathbf{u}_n + \Xi_{Pd}^{(2)} \dot{\mathbf{u}}_n + \Xi_{Pd}^{(3)} \ddot{\mathbf{u}}_n \quad (2.48)$$

$$\tilde{\mathbf{P}}_{n+1}^k = \Xi_{Pp}^{(1)} \mathbf{P}_n + \Xi_{Pp}^{(2)} \dot{\mathbf{P}}_n + \Xi_{Pp}^{(3)} \ddot{\mathbf{P}}_n + \Xi_{Pp}^{(4)} \Delta \mathbf{P} \quad (2.49)$$

$$(2.50)$$

Start nonlinear iteration. Solving for $\Delta\delta_{n+1}^{k+1}$ from

$$\begin{aligned} & [\Xi_{Ca}\mathbf{M} + \Xi_{Cv}\mathbf{C} + \Xi_{Cd} {}^t\mathbf{K}_{n+1}^k] \Delta\delta_{n+1}^{k+1} = \\ & - (\mathbf{M}\tilde{\mathbf{u}}_{n+1}^k + \mathbf{C}\tilde{\mathbf{u}}_{n+1}^k + \tilde{\mathbf{P}} - \tilde{\mathbf{F}}_{n+1}^k) \end{aligned} \quad (2.51)$$

where

$${}^t\mathbf{K}_{n+1}^k = \mathbf{K}_1 + \mathbf{K}_2(\mathbf{b} + \frac{4}{l_0^2}\mathbf{A}\mathbf{u}_{n+1}^k) + \mathbf{K}_3[\mathbf{u}_{n+1}^k(\mathbf{b} + \frac{4}{l_0^2}\mathbf{A}\mathbf{u}_{n+1}^k) + \epsilon_{n+1}^k] \quad (2.52)$$

Then correct the primary variables as follows.

$$\tilde{\mathbf{u}}_{n+1}^{k+1} = \tilde{\mathbf{u}}_{n+1}^k + \Xi_{Ca}\Delta\delta_{n+1}^{k+1} \quad (2.53)$$

$$\tilde{\mathbf{u}}_{n+1}^{k+1} = \tilde{\mathbf{u}}_{n+1}^k + \Xi_{Cv}\Delta\delta_{n+1}^{k+1} \quad (2.54)$$

$$\tilde{\mathbf{u}}_{n+1}^{k+1} = \tilde{\mathbf{u}}_{n+1}^k + \Xi_{Cd}\Delta\delta_{n+1}^{k+1} \quad (2.55)$$

$$\tilde{\mathbf{P}}_{n+1}^{k+1} = \Xi_{Cp}^{(1)} \mathbf{P}_n + \Xi_{Cp}^{(2)} \dot{\mathbf{P}}_n + \Xi_{Cp}^{(3)} \ddot{\mathbf{P}}_n + \Xi_{Cp}^{(4)} \Delta\mathbf{P} \quad (2.56)$$

until the solution converges

$$|\delta_{n+1}^{k+1} - \delta_{n+1}^k| < tolerance \quad (2.57)$$

Once the solution converges, update the primary variables at the end of the time step as follows.

$$\ddot{\mathbf{u}}_{n+1} = \ddot{\mathbf{u}}_n + (\tilde{\mathbf{u}}_{n+1}^{k+1} - \ddot{\mathbf{u}}_n)/(\Lambda_6 W_1) \quad (2.58)$$

$$\dot{\mathbf{u}}_{n+1} = \dot{\mathbf{u}}_n + \lambda_4 \ddot{\mathbf{u}}_n \Delta t + \lambda_5 (\ddot{\mathbf{u}}_{n+1} - \ddot{\mathbf{u}}_n) \Delta t \quad (2.59)$$

$$\mathbf{u}_{n+1} = \mathbf{u}_n + \lambda_1 \dot{\mathbf{u}}_n \Delta t + \lambda_2 \ddot{\mathbf{u}}_n \Delta t^2 + \lambda_3 (\ddot{\mathbf{u}}_{n+1} - \ddot{\mathbf{u}}_n) \Delta t^2 \quad (2.60)$$

	a-form	v-form	d-form
$\Xi_{Pd}^{(1)}$	1	1	1
$\Xi_{Pd}^{(2)}$	$\lambda_1 \Delta t$	$\lambda_1 \Delta t$	0
$\Xi_{Pd}^{(3)}$	$\lambda_2 \Delta t^2$	$(\lambda_2 - \frac{\lambda_3 \lambda_4}{\lambda_5}) \Delta t^2$	0
$\Xi_{Pv}^{(1)}$	0	0	0
$\Xi_{Pv}^{(2)}$	1	1	$1 - \frac{\Lambda_5 W_2 \lambda_1}{\lambda_3}$
$\Xi_{Pv}^{(3)}$	$\Lambda_4 W_1 \Delta t$	$(\Lambda_4 W_1 - \frac{\Lambda_5 W_2 \lambda_4}{\lambda_5}) \Delta t$	$(\Lambda_4 W_1 - \frac{\Lambda_5 W_2 \lambda_2}{\lambda_3}) \Delta t$
$\Xi_{Pa}^{(1)}$	0	0	0
$\Xi_{Pa}^{(2)}$	0	0	$-\frac{\Lambda_6 W_1 \lambda_1}{\lambda_3 \Delta t}$
$\Xi_{Pa}^{(3)}$	1	$1 - \frac{\Lambda_6 W_1 \lambda_4}{\lambda_5}$	$1 - \frac{\Lambda_6 W_1 \lambda_2}{\lambda_3}$
$\Xi_{Pp}^{(1)}$	1	1	1
$\Xi_{Pp}^{(2)}$	$(\Lambda_1 W_1 - \frac{\Lambda_3 W_3 \lambda_1}{\lambda_3}) \Delta t$	$(\Lambda_1 W_1 - \frac{\Lambda_3 W_3 \lambda_1}{\lambda_3}) \Delta t$	$(\Lambda_1 W_1 - \frac{\Lambda_3 W_3 \lambda_1}{\lambda_3}) \Delta t$
$\Xi_{Pp}^{(3)}$	$(\Lambda_2 W_2 - \frac{\Lambda_3 W_3 \lambda_2}{\lambda_3}) \Delta t^2$	$(\Lambda_2 W_2 - \frac{\Lambda_3 W_3 \lambda_2}{\lambda_3}) \Delta t^2$	$(\Lambda_2 W_2 - \frac{\Lambda_3 W_3 \lambda_2}{\lambda_3}) \Delta t^2$
$\Xi_{Pp}^{(4)}$	$\frac{\Lambda_3 W_3}{\lambda_3}$	$\frac{\Lambda_3 W_3}{\lambda_3}$	$\frac{\Lambda_3 W_3}{\lambda_3}$
Ξ_{Cd}	$\Lambda_3 W_3 \Delta t^2$	$\frac{\Lambda_3 W_3}{\lambda_5} \Delta t$	$\frac{\Lambda_3 W_3}{\lambda_3}$
Ξ_{Cv}	$\Lambda_5 W_2 \Delta t$	$\frac{\Lambda_5 W_2}{\lambda_5}$	$\frac{\Lambda_5 W_2}{\lambda_3 \Delta t}$
Ξ_{Ca}	$\Lambda_6 W_1$	$\frac{\Lambda_6 W_1}{\lambda_5 \Delta t}$	$\frac{\Lambda_6 W_1}{\lambda_3 \Delta t^2}$
$\Xi_{Cp}^{(1)}$	1	1	1
$\Xi_{Cp}^{(2)}$	$(\Lambda_1 W_1 - \frac{\Lambda_3 W_3 \lambda_1}{\lambda_3}) \Delta t$	$(\Lambda_1 W_1 - \frac{\Lambda_3 W_3 \lambda_1}{\lambda_3}) \Delta t$	$(\Lambda_1 W_1 - \frac{\Lambda_3 W_3 \lambda_1}{\lambda_3}) \Delta t$
$\Xi_{Cp}^{(3)}$	$(\Lambda_2 W_2 - \frac{\Lambda_3 W_3 \lambda_2}{\lambda_3}) \Delta t^2$	$(\Lambda_2 W_2 - \frac{\Lambda_3 W_3 \lambda_2}{\lambda_3}) \Delta t^2$	$(\Lambda_2 W_2 - \frac{\Lambda_3 W_3 \lambda_2}{\lambda_3}) \Delta t^2$
$\Xi_{Cp}^{(4)}$	$\frac{\Lambda_3 W_3}{\lambda_3}$	$\frac{\Lambda_3 W_3}{\lambda_3}$	$\frac{\Lambda_3 W_3}{\lambda_3}$

Table 2.2: Predictor multi-corrector coefficients for the incremental a-, v- and d-form representations

The predictor-corrector coefficients Ξ above in the corresponding a, v, and d-form are listed in Table 2.2.

We see that this approximation is of the exact same form as the approximation for u , and therefore in the U0 family of algorithms it results in a trapezoidal rule representation. In this case (U0 family of algorithms) we have an easily understood physical interpretation of the algorithm. The value of p calculated in the generic algorithm is simply a linear interpolation at time t_{n+W_1} . For the V0 family of algorithms, the approximation cannot be represented exactly in the form of a linear interpolation, and is therefore more difficult to physically interpret.

To fully understand the thought process, consider for a moment that one has the generic linear framework of the GSSSS family of algorithms and wished to apply these parent linear dynamic algorithms to a nonlinear dynamic problem. A possible option of doing so would be to use the approximations for u , \dot{u} , and \ddot{u} from the parent linear dynamic algorithm and then apply a simple trapezoidal rule to the internal force term p . For certain common algorithms it may be possible to identify the correct approximation, for example in the Newmark algorithm we would choose to evaluate p at time level t_{n+1} and the midpoint rule at $t_{n+\frac{1}{2}}$. In general, it is not at all trivial to identify the proper time level, especially when dealing with algorithms with controllable numerical dissipation. However, with the concept of the EOM time level described earlier, we now show the correct way to approximate p : at t_{n+W_1} . The resulting U0 family of algorithms based on this classical approach for the Duffing equation are shown in Eq. 2.61.

$$\begin{aligned}
M\left[\ddot{u}_n\left(1 - \frac{\lambda_2\Lambda_6W_1}{\lambda_3}\right) - \dot{u}_n\frac{\lambda_1\Lambda_6W_1}{\Delta t\lambda_3} + \Delta u\frac{\Lambda_6W_1}{\Delta t^2\lambda_3}\right] + (1 - W_1)p_n + W_1p_{n+1} &= 0 \\
p_n &= S1u_n + S2u_n^3 \\
p_{n+1} &= S1u_{n+1} + S2u_{n+1}^3 \quad (2.61)
\end{aligned}$$

For the U0 family of algorithms, to validate the fundamental ideas put forth, we consider some commonly known algorithms identified in the Table 2.3 below for illustration. Each algorithm is run once with the correctly evaluated internal force time level $p = (1 - W_1)p_n + W_1p_{n+1}$, and then again with p evaluated at an incorrect time level. It can be seen in Figs. 2.5-2.9, that using the correct EOM time level as well as considering the shift in the acceleration time level (Eq. 2.18) ideally results in second order accuracy in all three of the primary variables: displacement, velocity,

and acceleration for all the selected methods. When the EOM time level is violated the accuracy of the respective algorithm falls to only first order for all the primary variables.

The convergence plots were generated using $S1 = 2$ and $S2 = 1$ with initial conditions $u(0) = 1$ and $v(0) = 0$. All runs had an end time of 0.1 seconds, the "exact" solution employed 10,000 time steps, and the numerical solutions employed different number of time steps of 200, 150, 100, and 50, respectively. The incorrect time level shown for each algorithm was chosen arbitrarily.

Table 2.3: Selected U0 Algorithms used in examples and their corresponding EOM time level

Algorithms Family($\rho_{1\infty}, \rho_{2\infty}, \rho_{3\infty}$)	Common name	Value of W_1 (EOM time level)
U0/V0(1,1,1)	Mid-point Rule	0.5
U0(0.6,0.6,0.6)	Three-parameters optimal scheme (Generalized- α)	0.625
U0/V0(0.25,1.0,0.25)	U0/V0 optimal	0.8
U0(0.9,0.9,0.0728)	HHT- α	0.9321
U0(0.25,0.25,0)	WBZ	1.0

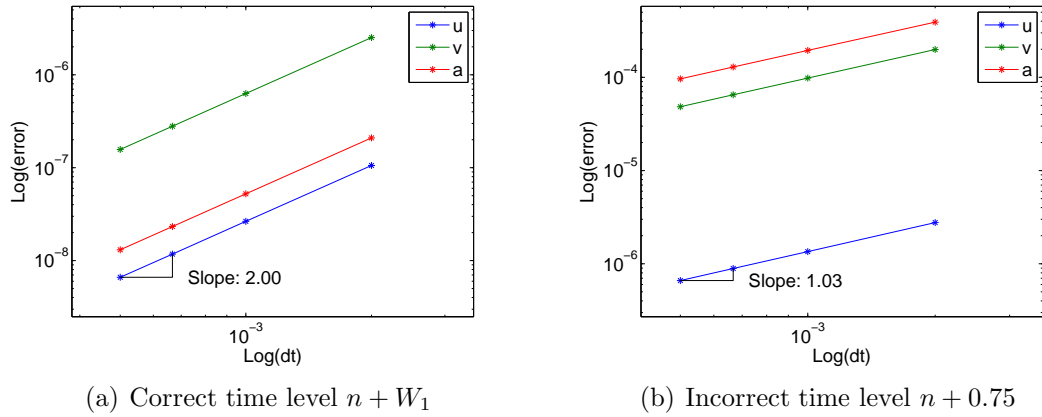
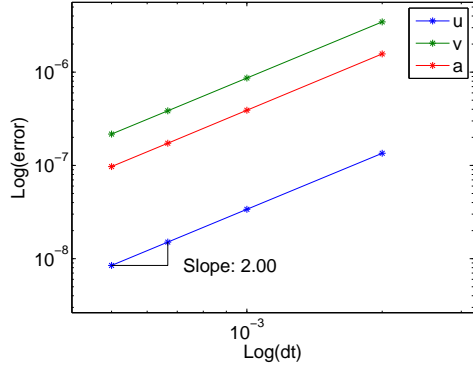
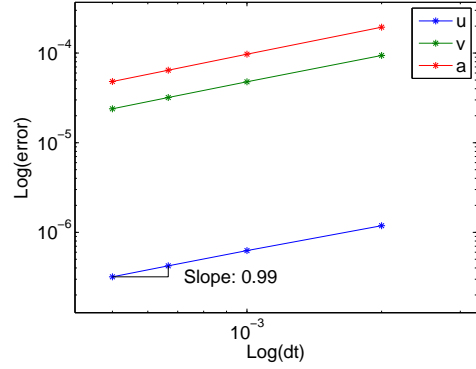


Figure 2.5: U0/V0(1,1,1) Classical method results for Duffing equation

For the V0 family of algorithms, the same problem was analyzed using the selected algorithms shown in Table 2.4. Notice that for the last algorithm the EOM time level is actually at $t_{n+1.1}$, beyond the end of the time step itself. Since there is no simple

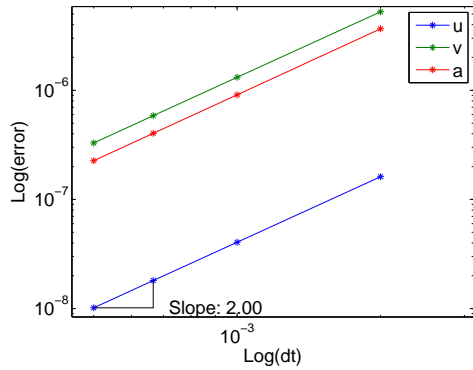


(a) Correct time level $n + W_1$

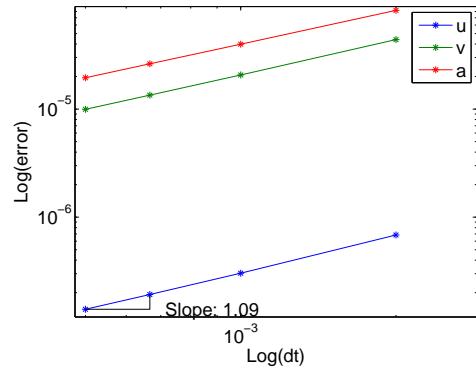


(b) Incorrect time level $n + 0.5$

Figure 2.6: $U_0(0.6,0.6,0.6)$ Classical method results for Duffing equation



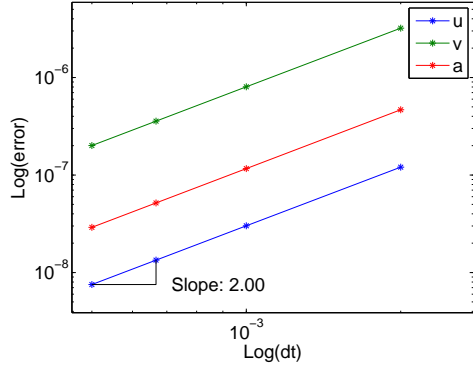
(a) Correct time level $n + W_1$



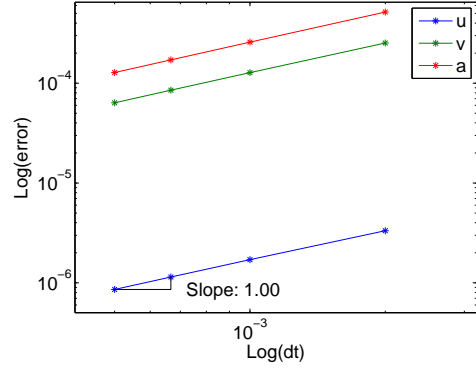
(b) Incorrect time level $n + 0.85$

Figure 2.7: $U_0(0.25,1.0,0.25)$ Classical method results for Duffing equation

and physically understandable approximation for the V_0 family, only the correct time level plots are shown (see Fig. 2.10). Unlike the V_0 , for the U_0 family of algorithms it is feasible that without knowing the correct EOM time level, one could choose incorrectly the time level for p as we previously demonstrated. For the V_0 family, without an intuitive representation such as the trapezoidal rule, it is not as easy to conceive how to form an incorrect time level approximation for p . As such, only the correct time level is shown. These results are equally important in that they show that a thorough understanding of the EOM time level enables one for the first time to show that even algorithms with controllable numerical dissipation are indeed second order accurate in displacement, velocity, *and* acceleration.

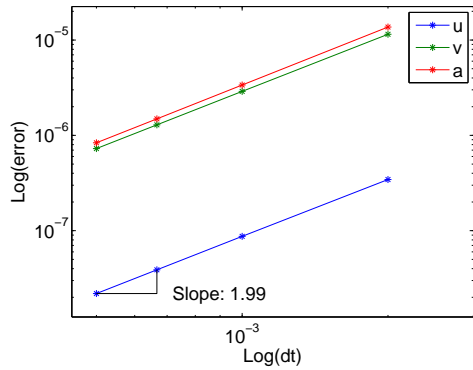


(a) Correct time level $n + W_1$

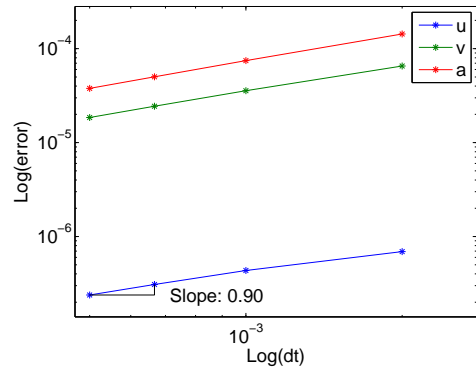


(b) Incorrect time level $n + 0.6$

Figure 2.8: $U_0(0.9,0.9,0.0728)$ Classical method results for Duffing equation



(a) Correct time level $n + W_1$



(b) Incorrect time level $n + 0.9$

Figure 2.9: $U_0(0.25,0.25,0)$ Classical method results for Duffing equation

2.9.2 "Symplectic-Momentum Based" Representations

Here, extending the parent linear dynamic algorithms to the nonlinear dynamic case is rather straightforward in that we simply use the same approximations as in the linear dynamic algorithms for u , v , and a for the latter case as well. Unlike the classical approach, here we do not approximate the internal force term independently, and the normalized time weighted residual approach detailed in Part II and Part III [20,21] explains the theoretical basis. The internal for $p(u)$ is evaluated using the approximation for u , thereby resulting in a generalized midpoint rule representation for the internal force. The resulting generic algorithmic representation is shown below.

Table 2.4: V0 Algorithms used in examples and their corresponding EOM time level

Algorithm: Family($\rho_{1\infty}, \rho_{2\infty}, \rho_{3\infty}$)	Common name	Value of W_1
V0(1,1,0)	Velocity based scheme	0.5
V0(0.6,0.6,0.6)		0.75
V0(0.9,0.9,0.0728)		0.5526
V0(0.25,0.25,0)		1.1

$$M\tilde{a} + p(\tilde{u}) = 0 \quad (2.62)$$

$$\tilde{a} = \ddot{u}_n \left(1 - \frac{\lambda_2 \Lambda_6 W_1}{\lambda_3}\right) - \dot{u}_n \frac{\lambda_1 \Lambda_6 W_1}{\Delta t \lambda_3} + \Delta u \frac{\Lambda_6 W_1}{\Delta t^2 \lambda_3} \quad (2.63)$$

$$\tilde{u} = \ddot{u}_n \Delta t^2 \left(\Lambda_2 W_2 - \frac{\lambda_2 \Lambda_3 W_3}{\lambda_3}\right) + \dot{u}_n \Delta t \left(\Lambda_1 W_1 - \frac{\lambda_1 \Lambda_3 W_3}{\lambda_3}\right) + u_n + \Delta u \frac{\Lambda_3 W_3}{\lambda_3} \quad (2.64)$$

Displacement Based Normalized Time Weighted Residual Approach: Dissipative Schemes in Unified Predictor Multi-Corrector Representation

For structural dynamic problems for the illustrative truss element with general internal force of the form

$$\mathbf{P} = \mathbf{K}_1 \mathbf{u} + \mathbf{K}_2 \epsilon + \mathbf{K}_3 \mathbf{u} \epsilon \quad (2.65)$$

the effective structural equation employing the displacement based normalized time weighted residual approach is given by

$$\mathbf{M}\tilde{\ddot{\mathbf{u}}} + \mathbf{C}\tilde{\dot{\mathbf{u}}} + \tilde{\mathbf{P}} = \tilde{\mathbf{F}} \quad (2.66)$$

where

$$\tilde{\mathbf{P}} = \mathbf{P}(\tilde{\mathbf{u}}) = \mathbf{K}_1 \tilde{\mathbf{u}} + (\mathbf{K}_2 + \mathbf{K}_3 \tilde{\mathbf{u}}) \epsilon(\tilde{\mathbf{u}}) \quad (2.67)$$

$$\epsilon(\tilde{\mathbf{u}}) = \mathbf{b}^T \tilde{\mathbf{u}} + \frac{2}{l_0^2} \tilde{\mathbf{u}}^T \mathbf{A} \tilde{\mathbf{u}} \quad (2.68)$$

Employ the Newton-Raphson method to iteratively solve for the nonlinear effective structural equation above:

At the beginning of time step, predict the state vectors

$$\tilde{\mathbf{u}}_{n+1}^k = \Xi_{Pa}^{(1)} \mathbf{u}_n + \Xi_{Pa}^{(2)} \dot{\mathbf{u}}_n + \Xi_{Pa}^{(3)} \ddot{\mathbf{u}}_n \quad (2.69)$$

$$\tilde{\dot{\mathbf{u}}}_{n+1}^k = \Xi_{Pv}^{(1)} \mathbf{u}_n + \Xi_{Pv}^{(2)} \dot{\mathbf{u}}_n + \Xi_{Pv}^{(3)} \ddot{\mathbf{u}}_n \quad (2.70)$$

$$\tilde{\ddot{\mathbf{u}}}_{n+1}^k = \Xi_{Pd}^{(1)} \mathbf{u}_n + \Xi_{Pd}^{(2)} \dot{\mathbf{u}}_n + \Xi_{Pd}^{(3)} \ddot{\mathbf{u}}_n \quad (2.71)$$

Start nonlinear iteration. Solving for $\Delta \delta_{n+1}^{k+1}$ from

$$\begin{aligned} & [\Xi_{Ca} \mathbf{M} + \Xi_{Cv} \mathbf{C} + \Xi_{Cd} \mathbf{K}_{n+1}^k] \Delta \delta_{n+1}^{k+1} = \\ & - (\mathbf{M} \tilde{\ddot{\mathbf{u}}}_{n+1}^k + \mathbf{C} \tilde{\dot{\mathbf{u}}}_{n+1}^k + \mathbf{K}_1 \tilde{\mathbf{u}}_{n+1}^k + \mathbf{K}_2 \tilde{\epsilon}_{n+1}^k + \mathbf{K}_3 \tilde{\mathbf{u}} \epsilon(\tilde{\mathbf{u}}_{n+1}^k) - \tilde{\mathbf{F}}_{n+1}^k) \end{aligned} \quad (2.72)$$

$${}^t \mathbf{K}_{n+1}^k = \mathbf{K}_1 + \mathbf{K}_2 \left(\mathbf{b} + \frac{4}{l_0^2} \mathbf{A} \tilde{\mathbf{u}}_{n+1}^k \right) + \mathbf{K}_3 \left[\tilde{\mathbf{u}}_{n+1}^k \left(\mathbf{b} + \frac{4}{l_0^2} \mathbf{A} \tilde{\mathbf{u}}_{n+1}^k \right) + \epsilon(\tilde{\mathbf{u}}_{n+1}^k) \right] \quad (2.73)$$

Then correct the primary variables as follows.

$$\tilde{\mathbf{u}}_{n+1}^{k+1} = \tilde{\mathbf{u}}_{n+1}^k + \Xi_{Ca} \Delta \delta_{n+1}^{k+1} \quad (2.74)$$

$$\tilde{\dot{\mathbf{u}}}_{n+1}^{k+1} = \tilde{\dot{\mathbf{u}}}_{n+1}^k + \Xi_{Cv} \Delta \delta_{n+1}^{k+1} \quad (2.75)$$

$$\tilde{\ddot{\mathbf{u}}}_{n+1}^{k+1} = \tilde{\ddot{\mathbf{u}}}_{n+1}^k + \Xi_{Cd} \Delta \delta_{n+1}^{k+1} \quad (2.76)$$

until the solution converges

$$|\delta_{n+1}^{k+1} - \delta_{n+1}^k| < tolerance \quad (2.77)$$

Once the solution converges, update the primary variables at the end of the time step as follows.

$$\ddot{\mathbf{u}}_{n+1} = \ddot{\mathbf{u}}_n + (\tilde{\ddot{\mathbf{u}}}_{n+1}^{k+1} - \ddot{\mathbf{u}}_n)/(\Lambda_6 W_1) \quad (2.78)$$

$$\dot{\mathbf{u}}_{n+1} = \dot{\mathbf{u}}_n + \lambda_4 \ddot{\mathbf{u}}_n \Delta t + \lambda_5 (\ddot{\mathbf{u}}_{n+1} - \ddot{\mathbf{u}}_n) \Delta t \quad (2.79)$$

$$\mathbf{u}_{n+1} = \mathbf{u}_n + \lambda_1 \dot{\mathbf{u}}_n \Delta t + \lambda_2 \ddot{\mathbf{u}}_n \Delta t^2 + \lambda_3 (\ddot{\mathbf{u}}_{n+1} - \ddot{\mathbf{u}}_n) \Delta t^2 \quad (2.80)$$

The predictor-corrector coefficients Ξ above in the corresponding a, v, and d-form are listed in Table 2.2.

For the respective algorithm under consideration via these representations, knowledge of the EOM time level is needed only to correctly account for accelerations due to the lack of any approximations outside the ones provided by the parent linear dynamic algorithms. That is, without any consideration of the EOM time level it is possible to extend the parent algorithms to nonlinear dynamic problems and achieve second order accuracy in both displacement and velocity, but it will appear that accelerations are only first order accurate (just as it does so in the linear dynamic case). As such, only the correctly evaluated time level plots are shown since disturbing the EOM time level would result in modification to the respective parent linear dynamic algorithm.

It can be seen from Figs. 2.11 and 2.12, that indeed we are again able to achieve second order accuracy in all the primary variables. The problem constants, initial conditions, and time steps are all the same as identified previously.

2.9.3 "Energy-Momentum Based" Representations

In contrast to the previous two options, an alternative for extending the linear dynamic framework of the GSSSS family of algorithms to nonlinear dynamic problems is via the energy-momentum based representations. This is in the context that they are able to provide algorithm designs (non-dissipative) which conserve linear and angular momentum, as well as energy exactly for conservative dynamic systems. Details with and without controllable numerical dissipation for such representations are described elsewhere [18, 19].

It involves a hybrid procedure where the strain is approximated independently from the displacement. Similar to the classical approach, for the U0 family of algorithms it defines the strain to be a linear approximation between t_n and t_{n+1} . For this case of the Duffing equation, we define strain to be $\epsilon = u^2$, making the governing equation now of the representation $\ddot{u} + S1u + S2u\epsilon = 0$. Again, the approximations for \ddot{u} and u come directly from the parent linear dynamic algorithms. The importance of understanding the EOM time level now is evident from the strain term, and is synonymous to the internal force in the classical approach. Without knowledge of the EOM time level, simply stating that we wish to approximate strain by a trapezoidal rule representation between t_n and t_{n+1} provides no basis for *where* to choose the time point between t_n and t_{n+1} . Considering the point collocation at time level t_{n+W_1} , we now have a justification for the time point at which to approximate the strain. Figs. 2.13-2.18 again show that if we correctly evaluate the strain at time level t_{n+W_1} , the resulting algorithms are second order accurate in all the variables, whereas any shift from this correct time level destroys the accuracy of the algorithms. The conservation aspects are described for the spring-mass system illustrated in the next example. The resulting generic algorithm for these energy-momentum based

representations is shown in below.

$$M\tilde{a} + p(\tilde{u}, \tilde{\epsilon}) = 0 \quad (2.81)$$

$$\tilde{a} = \ddot{u}_n \left(1 - \frac{\lambda_2 \Lambda_6 W_1}{\lambda_3}\right) - \dot{u}_n \frac{\lambda_1 \Lambda_6 W_1}{\Delta t \lambda_3} + \Delta u \frac{\Lambda_6 W_1}{\Delta t^2 \lambda_3} \quad (2.82)$$

$$\tilde{u} = \ddot{u}_n \Delta t^2 \left(\Lambda_2 W_2 - \frac{\lambda_2 \Lambda_3 W_3}{\lambda_3}\right) + \dot{u}_n \Delta t \left(\Lambda_1 W_1 - \frac{\lambda_1 \Lambda_3 W_3}{\lambda_3}\right) + u_n + \Delta u \frac{\Lambda_3 W_3}{\lambda_3} \quad (2.83)$$

$$\tilde{\epsilon} = \ddot{\epsilon}_n \Delta t^2 \left(\Lambda_2 W_2 - \frac{\lambda_2 \Lambda_3 W_3}{\lambda_3}\right) + \dot{\epsilon}_n \Delta t \left(\Lambda_1 W_1 - \frac{\lambda_1 \Lambda_3 W_3}{\lambda_3}\right) + \epsilon_n + \Delta \epsilon \frac{\Lambda_3 W_3}{\lambda_3} \quad (2.84)$$

Hybrid Displacement-Strain Based Normalized Weighted Residual Approach: Dissipative Schemes in Unified Predictor Multi-Corrector Representation

For structural dynamic problems of truss element with general internal force of the form

$$\mathbf{P} = \mathbf{K}_1 \mathbf{u} + \mathbf{K}_2 \epsilon + \mathbf{K}_3 \mathbf{u} \epsilon \quad (2.85)$$

the effective structural equation employing the hybrid displacement-strain based normalized weighted residual approach is given by

$$\tilde{\mathbf{M}}\ddot{\tilde{\mathbf{u}}} + \tilde{\mathbf{C}}\dot{\tilde{\mathbf{u}}} + \mathbf{K}_1 \tilde{\mathbf{u}} + \mathbf{K}_2 \tilde{\epsilon} + \mathbf{K}_3 \tilde{\mathbf{u}} \tilde{\epsilon} = \tilde{\mathbf{f}} \quad (2.86)$$

At the beginning of time step, predict the state vectors

$$\tilde{\mathbf{u}}_{n+1}^k = \Xi_{Pa}^{(1)} \mathbf{u}_n + \Xi_{Pa}^{(2)} \dot{\mathbf{u}}_n + \Xi_{Pa}^{(3)} \ddot{\mathbf{u}}_n \quad (2.87)$$

$$\tilde{\dot{\mathbf{u}}}_{n+1}^k = \Xi_{Pv}^{(1)} \mathbf{u}_n + \Xi_{Pv}^{(2)} \dot{\mathbf{u}}_n + \Xi_{Pv}^{(3)} \ddot{\mathbf{u}}_n \quad (2.88)$$

$$\tilde{\ddot{\mathbf{u}}}_{n+1}^k = \Xi_{Pd}^{(1)} \mathbf{u}_n + \Xi_{Pd}^{(2)} \dot{\mathbf{u}}_n + \Xi_{Pd}^{(3)} \ddot{\mathbf{u}}_n \quad (2.89)$$

$$\tilde{\epsilon}_{n+1}^k = \Xi_{P\epsilon}^{(1)} \epsilon_n + \Xi_{P\epsilon}^{(2)} \dot{\epsilon}_n + \Xi_{P\epsilon}^{(3)} \ddot{\epsilon}_n + \Xi_{P\epsilon}^{(4)} \Delta \epsilon \quad (2.90)$$

Start nonlinear iteration. Solving for $\Delta\delta_{n+1}^{k+1}$ from

$$\begin{aligned} & [\Xi_{Ca}\mathbf{M} + \Xi_{Cv}\mathbf{C} + \Xi_{Cd}\mathbf{K}_{n+1}^k]\Delta\delta_{n+1}^{k+1} = \\ & - (\mathbf{M}\tilde{\ddot{\mathbf{u}}}_{n+1}^k + \mathbf{C}\tilde{\dot{\mathbf{u}}}_{n+1}^k + \mathbf{K}_1\tilde{\mathbf{u}}_{n+1}^k + \mathbf{K}_2\tilde{\epsilon}_{n+1}^k + \mathbf{K}_3\tilde{\mathbf{u}}_{n+1}^k\tilde{\epsilon}_{n+1}^k - \tilde{\mathbf{F}}_{n+1}^k) \end{aligned} \quad (2.91)$$

$${}^t\mathbf{K}_{n+1}^k = \mathbf{K}_1 + \mathbf{K}_2(\mathbf{b} + \frac{4}{l_0^2}\mathbf{A}\mathbf{u}_{n+1}^k) + \mathbf{K}_3[\tilde{\mathbf{u}}_{n+1}^k(\mathbf{b} + \frac{4}{l_0^2}\mathbf{A}\mathbf{u}_{n+1}^k) + \tilde{\epsilon}_{n+1}^k] \quad (2.92)$$

Then correct the primary variables as follows.

$$\tilde{\ddot{\mathbf{u}}}_{n+1}^{k+1} = \tilde{\ddot{\mathbf{u}}}_{n+1}^k + \Xi_{Ca}\Delta\delta_{n+1}^{k+1} \quad (2.93)$$

$$\tilde{\dot{\mathbf{u}}}_{n+1}^{k+1} = \tilde{\dot{\mathbf{u}}}_{n+1}^k + \Xi_{Cv}\Delta\delta_{n+1}^{k+1} \quad (2.94)$$

$$\tilde{\mathbf{u}}_{n+1}^{k+1} = \tilde{\mathbf{u}}_{n+1}^k + \Xi_{Cd}\Delta\delta_{n+1}^{k+1} \quad (2.95)$$

$$\tilde{\epsilon}_{n+1}^{k+1} = \Xi_{C\epsilon}^{(1)}\epsilon_n + \Xi_{C\epsilon}^{(2)}\dot{\epsilon}_n + \Xi_{C\epsilon}^{(3)}\ddot{\epsilon}_n + \Xi_{C\epsilon}^{(4)}\Delta\epsilon \quad (2.96)$$

until the solution converges

$$|\delta_{n+1}^{k+1} - \delta_{n+1}^k| < tolerance \quad (2.97)$$

Once the solution converges, update the primary variables at the end of the time step as follows.

$$\ddot{\mathbf{u}}_{n+1} = \ddot{\mathbf{u}}_n + (\tilde{\ddot{\mathbf{u}}}_{n+1}^{k+1} - \ddot{\mathbf{u}}_n)/(\Lambda_6 W_1) \quad (2.98)$$

$$\dot{\mathbf{u}}_{n+1} = \dot{\mathbf{u}}_n + \lambda_4\ddot{\mathbf{u}}_n\Delta t + \lambda_5(\ddot{\mathbf{u}}_{n+1} - \ddot{\mathbf{u}}_n)\Delta t \quad (2.99)$$

$$\mathbf{u}_{n+1} = \mathbf{u}_n + \lambda_1\dot{\mathbf{u}}_n\Delta t + \lambda_2\ddot{\mathbf{u}}_n\Delta t^2 + \lambda_3(\ddot{\mathbf{u}}_{n+1} - \ddot{\mathbf{u}}_n)\Delta t^2 \quad (2.100)$$

$$\epsilon_{n+1} = \epsilon(\mathbf{u}_{n+1}) \quad (2.101)$$

The predictor-corrector coefficients Ξ above in the corresponding a, v, and d-form are

listed in Table 2.2.

As in the previous, the convergence plots were generated using $S1 = 2$ and $S2 = 1$ with initial conditions $u(0) = 1$ and $v(0) = 0$ and the same end time and steps as above. The incorrect time level is again chosen arbitrarily, but different from the examples shown for the classical approach. This is to further demonstrate that the proper time level is the only way to preserve second order accuracy.

2.10 Example: Non-linear Tetrahedral Spring-Mass System

In the previous example, we have shown the fundamental concepts underlying the EOM time level using a simple single-degree-of-freedom problem. As a demonstration for a mutli-DOF model problem, we consider the spring element described by Kuhl [22]. Using this spring element a system of four masses connected by six springs was constructed to form a tetrahedron, see Fig. 2.19(a). Each spring has initial length of 1, stiffness of 10^3 and mass 1. The initial position of the four nodes are given as:

$$[0.5, 3^{0.5}/2, 0, 0, 0, 0, 1, 0, 0, 0.5, 1/(2 * 3^{0.5}), (2/3)^{0.5}]^T$$

with given initial displacement of

$$[0, 0.5, 0.2, 0, 0, 0, 0, 0.8, 0, 0, 0, 0]^T$$

and initial velocity of

$$[0, 0, 6, 0, 0, 0, 0, 0, 0, 1, 3, 2]^T$$

The nonlinear dynamic response of the system over a period of 5 seconds with a time step size $\Delta t = 0.1s$ using the so-called energy-momentum conservation EMC can be seen in Figure 2.19. The spring material model is linear, but accounting for large displacement formulation, a nonlinear Green strain is applied.

2.10.1 Classical Approach

Again the above three options of extending the parent linear dynamic framework of the GSSSS family of algorithms to nonlinear problems are used as a demonstration of the importance of the EOM time level for a multi-DOF model problem. For this example we illustrate only the U0V0 optimal algorithm and the EMC simply for demonstration.

Fig. 2.20 shows the results of the classical approach with selected number of time steps of 50, 100, 150, and 200 respectively. The end time of 0.1 second was selected for illustration. The "exact" solution used 10,000 time steps. Again p takes the exact form of linear interpolation for the U0 family of representations. For illustration, the particular algorithm used was U0/V0(0.25,1.0,0.25) resulting in the EOM time level of $n + W_1 = n + 0.8$. To demonstrate incorrect time level selection, the results leading to a loss of accuracy when the internal force was approximated as $n + 0.9$ are shown.

2.10.2 "Symplectic-Momentum Based" Representations

Using the symplectic-momentum based representations, the multi-DOF spring-mass system was simulated using the same U0/V0(0.25,1,0.25) algorithm and number of steps as in the classical case. Using a consistent time level to compare the numerical results to the "exact", we are able for the first time show that indeed all GSSSS family of algorithms are second order accurate in acceleration as well as displacement and velocity, even for algorithms with numerical dissipation. See Fig. 2.21.

2.10.3 "Energy-Momentum Based" Representations

The same algorithm and parameters as in the previous were used first approximating strain at the correct $n + W_1 = n + 0.8$ time level, and then again incorrectly at time $n + 0.75$. We can see in Fig. 2.22 that yet again only the correct time level $n + W_1$

will result in second order accuracy in all the three variables.

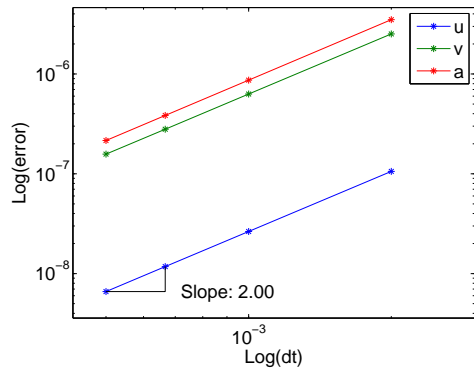
The energy and momentum conserving representations (non-dissipative) provides an algorithm which conserves linear momentum, angular momentum, and energy exactly. In this hybrid displacement-strain based formulation, the decision for how to correctly approximate the strain is not simply an extension of the linear dynamic framework of the GSSSS family of algorithms because no such term exists in the linear dynamic formulation. Utilizing the correct time level it is shown in Figure 2.23 that the method exactly conserves energy, linear momentum, and angular momentum while being second order accurate in time in all the primary variables.

To demonstrate again that the time level of the equation of motion needs to be carefully accounted for in this approximation, the same simulation as above was run with a purposely shifted time level for the strain. In Figure 2.24 we see that both linear and angular momentum are still conserved, but the energy of the system now grows without bound and the order of accuracy degrades to first order.

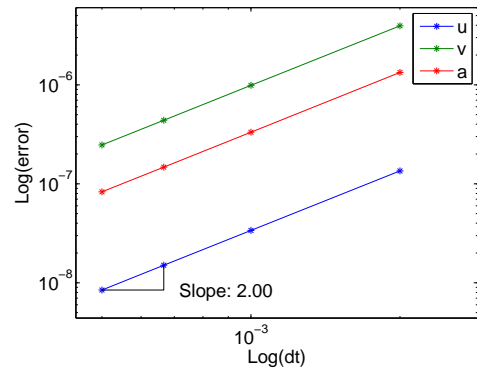
2.11 Conclusion

In Part I of this exposition, we have described the theoretical basis and the significance and importance of the "time level" at which the equation of motion needs to be integrated. It was shown that there is a specific time level at which all the second order accurate time integration algorithms contained within the GSSSS framework which encompasses LMS methods, preserve this same order of time accuracy for all the primary variables (includes non-dissipative and dissipative schemes). It was shown that various options exist for extension of the parent linear dynamic algorithms to nonlinear dynamic problems in order to maintain the second order accuracy of the parent algorithms, if and only if all terms are approximated consistently at the same time level t_{n+W_1} .

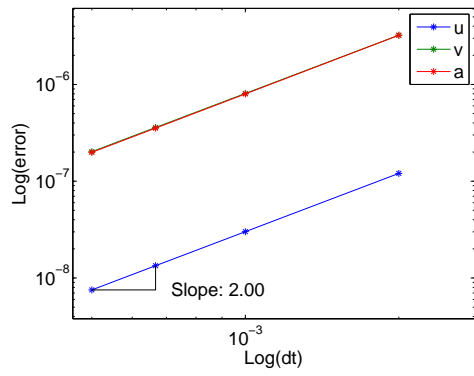
Also, as a result of an in depth understanding this time level, it was shown that for the first time, we are indeed able to demonstrate that algorithms with controllable numerical dissipation are also inherently second order accurate in their acceleration term, as well as in the displacement and velocity. This time level plays an important role in accurately and correctly interpreting acceleration results, and also for precisely applying the linear dynamic framework to nonlinear dynamic situations. Part II [20] and Part III [21] detail applications and extensions to nonlinear dynamic problems with and without numerical dissipation.



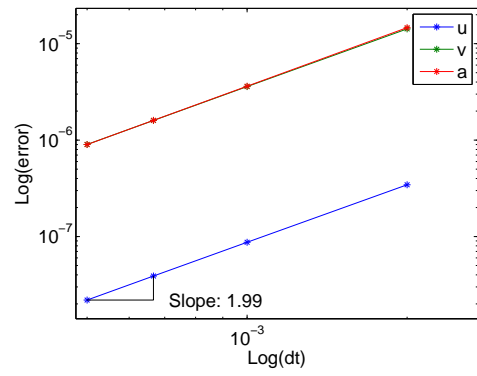
(a) $V_0(1,1,0)$



(b) $V_0(0.6,0.6,0.6)$

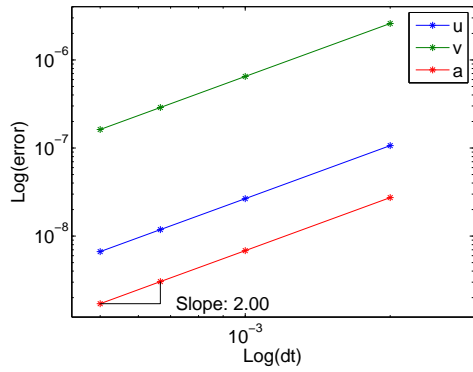


(c) $V_0(0.9,0.9,0.0728)$

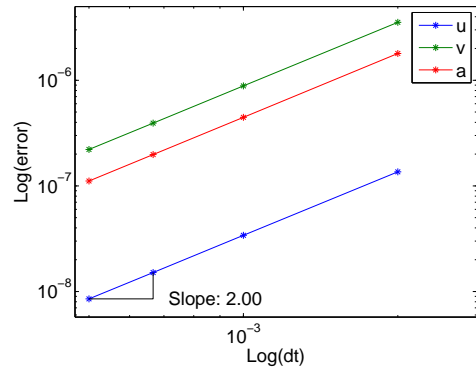


(d) $V_0(0.25,0.25,0)$

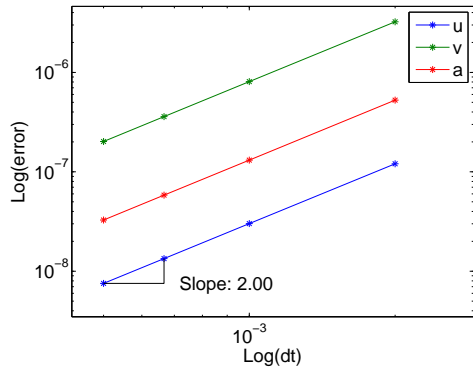
Figure 2.10: V_0 family classical approach results for Duffing equation



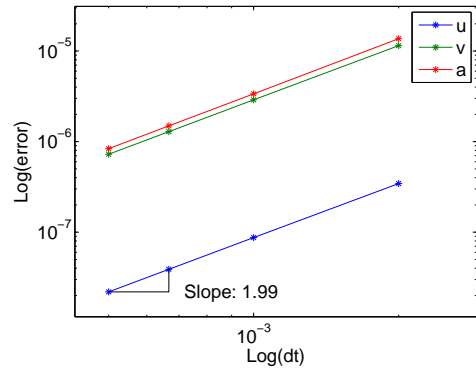
(a) $U_0/V_0(1,1,1)$



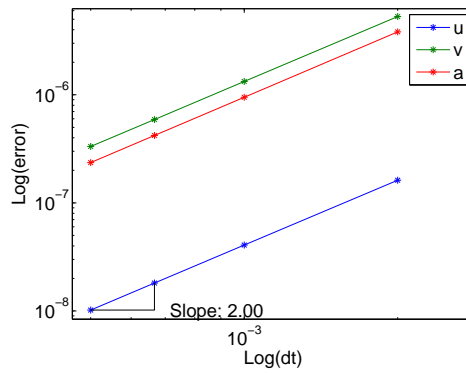
(b) $U_0(0.6,0.6,0.6)$



(c) $U_0(0.9,0.9,0.0728)$

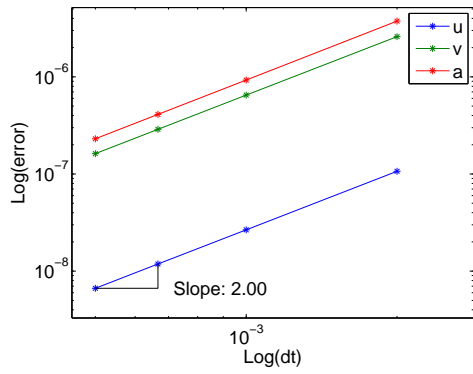


(d) $U_0(0.25,0.25,0)$

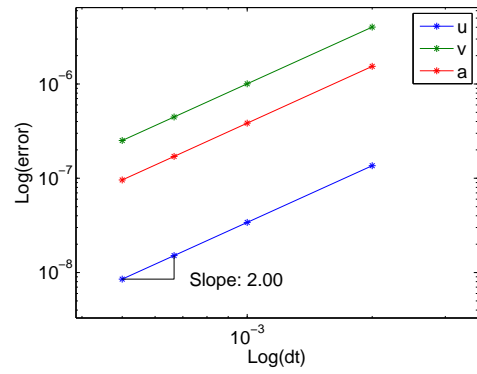


(e) $U_0/V_0(0.25,1.0,0.25)$

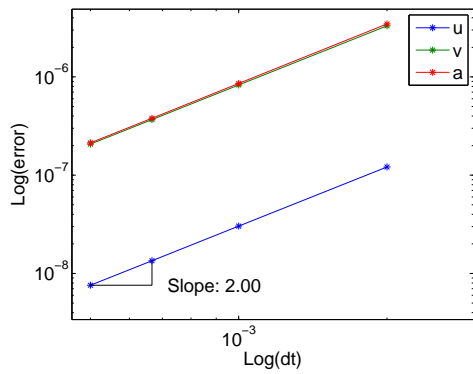
Figure 2.11: U_0 family symplectic based method results for Duffing equation



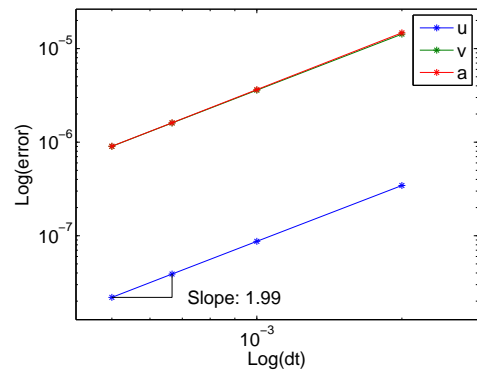
(a) $V0(1,1,0)$



(b) $V0(0.6,0.6,0.6)$

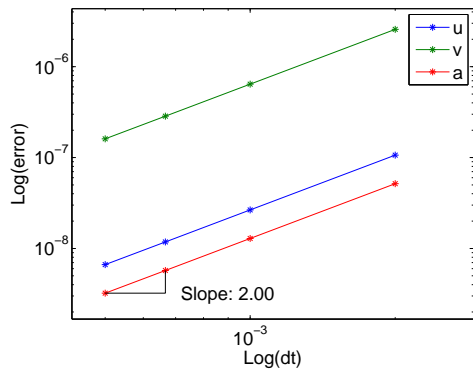


(c) $V0(0.9,0.9,0.0728)$

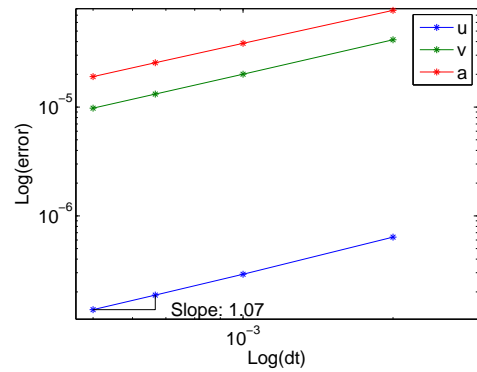


(d) $V0(0.25,0.25,0)$

Figure 2.12: $V0$ family symplectic based method results for Duffing equation

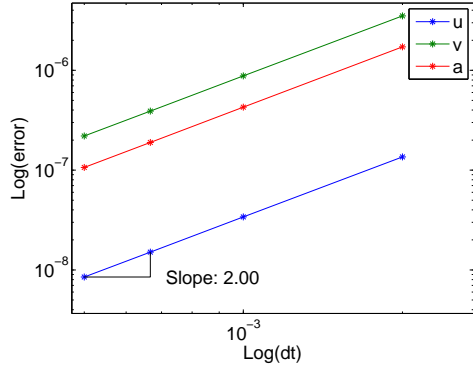


(a) Correct time level $n + W_1$

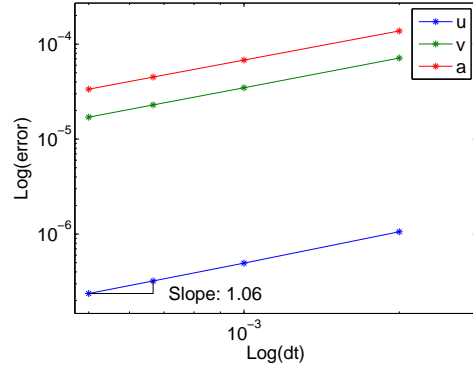


(b) Incorrect time level $n + 0.6$

Figure 2.13: $U0/V0(1,1,1)$ Energy and momentum conserving method results for Duffing equation

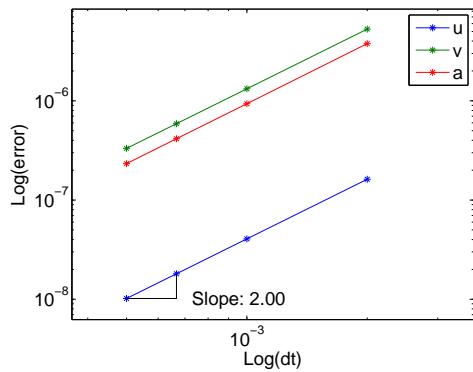


(a) Correct time level $n + W_1$

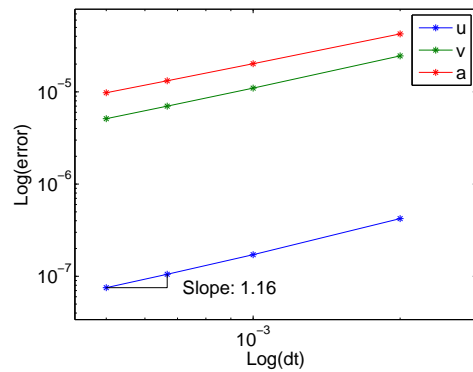


(b) Incorrect time level $n + 0.8$

Figure 2.14: $U_0(0.6,0.6,0.6)$ Energy and momentum conserving method results for Duffing equation

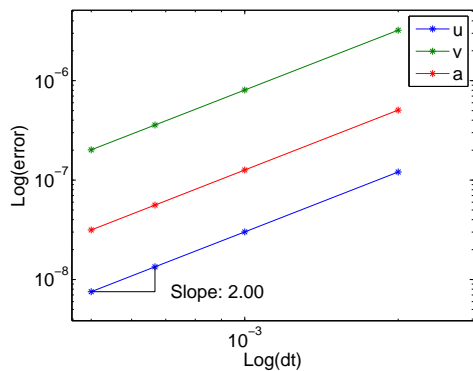


(a) Correct time level $n + W_1$

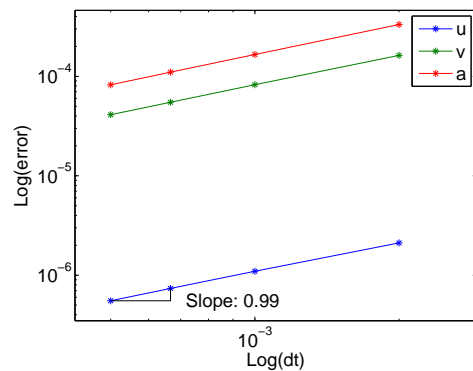


(b) Incorrect time level $n + 0.85$

Figure 2.15: $U_0(0.25,1.0,0.25)$ Energy and momentum conserving method results for Duffing equation

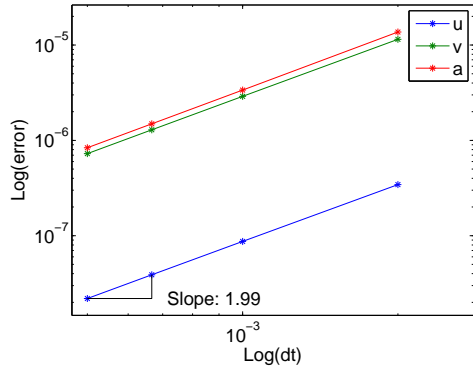


(a) Correct time level $n + W_1$

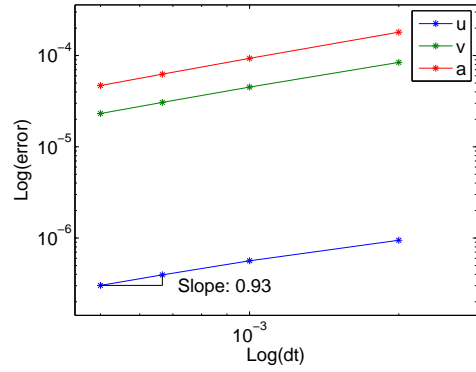


(b) Incorrect time level $n + 0.5$

Figure 2.16: $U_0(0.9,0.9,0.0728)$ Energy and momentum conserving method results for Duffing equation

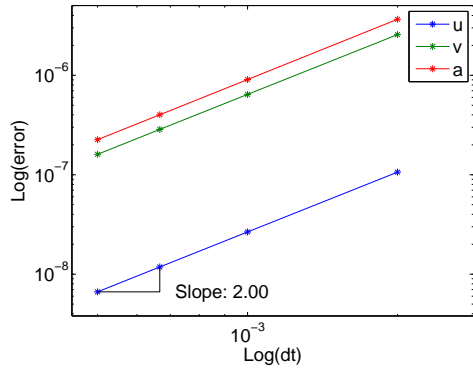


(a) Correct time level $n + W_1$

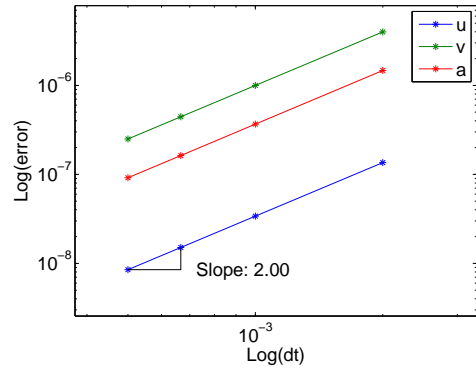


(b) Incorrect time level $n + 0.75$

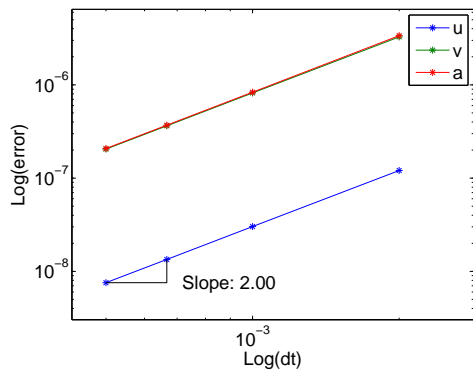
Figure 2.17: $U_0(0.25,0.25,0)$ Energy and momentum conserving method results for Duffing equation



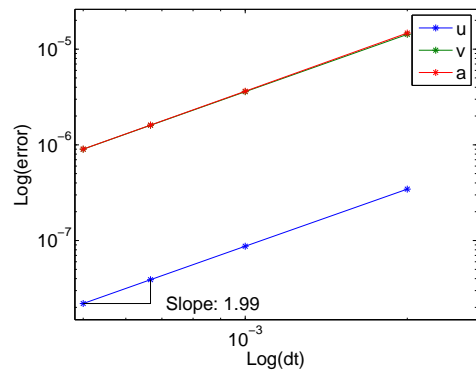
(a) $V_0(1,1,0)$



(b) $V_0(0.6,0.6,0.6)$

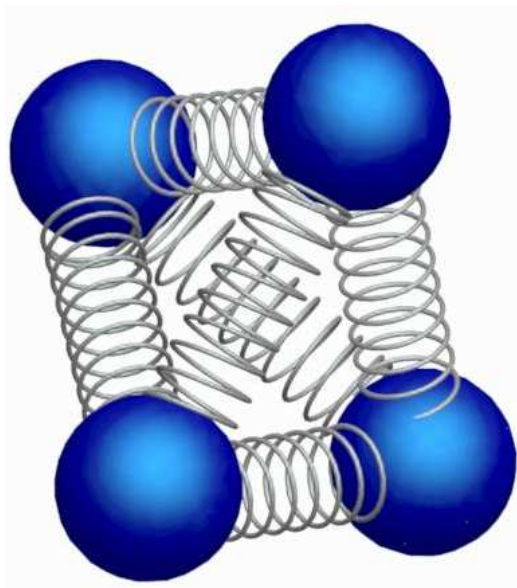


(c) $V_0(0.9,0.9,0.0728)$

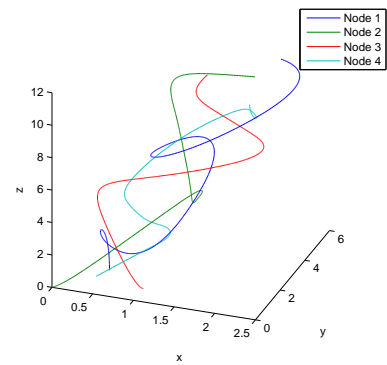


(d) $V_0(0.25,0.25,0)$

Figure 2.18: V_0 family Energy and momentum conserving method results for Duffing equation

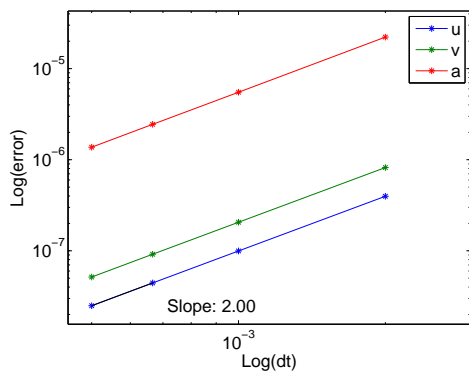


(a) Tetrahedral spring-mass system

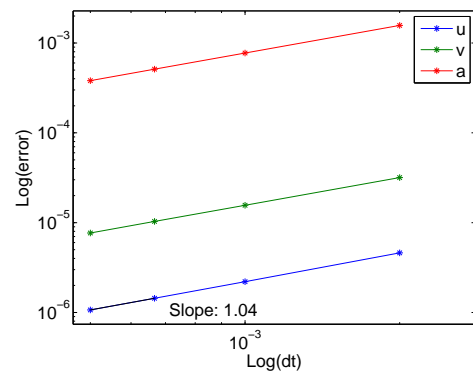


(b) Dynamic response $U0/V0(1,1,1)$

Figure 2.19: Dynamic system response



(a) Correct time level $n + W_1$



(b) Incorrect time level $n + 0.9$

Figure 2.20: $U0/V0(0.25,1,0.25)$ Classical method results for spring-mass system

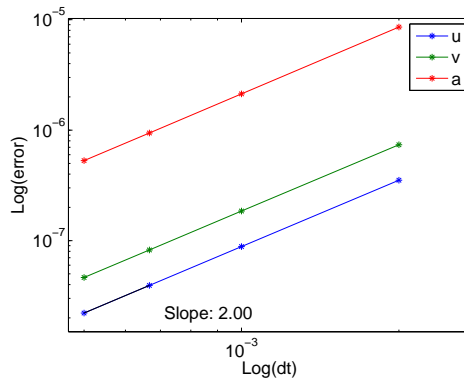
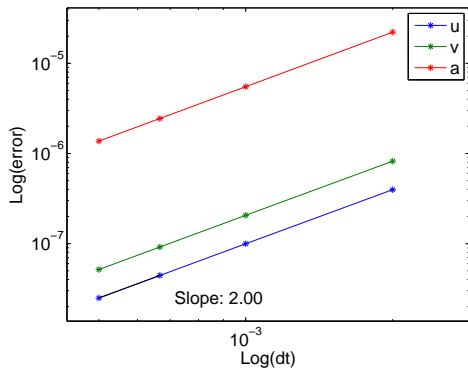
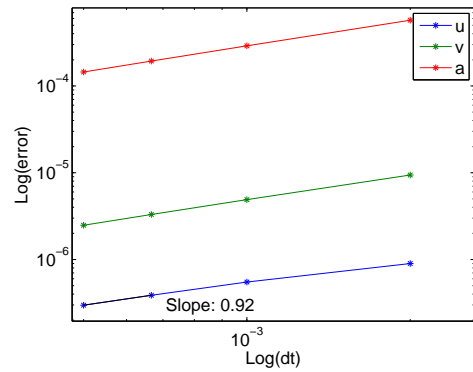


Figure 2.21: U0(0.25,1,0.25) Symplectic based approach results for spring-mass system

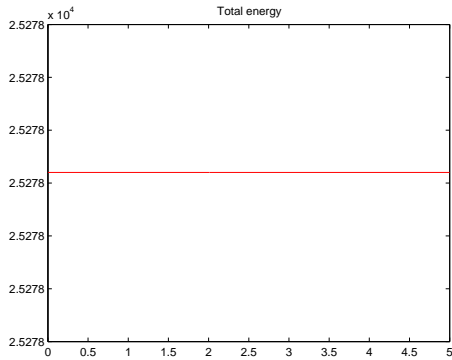


(a) Correct time level $n + W_1$

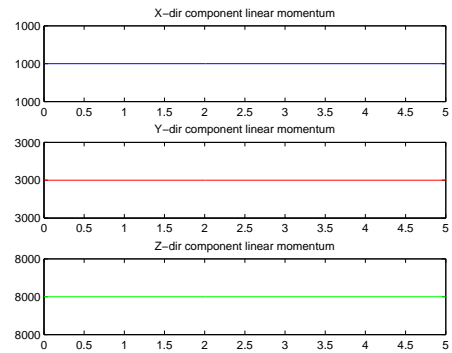


(b) Incorrect time level $n + 0.75$

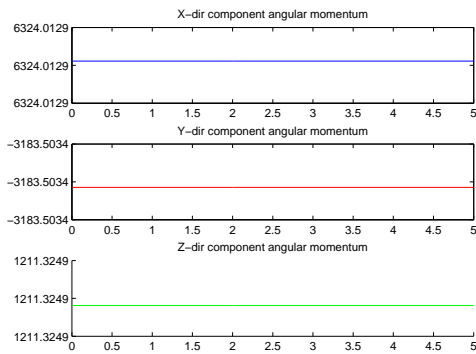
Figure 2.22: U0/V0(0.25,1,0.25) Energy and momentum conserving approach results for spring-mass system



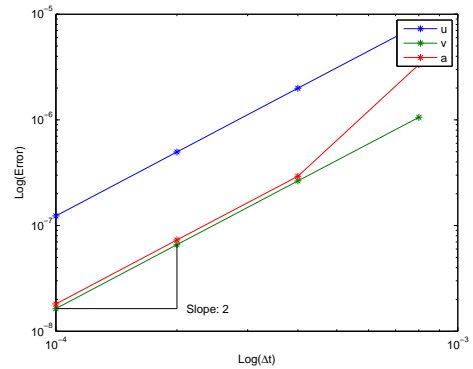
(a) Energy vs. time



(b) Linear momentum vs. time

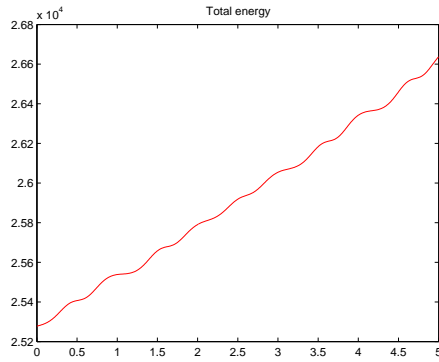


(c) Angular momentum vs. time

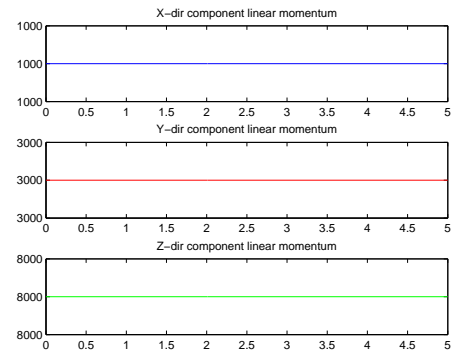


(d) Convergence

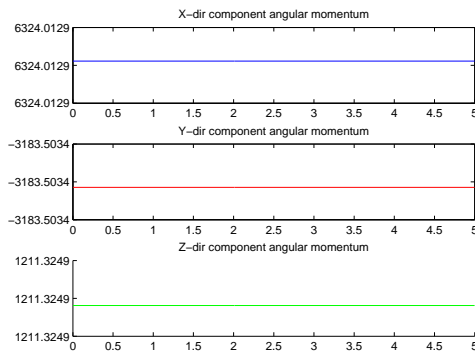
Figure 2.23: Conservation and accuracy using the correct time level , $U0/V0(1,1,1)$ or $V0(1,1, \text{any } \rho_{3\infty})$



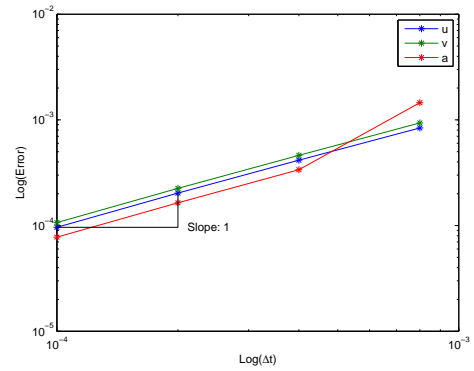
(a) Energy vs. time



(b) Linear momentum vs. time



(c) Angular momentum vs. time



(d) Convergence

Figure 2.24: Conservation and accuracy using the incorrect time level, $U0/V0(1,1,1)$ or $V0(1,1, \text{any } \rho_{3\infty})$

Chapter 3

Precise computation of algorithmic accelerations for non-dissipative and dissipative methods under the class of LMS methods with single step and single solve

3.1 Introduction

The present chapter describes the accurate and precise evaluation of acceleration computations which play an important role in general structural dynamics applications. Computational algorithms for time dependent problems are mostly designed after the semi-discretization is done on the partial differential equations governing the equations of motion. In particular, these computational algorithms are predominantly the class of linear multi-step (LMS) methods that are widely used for general engi-

neering applications and most commercial software. Numerous research efforts have been undertaken in designing computational algorithms over the past fifty years or so starting with the Houbolt method, Newmark methods and the like. The two principal classes of algorithms are non-dissipative methods such as Newmark, the velocity based scheme, midpoint rule, and controllable numerical dissipative methods to control high frequency behavior encountered in stiff dynamical systems. It is well known that numerous application areas exist in science and engineering wherein acceleration calculations need to be precise. Typical applications encompass determination of g-forces encountered in dynamical systems to include impact applications where it is extremely useful to have a knowledge of the accelerations imparted at critical sections of vehicle structures carrying sensitive electronics, in the evaluation of base accelerations imparted due to missiles/torpedos and the like in the vicinity of ship structures which is critical for assessments of ship dynamics, earthquake engineering applications involving base ground motion, and the like. The fundamental question that is posed regarding all these developments to-date is: "are the accelerations that are being computed accurate"? Alternatively, this chapter presents the deficiencies and drawbacks that have plagued the community at large to-date and provides a fundamental resolution to the accurate computation of accelerations for both non-dissipative and dissipative methods with a clear resolution, and consequently puts this matter to rest so that future developments can now accurately implement the dynamics for general applications precisely. For illustration, the focus is on the class of LMS methods.

Recently, Zhou and Tamma [4–6, 13, 23] described a unified theory underlying computational algorithms for designing computational algorithms for time dependent phenomenon. In particular, focusing attention on the class of LMS methods for general structural dynamics applications which are the predominant methods in most commercial software, Zhou and Tamma described a general framework under the umbrella

of generalized single step single solve [GSSSS] family of algorithms. This novel framework encompasses all of the LMS methods that have been developed over the past fifty years, and also includes new methods that have not been available to-date, which are optimal for various applications in structural dynamics. A brief highlight of the so-called big picture follows next so as to put into context the subtle issues and the underlying resolution of the existing drawbacks and deficiencies prevalent in acceleration computations to-date. The GSSSS family of algorithms can be viewed as the fully discretized time integration algorithms for the equation of motion which contains all past and new computational developments. The beauty of this framework is that a single precise time integration module is all that is needed to be implemented which provides all possible algorithms as available choices and options for the analyst to use. Basically, within this framework there exist two distinct families of algorithms termed as constrained u (displacement-overshoot aspects) and constrained v (velocity overshoot aspects) family of algorithms. They simply characterize the overshoot behavior on the displacement or the velocity fields. For example, the Newmark [7] algorithm which is a non-dissipative method belongs to the U0 family with particular characteristics termed as U0-V0; that is it does not have any overshoot behavior for the displacement or velocity. Alternately, the so-called HHT-alpha [15], is a numerically controllable dissipative method that was introduced to control high frequency behavior and it is a U0-V1 algorithm belonging to the U0 family (same as Newmark), which is characterized by no overshoot in displacement and first order overshoot in velocity field. These overshoot characteristics play an important role in that they enter the computations and high overshoot behavior may cause nonlinear iterations in implicit dynamics calculations to not converge. There are numerous algorithms that are contained within the GSSSS family including other more recently developed optimal algorithms that are preferable for many engineering applications [4].

The Issue: The fully discretized equation of motion is valid at all time levels during

the computations, and consequently, these time marching methods take the initial values at time level n and march to the next time level at $n+1$. In this process, they evaluate the primary variables u , v , and a at the next time level $n+1$. In doing these computations, while the displacement and velocity fields are indeed evaluated precisely at $n+1$, the acceleration computations that are reported in the literature and the like, are, in general, not accurately determined and/or are being misinterpreted by the research and commercial computational community at large. This precise determination of the exact time level at which the equation of motion needs to be evaluated, and the shift in time level has been shown recently in Ref. [24] which results in the following reconstruction of the GSSSS family of algorithms.

$$\begin{aligned}
& M[\ddot{u}_{n-\phi} + \Lambda_6 W_1(\ddot{u}_{n+1-\phi} - \ddot{u}_{n-\phi})] \\
& + C(\dot{u}_n + \Lambda_4 W_1 \dot{u}_{n-\phi} \Delta t + \Lambda_5 W_2(\ddot{u}_{n+1-\phi} - \ddot{u}_{n-\phi}) \Delta t] \\
& + K[u_n + \Lambda_1 W_1 \dot{u}_n \Delta t + \Lambda_2 W_2 \ddot{u}_{n-\phi} \Delta t^2 + \Lambda_3 W_3(\ddot{u}_{n+1-\phi} - \ddot{u}_{n-\phi}) \Delta t^2] \\
& = (1 - W_1)f_n + W_1 f_{n+1}
\end{aligned} \tag{3.1}$$

$$\phi = W_1(\Lambda_6 - 1) \tag{3.2}$$

It is this time level shift (ϕ) of the acceleration which has caused significant misinterpretation of how to correctly understand accelerations for dissipative algorithms. For example while it is true (actually happens to be true) for the Newmark method that a indeed is accurately computed at time level $n+1$, for most of the other methods this is not the general case. As a demonstration, Fig. 3.1 shows convergence plots using the traditional (misinterpreted) method for the single degree of freedom problem described later in numerical examples. We see that for the Newmark method

and the midpoint rule (non-dissipative methods) that the order of convergence for all three primary variables (u , v , a) is indeed two. For these two special cases, is it true that the acceleration resulting from the algorithm falls at the time $n+1$. However, for the non-dissipative velocity based scheme [8] in the sense of LMS methods, this is not true (a comes out to be first order only). For all other algorithms, including the non-dissipative velocity based scheme shown, the acceleration time level resulting from the algorithm is shifted from displacement and velocity time levels, and thus incorrectly yields only first order accurate results. It is also routinely shown in the

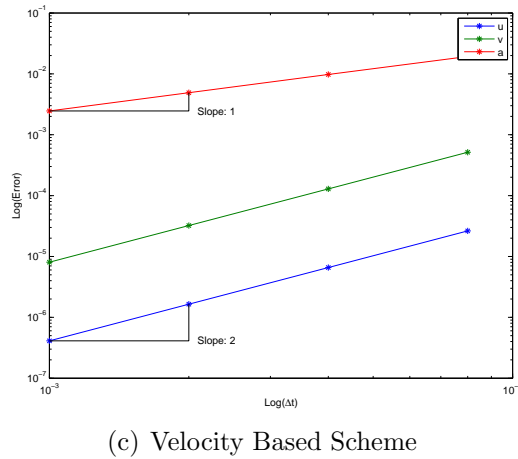
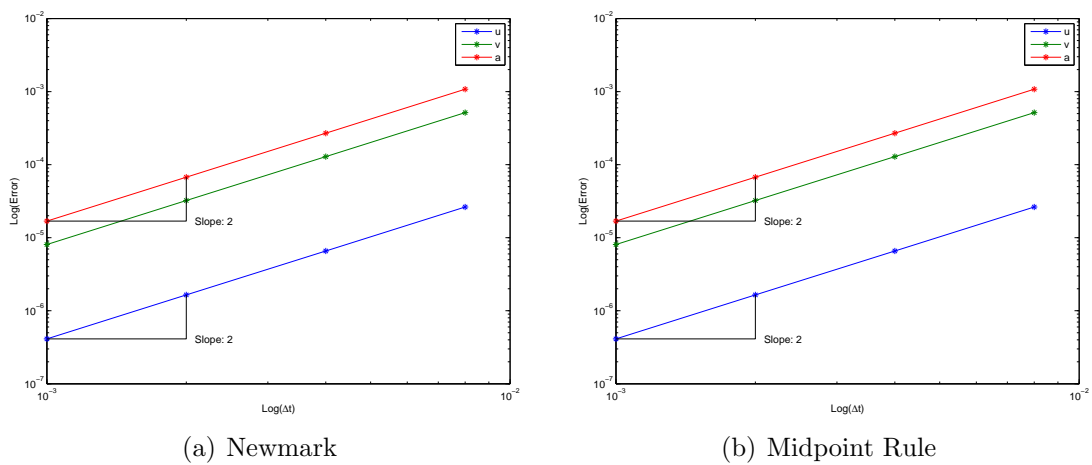


Figure 3.1: Traditional convergence of algorithms without dissipation

literature that for algorithms which include numerical dissipation the order of accuracy in the acceleration is less than two. Fig. 3.2 shows just one example of this using

the well known generalized- α method (and is taken from Ref. [1]).

There exists a fundamental misconception in the use and interpretation of the term "acceleration and its time level", and this chapter provides a deep insight and a resolution to the issue once and for all, with the entire LMS class of algorithms chosen simply for illustration.

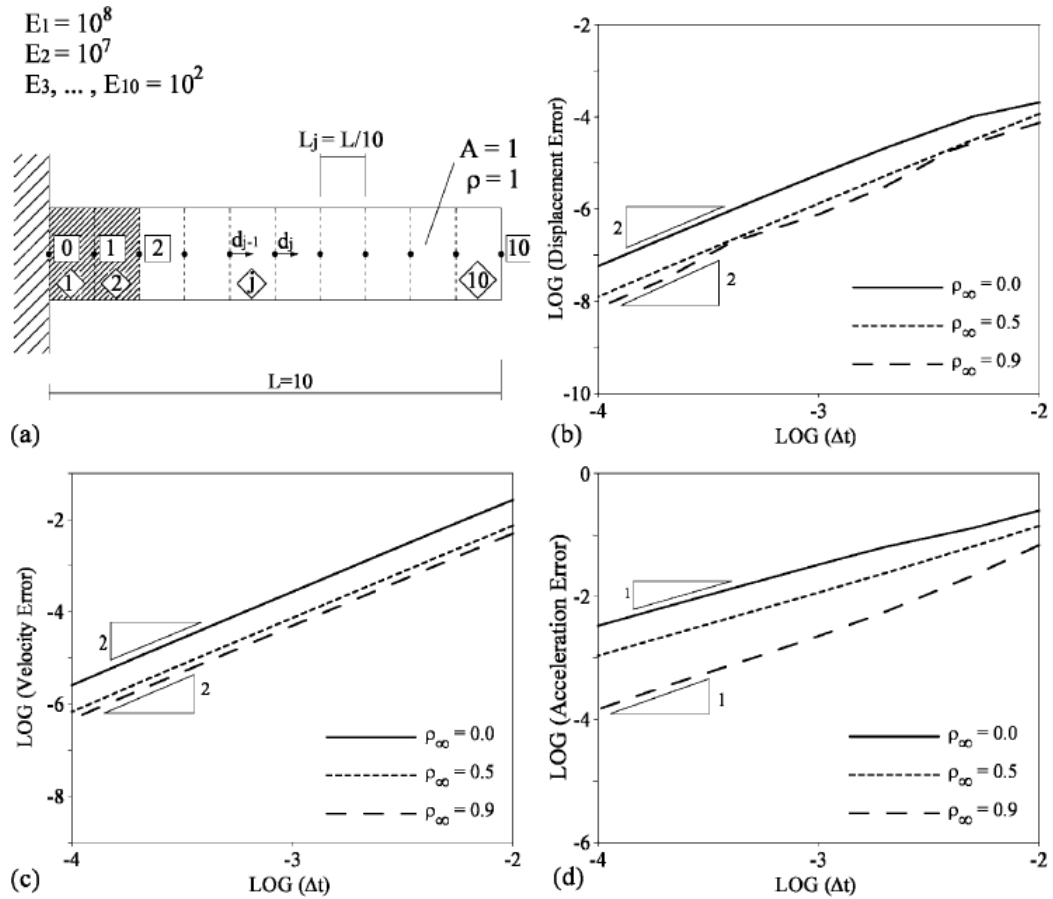


Figure 3.2: Fig. from Ref [1] showing u and v second order accurate, but a is only cited to be first order accurate using the generalized- α method

Understanding the time level at which the acceleration is computed in general computational algorithms is the key to showing proper accuracy. Again, the fundamental concept is that, in general computational algorithms, the acceleration is strictly *not* being calculated at time level t_{n+1} . How the acceleration time level is calculated and how to describe an accurate convergence plot which shows correct accuracy will be

discussed at length below based on the generalized single step single solve (GSSSS) algorithms developed by Zhou and Tamma [4] for the entire class of LMS methods that are predominant in most commercial software.

3.2 U0 family acceleration time level

Eq. 3.1 encompasses all the LMS class of algorithms to-date. Within this, of interest are the class of U0 and V0 algorithms which are the most competitive [4]. To gain a physical interpretation of acceleration time level shift denoted by ϕ shown in Eq. 3.2, ϕ needs to be viewed separately for each of the U0 and V0 families of algorithms due to the fact that values of W_1 and Λ_6 are not the same. For the case of the family of U0 algorithms we have:

$$W_1 = \frac{1}{1 + \rho_{3\infty}} \quad (3.3)$$

$$\Lambda_6 = \frac{2 + \rho_{1\infty} + \rho_{2\infty} + \rho_{3\infty} - \rho_{1\infty}\rho_{2\infty}\rho_{3\infty}}{(1 + \rho_{1\infty})(1 + \rho_{2\infty})}; \quad (3.4)$$

Combining Eqs. 3.2 and 3.3 to rewrite ϕ in terms of the various ρ_∞ 's:

$$\phi(U0) = \frac{1}{1 + \rho_{2\infty}} - \frac{\rho_{1\infty}}{1 + \rho_{1\infty}} \quad (3.5)$$

Recall that ϕ was defined to be the offset of acceleration from time level t_n . As such, the resulting values of acceleration for a given time step between t_n and t_{n+1} are actually at time $t_{n-\phi+1}$. Fig. 3.3a shows this time level vs. $\rho_{1\infty}$ and $\rho_{2\infty}$. We see that for the family of U0 algorithms the calculated accelerations occur at a time between t_n and t_{n+1} . In current practice it is being assumed by all researchers, that what all algorithms yield is the value of acceleration at time t_{n+1} ; this results in a maximum

possible error in time of Δt , as demonstrated later.

3.3 V0 family acceleration time level

In a similar fashion to the U0 family of algorithms, a physical interpretation of ϕ can also be shown by considering the associated possible range of values. For the V0 family of algorithms we have:

$$W_1 = \frac{-1}{2} + \frac{1}{1 + \rho_{1\infty}} + \frac{1}{1 + \rho_{2\infty}} \quad (3.6)$$

$$\Lambda_6 = \frac{2(2 + \rho_{1\infty} + \rho_{2\infty} + \rho_{3\infty} - \rho_{1\infty}\rho_{2\infty}\rho_{3\infty})}{(1 + \rho_{3\infty})(3 + \rho_{1\infty} + \rho_{2\infty} - \rho_{1\infty}\rho_{2\infty})}; \quad (3.7)$$

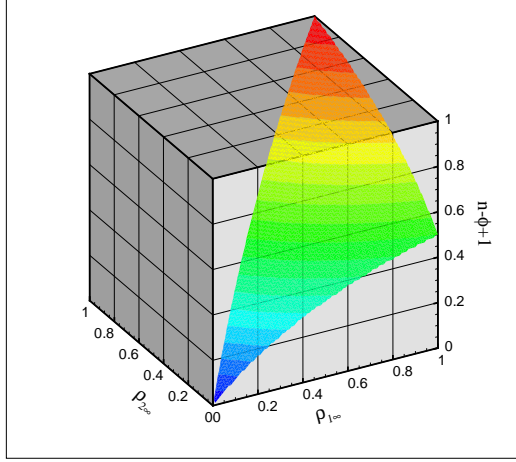
Combining Eqs. 3.2 and 3.6 to rewrite ϕ in terms of the different ρ_∞ 's, we have:

$$\phi(V0) = \frac{1}{1 + \rho_{3\infty}} - \frac{1}{2} \quad (3.8)$$

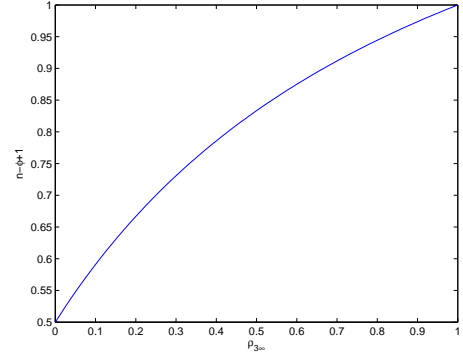
From Eq. 3.8 we see that it is possible to describe and construct a plot of acceleration time level vs. $\rho_{3\infty}$ (see Fig. 3.3b). Notice that in the case of V0, the acceleration resulting from a single time step between t_n and t_{n+1} is located between $t_{n+\frac{1}{2}}$ and t_{n+1} resulting in a maximum possible error in time of $\frac{\Delta t}{2}$, as demonstrated later.

3.4 Implications of a shifted acceleration time level

To fully understand the impact of the above, consider the hypothetical situation of running a simulation to an end time of $\frac{1}{2}$ second with a time step size $\Delta t = 0.1$ with the algorithm where $t_{n-\phi+1} = t_n$ (U0($\rho_{1\infty} = \rho_{2\infty} = \rho_{3\infty} = 0$)). The output of this hypothetical simulation would result in values of displacement, velocity, and



(a) U0 Family



(b) V0 Family

Figure 3.3: Time level of the resulting acceleration over time step t_n to t_{n+1}

acceleration at times seen in the following table.

Time step number	Time	Displacement	Velocity	Acceleration
1	0.1	$u(0.1)$	$v(0.1)$	$a(0.0)$
2	0.2	$u(0.2)$	$v(0.2)$	$a(0.1)$
3	0.3	$u(0.3)$	$v(0.3)$	$a(0.2)$
4	0.4	$u(0.4)$	$v(0.4)$	$a(0.3)$
5	0.5	$u(0.5)$	$v(0.5)$	$a(0.4)$

An outcome of this shift in the acceleration time level is that the notation implied (naively) in the selected algorithm leads to an interpretation that is somewhat misleading. The algorithm in the a-form solves for Δa (meaning we now have $a_{n-\phi+1}$), and then uses the updates to obtain values of v_{n+1} and u_{n+1} . As a result, what would be considered the output/results are actually values of u and v at $n\Delta t$ over the duration of the runtime and values of a at $(n-\phi+1)\Delta t$, even though over the duration of the runtime the equation of motion was satisfied consistently at the same time level. As such, any algorithm that does not satisfy $\phi = 0$ produces data that is very easily misrepresented. The current understanding before the findings of this chapter was that all time integration algorithms simply returned values of u , v , and a at time t_{n+1} .

This is strictly not true, in general. Therefore, the correct time history of u , v , and a should in fact appear with accelerations misaligned with displacements and velocities. To reiterate, if the goal of the hypothetical simulation described above was to obtain values of u , v , and a at time $t = 0.5$, in general a numerically dissipative algorithm will *not* provide all three quantities correctly. This concept has some extremely important implications about the manner in which numerically dissipative algorithms, as well as some non-dissipative methods, are used with respect to accelerations.

3.5 Interpretation and description of a consistent convergence plot

We now are able to address the fact that in general, for numerically dissipative algorithms, to-date one is not able to demonstrate that the accelerations are second order accurate but do obtain second order time accuracy for both displacements and velocities. For non-dissipative algorithms such as the velocity based scheme, this also happens to be the case. To understand the resulting consequences of the shift in the time level which the algorithm returns on a convergence plot, we first must focus on how the plot is produced.

To construct a converge plot (notice here convergence is always meant to mean convergence in time, not convergence of the spatial discretization), the same problem is run multiple times with only a change in the time step size. The time step is gradually reduced and the slope of the result yields the order of accuracy of the variable. An end time of the simulation is chosen and the resulting displacement, velocity, and acceleration are compared to values of the exact solution at that time. If, as in most practical problems, there is no exact known analytic solution, then the simulation is run with a time step which is very small in comparison to the others and this run is considered to be the benchmark solution. The log of the error in the numerical solu-

Algorithms when $\phi = 0$

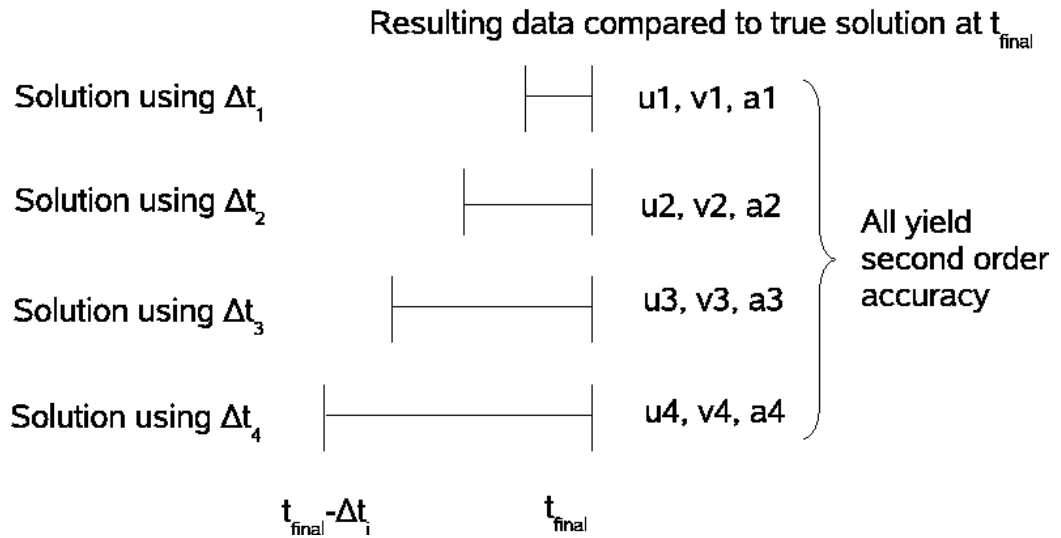


Figure 3.4: Example of the final time steps used in creating a convergence plot when $\phi = 0$

tion is then plotted versus the log of the time step size used in the solution. The slope of the line generated by these points is then the convergence rate of the algorithm.

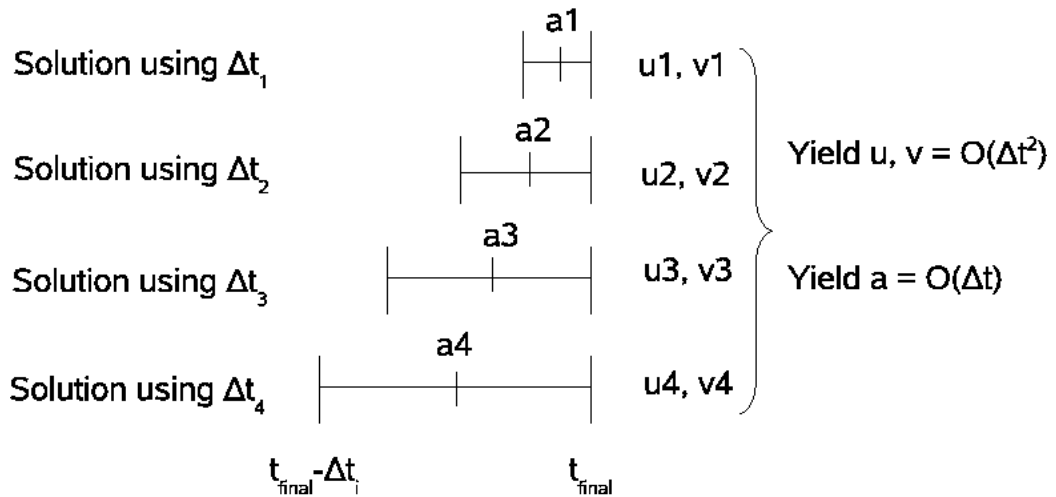
To visualize how the final step of the simulations used to generate the acceleration convergence plot looks like, Figs. 3.4 and 3.5 are provided for illustration. The assumption is the simulation will be run using several different time step sizes, for example: $\Delta t_1 = \Delta t$, $\Delta t_2 = 2\Delta t$, $\Delta t_3 = 3\Delta t$, and $\Delta t_4 = 4\Delta t$. Without properly accounting for the acceleration offset, the results will show second order time accuracy for all three quantities only if $\phi = 0$, and otherwise will show second order time accuracy for u and v , and only first order time accuracy in a as reported in the literature [1]. If the time step sizes are chosen as to properly align accelerations (see bottom of Fig. 3.5), then the comparison of the numerical solutions to the exact solution are occurring at the same time level. It is then, that these normally misrepresented accelerations can be readily proven to be second order accurate.

To properly describe a convergence plot in which the value of acceleration from the

Traditional convergence plot

Algorithms when $\phi \neq 0$ (Shown as ~ 0.5)

Resulting data compared to true solution at t_{final}



Acceleration aligned convergence plot

Algorithms when $\phi \neq 0$ (Shown as ~ 0.5)

Resulting data compared to true solution at t_{final}

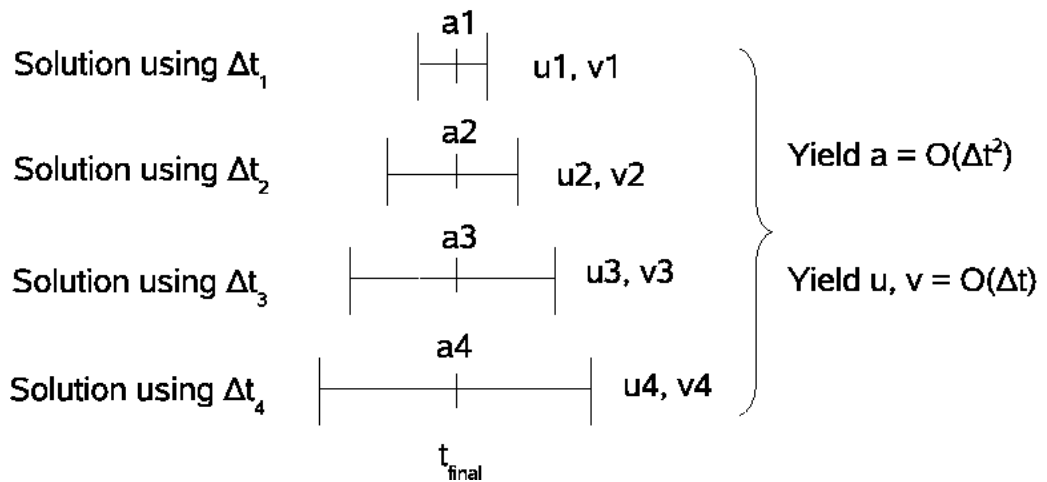


Figure 3.5: Example of the final time steps used in creating a convergence plot when $\phi \neq 0$

numerical simulations are compared with the acceleration of the exact solution at precisely the same time, very close attention must be paid to the selection of the time step size for each simulation used in generating the plot. As an example, assume that the convergence plot will have 4 data points meaning that one has 4 numerical solutions which are being compared to an exact solution. Instead of defining 4 different time step sizes, let us define the total number of steps for each of the 4 solutions that we will use. Then calculate the corresponding time step size as follows:

$$\Delta t = \frac{endtime}{steps + \phi} \quad (3.9)$$

It is shown below in the different numerical examples that using this procedure in fact results in properly obtaining second order accurate accelerations, thereby validating the assumption of the offset in the acceleration time level. The first example problem is a simple linear dynamic single degree of freedom problem for which we have an exact solution. The next example is a multi-DOF dynamic problem with nonlinear strain definition (Green strain). For illustration, a general purpose dynamic research code was developed to simulate any combination of springs and masses. Having tested the current assertions on codes ranging from single DOF to multi-DOF and both linear and non-linear dynamic problems, we have great confidence that the current work will advance the field of general dissipative (and non-dissipative as well) LMS class of algorithms. This also will create new, and never before addressed areas of research in the field.

3.6 Single degree of freedom (SDOF) example

Consider again the single degree of freedom Duffing equation defined by Eq. 2.37. For this system it was shown in Fig. 3.1c for the velocity based scheme (V0(1,1,0)) the acceleration appears to be only first order accurate. Using the concepts described

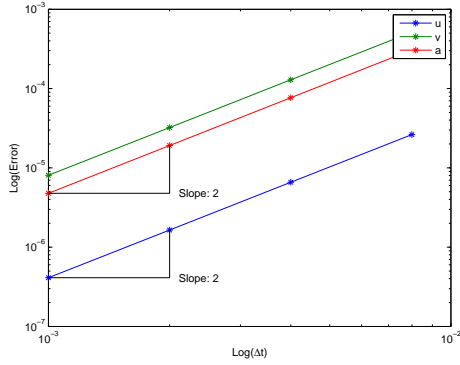


Figure 3.6: Velocity based scheme - acceleration aligned convergence plot

above and Eq. 3.2 for the V0 family we are now able to understand why: the ϕ value for the velocity based scheme is 0.5. This means that the accelerations resulting from the algorithm do not align with the displacement and velocity, and care must be taken to properly represent the converge rate. Fig. 3.6 shows the resulting convergence plot taking into account this shift; notice that now all three primary variables show the expected order of accuracy.

To demonstrate that the above procedure indeed yields expected second order time accurate accelerations over the entire range of ϕ , four common algorithms which include numerical dissipation in the U0 family of algorithms were selected as well as their four V0 counterparts. These chosen algorithms are summarized in the table below.

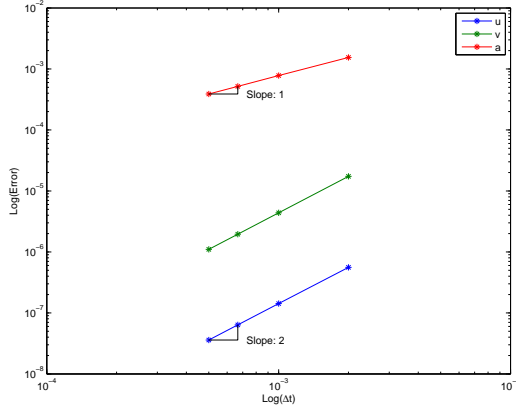
Algorithm (Family($\rho_{1\infty}, \rho_{2\infty}, \rho_{3\infty}$))	Common name	Accel. time level ($t_{n-\phi+1}$)
U0(0,0,0)	WBZ	0.0
U0(0.25,1,0.25)	U0V0-optimal	0.7
U0(0.5,0.5,0.5)	Generalized- α	0.6667
U0(0.8,0.8,0.125)	HHT- α	0.8889
V0(0,0,0)		0.5
V0(0.25,1,0.25)	U0V0-optimal	0.7
V0(0.5,0.5,0.5)		0.8333
V0(0.8,0.8,0.125)		0.6111

For each of the eight algorithms described above, convergence plots are shown using 4 numerical solutions. Two convergence plots were generated for each algorithm: one without accounting for acceleration offset and one in which accelerations were aligned via Eq. 3.9. These convergence plots are shown for both the symplectic-momentum based method and the energy-momentum based method, both described in the previous chapter, as proof that this method works regardless of the particular formulation being used.

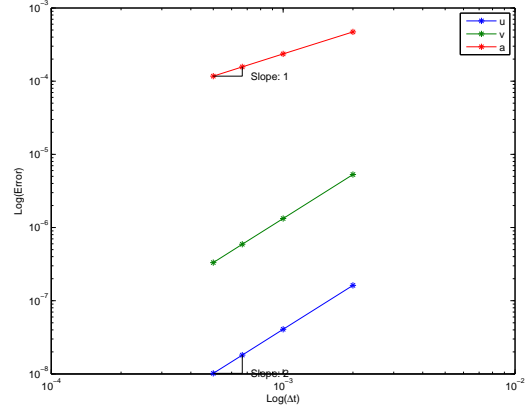
It is consistently shown in Figs. 3.7-3.14 that in the traditional convergence plot the slope of u and v is 2.0 while the slope of a is 1.0. Once aligned correctly, the proper convergence plots now indeed show the acceleration has a precise slope of 2.0.

3.7 Example: Non-linear Tetrahedral Spring-Mass System

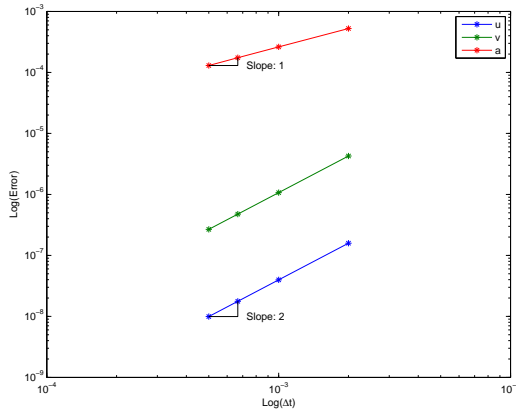
Consider another example, which is a tetrahedral spring-mass system depicted in Fig. 3.15a constructed of six springs and four masses. Each spring has initial length of 1m, stiffness of 10^3N/m , and mass 1kg. The spring is characterized by geometrically nonlinear strain (Green strain). The initial position of the four nodes are $[0.5, 3^{0.5}/2, 0, 0, 0, 0, 1, 0, 0, 0.5, 1/(2 * 3^{0.5}), (2/3)^{0.5}]^T\text{m}$ with given initial displacement



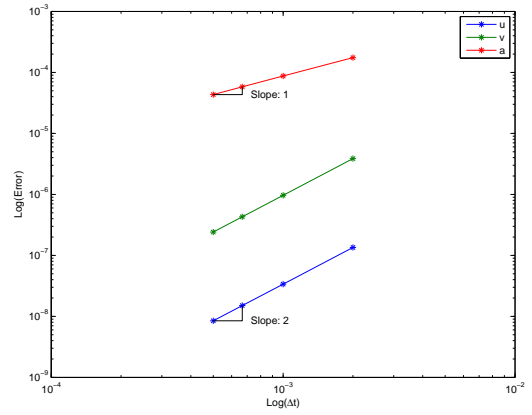
(a) $U_0(0,0,0)$



(b) $U_0(0.25,1,0.25)$



(c) $U_0(0.5,0.5,0.5)$



(d) $U_0(0.8,0.8,0.125)$

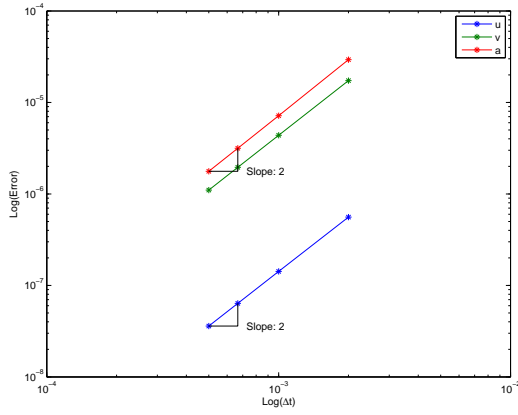
Figure 3.7: Symplectic-momentum based Duffing equation U_0 family traditional convergence plots

and velocity seen in the following equations. The dynamic response of the system over a 5 second time span with a time step size $\Delta t = 0.1s$ using the so-called energy-momentum method can be seen in Fig. 3.15b.

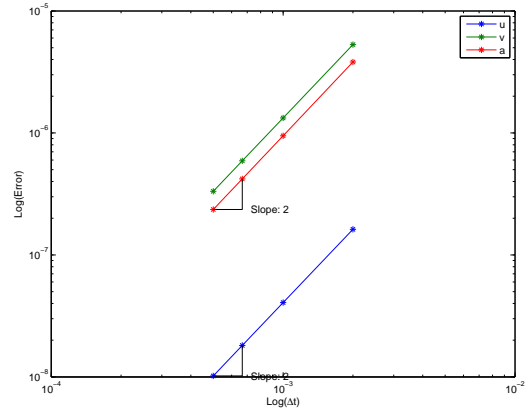
$$u_0 = [0, 0, 6, 0, 0, 0, 0, 0, 0, 0, 1, 3, 2]^T m \quad (3.10)$$

$$v_0 = [0, 0, 6, 0, 0, 0, 0, 0, 0, 0, 1, 3, 2]^T m/s \quad (3.11)$$

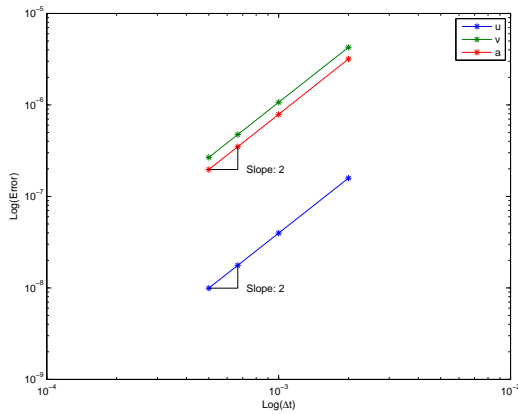
The so called symplectic-momentum based representation and the energy-momentum



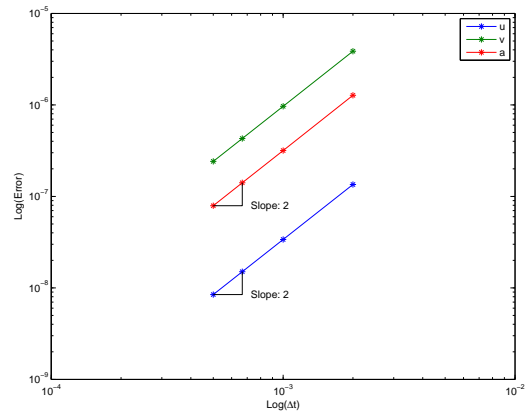
(a) $U0(0,0,0)$



(b) $U0(0.25,1,0.25)$



(c) $U0(0.5,0.5,0.5)$

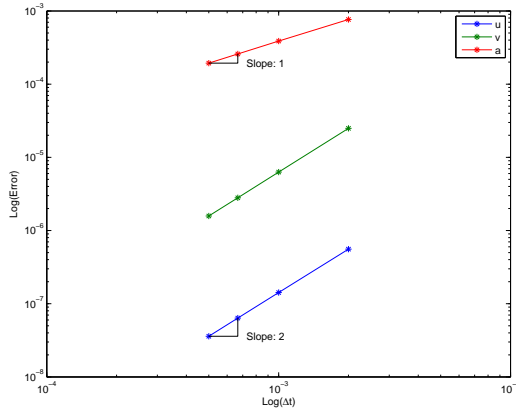


(d) $U0(0.8,0.8,0.125)$

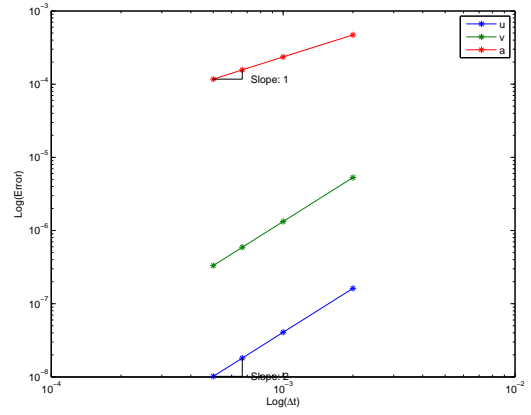
Figure 3.8: Symplectic-momentum based Duffing equation $U0$ family acceleration aligned convergence plots

based representation discussed in the previous chapter are used here as demonstration. The classical approach discussed previously is not repeated here due to its lack of conservation of key properties which make it an undesirable approach for real world problems. The following figures demonstrate that correctly accounting for the acceleration time level results in proper second order accurate accelerations regardless of the strain formulation being used for various algorithms with controllable dissipation in both the $U0$ and $V0$ family of algorithms. The misaligned (traditional) convergence plots are shown first followed by the correctly aligned plots.

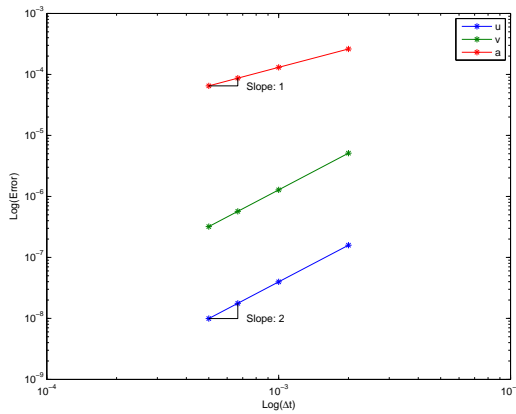
In the appendix additional numerical examples can be found. These examples in-



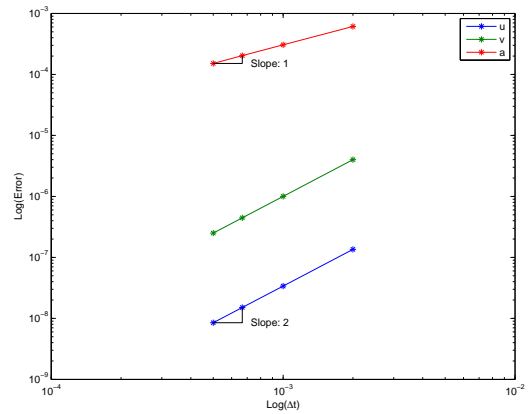
(a) $V0(0,0,0)$



(b) $V0(0.25,1,0.25)$



(c) $V0(0.5,0.5,0.5)$



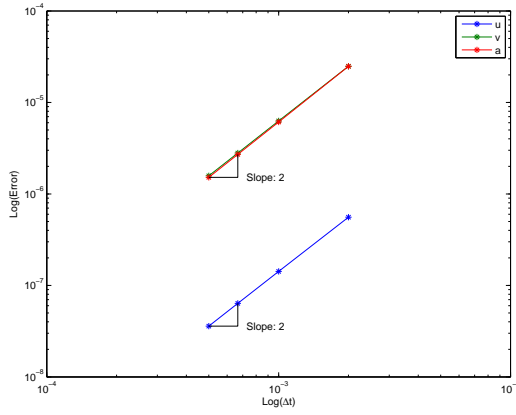
(d) $V0(0.8,0.8,0.125)$

Figure 3.9: Symplectic-momentum based Duffing equation $V0$ family traditional convergence plots

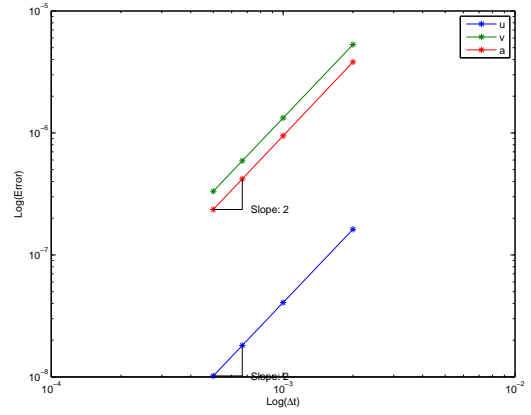
clude a four mass-spring system with a given only initial displacement and a simple pendulum given only initial velocity. With these additional examples it can be seen that regardless of initial conditions, algorithm, and geometric/material properties the concept of a correctly aligned convergence plot holds true.

3.8 Conclusion

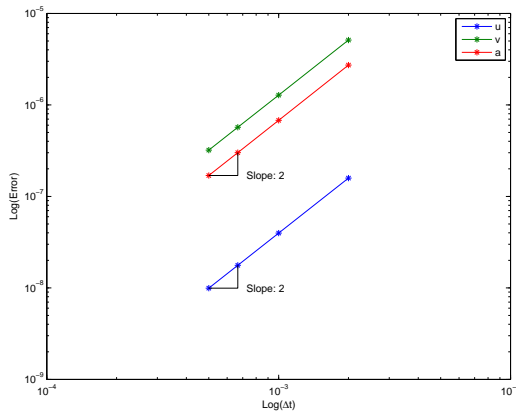
This chapter described the accurate and precise evaluation of acceleration computations useful for structural dynamic simulations. The focus was on the class of LMS methods which are predominantly used in most commercial and research software.



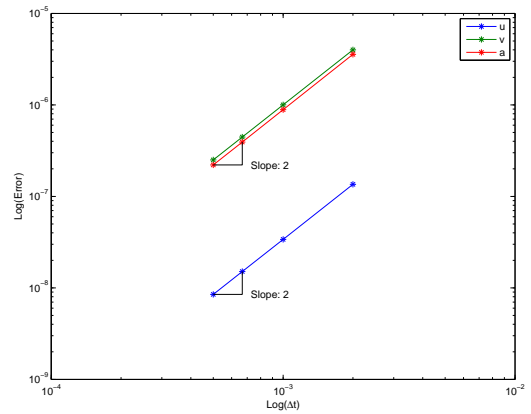
(a) $V0(0,0,0)$



(b) $V0(0.25,1,0.25)$



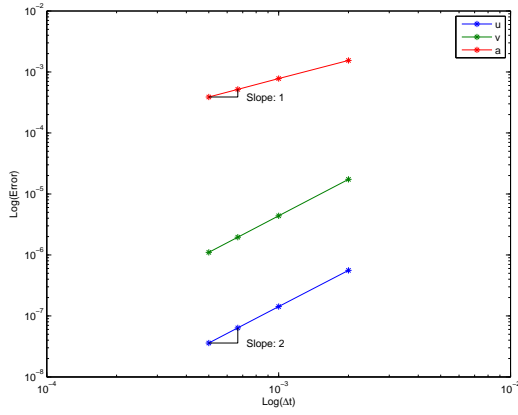
(c) $V0(0.5,0.5,0.5)$



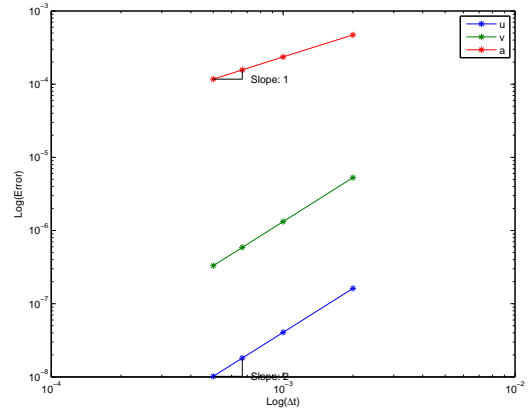
(d) $V0(0.8,0.8,0.125)$

Figure 3.10: Symplectic-momentum based Duffing equation $V0$ family acceleration aligned convergence plots

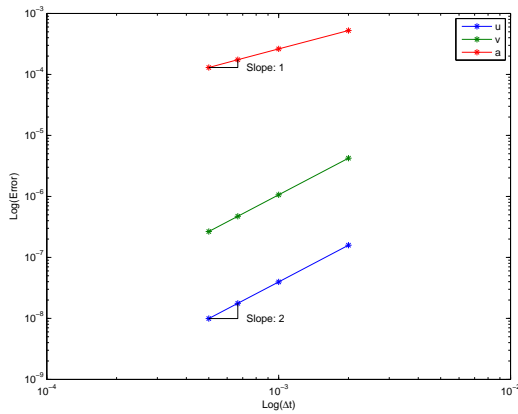
They include non-dissipative and dissipative methods. The fundamental problem is that very little is understood to-date regarding the time levels at which these computations need to take place precisely to achieve the desired order of accuracy which is targeted at second order for the class of LMS methods. The literature to-date is filled with misconceptions about how to evaluate these acceleration computations accurately and precisely. These LMS methods have been developed under the umbrella of GSSSS algorithms which encompass all the developments to-date since the past fifty years and contain both non-dissipative and dissipative methods. The fundamental problem is that with the exception of the standard Newmark and midpoint rule, the acceleration computations of all the others are not trivial and the resulting



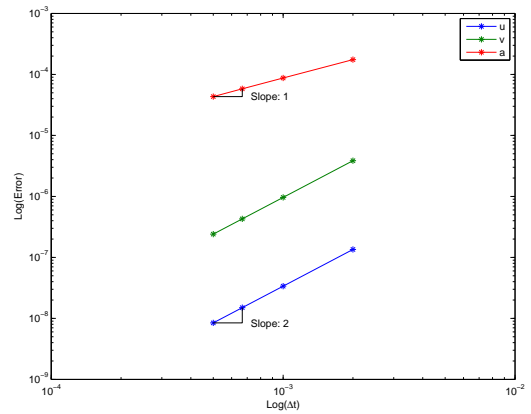
(a) $U_0(0,0,0)$



(b) $U_0(0.25,1,0.25)$



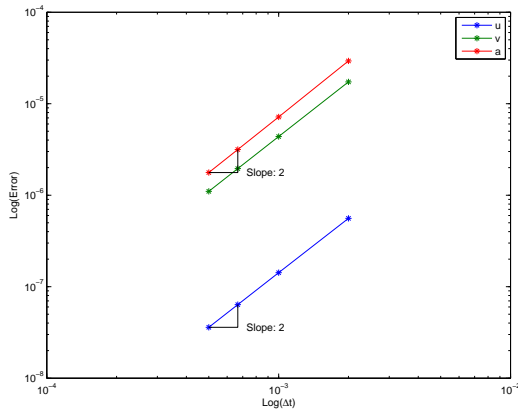
(c) $U_0(0.5,0.5,0.5)$



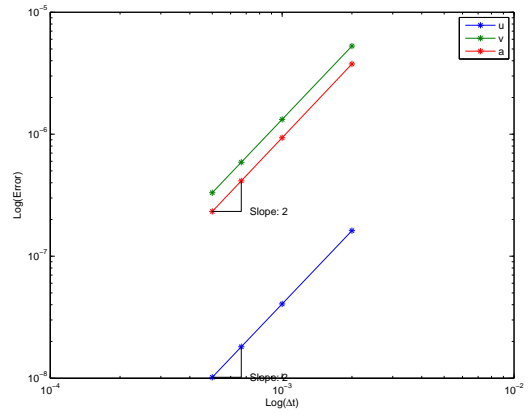
(d) $U_0(0.8,0.8,0.125)$

Figure 3.11: Energy-momentum based Duffing equation U_0 family traditional convergence plots

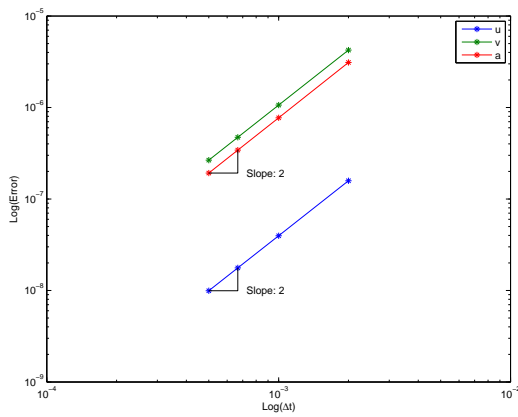
computations to achieve second order accuracy are not strictly at time level $n+1$. In this regard, we described the subtle issues and some noteworthy perspectives to accurately obtain the acceleration computations for general algorithms and the precise time levels that underlie their basic developments. We have provided a closure and an in-depth understanding to estimate accurately the accelerations and to ensure that they are second order accurate. Several simple numerical examples demonstrated the basic ideas.



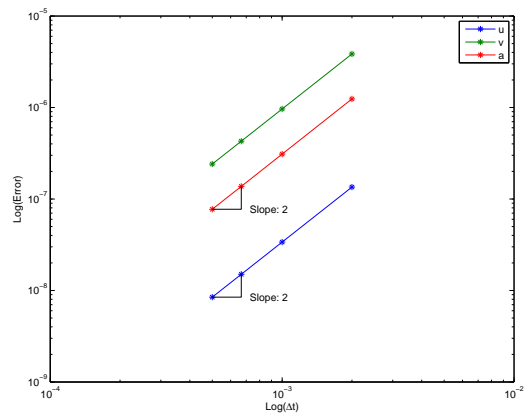
(a) $U_0(0,0,0)$



(b) $U_0(0.25,1,0.25)$

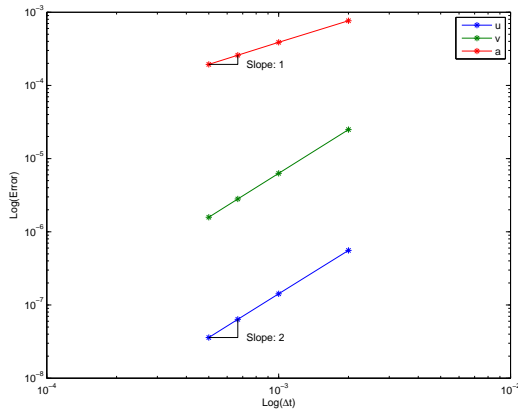


(c) $U_0(0.5,0.5,0.5)$

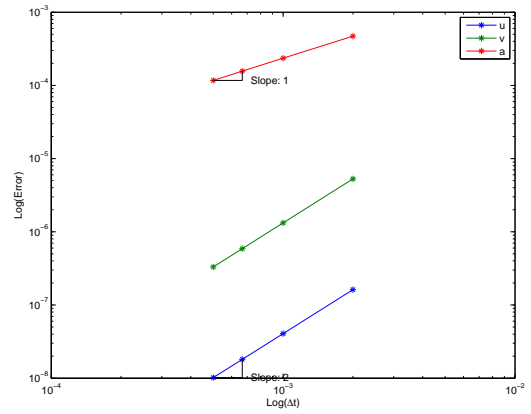


(d) $U_0(0.8,0.8,0.125)$

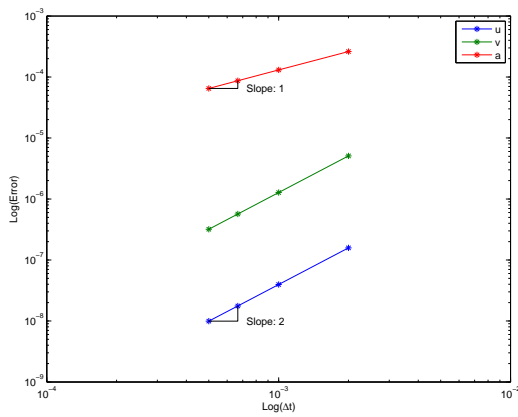
Figure 3.12: Energy-momentum based Duffing equation U_0 family acceleration aligned convergence plots



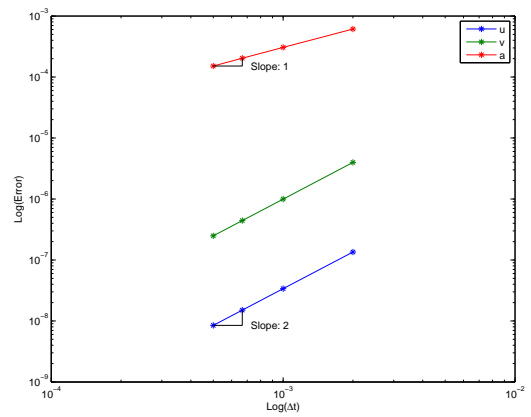
(a) $V0(0,0,0)$



(b) $V0(0.25,1,0.25)$

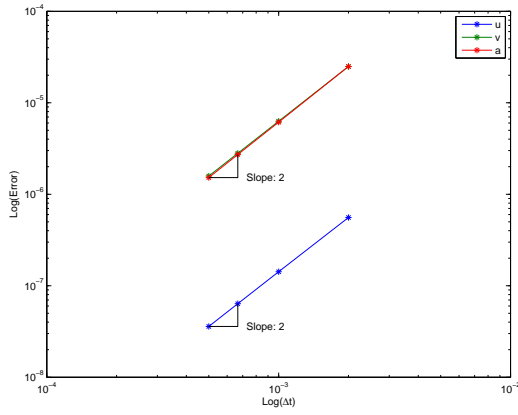


(c) $V0(0.5,0.5,0.5)$

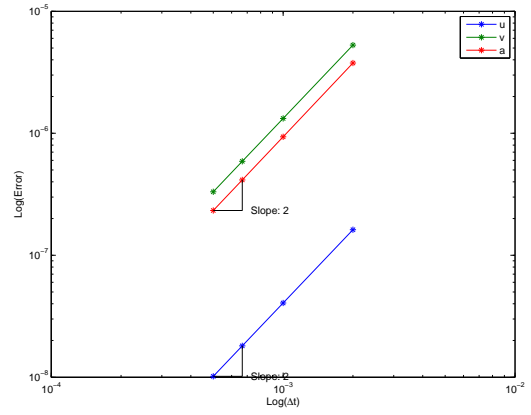


(d) $V0(0.8,0.8,0.125)$

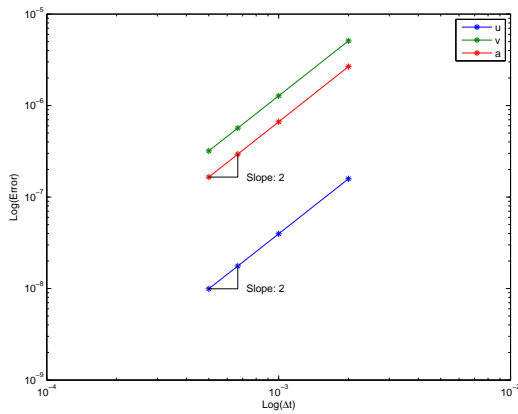
Figure 3.13: Energy-momentum based Duffing equation $V0$ family traditional convergence plots



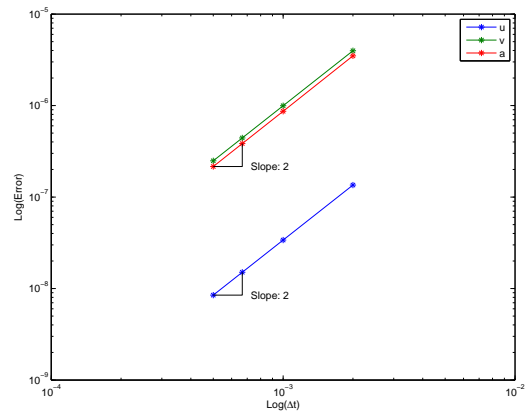
(a) $V0(0,0,0)$



(b) $V0(0.25,1,0.25)$

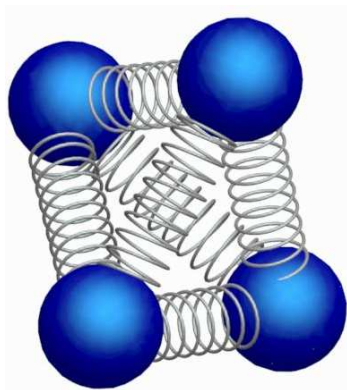


(c) $V0(0.5,0.5,0.5)$

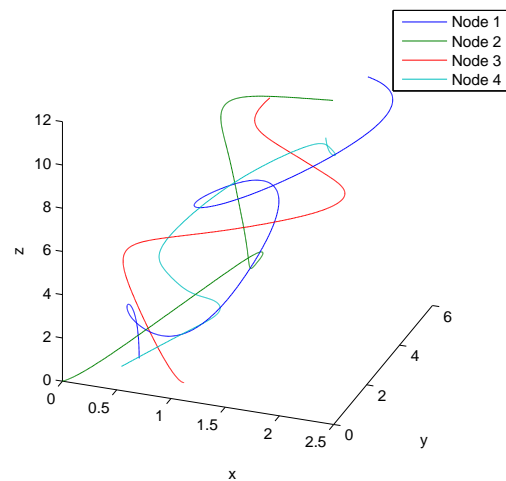


(d) $V0(0.8,0.8,0.125)$

Figure 3.14: Energy-momentum based Duffing equation $V0$ family acceleration aligned convergence plots

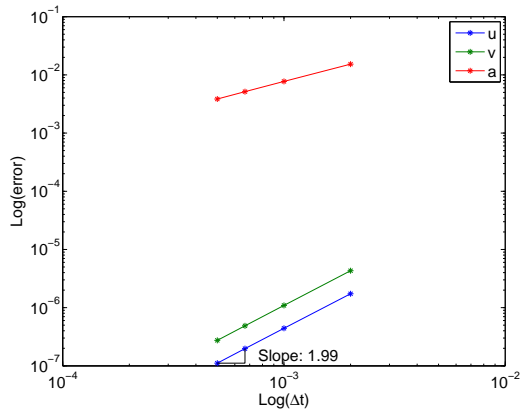


(a) Geometry

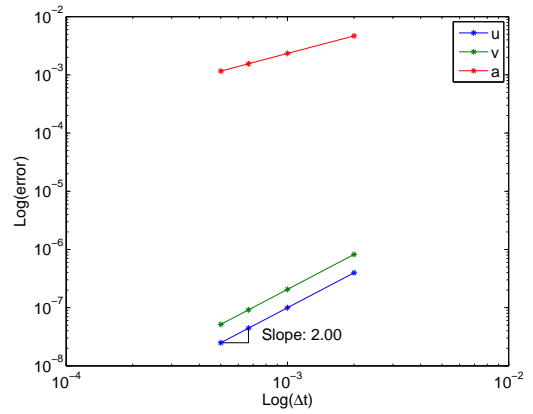


(b) Dynamic Response

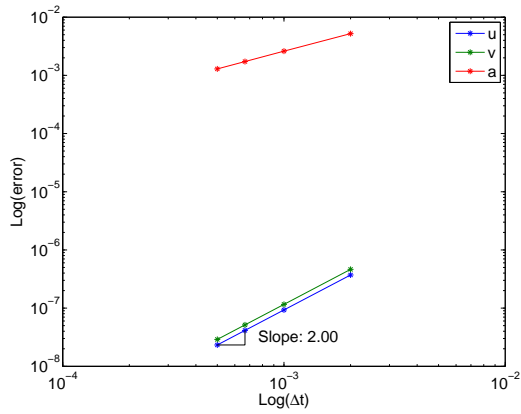
Figure 3.15: Tetrahedral spring-mass



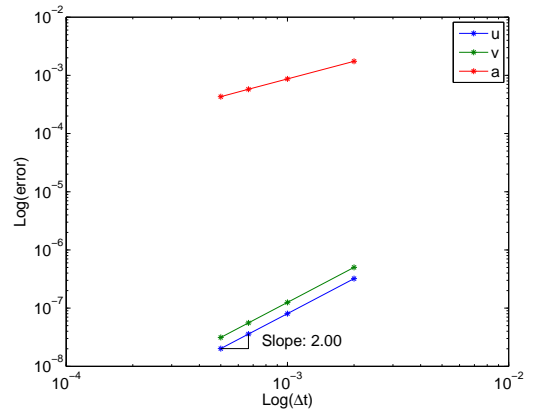
(a) $U0(0,0,0)$



(b) $U0(0.25,1,0.25)$

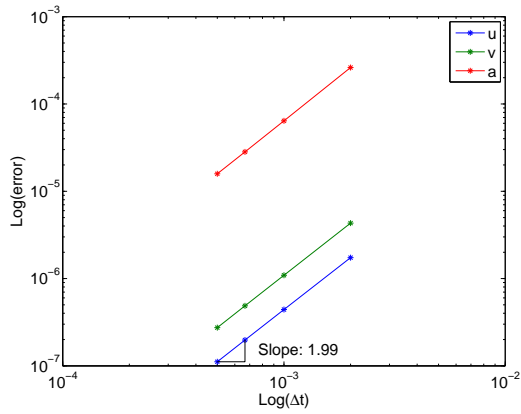


(c) $U0(0.5,0.5,0.5)$

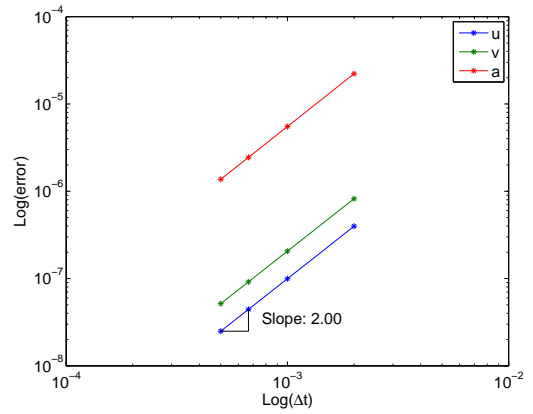


(d) $U0(0.8,0.8,0.125)$

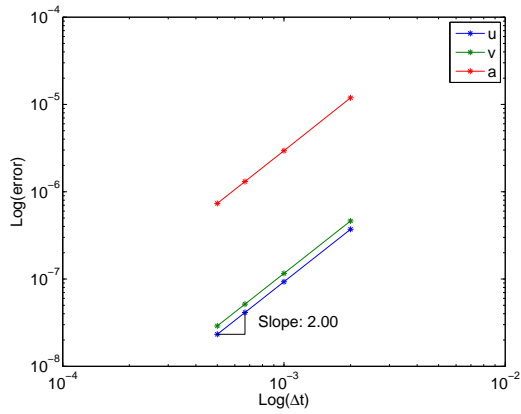
Figure 3.16: Energy-momentum based tetrahedral spring-mass $U0$ family traditional convergence plots



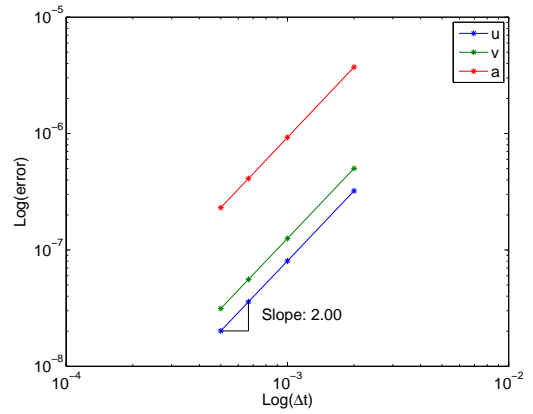
(a) $U0(0,0,0)$



(b) $U0(0.25,1,0.25)$

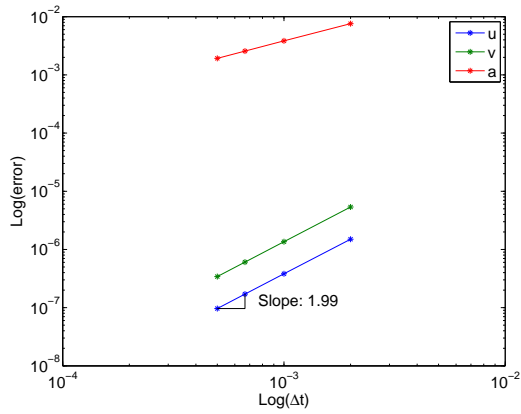


(c) $U0(0.5,0.5,0.5)$

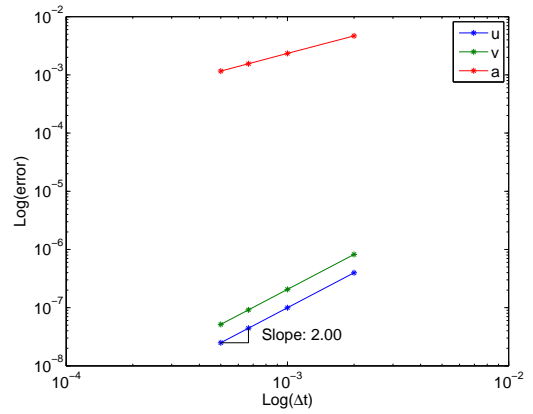


(d) $U0(0.8,0.8,0.125)$

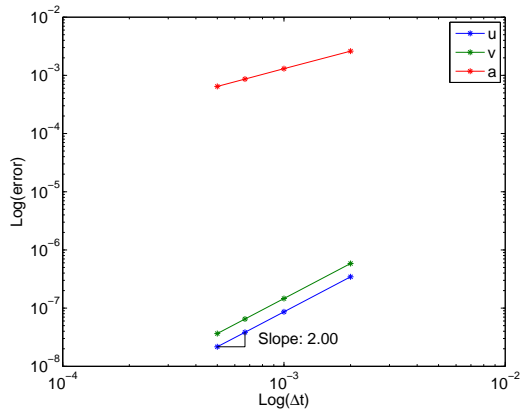
Figure 3.17: Energy-momentum based tetrahedral spring-mass $U0$ family acceleration aligned convergence plots



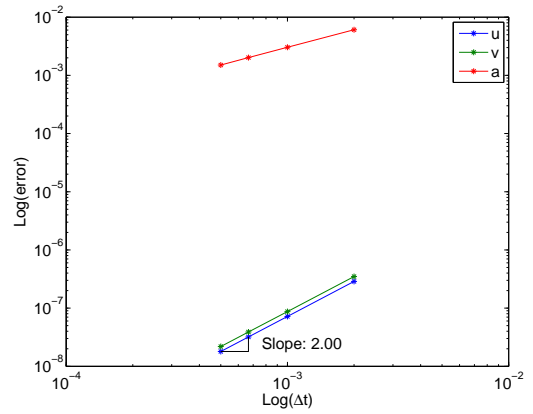
(a) $V0(0,0,0)$



(b) $V0(0.25,1,0.25)$

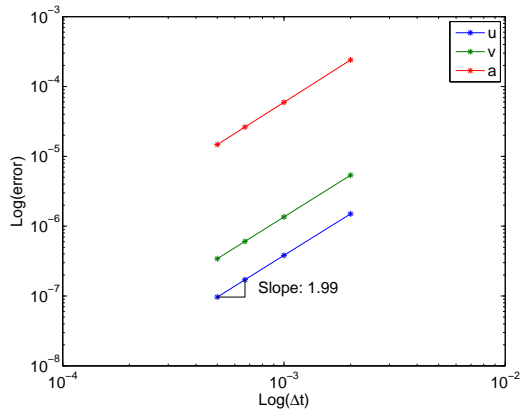


(c) $V0(0.5,0.5,0.5)$

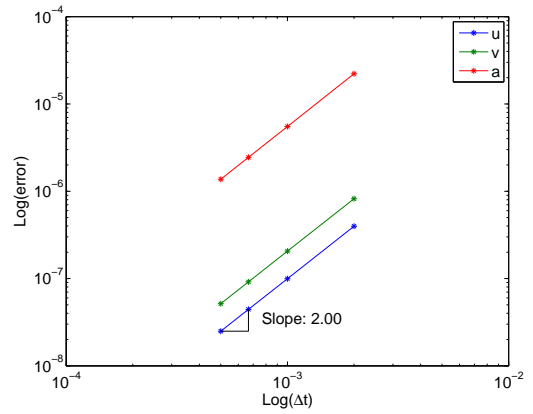


(d) $V0(0.8,0.8,0.125)$

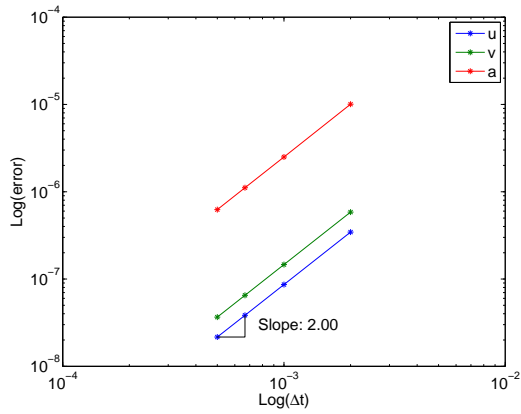
Figure 3.18: Energy-momentum based tetrahedral spring-mass $V0$ family traditional convergence plots



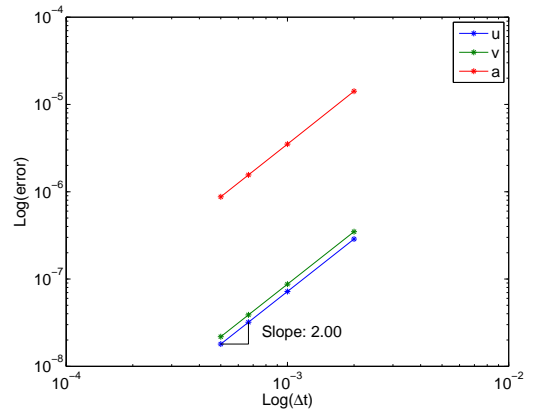
(a) $V0(0,0,0)$



(b) $V0(0.25,1,0.25)$

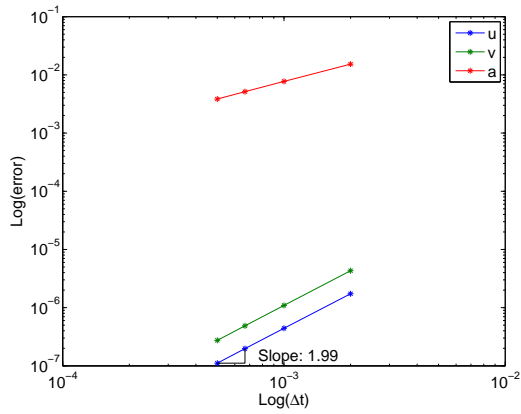


(c) $V0(0.5,0.5,0.5)$

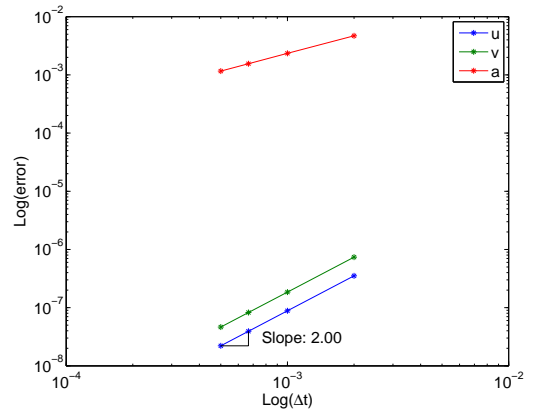


(d) $V0(0.8,0.8,0.125)$

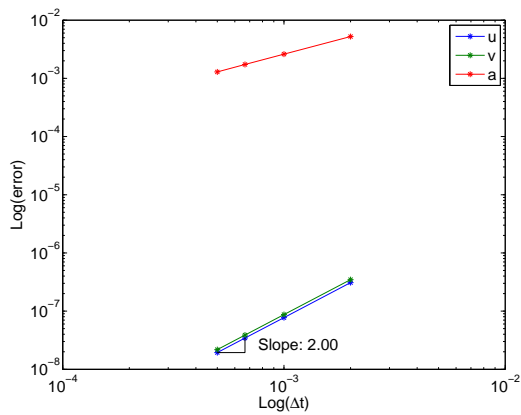
Figure 3.19: Energy-momentum based tetrahedral spring-mass $V0$ family acceleration aligned convergence plots



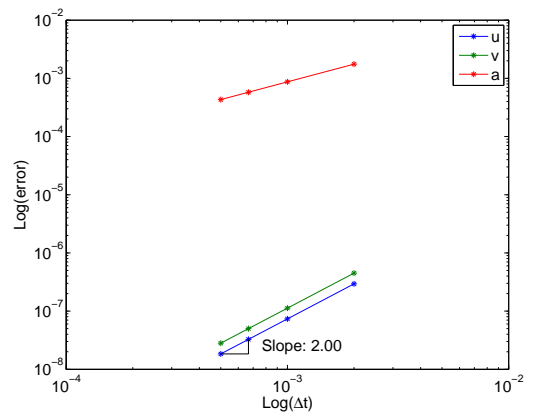
(a) $U_0(0,0,0)$



(b) $U_0(0.25,1,0.25)$

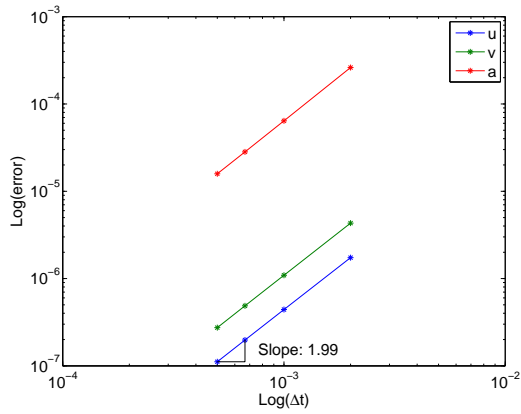


(c) $U_0(0.5,0.5,0.5)$

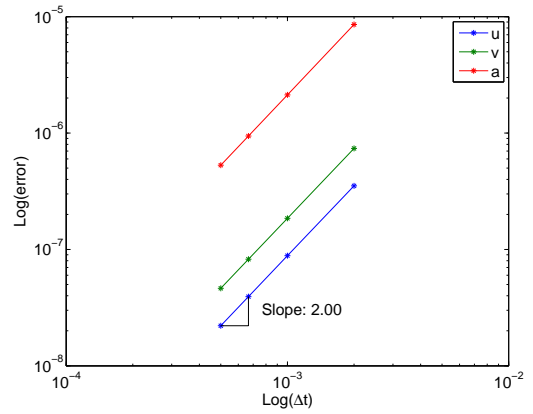


(d) $U_0(0.8,0.8,0.125)$

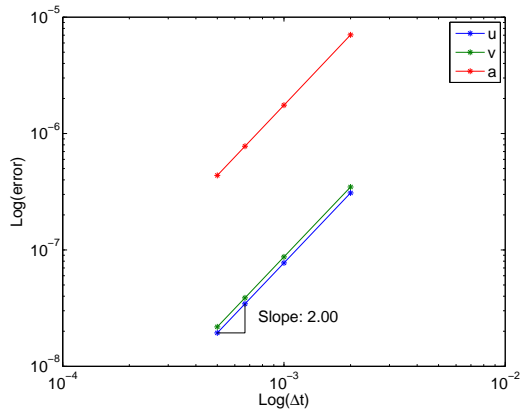
Figure 3.20: Symplectic-momentum based tetrahedral spring-mass U_0 family traditional convergence plots



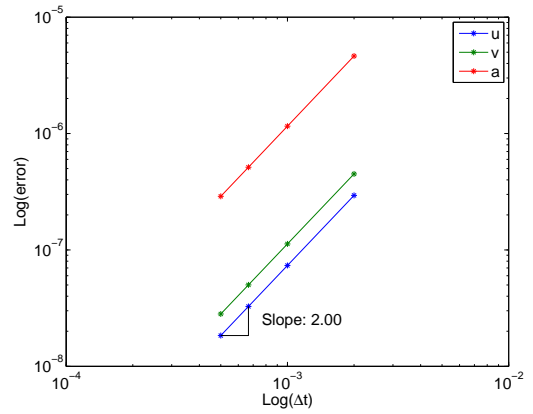
(a) $U0(0,0,0)$



(b) $U0(0.25,1,0.25)$

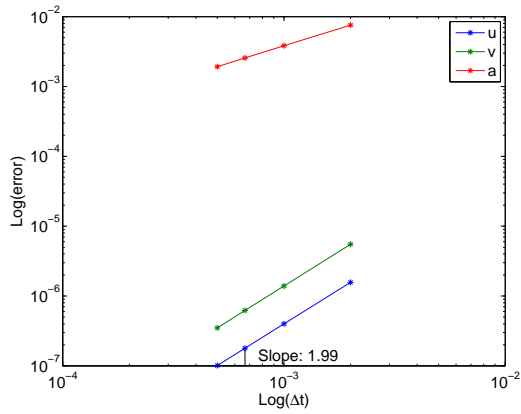


(c) $U0(0.5,0.5,0.5)$

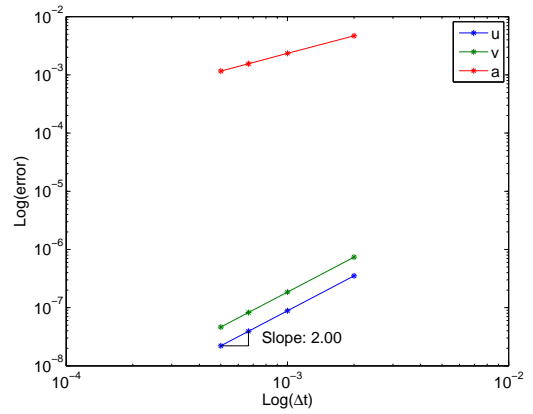


(d) $U0(0.8,0.8,0.125)$

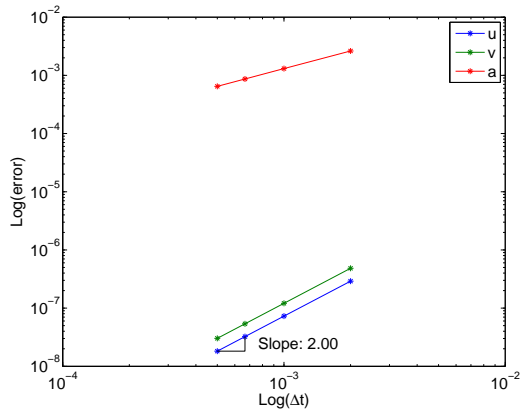
Figure 3.21: Symplectic-momentum based tetrahedral spring-mass $U0$ family acceleration aligned convergence plots



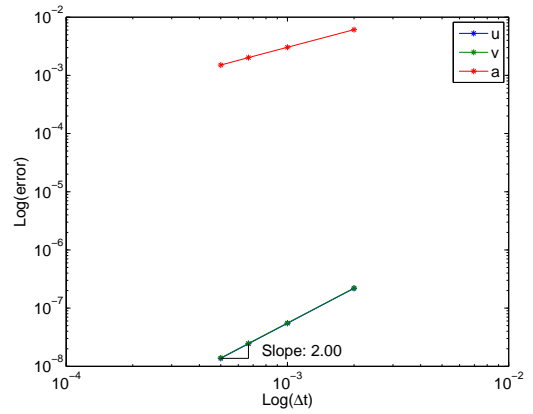
(a) $V0(0,0,0)$



(b) $V0(0.25,1,0.25)$

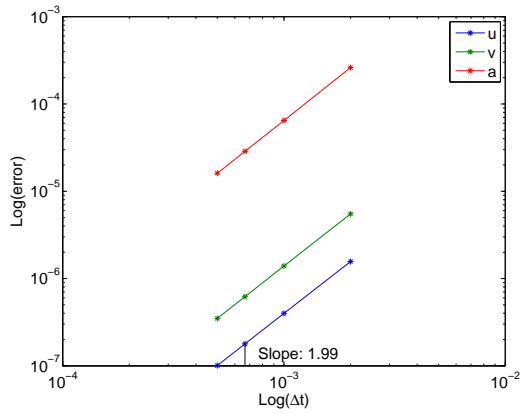


(c) $V0(0.5,0.5,0.5)$

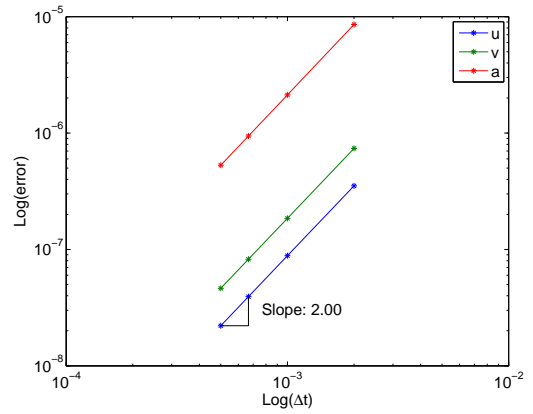


(d) $V0(0.8,0.8,0.125)$

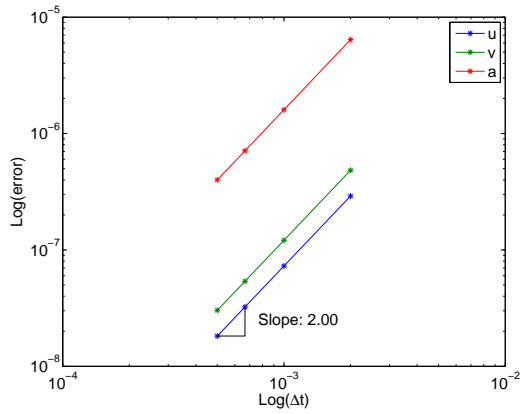
Figure 3.22: Symplectic-momentum based tetrahedral spring-mass $V0$ family traditional convergence plots



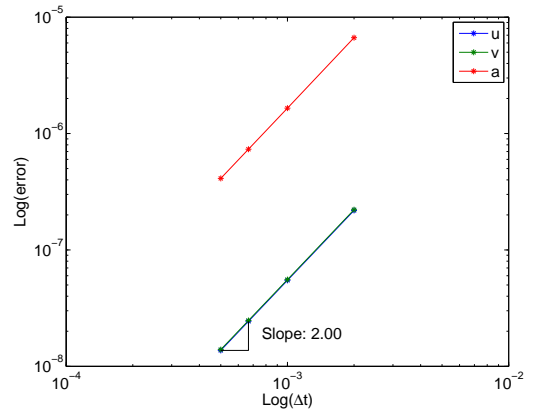
(a) $V0(0,0,0)$



(b) $V0(0.25,1,0.25)$



(c) $V0(0.5,0.5,0.5)$



(d) $V0(0.8,0.8,0.125)$

Figure 3.23: Symplectic-momentum based tetrahedral spring-mass $V0$ family acceleration aligned convergence plots

Chapter 4

Observations on the long term stability of algorithms for numerically stiff non-linear problems

4.1 Introduction

In this chapter an investigation on the stability of algorithms intended for long-duration simulation of numerically stiff nonlinear problems is presented. Typical engineering structures often involve structural components that are partly rigid and partly flexible. The major bottleneck is that for the solution of this class of problems, no single time stepping method to-date has been shown to complete the solution for long term desired time duration within a reasonable computational time whilst preserving the physics of the problem. In particular, we are interested in the class of problems that are truly conservative systems. The difficulty is that when a spatial

finite element mesh is used to discretize the structure, there exist both low and high frequency components participating and dominating the problem. Mostly the non-linear dynamic response for inertial structures is low frequency dominated; however, high frequencies also participate which poses the various complexities. Theoretically, if damping is to be ignored, for conservative systems which are of interest here, then physical quantities such as energy, linear and angular momentum need to be conserved. This is the true physical situation. Unfortunately, because of the participation of a range of low and high frequency modes induced from spatial discretization due to the finite element spatial mesh, this causes numerical problems in integrating the equations of motion. This particularly becomes a nuisance for rigid-flexible structural systems. That is, either the analysis fails after a certain time duration and/or there is significant loss of the true physics of the problem at hand. It should be noted that the term "stability" in this chapter is not intended to mean traditional analytic algorithmic stability analysis. Such analysis is impossible for the nonlinear problems presented here. With no mathematical way to define stability for the given problem, instead stability is taken to mean stability of the solution in terms of measurable quantities such as energy.

4.2 History

Stiff ordinary differential equations have long been studied by mathematicians, but largely, methods used for solving them are impractical in several situations dealing with practical finite element analysis encountered in computational structural dynamics. ODEs are considered to be "stiff" when there is a large disparity in the range of the eigenvalues. That is to say, if the solution being sought has a significantly slower response than nearby solutions contained within the same problem. Clearly in the area of structural dynamics this situation frequently arises. Take for example the dynamic response of a vehicle rolling on rough terrain. The shocks imparted on

the vehicle may attempt to keep the high frequencies which occur at the wheels away from passengers, but in the computational model they are unavoidable. Basically there exist structural components that are both rigid and flexible within the same structural system being analyzed.

The ability to solve such problems certainly exists, but is computationally intensive, it is the ability to solve them in a computationally tractable practical way which does not exist to-date. The situation is more complicated when factors such as conservation of energy, linear and angular momentum need to be preserved in conservative dynamic systems. Given unlimited computational power these problems can be easily solved by an explicit method with a sufficiently small time step size. This defeats the purpose, since in practice, for models with thousands or even millions of degrees of freedom, such a time step is prohibitively slow and long term solutions are not reasonable. It is for these situations that implicit algorithms are often preferable for the class of inertial dynamic problems in order to be able to solve stiff problems in a reasonable amount of time. This is the focus of the present work. The dilemma is that unconditionally stable implicit methods are mostly preferred for the class of inertial problems that are of interest here in contrast to the conditionally stable explicit methods; however, on one hand constant time step based non-dissipative methods which can ensure conservation of properties often fail to complete the analysis due to convergence issues for stiff problems, while dissipative methods may in certain instances complete the analysis but with significant loss in conserving the true physics of the situation. Furthermore, the latter needs a tuning parameter to be selected which is arbitrary and there is no clear way of ensuring the quality of the solution and/or completion of the solution. More technically, unconditional stability proofs are difficult to establish for dissipative methods, while a certain degree of proof in the sense of energy stability can be established for selected conserving algorithms which do not inherit dissipation.

In the past 40 years or so there have been a multitude of algorithms designed which fall under the umbrella of implicit methods which are typically used to solve all types of stiff problems in the area of computational structural dynamics. These include non-dissipative methods such as the Newmark method, mid-point rule etc., and dissipative methods such as the WBZ, Generalized- α , and HHT- α , optimal U0-V0 family of algorithms with the so-called controllable dissipation. All of these have been unified via a novel procedure under the framework of GSSSS algorithms by Zhou and Tamma [4, 5] which encompass most of the developments to-date over the past fifty years with new avenues that provide features for optimal design of algorithms based on the overshoot behavior. While non-dissipative conserving methods are most preferred since they tend to preserve the physics of the problem (that is, they enable conservation of physical quantities such as energy, linear and angular momentum), the difficulty is that they eventually cause failure in the nonlinear Newton iterations and force the analysis to stop when long term simulations are desired. On the other hand, controllable dissipation (a tuning parameter often termed as the value of the spectral radius at infinity is an integral part of the algorithm which is related to the so-called spurious root), is often employed to remove or filter the high frequencies which cause problems due to the spatially unresolved high frequencies generated from a finite element mesh which participate in the solution. Artificially removing these high frequencies (which are very much a part of the physical system) destroys the physics of the problem (conservation of energy and angular momentum are not guaranteed), but this is the trade off one has to make to be able to complete the analysis. With no tangible metrics that exist to-date or descriptions to qualitatively provide a measure to accurately describe the stiffness of a problem over the desired analysis duration, the current state-of-the-art is to select an amount of dissipation which hopefully maintains the maximum amount of the physics of the problem while still allowing completion of the analysis. It is here that existing approaches currently cannot provide a satisfactory

answer and we are forced to step into the realm of "trial and error" which is often a guessing game to ensure completion of the analysis.

A brief highlight of the role of dissipation and its relation to the spectral characteristics follow next for computational algorithms. To see how numerical dissipation affects the frequencies involved in a problem, a spectral radius plot for the algorithm is readily constructed which describes the algorithmic behavior over a wide range of frequencies. The limitation is that such characterizations hold for mostly linear dynamic problems and there is no guarantee that the basic features are applicable to nonlinear dynamic situations. For linear dynamics, based on spectral analysis, a spectral radius plot shows the spectral radius of the algorithm versus $\Delta t/T$. See Figure 4.1 for example. For a conservative systems, a spectral radius of 1.0 would mean that there is no dissipation (alternately, it implies energy conservation) and all possible frequencies involved in the system play an active role. As the spectral radius goes to zero there is increasingly more and more dissipation. For example, given a very stiff system (system with both very low and very high frequencies) it can be seen in the figure for the HHT algorithm shown, frequencies of the range $\Delta T/T$ 0.5 begin to be damped out of the system. As such, the problem can be made less stiff by carefully selecting the appropriate time step size, thereby removing the participation of the high frequencies to an extent without destroying the low frequencies and possibly allowing completion of analysis.

4.3 Motivation

The motivation of the current study is to investigate which algorithms are viable for use in the long term simulation of numerically stiff systems. One technique which may aide in ensuring a physically stable solution (in terms of energy) is presented. This is addressed and explored in this exposition under the framework of LMS methods

since they are the most commonly employed in commercial and research software for finite element analysis over the decades. The fundamental issue at hand is does one select an appropriate non-dissipative algorithm and face the consequence of failure after a certain time duration, or does one resort to simply choosing a single time stepping dissipative algorithm and being forced to live with the attached loss of realistic physics with no clear guarantee that even this would resolve the problem at hand. The basic strategy advocated here for rigid-flexible dynamic systems is to rely on the conservation properties of the energy and momentum conserving algorithm (EMM), while resorting, if (when) necessary to a smaller time step when the non-linear iterations cease to converge. It is seen that this is the only technique which can assure that long term simulations remain realistic in terms of energy. As illustration, we focus attention on the family of GSSSS algorithms for which we clearly understand how and at precisely correct time levels to accurately integrate the equations of motion for nonlinear dynamics with/without controllable dissipation and EMM.

4.4 Numerical example: Rigid-flexible beam system

Recently in work by Armero [3] a structural system containing both rigid and flexible beam/spring elements has been shown to cause difficulties for numerical simulations. The general setup of the problem can be seen in Figure 4.2. The masses and stiffness parameters are as follows. Though the problem is described by Armero as having rigid elements they are not truly analytic rigid bodies. Instead, to avoid converting the problem to the realm of multibody dynamics which would require the addition of constraint equations, the "rigid" elements are simply spring elements with a very high stiffness.

$$m_b = 0.2, m_a = 2.0, \kappa = 500, C_b = 10$$

For the current study the rigid bars were taken to have a stiffness of $1e6$, or $2000x$ the

stiffness of the "soft" bars. A non-linear iteration tolerance of $1e-6$ was required on the norm of the residual. The critical timestep for explicit schemes for an analogous linear system would be $2.82e-3$ seconds. As implicit schemes are only computationally useful when a timestep size larger the critical explicit timestep is used, the timestep size chosen for demonstration was 0.05 seconds (between 10 and 20x the critical timestep for explicit schemes). The algorithms of utmost interest here are those which can be expected to provide physically meaningful results over long duration simulation without the "black magic" required to a priori select appropriate tuning parameters. As such, EMM and the (symplectic) midpoint rule are the main focus, though a scheme with controllable numerical dissipation is shown as well for demonstration. The EMM implementation is exactly that described by Armero in [3].

4.5 Results

To get a solution with which to compare the results of the implicit schemes, the explicit central difference algorithm was used with a timestep size of $1e-6$ seconds. The results of this very small timestep explicit scheme are shown in Fig 4.3. Armero noticed that there was often a sudden increase in the potential energy stored in the axial elements prior to jumps in energy or failure of nonlinear convergence. To further his concept the percentage of the total energy is shown separately for the bending, axial, and rigid elements. As intuition would suggest the potential energy stored in the rigid bars is very low. That is, the rigid bars remain essentially undeformed throughout the solution.

To demonstrate the solution for a algorithms with controllable numerical dissipation the U0V0 algorithm with $\rho_\infty = 0.5$ and $\rho_\infty = 0.9$ are shown in Fig. 4.4 and 4.5. As expected the total energy is the system decays in time. Though with proper tuning of ρ_∞ this loss of energy may be minimized to maintain as much of the true physics

of the problem as possible.

In agreement with the results presented by Armero, at the selected timestep EMM fails due to nonconvergence of the nonlinear Newton iteration (at the very first timestep) and the midpoint rule produces a meaningless result in which energy grows well beyond its physical bound, as seen in Fig. 4.6.

It is the addition of an adaptive timestep size for EMM that produces intriguing results. When the number of non-linear iterations became larger than 1,000, the current step was halted and restarted with a timestep 10 percent of the original. Upon completion of any step the timestep size was set back to the original value of 0.05. The resulting simulation was able to complete the 20 second duration, maintain exact energy conservation, and prevent potential energy from erroneously growing in the rigid bars, Fig. 4.7.

Interestingly, the timestep size reduction was only needed for the first 14 steps of the simulation, Fig. 4.8. That is, for the first 14 steps of the solution the number of non-linear iterations reached 1,000 and the step was restarted with a timestep size of 0.005, which produced a converged solution. After the first 14 steps (0.07 seconds) the Newton iterations were able to converge readily with the original timestep size of 0.05. Thus, this primitive method of adaptive timestepping was able to produce a good solution up to the 20 second duration in 413 steps, only 13 more steps than the midpoint rule solution (which generated a very poor result).

It is in this way that the main downfall of EMM, its difficulty in non-linear convergence, actually becomes a benefit in that it provides a measure upon which the adaptive timestep size can be based. The midpoint rule rarely seems to have a problem in non-linear iterations but produces a result which is non-physical. As such, without some artificial bound on change in energy per step, there seems to be no good way to adaptively monitor the solution to ensure meaningful results for the midpoint

rule.

4.6 A preliminary investigation on accuracy

In an attempt to try to understand the requirements on the timestep size required to obtain a converged solution a theoretical question was posed: What is the largest timestep that can be taken which produces a maximum of 5 percent error over a duration of 5 seconds? In this preliminary study we compared the explicit central difference method, the implicit midpoint rule, and the implicit EMM for the beam problem described above. The non-linear iteration tolerance was $1e-6$ on the norm of the residual. To identify if the stiffness of the problem was a major factor in this required timestep size we ran two cases: 1) stiffness of the rigid bars= $1e6$, 2) stiffness of the rigid bars= $1e3$. In this manner we are able to compare the stiff problem with the one where the range of frequencies is not nearly as high. The explicit central difference solution using a timestep size of $\Delta t = 0.000001$ was considered as the "exact" solution. The results can be seen in Figs 4.9-4.16.

Surprisingly, the result of this study shows that for the stiff problem (rigid bar stiffness = $1e6$) the implicit schemes showed ZERO advantage over the explicit in obtaining a converged solution (where converged is considered within 5 percent error). That is, to have a maximum of 5 percent error, they all needed a timestep size of 0.001 seconds. Given the very large additional computational cost of the implicit schemes, for this problem under these conditions it would seem the explicit scheme is by far the best algorithm to use.

With the non-stiff problem, we start to see a bit of an increase in the required timestep size for the implicit schemes. All 3 implicit schemes were able to complete the 5 second duration within 5 percent error using a timestep size of 0.003 while the explicit central difference required 0.002. As the computational advantage of the explicit is far more

than 1.5x, under these conditions for this problem it would seem that the explicit scheme is superior even for the non-stiff problem.

4.7 Conclusion

In this chapter an adaptive method for enabling long duration simulation with the energy momentum method was outlined. It was shown that with this adaptivity for the given problem EMM seems to be the only algorithm capable of ensuring physically realistic long term simulation. The symplectic midpoint rule, for comparable computational cost, produces a result in which axial potential energy builds up in the rigid bars until they oscillate wildly and in a very non-physical manner. Though EMM does not have any numerical dissipation, it seems to not allow energy to build up in the high frequency modes (axial oscillation in the rigid bars).

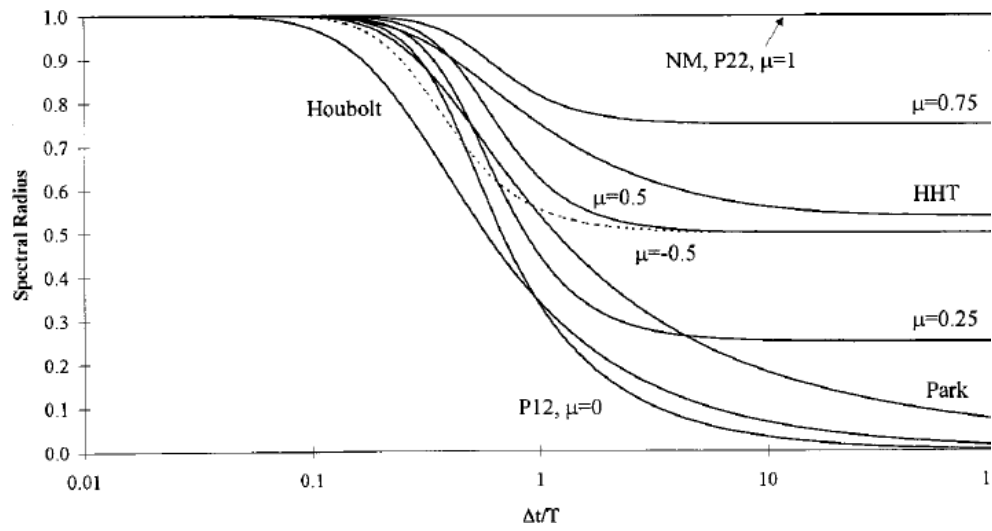


Figure 4.1: Sample spectral radius plot from [2]

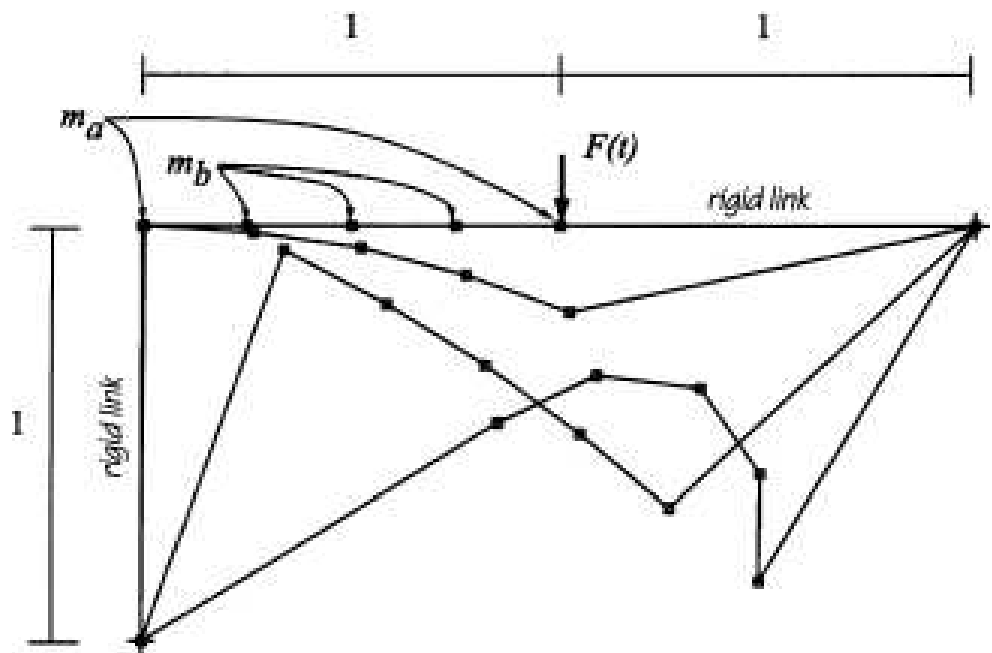
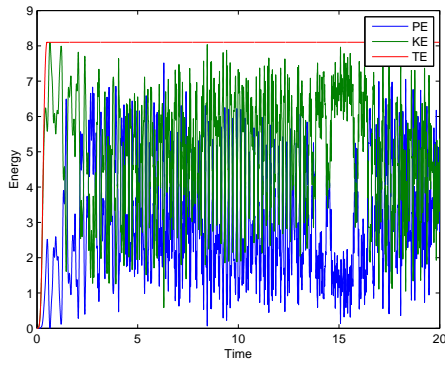
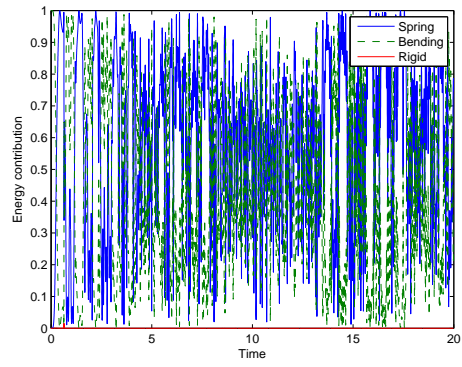


Figure 4.2: A rigid-flexible beam system from [3]

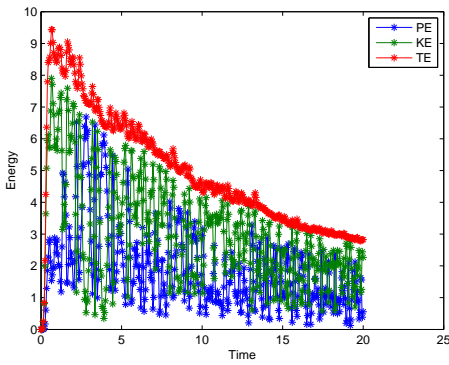


(a)

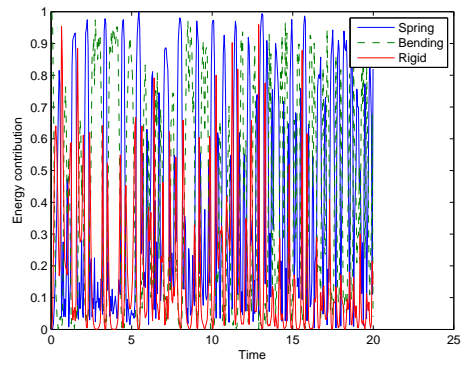


(b)

Figure 4.3: Central difference energy composition ($\Delta t = 0.000001$)

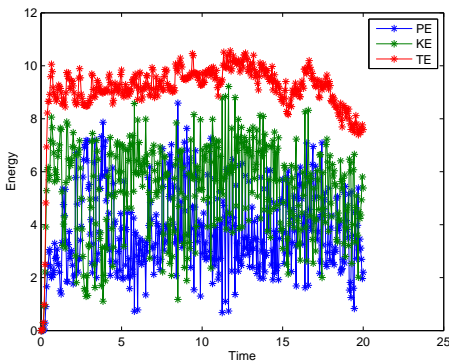


(a)

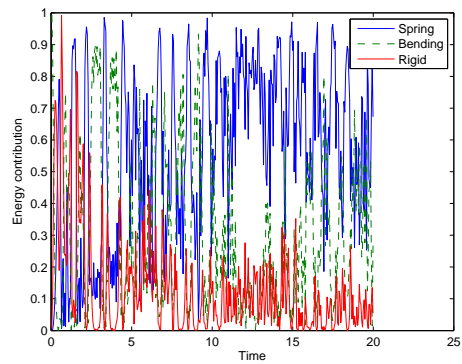


(b)

Figure 4.4: U0V0 $\rho_\infty = 0.5$ energy composition ($\Delta t = 0.05$)

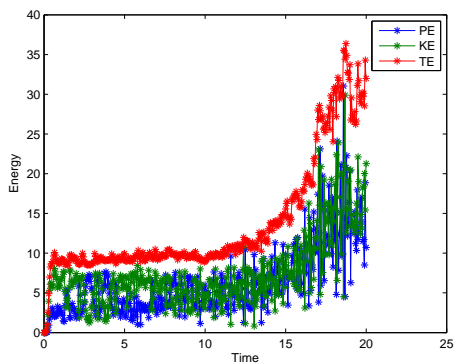


(a)

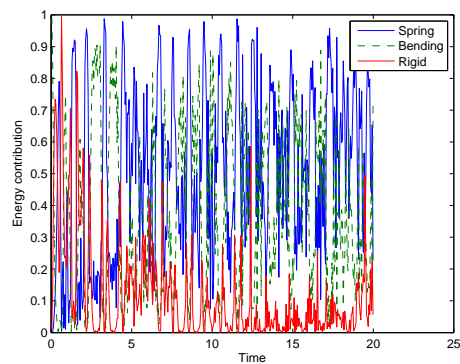


(b)

Figure 4.5: U0V0 $\rho_\infty = 0.9$ energy composition ($\Delta t = 0.05$)

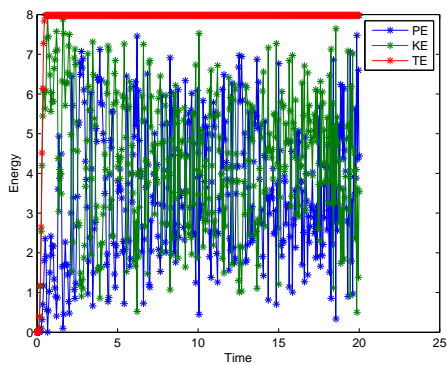


(a)

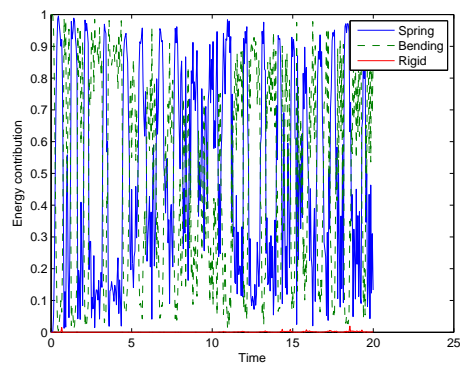


(b)

Figure 4.6: Midpoint rule energy composition ($\Delta t = 0.05$)



(a)



(b)

Figure 4.7: EMM energy composition ($\Delta t = 0.05$)

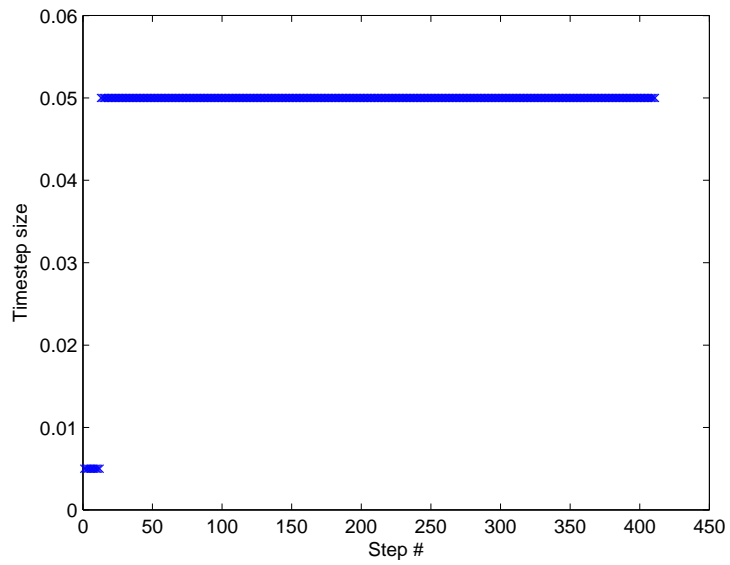


Figure 4.8: Adaptive timestep history

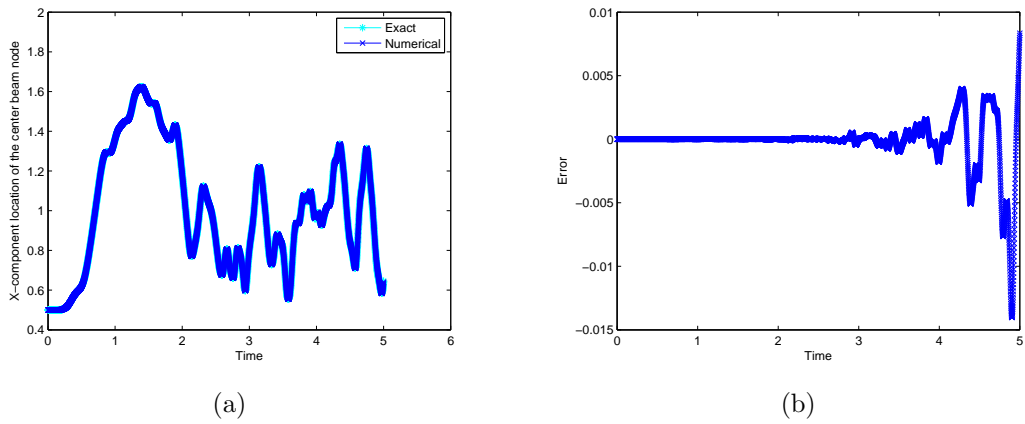
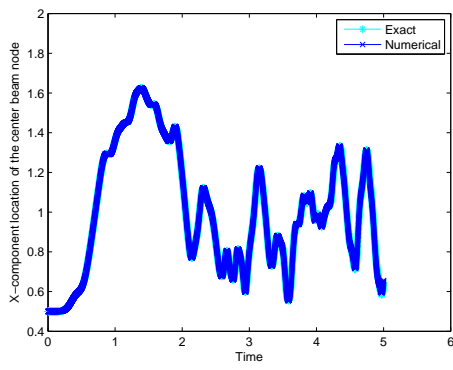
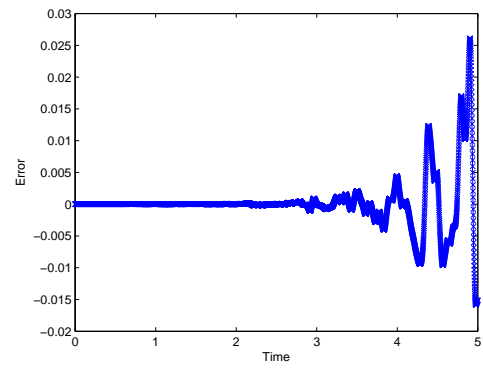


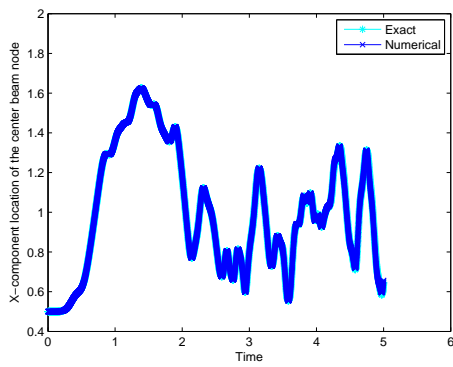
Figure 4.9: $1e6$ stiffness central difference 5 percent error accuracy limit ($\Delta t = 0.001$)



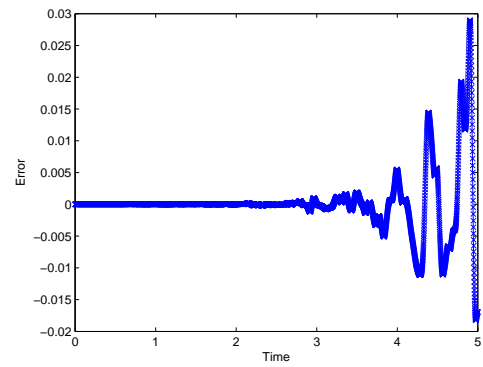
(a)



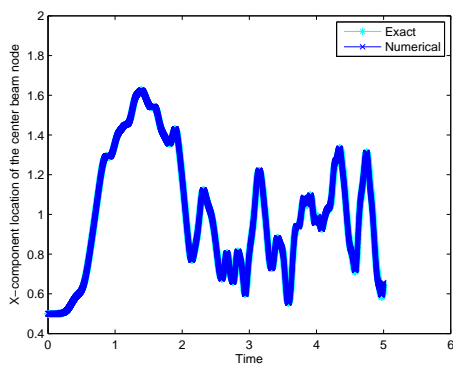
(b)

Figure 4.10: $1e6$ stiffness EMM 5 percent error accuracy limit ($\Delta t = 0.001$)

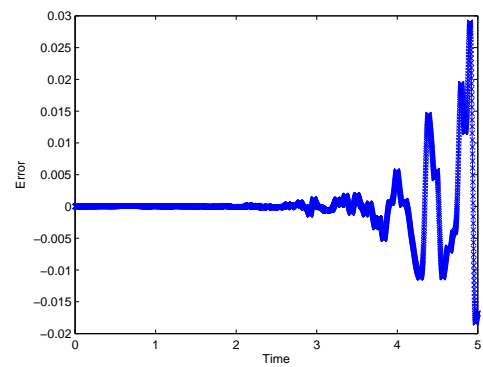
(a)



(b)

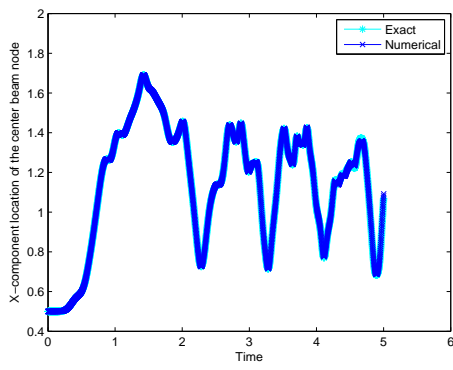
Figure 4.11: $1e6$ stiffness midpoint rule 5 percent error accuracy limit ($\Delta t = 0.001$)

(a)

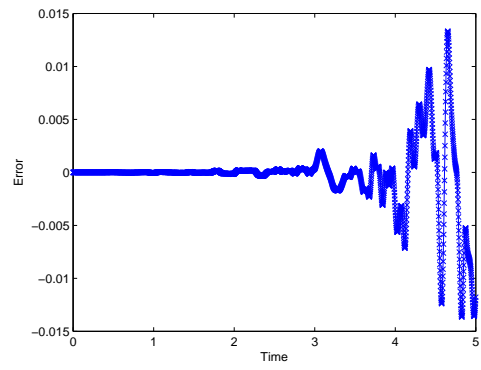


(b)

Figure 4.12: $1e6$ stiffness U0V0 $\rho_\infty = 0.5$ 5 percent error accuracy limit ($\Delta t = 0.001$)

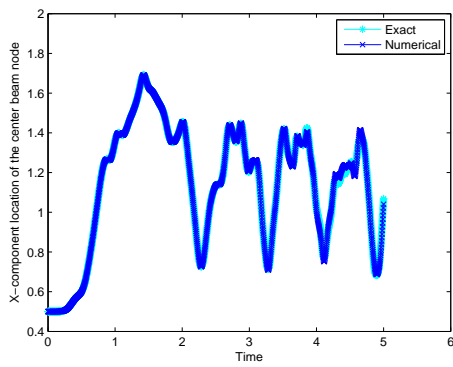


(a)

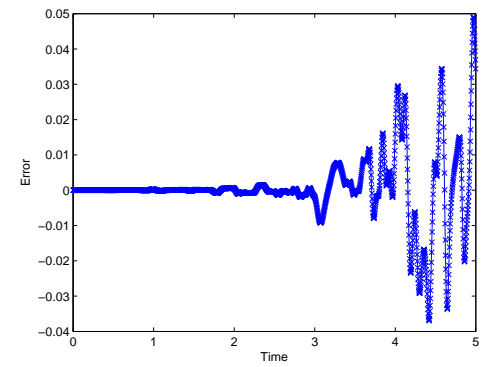


(b)

Figure 4.13: $1e3$ stiffness central difference 5 percent error accuracy limit ($\Delta t = 0.002$)

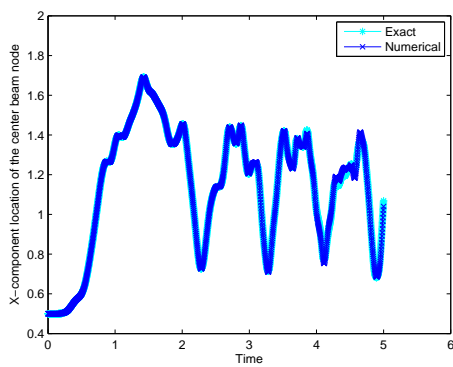


(a)

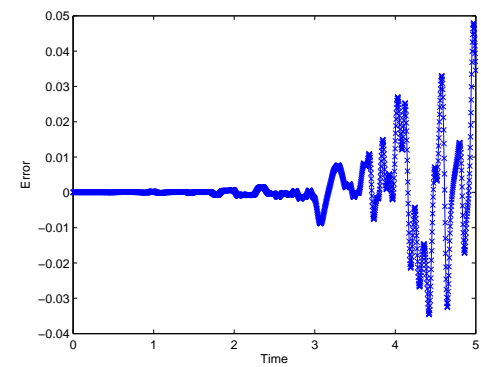


(b)

Figure 4.14: $1e3$ stiffness EMM 5 percent error accuracy limit ($\Delta t = 0.003$)

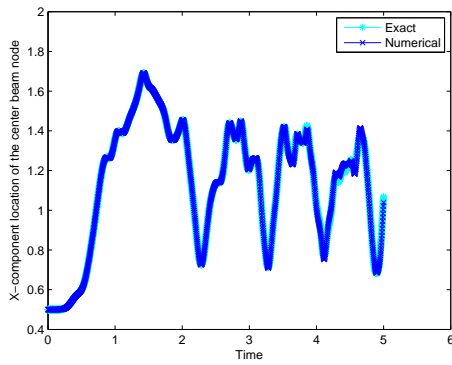


(a)

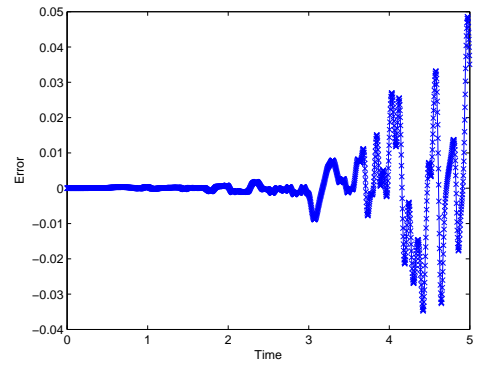


(b)

Figure 4.15: $1e3$ stiffness midpoint rule 5 percent error accuracy limit ($\Delta t = 0.003$)



(a)



(b)

Figure 4.16: $1e3$ stiffness U0V0 $\rho_\infty = 0.5$ 5 percent error accuracy limit ($\Delta t = 0.003$)

Chapter 5

Concluding Remarks

The present research described first an overview and introduction to the area of computational structural dynamics and integration used for conducting linear/nonlinear dynamics simulations. Some noteworthy perspectives and challenges were posed and still unresolved and subtle but important issues were highlighted including challenges faced to-date.

The main claims and contributions of this research included the following:

a) We first addressed the proper treatment and precise evaluations for integrating the equations of motion for linear and nonlinear dynamic situations. The focus was on the class of the so-called LMS methods simply for illustration of the basic ideas and since they continue to be the most popular in research and commercial codes to-date. The significance of the present developments is noteworthy since we demonstrated through theoretical developments and numerous numerical examples, that failure to enact proper treatment for the respective non-dissipative and dissipative algorithms will lead to inaccurate analysis and inferences that will be misleading. Furthermore, this is the first time that such a proper and rigorous treatment exists to help analysts to carefully implement computational algorithms for general dynamics applications

in the future.

b) We also provided a precise determination of acceleration time levels which is yet another important contribution. To-date, the literature is filled with misconceptions and lacks an in depth understanding of the underlying problem. As a consequence, inaccurate interpretations and incorrect analysis results for general structural dynamics applications. The relevant issues have been completely resolved once and for all in this research, with an in depth understanding of how to perform the respective computations accurately and precisely. The theoretical developments as well as the numerical simulations described the significance of the determinations and the ability to indeed maintain the required order of accuracy for faster convergence.

c) The final contribution addressed the notorious issue of long term dynamics and stiff nonlinear dynamics applications with a proof of concept and suggested strategies to foster such simulations. This is one of the most challenging of the issues to-date in computational structural dynamics and the present contributions explored new avenues not available currently.

Future research directions and recommendations include:

- Further investigate, the notion of algorithmic adaptivity and provide theoretical background for underlying stiff dynamic systems.
- Conduct rigorous analysis of stiff dynamics applications with different material models and ensure long term simulations.
- Extend the range of applications to the case of multi-body dynamics and their relevant practical applications.
- Conduct studies of nonlinear dynamics applications with energy-momentum and symplectic-momentum conserving algorithms to identify the pros and cons.

- Extend developments to large scale structural dynamics applications involving hundreds and thousands of degrees of freedom, and parallel computing strategies on high performance computing platforms.

Bibliography

- [1] S. Erlicher, L. Bonaventura, and O. S. Bursi. The Analysis of the Generalized- α Method for Nonlinear Dynamic Problems. *Computational Mechanics*, 28:83–104, 2002.
- [2] T. C. Fung. Complex-Time-Step Newmark Methods with Controllable Numerical Dissipation. *International Journal for Numerical Methods in Engineering*, 41:65–93, 1998.
- [3] F. Armero and I. Romero. On the formulation of high-frequency dissipative time-stepping algorithms for nonlinear structural dynamics. Part I: low-order methods for two model problems and nonlinear elastodynamics. *Computer Methods in Applied Mechanics and Engineering*, 190:2603–2649, 2001.
- [4] X. Zhou and K. K. Tamma. Design, Analysis, and Synthesis of Generalized Single Step Single Solve and Optimal Algorithms for Structural Dynamics. *International Journal for Numerical Methods in Engineering*, 59:597–668, 2004.
- [5] X. Zhou and K. K. Tamma. Algorithms by Design with Illustrations to Solid and Structural Mechanics/Dynamics. *International Journal for Numerical Methods in Engineering*, 66:1738–1790, 2006.

- [6] X. Zhou and K. K. Tamma. A New Unified Theory Underlying Time Dependent Linear First-Order Systems: A Prelude to Algorithms by Design. *International Journal for Numerical Methods in Engineering*, 60:1699–1740, 2004.
- [7] N. M. Newmark. A Method of Computation for Structural Dynamics. *Journal for American Society of Civil Engineers*, 1:67–94, 1959.
- [8] K. K. Tamma and R. R. Namburu. Applicability and Evaluation of An Implicit Self-Starting Unconditionally Stable Methodology for Dynamics of Structures. *Computers and Structures*, 34:835–842, 1990.
- [9] H. M. Hilber. *Analysis and Design of Numerical Integration Methods in Structural Dynamics*. PhD thesis, University of California at Berkeley, College of Engineering, University of California, Berkeley, California, 1977.
- [10] W. L. Wood, M. Bossak, and O. C. Zienkiewicz. An Alpha Modification of Newmark’s Method. *International Journal for Numerical Methods in Engineering*, 15:1562–1566, 1980.
- [11] H. P. Shao and C. W. Cai. The Direct Integration Three-Parameters Optimal Schemes for Structural Dynamics. In *Proceeding of the International Conference: Machine Dynamics and Engineering Applications*, pages C16–C20. Xi’an Jiaotong University Press, 1988.
- [12] A. Hoitink, S. Masuri, X. Zhou, and K. K. Tamma. On the Precise Evaluation of Acceleration Computations for General Structural Dynamic Application: Issues and Noteworthy Perspectives. *Communications in Numerical Methods in Engineering (In Review)*.
- [13] K. K. Tamma, X. Zhou, and D. Sha. The Time Dimension: A Theory of Development/Evolution, Classification, Characterization and Design of Computational

- Algorithms for Transient/Dynamic Applications. *Archives in Computational Mechanics*, 7(2):67–290, 2000.
- [14] K. K. Tamma, X. Zhou, and D. Sha. A Theory of Development and Design of Generalized Integration Operators for Computational Structural Dynamics. *International Journal of Numerical Methods in Engineering*, 50:1619–1664, 2001.
- [15] H. M. Hilber, T. J. R. Hughes, and R. L. Taylor. Improved Numerical Dissipation for Time Integration Algorithms in Structural Dynamics. *Earthquake Engineering and Structural Dynamics*, 5:283–292, 1977.
- [16] V. A. Leontiev. Extension of LMS formulations for L-Stable Optimal Integration Methods with U0-V0 Overshoot Properties in Structural Dynamics: The Level-Symmetric (LS) Integration Methods. *International Journal for Numerical Methods in Engineering*, 71:1598–1632, 2007.
- [17] J. Chung and G. Hulbert. A Time Integration Method for Structural Dynamics With Improved Numerical Dissipation: The Generalized α -Method. *Journal of Applied Mechanics*, 30:371–375, 1993.
- [18] S. Masuri, A. Hoitink, X. Zhou, and K. K. Tamma. Algorithms by Design: A Novel Normalized Time Weighted Residual Methodology and Design Leading to a Family of Energy-Momentum Conserving Algorithms for Nonlinear Structural Dynamics. *International Journal of Numerical Methods in Engineering (In Review)*.
- [19] S. Masuri, A. Hoitink, X. Zhou, and K. K. Tamma. Algorithms by Design: Optimal Energy-Momentum Based Controllable Numerical Dissipative Schemes in the Sense of Linear Multi-Step (LMS) Methods for Nonlinear Structural Dynamics. *Computer Methods in Applied Mechanics and Engineering (In Review)*.

- [20] S. Masuri, A. Hoitink, X. Zhou, and K. K. Tamma. Algorithms by Design: Part II - A Novel Normalized Time Weighted Residual Methodology and Design Leading to a Family of Symplectic-Momentum Conserving Algorithms for Nonlinear Structural Dynamics. *Communications in Applied Numerical Methods in Engineering (In Review)*.
- [21] S. Masuri, A. Hoitink, X. Zhou, and K. K. Tamma. Algorithms by Design: Part III - A Novel Normalized Time Weighted Residual Methodology and Design of Optimal Symplectic-Momentum Based Controllable Numerical Dissipative Algorithms for Nonlinear Structural Dynamics. *Communications in Applied Numerical Methods in Engineering (In Review)*.
- [22] K. Kuhl and M. Crisfield. Energy-Conserving and Decaying Algorithms in Nonlinear Structural Dynamics. *International Journal for Numerical Methods in Engineering*, 45:569–599, 1999.
- [23] K. K. Tamma, R. Kanapady, X. Zhou, and D. Sha. Recent Advances in Computational Structural Dynamics Algorithms. In *Seventh Int-Conf. on Recent Advances in Structural Dynamics ISVR*, Southampton, UK, July 2000.
- [24] A. Hoitink, S. Masuri, X. Zhou, and K. K. Tamma. Algorithms by Design: Part I - On the Hidden Point Collocation Within LMS Methods and Implications for Nonlinear Dynamics Applications. *International Journal for Computational Methods in Engineering Science and Mechanics (In Review)*.

Appendix A

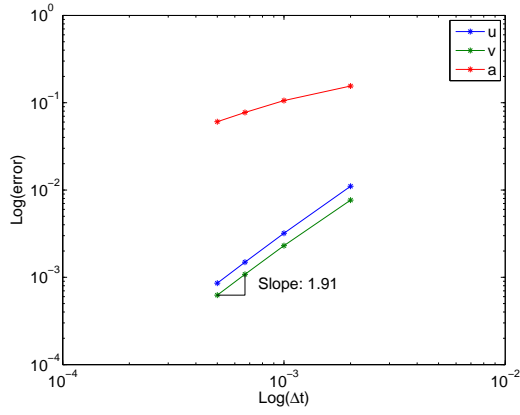
Acceleration Alignment: Further Numerical Examples

A.1 Four Spring-mass system

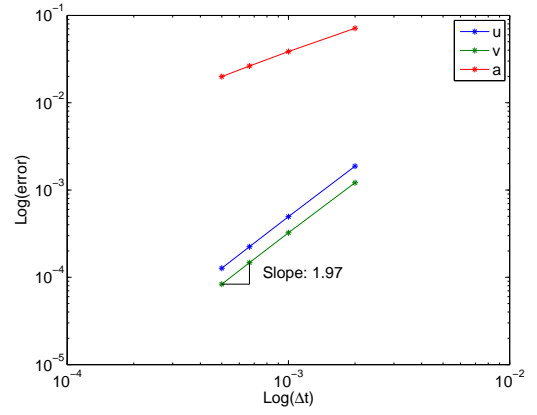
In this numerical example the same nonlinear spring element is used to connect four masses using four springs. The nodal locations of the masses are $[0,0,0;1,0,0;1,1,0;0,1,0]$. The masses are connected to form a square in the XY plane. Each mass is 1 unit, the spring stiffness is 1000 units, and the natural length of each spring is 1 unit. For this example the system is given only an initial displacement of $[0,0,0,0,0,0,0,0,0,0,1]$, that is node 4 is displaced one unit in the Z direction. The following figures again demonstrate the importance of the acceleration time level in creating correct convergence plots.

A.2 The Simple Pendulum

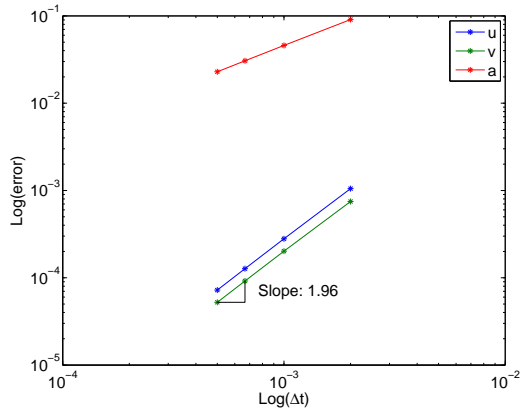
In this final example the same nonlinear spring element is used in a standard pendulum configuration. The nodal locations of the masses are $[0,0,0;0,-3.0443,0]$. The



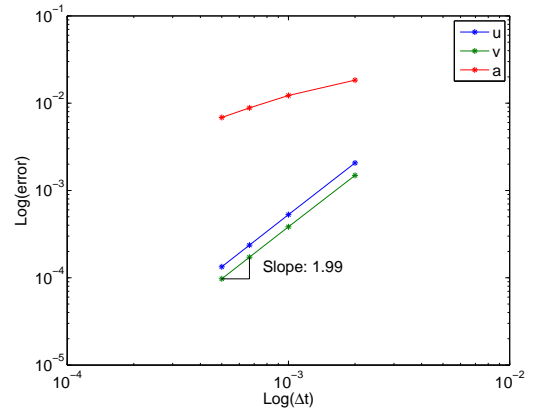
(a) $U0(0,0,0)$



(b) $U0(0.25,1,0.25)$



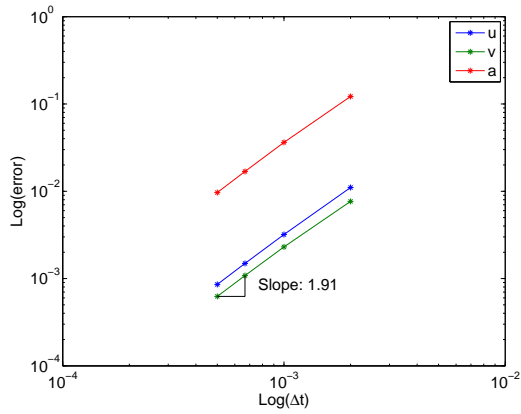
(c) $U0(0.5,0.5,0.5)$



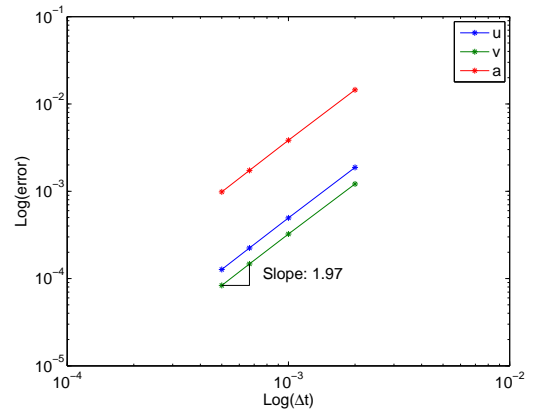
(d) $U0(0.8,0.8,0.125)$

Figure A.1: Energy-momentum based four spring-mass $U0$ family traditional convergence plots

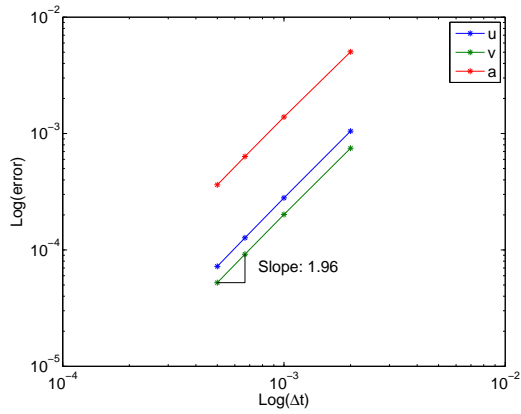
node at $(0,0,0)$ is fixed while the other node has a mass of 10 units. The spring stiffness is 1000 units and has a natural length of 3.0443 units. For this example the system is given only an initial velocity of $[0,0,0,7.725,0,0]'$, that is node 2 is given a velocity in the position X direction, making the pendulum begin circular motion.



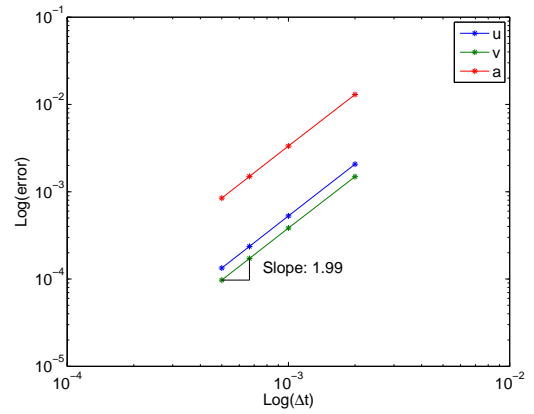
(a) $U0(0,0,0)$



(b) $U0(0.25,1,0.25)$

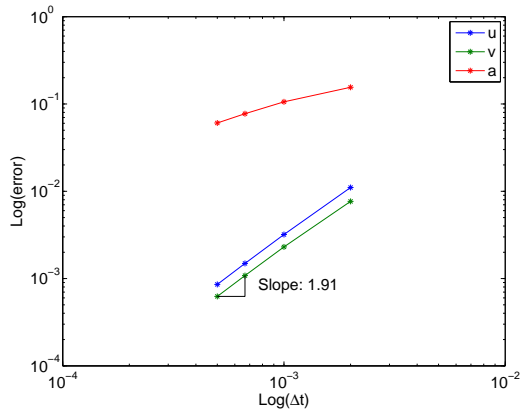


(c) $U0(0.5,0.5,0.5)$

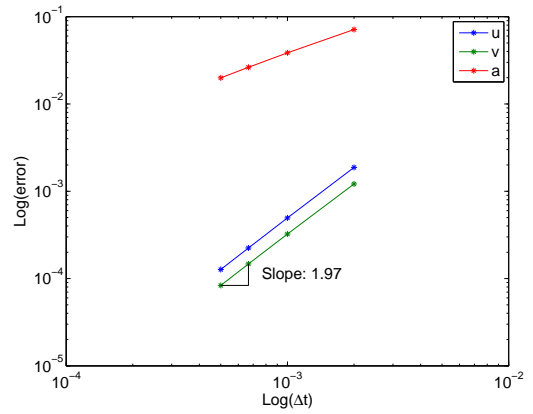


(d) $U0(0.8,0.8,0.125)$

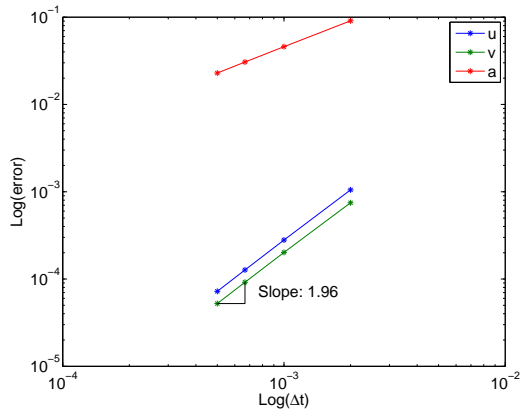
Figure A.2: Energy-momentum based four spring-mass $U0$ family acceleration aligned convergence plots



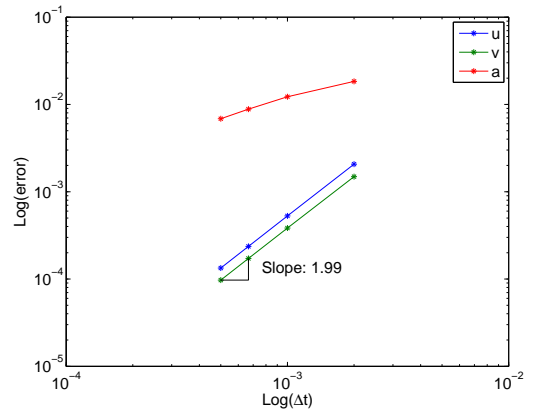
(a) $V0(0,0,0)$



(b) $V0(0.25,1,0.25)$

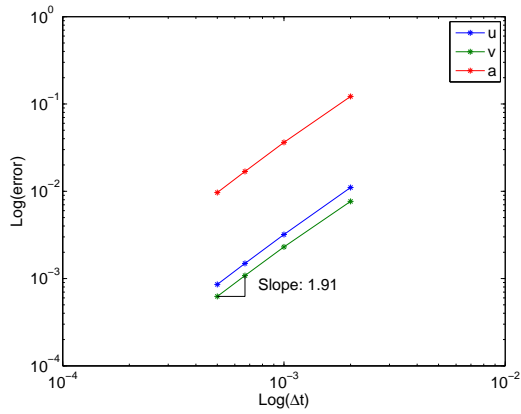


(c) $V0(0.5,0.5,0.5)$

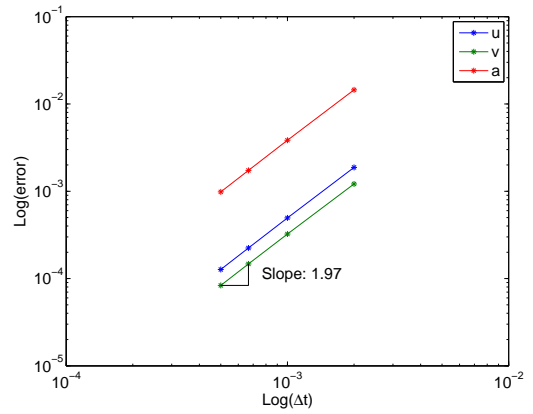


(d) $V0(0.8,0.8,0.125)$

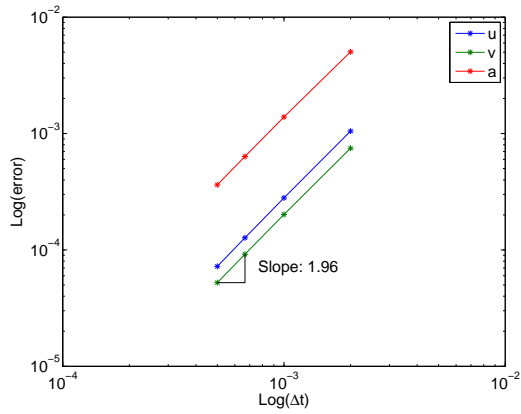
Figure A.3: Energy-momentum based four spring-mass $V0$ family traditional convergence plots



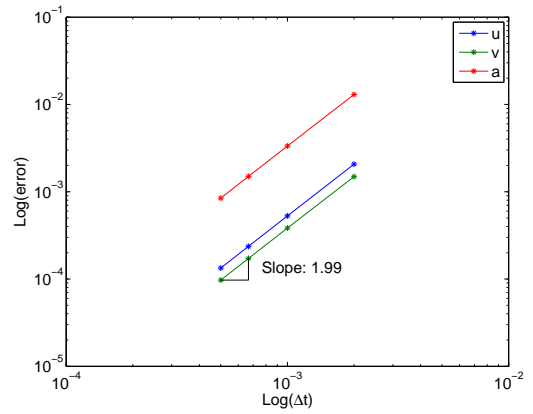
(a) $V0(0,0,0)$



(b) $V0(0.25,1,0.25)$

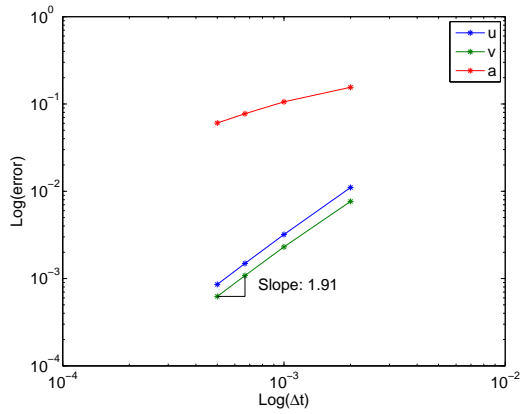


(c) $V0(0.5,0.5,0.5)$

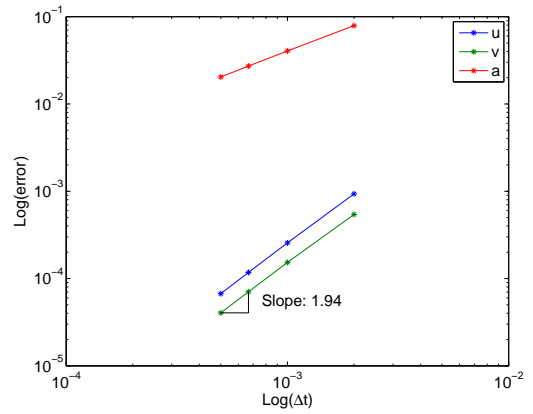


(d) $V0(0.8,0.8,0.125)$

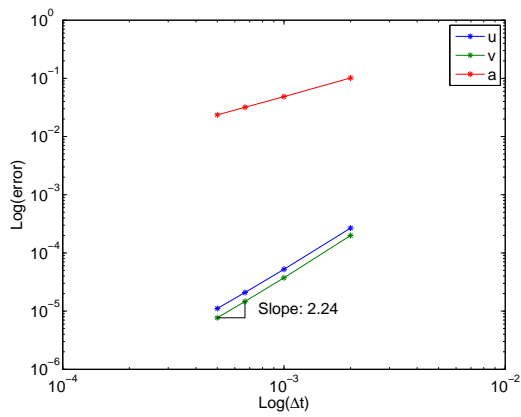
Figure A.4: Energy-momentum based four spring-mass $V0$ family acceleration aligned convergence plots



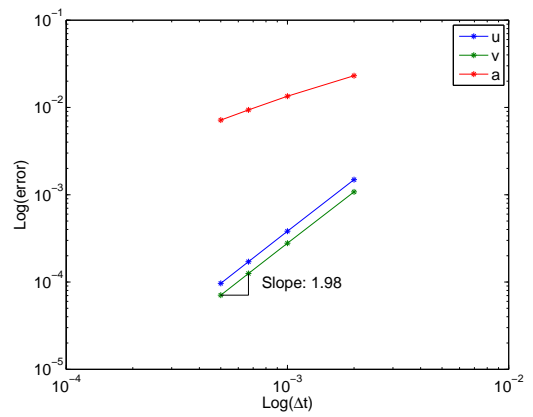
(a) $U0(0,0,0)$



(b) $U0(0.25,1,0.25)$

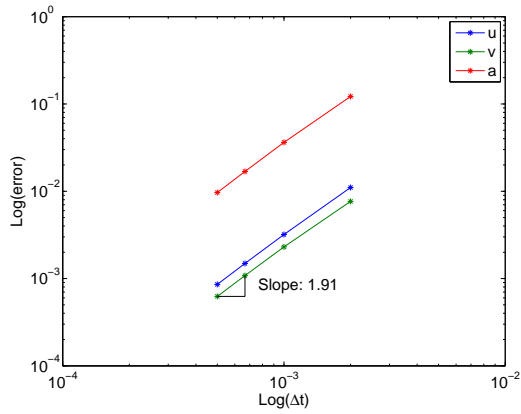


(c) $U0(0.5,0.5,0.5)$

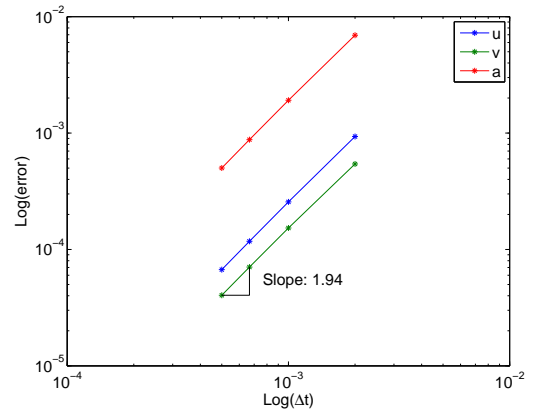


(d) $U0(0.8,0.8,0.125)$

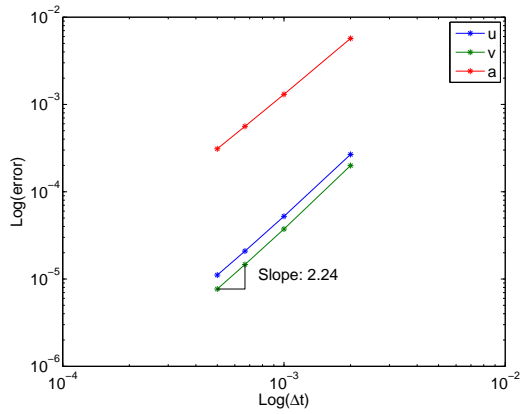
Figure A.5: Symplectic-momentum based four spring-mass $U0$ family traditional convergence plots



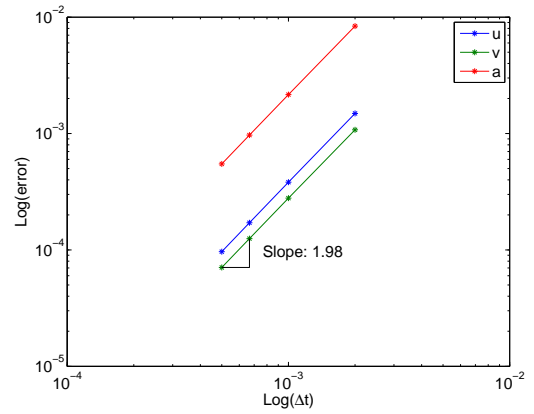
(a) $U0(0,0,0)$



(b) $U0(0.25,1,0.25)$

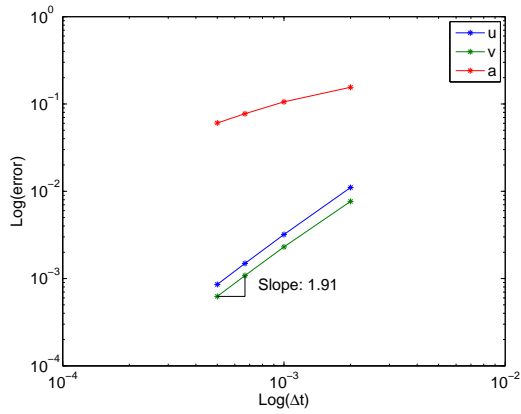


(c) $U0(0.5,0.5,0.5)$

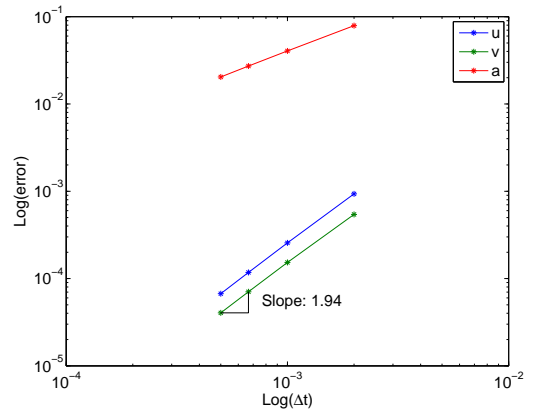


(d) $U0(0.8,0.8,0.125)$

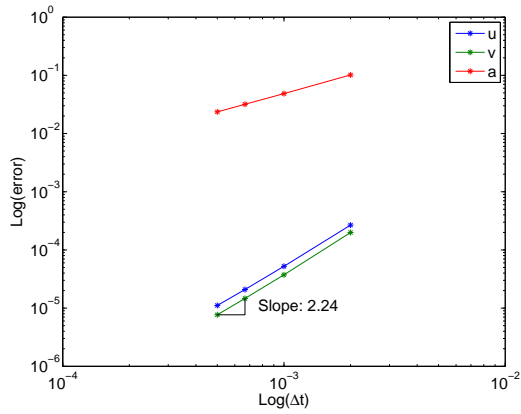
Figure A.6: Symplectic-momentum based four spring-mass $U0$ family acceleration aligned convergence plots



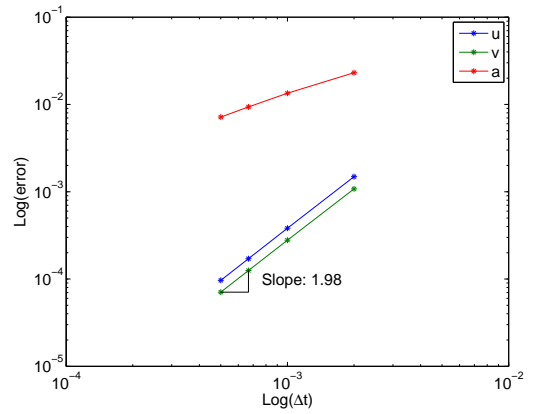
(a) $V0(0,0,0)$



(b) $V0(0.25,1,0.25)$

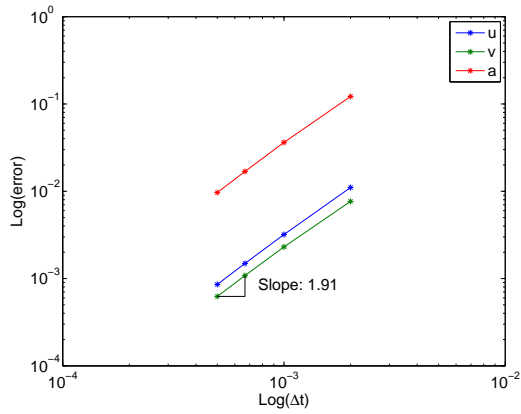


(c) $V0(0.5,0.5,0.5)$

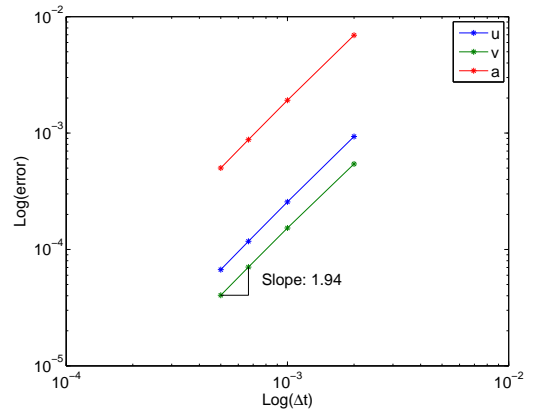


(d) $V0(0.8,0.8,0.125)$

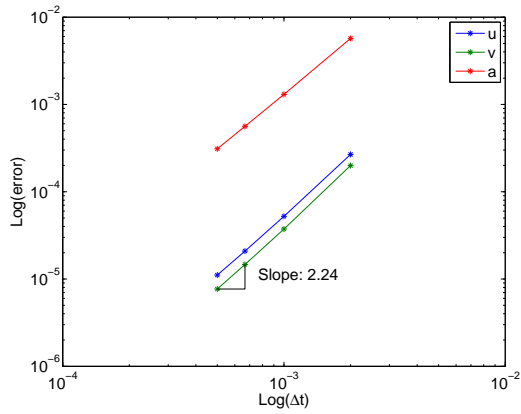
Figure A.7: Symplectic-momentum based four spring-mass $V0$ family traditional convergence plots



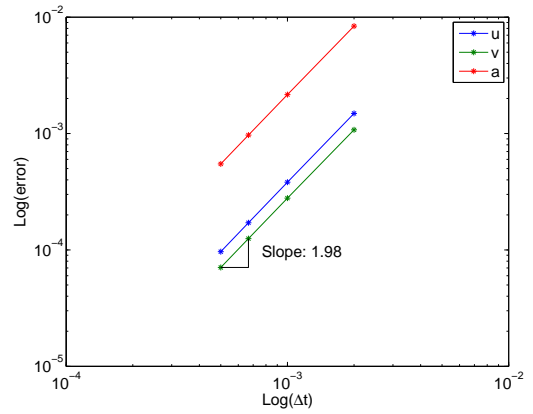
(a) $V0(0,0,0)$



(b) $V0(0.25,1,0.25)$

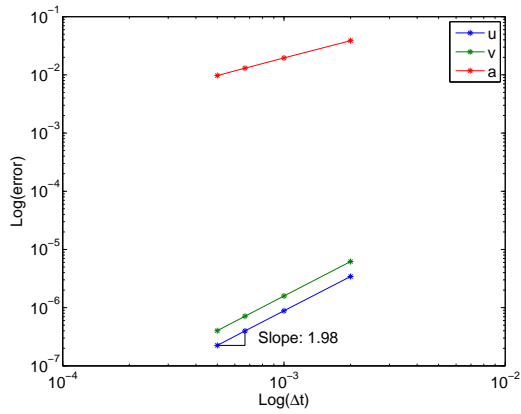


(c) $V0(0.5,0.5,0.5)$

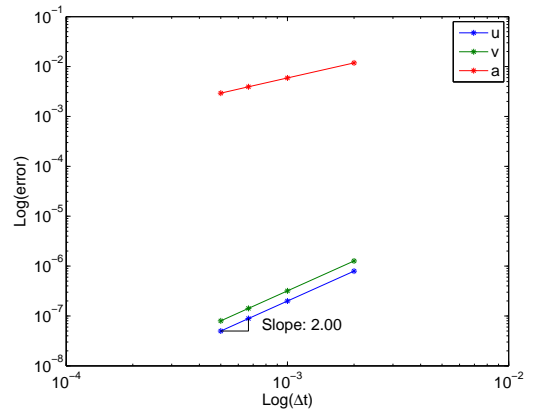


(d) $V0(0.8,0.8,0.125)$

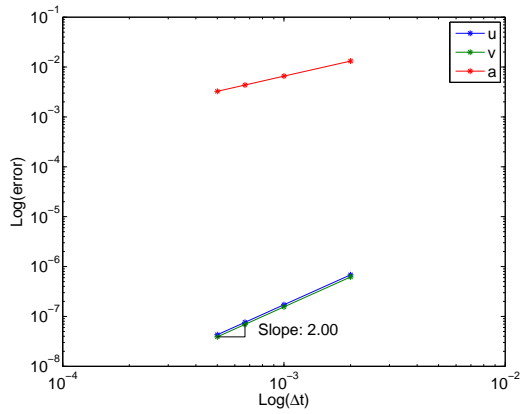
Figure A.8: Symplectic-momentum based four spring-mass $V0$ family acceleration aligned convergence plots



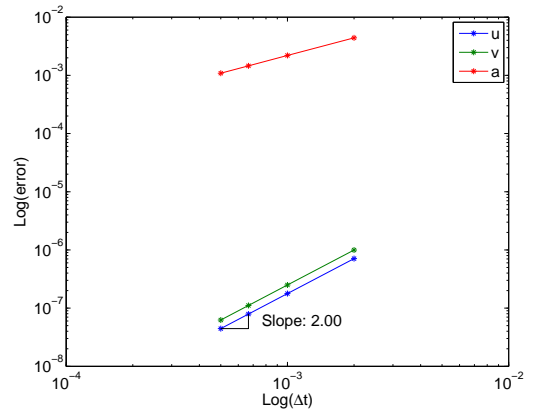
(a) $U_0(0,0,0)$



(b) $U_0(0.25,1,0.25)$

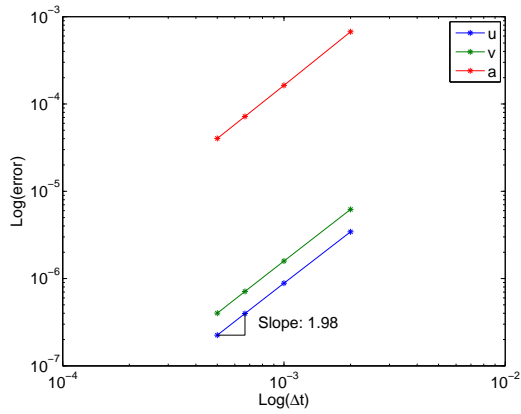


(c) $U_0(0.5,0.5,0.5)$

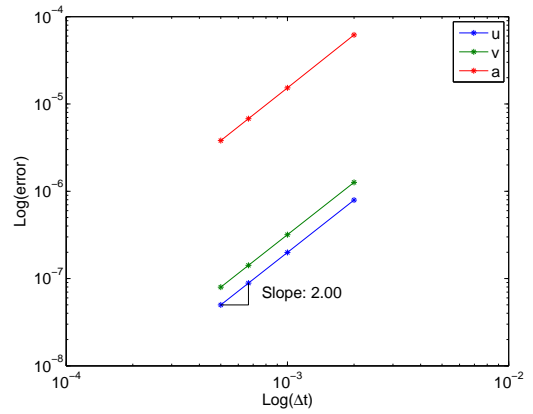


(d) $U_0(0.8,0.8,0.125)$

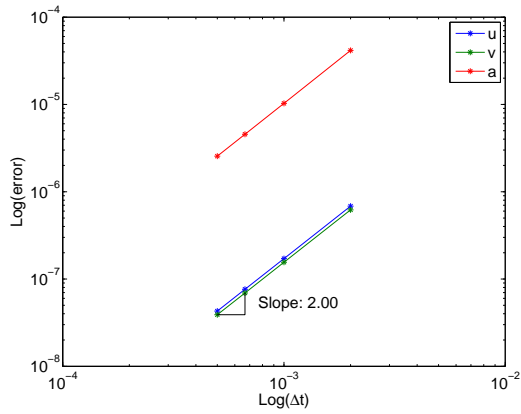
Figure A.9: Energy-momentum based simple pendulum U_0 family traditional convergence plots



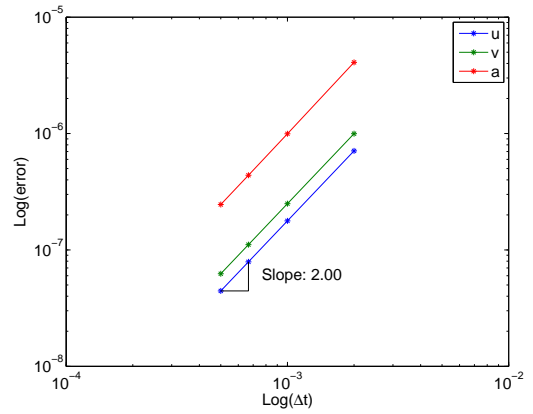
(a) $U0(0,0,0)$



(b) $U0(0.25,1,0.25)$

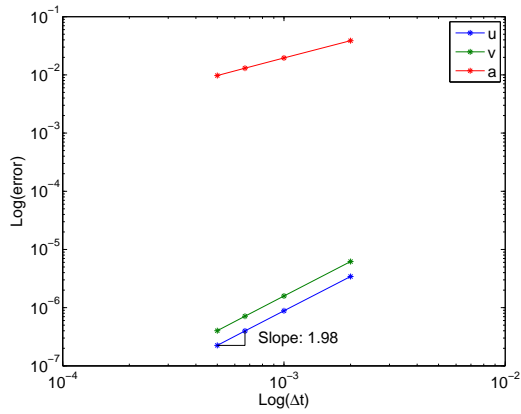


(c) $U0(0.5,0.5,0.5)$

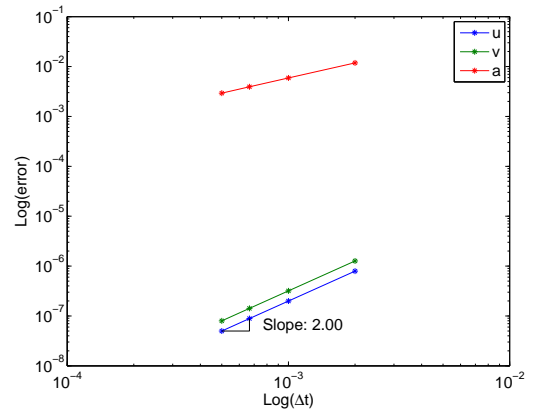


(d) $U0(0.8,0.8,0.125)$

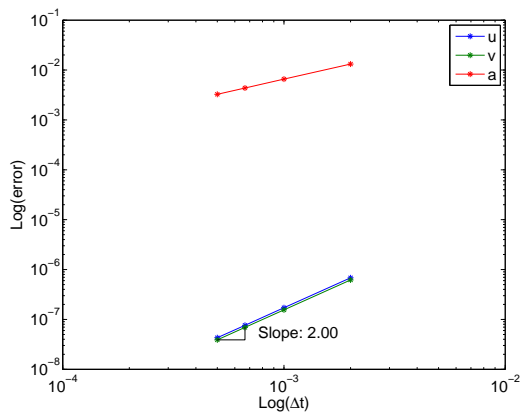
Figure A.10: Energy-momentum based simple pendulum $U0$ family acceleration aligned convergence plots



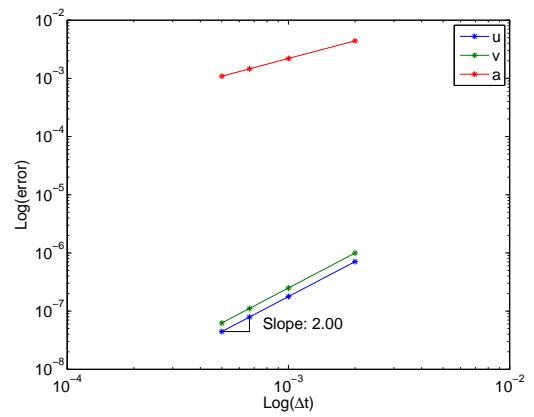
(a) $V0(0,0,0)$



(b) $V0(0.25,1,0.25)$

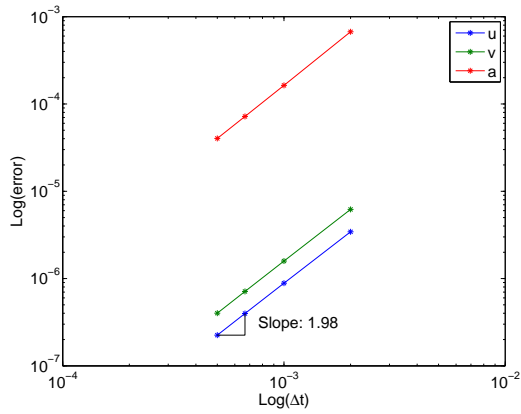


(c) $V0(0.5,0.5,0.5)$

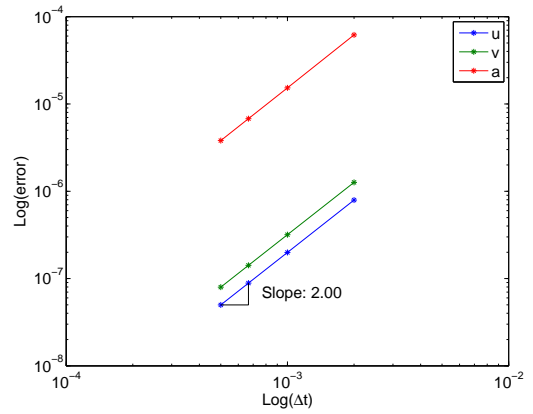


(d) $V0(0.8,0.8,0.125)$

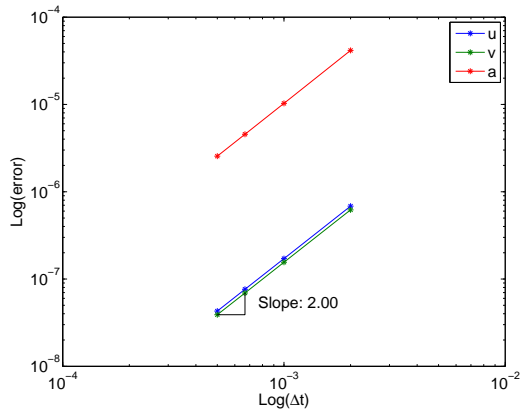
Figure A.11: Energy-momentum based simple pendulum $V0$ family traditional convergence plots



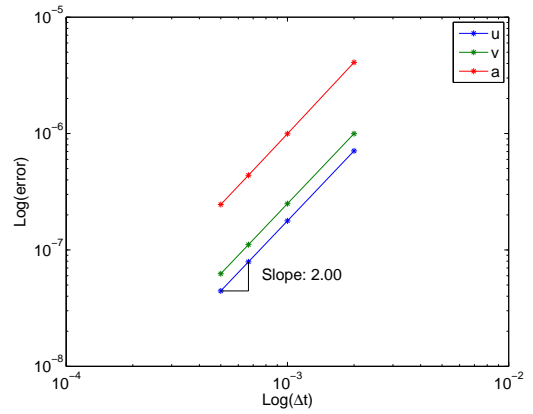
(a) $V0(0,0,0)$



(b) $V0(0.25,1,0.25)$

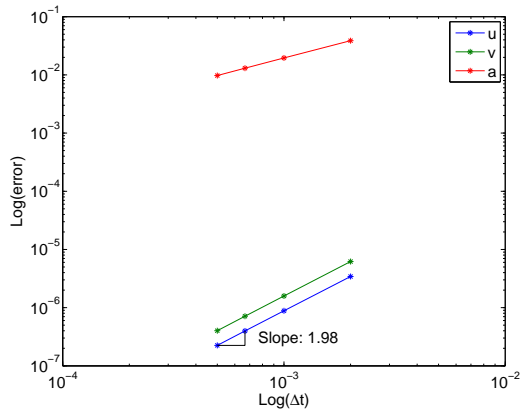


(c) $V0(0.5,0.5,0.5)$

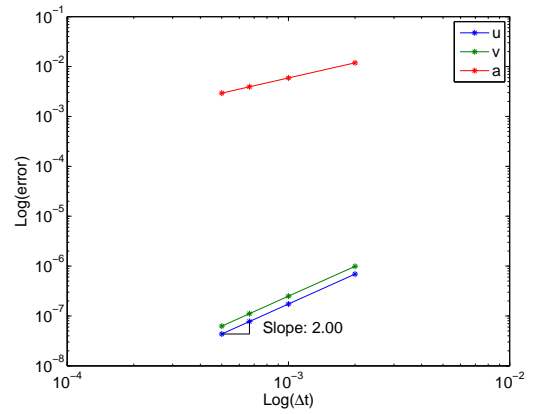


(d) $V0(0.8,0.8,0.125)$

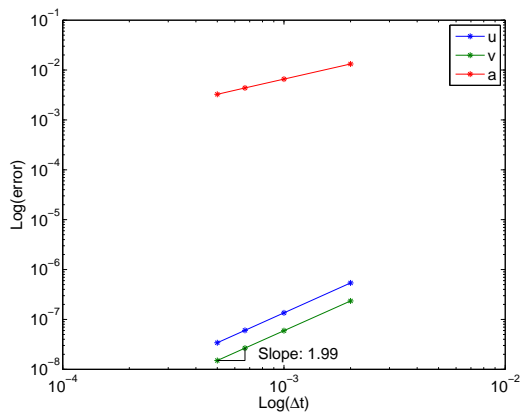
Figure A.12: Energy-momentum based simple pendulum $V0$ family acceleration aligned convergence plots



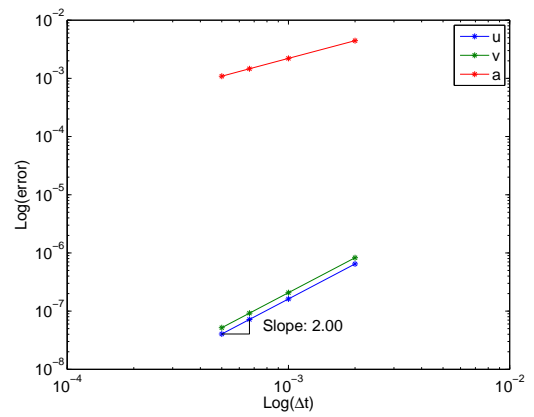
(a) $U_0(0,0,0)$



(b) $U_0(0.25,1,0.25)$

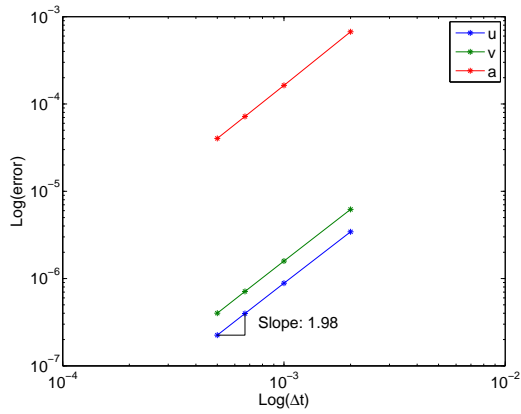


(c) $U_0(0.5,0.5,0.5)$

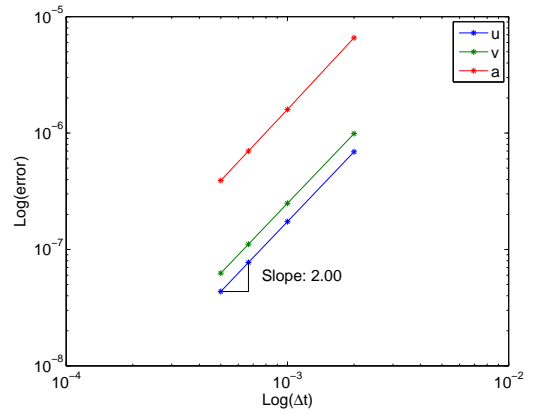


(d) $U_0(0.8,0.8,0.125)$

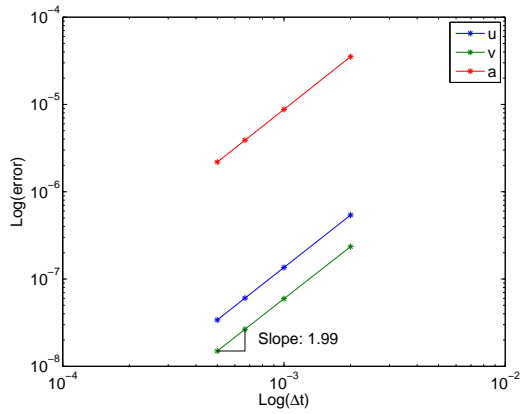
Figure A.13: Symplectic-momentum based simple pendulums U_0 family traditional convergence plots



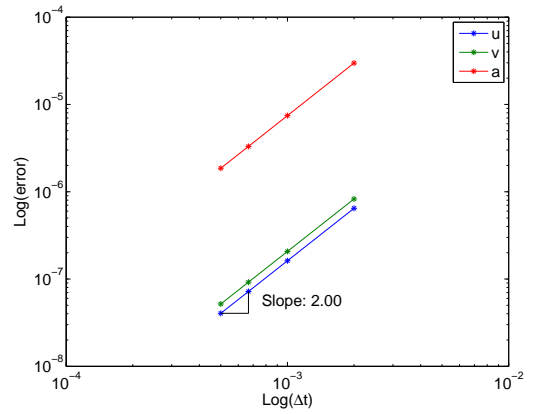
(a) $U_0(0,0,0)$



(b) $U_0(0.25,1,0.25)$

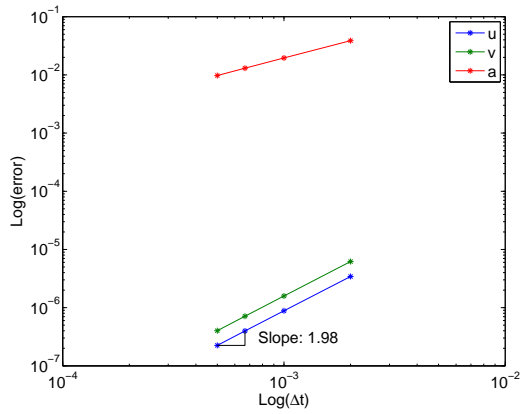


(c) $U_0(0.5,0.5,0.5)$

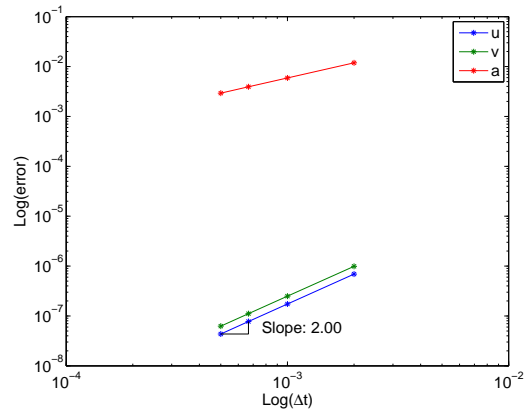


(d) $U_0(0.8,0.8,0.125)$

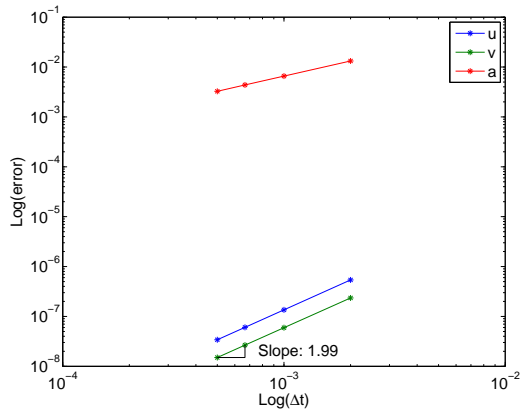
Figure A.14: Symplectic-momentum based simple pendulum U_0 family acceleration aligned convergence plots



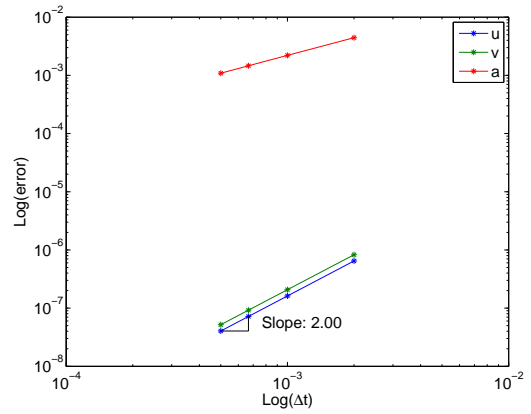
(a) $V0(0,0,0)$



(b) $V0(0.25,1,0.25)$

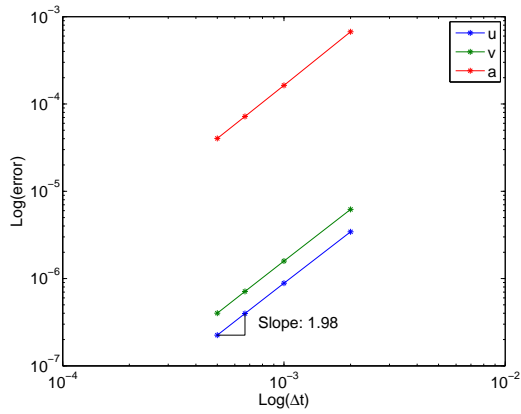


(c) $V0(0.5,0.5,0.5)$

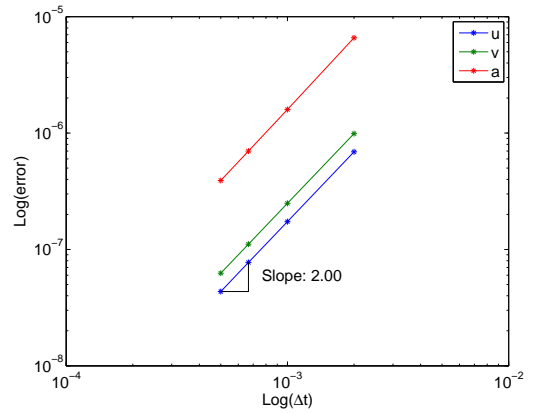


(d) $V0(0.8,0.8,0.125)$

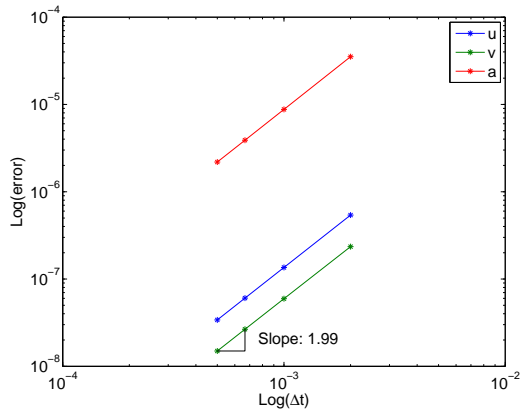
Figure A.15: Symplectic-momentum based simple pendulum $V0$ family traditional convergence plots



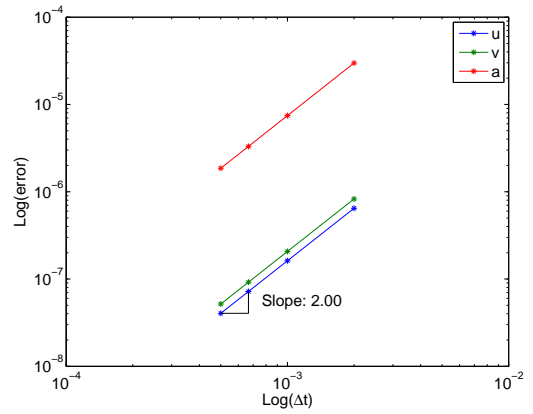
(a) $V0(0,0,0)$



(b) $V0(0.25,1,0.25)$



(c) $V0(0.5,0.5,0.5)$



(d) $V0(0.8,0.8,0.125)$

Figure A.16: Symplectic-momentum based simple pendulum $V0$ family acceleration aligned convergence plots

Appendix B

Programming Codes and Input Data

```

clear all;

% declare all the global variables here
global fixed A b W opl dupl irep h Xo elements nodes numNodes
global numElements fext masses
global tend method problem tol1 tol2;
global dp dn vn an ah dh en eh;

% get input
[ialgo,rho1,rho2,rho3,irep,tol,t,h,tend] = inputParams;

% formulate the mass matrix
mass = zeros(numNodes*3,numNodes*3);
for i=1:numNodes
    mass(3*i,3*i)=masses(i);
    mass(3*i-1,3*i-1)=masses(i);
    mass(3*i-2,3*i-2)=masses(i);
end

% initialize relevant matrices
% A is general to any element
A=1/4*[eye(3,3) -1*eye(3,3);-1*eye(3,3) eye(3,3)];

% b is element specific
for i = 1:numElements
    x01=nodes(elements(i,1),1);
    y01=nodes(elements(i,1),2);

```



```

zo1=nodes(elements(i,1),3);
xo2=nodes(elements(i,2),1);
yo2=nodes(elements(i,2),2);
zo2=nodes(elements(i,2),3);
b(1:6,i)=4/elements(i,4)^2*A*[xo1;yo1;zo1;xo2;yo2;zo2];
end

```

```

Xo=zeros(numNodes,1);
for i=1:numNodes
    Xo(3*i-2:3*i)=[nodes(i,1);nodes(i,2);nodes(i,3)];
end

```

% calculate initial accelerations

```
fint=zeros(numNodes*3,1);
```

```

for i = 1:numElements
    % get nodes of element i:
    node1=elements(i,1);
    node2=elements(i,2);
    % get initial displacement for element i:
    un(1:3,1)=dn(node1*3-2:node1*3);
    un(4:6,1)=dn(node2*3-2:node2*3);
    % get initial velocity for element i:
    undot(1:3,1)=vn(node1*3-2:node1*3);
    undot(4:6,1)=vn(node2*3-2:node2*3);
    % get element length and EA:

```

```

    Lo=elements(i,4);
    EA=elements(i,3);
    % get initial strain
    en(i)=b(:,i)'\*un+2/Lo^2*\un'*A*\un;
    % get initial internal force for this element
    elementFint=EA*Lo*(b(:,i)+4/Lo^2*A*\un)*en(i);
    fint((node1*3-2):(node1*3),1)=...
        fint((node1*3-2):(node1*3),1)+elementFint(1:3,1);
    fint((node2*3-2):(node2*3),1)=...
        fint((node2*3-2):(node2*3),1)+elementFint(4:6,1);
end

% remove rows/cols for fixed nodes
massSolve=mass;
for i=numNodes*3:-1:1
    if fixed(i)==1
        massSolve(i,:)=[];
        massSolve(:,i)=[];
    end
end

fextSolve=fext;
for i=numNodes*3:-1:1
    if fixed(i)==1
        fextSolve(i)=[];
    end
end

fintSolve=fint;

```

```

for i=numNodes*3:-1:1
    if fixed(i)==1
        fintSolve(i)=[];
    end
end

% solve for initial acceleration
anSolve = inv(massSolve)*(fextSolve - fintSolve);

% put back zeros in an
an=zeros(numNodes,1);
numFixed=0;
for i=1:numNodes*3
    if fixed(i)==0
        an(i)=anSolve(i-numFixed);
    else
        numFixed=numFixed+1;
    end
end

% get en_dot and en_dot_dot:
for i = 1:numElements
    % get nodes of element i:
    node1=elements(i,1);
    node2=elements(i,2);
    % get initial displacement for element i:

```

```

un(1:3,1)=dn(node1*3-2:node1*3);
un(4:6,1)=dn(node2*3-2:node2*3);
% get initial velocity for element i:
undot(1:3,1)=vn(node1*3-2:node1*3);
undot(4:6,1)=vn(node2*3-2:node2*3);
% get initial acceleration for element i:
undotdot(1:3,1)=an(node1*3-2:node1*3);
undotdot(4:6,1)=an(node2*3-2:node2*3);
en_dot(i) = (b(:,i) + 4/Lo^2*A*un)'*undot;
en_dot_dot(i) = (b(:,i) + 4/Lo^2*A*un)'*undotdot + ...
                4/Lo^2*undot'*A*undot;

end

% initialize result vectors
linearmomentum=zeros(1,4);
angularmomentum=zeros(1,4);

% get numerical method parameters
[W,opl,dupl] = gssss (ialgo,rho1,rho2,rho3);

% get the predictor corrector coeff
[xsip,xsic] = getXsi(W,opl,dupl,h,irep);

% store the initial condition into result
istep = 1;
time(istep)=t;

```

```

% position
position(:,istep) = Xo + dn;
velocity(:,istep) = vn;
displacement(:,istep) = dn;

% total energy
[KE(istep),PE(istep),totalenergy(istep)] = getenergy(dn,vn);

% linear momentum
linearmomentum(istep,:) = getLinearMomentum(vn);

% angular momentum
angularmomentum(istep,:) = getAngularMomentum(dn,vn);

%%%%%%%%% START THE TIME INTEGRATION PROCEDURE %%%%%%%%%%
steps=ceil(tend/h)+1;

while (istep<steps)
    t = h*istep
    istep = istep + 1;
    time(istep)=t;

    % cut down for fixed nodes
    bSolve=b;
    dnSolve=dn;
    vnSolve=vn;
    anSolve=an;
    enSolve=en;
    en_dotSolve=en_dot;

```

```

en_dot_dotSolve=en_dot_dot;
for i=numNodes*3:-1:1
    if fixed(i)==1
        bSolve(i)=[];
        dnSolve(i)=[];
        vnSolve(i)=[];
        anSolve(i)=[];
    end
end

% cut down A for fixed nodes
ASolve=A;
for i=numNodes*3:-1:1
    if fixed(i)==1
        ASolve(i,:)=[];
    end
end

for j=numNodes*3:-1:1
    if fixed(j)==1
        ASolve(:,j)=[];
    end
end

% ==== Predict ==== %
%displacement at n+1

```

```

dpSolve = xsip(1,1)*dnSolve + xsip(1,2)*vnSolve +...
          xsip(1,3)*anSolve;
%diplacememnt at hat time level
dhSolve = xsip(2,1)*dnSolve + xsip(2,2)*vnSolve +...
          xsip(2,3)*anSolve;
%velocity at hat time level
vhSolve = xsip(3,1)*dnSolve + xsip(3,2)*vnSolve +...
          xsip(3,3)*anSolve;
%acceleration at hat time level
ahSolve = xsip(4,1)*dnSolve + xsip(4,2)*vnSolve +...
          xsip(4,3)*anSolve;
%strain at hat time level
%first put back zeros in dp
dp=zeros(numNodes,1);
numFixed=0;
for i=1:numNodes*3
    if fixed(i)==0
        dp(i)=dpSolve(i-numFixed);
    else
        numFixed=numFixed+1;
    end
end
%then loop over each element to calculate
%the predicted strain
for i = 1:numElements
    % get nodes of element i:
    node1=elements(i,1);

```

```

node2=elements(i,2);
% get initial displacement for element i:
up(1:3,1)=dp(node1*3-2:node1*3);
up(4:6,1)=dp(node2*3-2:node2*3);
% get element length and EA:
Lo=elements(i,4);
EA=elements(i,3);
% get initial strain
ep(i)=b(:,i)'up+2/Lo2up'*A*up;
dele(i) = ep(i) - en(i);
eh(i) = en(i) + opl(3)*W(3)/dupl(3)*dele(i) +...
        (opl(1)*W(1)-opl(3)*W(3)*dupl(1)/...
        dupl(3))*h*en_dot(i)...
        (opl(2)*W(2)-opl(3)*W(3)*dupl(2)/...
        dupl(3))*h*en_dot_dot(i);
end

% start the N/L Iteration %
converge = 0;
nliter=0;

while (converge ~= 1)
    nliter=nliter+1;

    % put back zeros in dh, dp
    dh=zeros(numNodes,1);

```



```

dp=zeros(numNodes,1);

numFixed=0;
for i=1:numNodes*3
    if fixed(i)==0
        dh(i)=dhSolve(i-numFixed);
        dp(i)=dpSolve(i-numFixed);
    else
        numFixed=numFixed+1;
    end
end

% get the tangent stiffness matrix & internal force
[fint, ktang] = getFandK;

ktangSolve=ktang;
for i=numNodes*3:-1:1
    if fixed(i)==1
        ktangSolve(i,:)=[];
        ktangSolve(:,i)=[];
    end
end

fintSolve=fint;
for i=numNodes*3:-1:1
    if fixed(i)==1
        fintSolve(i)=[];
    end
end

```

```

        end
    end

    % form the jacobian matrix    %
    jac = xsic(4)*massSolve + xsic(2)*ktangSolve;

    % get the residual vector %
    res = fextSolve - massSolve*ahSolve - fintSolve;

    % solve for the delta    %
    del = inv(jac)*res;

    % ===== Correct ===== %
    %displacement at n+1 time level
    dpSolve = dpSolve + xsic(1)*del;
    %displacement at hat time level
    dhSolve = dhSolve + xsic(2)*del;
    %velocity at hat time level
    vhSolve = vhSolve + xsic(3)*del;
    %acceleration at hat time level
    ahSolve = ahSolve + xsic(4)*del;
    %strain at hat time level
    %first put back zeros in dp
    dp=zeros(numNodes,1);
    numFixed=0;
    for i=1:numNodes*3
        if fixed(i)==0

```

```

        dp(i)=dpSolve(i-numFixed);
    else
        numFixed=numFixed+1;
    end
end
end
%then loop over each element to calculate the
%new strain value
for i = 1:numElements
    % get nodes of element i:
    node1=elements(i,1);
    node2=elements(i,2);
    % get initial displacement for element i:
    up(1:3,1)=dp(node1*3-2:node1*3);
    up(4:6,1)=dp(node2*3-2:node2*3);
    % get element length and EA:
    Lo=elements(i,4);
    EA=elements(i,3);
    % get initial strain
    ep(i)=b(:,i)'up+2/Lo2up'*A*up;
    dele(i) = ep(i) - en(i);
    eh(i) = en(i) + opl(3)*W(3)/dupl(3)*dele(i) +...
    (opl(1)*W(1)-opl(3)*W(3))*...
    dupl(1)/dupl(3))*h*en_dot(i)...
    +(opl(2)*W(2)-opl(3)*W(3))*...
    dupl(2)/dupl(3))*h*en_dot_dot(i);
end
end

```

```

    % check for convergence %
    atol1 = norm(res);
    atol2 = norm(del);
    if (atol1<tol1) && (atol2<tol2)
        nliter;
        converge = 1;
    end

end % end of nonlinear iteration (solution converged)

% need to stick ahSolve into a non-reduced ah
% for final updates
ah=zeros(numNodes,1);
numFixed=0;
for i=1:numNodes*3
    if fixed(i)==0
        ah(i)=ahSolve(i-numFixed);
    else
        numFixed=numFixed+1;
    end
end

end

% ==== Design updates at the end of time step ==== %

% acceleration
ap = an + (ah-an)/opl(6)/W(1);
% velocity

```

```

vp = vn + dupl(4)*an*h + dupl(5)*(ap-an)*h;
% displacement
dp = dn + dupl(1)*vn*h + dupl(2)*an*h*h + ...
    dupl(3)*(ap-an)*h*h;

% ==== Get ready to the next time step ==== %
%displacement
dn = dp;
%velocity
vn = vp;
%acceleration
an = ap;
%strain
for i = 1:numElements
    % get nodes of element i:
    node1=elements(i,1);
    node2=elements(i,2);
    % get displacement for element i:
    un(1:3,1)=dn(node1*3-2:node1*3);
    un(4:6,1)=dn(node2*3-2:node2*3);
    % get velocity for element i:
    undot(1:3,1)=vn(node1*3-2:node1*3);
    undot(4:6,1)=vn(node2*3-2:node2*3);
    % get acceleration for element i:
    undotdot(1:3,1)=an(node1*3-2:node1*3);
    undotdot(4:6,1)=an(node2*3-2:node2*3);
    % get element length and EA:

```

```

    Lo=elements(i,4);
    EA=elements(i,3);
    % get initial strain
    en(i)=b(:,i)'un+2/Lo2un'*A*un;
    en_dot(i) = (b(:,i) + 4/Lo2*A*un)'*undot;
    en_dot_dot(i) = (b(:,i) + 4/Lo2*A*un)'*undotdot ...
                    + 4/Lo2*undot'*A*undot;

end

% ==== get result ====:

% position
position(:,istep) = Xo + dp;
velocity(:,istep) = vp;
displacement(:,istep) = dp;

% total energy
[KE(istep),PE(istep),totalenergy(istep)] = getenergy(dp,vp);

% linear momentum
linearmomentum(istep,:) = getLinearMomentum(vp);

% angular momentum
angularmomentum(istep,:) = getAngularMomentum(dp,vp);

end

```

```

% this function returns the GSSSS parameters

function [W,OPL,DUPL] = gssss ( IALGO, RH01, RH02, RH03 )

switch IALGO

% U0 Family
    case 100
        W(1) = 1.0/(1.+RH03);
        W(2) = 1.0/(1.+RH03);
        W(3) = 1.0/(1.+RH03);
        OPL(1) = 1.;
        OPL(2) = 0.50;
        OPL(3) = 1./((1.+RH01)*(1.+RH02));
        OPL(4) = 1.;
        OPL(5) = (3.+RH01+RH02-RH01*RH02)/...
                (2.*(1.+RH01)*(1.+RH02));
        OPL(6) = (2.+RH01+RH02+RH03-RH01*RH02*RH03)/...
                ((1.+RH01)*(1.+RH02));
        DUPL(1) = OPL(1);
        DUPL(2) = OPL(2);
        DUPL(3) = OPL(3);
        DUPL(4) = OPL(4);
        DUPL(5) = OPL(5);

% V0 family
    case 200
        W(1) = (3.+RH01+RH02-RH01*RH02)/...

```

```

                (2.*(1.+RHO1)*(1.+RHO2));
W(2)    = 2.0/((1.+RHO1)*(1.+RHO2));
W(3)    = 2.0/((1.+RHO1)*(1.+RHO2));
OPL(1)  = 1.;
OPL(2)  = 0.50;
OPL(3)  = 1./(2.*(1.+RHO3));
OPL(4)  = 1.;
OPL(5)  = 1./(1.+RHO3);
OPL(6)  = 2.*(2.+RHO1+RHO2+RHO3-RHO1*RHO2*RHO3)/...
                ((3.+RHO1+RHO2-RHO1*RHO2)*(1.+RHO3));
DUPL(1) = OPL(1);
DUPL(2) = OPL(2);
DUPL(3) = OPL(3);
DUPL(4) = OPL(4);
DUPL(5) = OPL(5);

    end

return

```



```

% this function computes the predictor/corrector coefficient
% given the DNA markers and the type of representation

function [xsip,xsic] = getXsi(W, opl, dupl, h, irep)
    switch irep

% True incremental d form %
        case 1
            xsip(1,1) = 1.0;
            xsip(1,2) = 0.0;
            xsip(1,3) = 0.0;
            xsip(2,1) = 1.0;
            xsip(2,2) = (opl(1)*W(1) - opl(3)*W(3)*dupl(1)/...
                dupl(3))*h;
            xsip(2,3) = (opl(2)*W(2) - opl(3)*W(3)*dupl(2)/...
                dupl(3))*h*h;
            xsip(3,1) = 0.0;
            xsip(3,2) = 1.0 - opl(5)*W(2)*dupl(1)/dupl(3);
            xsip(3,3) = (opl(4)*W(1) - opl(5)*W(2)*dupl(2)/...
                dupl(3))*h;
            xsip(4,1) = 0.0;
            xsip(4,2) = -opl(6)*W(1)*dupl(1)/dupl(3)/h;
            xsip(4,3) = 1.0 - opl(6)*W(1)*dupl(2)/dupl(3);
            xsic(1) = 1.0;
            xsic(2) = opl(3)*W(3)/dupl(3);
            xsic(3) = opl(5)*W(2)/dupl(3)/h;
            xsic(4) = opl(6)*W(1)/dupl(3)/h/h;

```

```
% Pseudo incremental d form %
```

```
case 2
```

```
xsip(1,1) = 1.0;
```

```
xsip(1,2) = (dupl(1)-dupl(3)*opl(1)*W(1)/...  
            opl(3)/W(3))*h;
```

```
xsip(1,3) = (dupl(2)-dupl(3)*opl(2)*W(2)/...  
            opl(3)/W(3))*h*h;
```

```
xsip(2,1) = 1.0;
```

```
xsip(2,2) = 0.0;
```

```
xsip(2,3) = 0.0;
```

```
xsip(3,1) = 0.0;
```

```
xsip(3,2) = 1.0 - opl(1)*opl(5)*W(1)*W(2)/...  
            opl(3)/W(3);
```

```
xsip(3,3) = (opl(4)*W(1)-opl(2)*opl(5)*W(2)*W(2)/...  
            opl(3)/W(3))*h;
```

```
xsip(4,1) = 0.0;
```

```
xsip(4,2) = -opl(1)*opl(6)*W(1)*W(1)/opl(3)/W(3)/h;
```

```
xsip(4,3) = 1.0 - opl(2)*opl(6)*W(1)*W(2)/opl(3)/W(3);
```

```
xsic(1) = dupl(3)/opl(3)/W(3);
```

```
xsic(2) = 1.0;
```

```
xsic(3) = opl(5)*W(2)/opl(3)/W(3)/h;
```

```
xsic(4) = opl(6)*W(1)/opl(3)/W(3)/h/h;
```

```
% True incremental v form %
```

```
case 3
```

```
xsip(1,1) = 1.0;
```

```

xsip(1,2) = dupl(1)*h;
xsip(1,3) = (dupl(2)-dupl(3)*dupl(4)/dupl(5))*h*h;
xsip(2,1) = 1.0;
xsip(2,2) = opl(1)*W(1)*h;
xsip(2,3) = (opl(2)*W(2)-opl(3)*dupl(4)*W(3)/...
            opl(5))*h*h;
xsip(3,1) = 0.0;
xsip(3,2) = 1.0;
xsip(3,3) = (opl(4)*W(1)-opl(5)*dupl(4)*W(2)/...
            dupl(5))*h;
xsip(4,1) = 0.0;
xsip(4,2) = 0.0;
xsip(4,3) = 1.0-opl(6)*dupl(4)*W(1)/dupl(5);
xsic(1) = dupl(3)*h/dupl(5);
xsic(2) = opl(3)*W(3)*h/dupl(5);
xsic(3) = opl(5)*W(2)/dupl(5);
xsic(4) = opl(6)*W(1)/dupl(5)/h;

```

% Pseudo incremental v form %

case 4

```

xsip(1,1) = 1.0;
xsip(1,2) = dupl(1)*h;
xsip(1,3) = (dupl(2)-dupl(3)*opl(4)*W(1)/...
            opl(5)/W(2))*h*h;
xsip(2,1) = 1.0;
xsip(2,2) = opl(1)*W(1)*h;
xsip(2,3) = (opl(2)*W(2)-opl(3)*opl(4)*W(1)*W(2)/...

```

```

                                opl(5)/W(2))*h*h;
xsip(3,1) = 0.0;
xsip(3,2) = 1.0;
xsip(3,3) = 0.0;
xsip(4,1) = 0.0;
xsip(4,2) = 0.0;
xsip(4,3) = 1.0-opl(4)*opl(6)*W(1)*W(1)/...
                                opl(5)/W(2);
xsic(1) = dupl(3)*h/opl(5)/W(2);
xsic(2) = opl(3)*W(3)*h/opl(5)/W(2);
xsic(3) = 1.0;
xsic(4) = opl(6)*W(1)/opl(5)/W(2)/h;

% incremental a - form %
case 5
xsip(1,1) = 1.0;
xsip(1,2) = dupl(1)*h;
xsip(1,3) = dupl(2)*h*h;
xsip(2,1) = 1.0;
xsip(2,2) = opl(1)*W(1)*h;
xsip(2,3) = opl(2)*W(2)*h*h;
xsip(3,1) = 0.0;
xsip(3,2) = 1.0;
xsip(3,3) = opl(4)*W(1)*h;
xsip(4,1) = 0.0;
xsip(4,2) = 0.0;
xsip(4,3) = 1.0;

```

```
xsic(1) = dupl(3)*h*h;  
xsic(2) = opl(3)*W(3)*h*h;  
xsic(3) = opl(5)*W(2)*h;  
xsic(4) = opl(6)*W(1);
```

```
end
```

```
return
```

```

function [fint, kt] = getFandK

global A b dn vn an dp dh eh dh m1 m2 m3 m4 W opl dupl h Xo;
global elements numNodes numElements method strainh problem;

fint=zeros(numNodes*3,1);
kt=zeros(numNodes*3,numNodes*3);

for i = 1:numElements

    % get nodes for this element:
    node1=elements(i,1);
    node2=elements(i,2);

    % get displacement at h time level for this element:
    uh(1:3,1)=dh(node1*3-2:node1*3);
    uh(4:6,1)=dh(node2*3-2:node2*3);

    % get displacement at p time level for this element:
    up(1:3,1)=dp(node1*3-2:node1*3);
    up(4:6,1)=dp(node2*3-2:node2*3);

    % get strainh for this element:
    strainh=eh(i);

    EA = elements(i,3);
    Lo = elements(i,4);

switch method
case 1 % classical time weighted residual approach
[elFint,elKt]=getFK1(node1,node2,h,elements(i,3),elements(i,4),i);

```

```

case 2 % displacement based normalized time weighted residual
      % approach
[elFint,elKt]=getFK2(node1,node2,h,elements(i,3),elements(i,4),i);
case 3 % hybrid displacement-strain based normalized
      % time weighted residual approach
[elFint,elKt]=getFK3(node1,node2,h,elements(i,3),elements(i,4),i);
end

fint((node1*3-2):(node1*3),1)=fint((node1*3-2):(node1*3),1)+...
      elFint(1:3,1);
fint((node2*3-2):(node2*3),1)=fint((node2*3-2):(node2*3),1)+...
      elFint(4:6,1);
kt((node1*3-2):(node1*3),(node1*3-2):(node1*3))=...
      kt((node1*3-2):(node1*3),(node1*3-2):(node1*3))+elKt(1:3,1:3);
kt((node1*3-2):(node1*3),(node2*3-2):(node2*3))=...
      kt((node1*3-2):(node1*3),(node2*3-2):(node2*3))+elKt(1:3,4:6);
kt((node2*3-2):(node2*3),(node1*3-2):(node1*3))=...
      kt((node2*3-2):(node2*3),(node1*3-2):(node1*3))+elKt(4:6,1:3);
kt((node2*3-2):(node2*3),(node2*3-2):(node2*3))=...
      kt((node2*3-2):(node2*3),(node2*3-2):(node2*3))+elKt(4:6,4:6);
end

```

```
function[elFint,elKt] = getFK1(node1,node2,dt,EA,Lo,elementNum)
```

```
global A b dn dp dh vn an Xo opl dupl W
```

```
un(1:3,1)=dn(node1*3-2:node1*3);
```

```
un(4:6,1)=dn(node2*3-2:node2*3);
```

```
uh(1:3,1)=dh(node1*3-2:node1*3);
```

```
uh(4:6,1)=dh(node2*3-2:node2*3);
```

```
up(1:3,1)=dp(node1*3-2:node1*3);
```

```
up(4:6,1)=dp(node2*3-2:node2*3);
```

```
unDOT(1:3,1)=vn(node1*3-2:node1*3);
```

```
unDOT(4:6,1)=vn(node2*3-2:node2*3);
```

```
unDOTDOT(1:3,1)=an(node1*3-2:node1*3);
```

```
unDOTDOT(4:6,1)=an(node2*3-2:node2*3);
```

```
strainn=b(:,elementNum)'*un+2/Lo^2*un'*A*un;
```

```
strainp=b(:,elementNum)'*up+2/Lo^2*up'*A*up;
```

```
strainnDOT=(b(:,elementNum)+4/Lo^2*A*un)'*unDOT;
```

```
strainnDOTDOT=(b(:,elementNum)+4/Lo^2*A*un)'*unDOTDOT+...
```

```
4/Lo^2*unDOT'*A*unDOT;
```

```
strainh=strainn+W(3)*opl(3)/dupl(3)*(strainp-strainn)+...
```

```
(W(1)*opl(1)-W(3)*opl(3)*dupl(1)/dupl(3))*dt*strainnDOT+...
```



```

(W(2)*opl(2)-W(3)*opl(3)*dupl(2)/dupl(3))*dt^2*strainnDOTDOT;

u_strain_n = un*strainn;
u_strain_p = up*strainp;
u_strain_nDOT = unDOT*strainn + ...
                un*(b(:,elementNum) + 4/Lo^2*A*un) '*unDOT;
u_strain_nDOTDOT = unDOTDOT*strainn + 2*unDOT*(b(:,elementNum)...
                + 4/Lo^2*A*un) '*unDOT + ...
                un*(4/Lo^2*unDOT '*A)*unDOT + ...
                un*(b(:,elementNum) + 4/Lo^2*A*un) '*unDOTDOT;

u_strain_h = u_strain_n + ...
            W(3)*opl(3)/dupl(3)*(u_strain_p-u_strain_n)+...
            (W(1)*opl(1)-W(3)*opl(3)*dupl(1)/dupl(3))*dt*u_strain_nDOT+...
            (W(2)*opl(2)-W(3)*opl(3)*dupl(2)/dupl(3))*dt^2*u_strain_nDOTDOT;

elFint=EA*Lo*b(:,elementNum)*strainh + EA*4/Lo*A*u_strain_h;

elKt= EA*4/Lo*A*strainp + EA*Lo*(b(:,elementNum) + ...
        4/Lo^2*A*up)*(b(:,elementNum) + 4/Lo^2*A*up)';

```

```

function[elFint,elKt] = getFK3(node1,node2,dt,EA,Lo,elementNum)

global A b dn dp dh vn an ah Xo opl dupl W irep h strainh

un(1:3,1)=dn(node1*3-2:node1*3);
un(4:6,1)=dn(node2*3-2:node2*3);

uh(1:3,1)=dh(node1*3-2:node1*3);
uh(4:6,1)=dh(node2*3-2:node2*3);

up(1:3,1)=dp(node1*3-2:node1*3);
up(4:6,1)=dp(node2*3-2:node2*3);

unDOT(1:3,1)=vn(node1*3-2:node1*3);
unDOT(4:6,1)=vn(node2*3-2:node2*3);

unDOTDOT(1:3,1)=an(node1*3-2:node1*3);
unDOTDOT(4:6,1)=an(node2*3-2:node2*3);

uhDOTDOT(1:3,1)=ah(node1*3-2:node1*3);
uhDOTDOT(4:6,1)=ah(node2*3-2:node2*3);

upDOTDOT = (uhDOTDOT-(1-opl(6)*W(1))*unDOTDOT)/(opl(6)*W(1));

upDOT = unDOT + dupl(4)*unDOTDOT*h + dupl(5)*h*(upDOTDOT-unDOTDOT);

elFint=EA*Lo*(b(:,elementNum)+4/Lo^2*A*uh)*strainh;

```

```
e1Kt=(EA*4/Lo*A)*strainh + ...  
      EA*Lo*(b(:,elementNum)+4/Lo^2*A*uh)*...  
      (b(:,elementNum)+4/Lo^2*A*up)';
```

```

function[elFint,elKt] = getFK2(node1,node2,dt,EA,Lo,elementNum)

global A b dn dp dh vn an Xo opl dupl W

uh(1:3,1)=dh(node1*3-2:node1*3);
uh(4:6,1)=dh(node2*3-2:node2*3);

% strain evaluated using its definition
strainh=b(:,elementNum)'*uh+2/Lo^2*uh'*A*uh;

elFint=EA*Lo*(b(:,elementNum)+4/Lo^2*A*uh)*strainh;

elKt=(EA*4/Lo*A*)*strainh +...
      EA*Lo*(b(:,elementNum)+4/Lo^2*A*uh)*...
      (b(:,elementNum)+4/Lo^2*A*uh)';

```

```

function [KE,PE,energy] = getenergy(dn,v)

global masses numElements elements Xo en

numNodes=size(dn)/3;
% get current position
d = Xo + dn;

% calculate the kinetic energy
KE=0;
for i = 1:numNodes
    vx = v(i*3-2);
    vy = v(i*3-1);
    vz = v(i*3);
    KE=KE+0.5*masses(i)*(vx^2+vy^2+vz^2);
end

% calculate the potential energy
PE = 0;
for i = 1:numElements
    node1=elements(i,1);
    node2=elements(i,2);
    x1 = d(node1*3-2);
    y1 = d(node1*3-1);
    z1 = d(node1*3);
    x2 = d(node2*3-2);
    y2 = d(node2*3-1);

```

```
z2 = d(node2*3);  
L=((x1-x2)^2+(y1-y2)^2+(z1-z2)^2)^0.5;  
Lo=elements(i,4);  
PE = PE + 1/2*elements(i,3)*(L^2-Lo^2)^2/(4*Lo^3);  
end  
  
energy = KE + PE;
```

```
function momentum = getLinearMomentum(v)

global masses numNodes;

%initialize
Lx=0;
Ly=0;
Lz=0;

for i=1:numNodes
Lx = Lx + masses(i)*v(3*i-2);
Ly = Ly + masses(i)*v(3*i-1);
Lz = Lz + masses(i)*v(3*i);
end

Total_momentum = sqrt(Lx^2 + Ly^2 + Lz^2);

momentum = [Lx Ly Lz Total_momentum];
```

```

function momentum = getAngularMomentum(d,v)

global masses Xo numNodes;

% get position at n time level
Xn = Xo + d;

% initialize
Jx=0;
Jy=0;
Jz=0;

for i=1:numNodes
Jx=Jx+masses(i)*v(3*i)*Xn(3*i-1)-masses(i)*v(3*i-1)*Xn(3*i);
Jy=Jy+masses(i)*v(3*i-2)*Xn(3*i)-masses(i)*v(3*i)*Xn(3*i-2);
Jz=Jz+masses(i)*v(3*i-1)*Xn(3*i-2)-masses(i)*v(3*i-2)*Xn(3*i-1);
end

Totalmomentum = sqrt(Jx^2 + Jy^2 + Jz^2);

momentum = [Jx Jy Jz Totalmomentum];

```



```

%Input data
function [ialgo,rho1,rho2,rho3,irep,tol,t,h,tend] = inputParams()

global elements nodes masses fext numNodes numElements dn vn
global fixed method irep tol1 tol2 h t tend;

%===== input =====

% define problem
problem = 'bell'; % options: tet,bell,pend
ic = 'u'; % options: u,v,uv
EA = 1e2;
method = 2; % 1= classical time weighted residual approach
           % 2= displacement based normalized
           % time weighted residual approach
           % 3= hybrid displacement-strain based normalized time
           % weighted residual approach

% choose algorithms
ialgo = 100;
rho1 = 1;
rho2 = 1;
rho3 = 1;
%rho3 = (1-rho1*rho2)/(rho1 + rho2 + rho1*rho2) %for HHT alpha
irep = 3; % form of representation
tol = 1e-14;
tol1 = tol*1e6; % tolerance value for residual convergence

```

```

tol2 = tol;          % tolerance value for delta convergence

% time step size
h = 1;
% tstart
tstart = 0.0;
% t
t = tstart;
% tend
tend = 200;

%=====

switch problem
    case 'tet'
        %%%% tet-spring problem %%%%
        switch ic
            case 'u'
                %%%% initial displacement %%%%
                % enter elements as row: node1 node2 EA Lo
                elements = [ 1 2 EA 1; 2 3 EA 1; 1 3 EA 1;...
                            1 4 EA 1; 2 4 EA 1; 3 4 EA 1];
                [numElements,junk]=size(elements);
                % enter nodes as row: Xo, Yo, Zo
                nodes = [0.5,3^0.5/2,0;0,0,0;1,0,0;...
                        0.5,1/(2*3^0.5),(2/3)^0.5];
                [numNodes,junk]=size(nodes);

```

```

% enter masses as: node(row) mass
masses=[1;1;1;1];

% enter ICs as: node(row) x-dir, y-dir, z-dir
dn = [0,0,0,0,0.1,0,0,1,0,1,0,1]';
vn = [0,0,0,0,0,0,0,0,0,0,0,0]';
fixed = zeros(3*numNodes,1);
fext = zeros(3*numNodes,1);

case 'v'

%%% initial velocity %%%%
% enter elements as row: node1 node2 EA Lo
elements = [ 1 2 EA 1; 2 3 EA 1; 1 3 EA 1;...
            1 4 EA 1; 2 4 EA 1; 3 4 EA 1];

[numElements,junk]=size(elements);

% enter nodes as row: Xo, Yo, Zo
nodes = [0.5,3^0.5/2,0;0,0,0;1,0,0;...
        0.5,1/(2*3^0.5),(2/3)^0.5];

[numNodes,junk]=size(nodes);

% enter masses as: node(row) mass
masses=[1;1;1;1];

% enter ICs as: node(row) x-dir, y-dir, z-dir
dn = [0,0,0,0,0,0,0,0,0,0,0,0]';
vn = [0.5,0,0,5,0,0,0,0.5,0,0,0,5]';
fixed = zeros(3*numNodes,1);
fext = zeros(3*numNodes,1);

case 'uv'

%%% initial disp and vel %%%%
% enter elements as row: node1 node2 EA Lo

```

```

elements = [ 1 2 EA 1; 2 3 EA 1; 1 3 EA 1;...
            1 4 EA 1; 2 4 EA 1; 3 4 EA 1];
[numElements,junk]=size(elements);
% enter nodes as row: Xo, Yo, Zo
nodes = [0.5,3^0.5/2,0;0,0,0;...
        1,0,0;0.5,1/(2*3^0.5),(2/3)^0.5];
[numNodes,junk]=size(nodes);
% enter masses as: node(row) mass
masses=[1;1;1;1];
% enter ICs as: node(row) x-dir, y-dir, z-dir
dn = [1,0,0,0,1,0,0,0,0,0,0,1]';
vn = [1,0,0,1,0,0,0,0,0,1,0,1]';
fixed = zeros(3*numNodes,1);
fext = zeros(3*numNodes,1);

end

case 'bell'
    %%%% fourmass problem %%%%
    switch ic
        case 'u'
            %%%% initial displacement %%%%
            % enter elements as row: node1 node2 EA Lo
            elements = [ 1 2 EA 1; 2 3 EA 1; 3 4 EA 1; 1 4 EA 1];
            [numElements,junk]=size(elements);
            % enter nodes as row: Xo, Yo, Zo
            nodes = [0,0,0;1,0,0;1,1,0;0,1,0];
            [numNodes,junk]=size(nodes);

```

```

% enter masses as: node(row) mass
masses=[1;1;1;1];
% enter ICs as: x-dir, y-dir, z-dir for node1:nodeN
% in a column vector
dn = [0,0,0,0,0,0,0,0,0,0,0,0,1]';
vn = [0,0,0,0,0,0,0,0,0,0,0,0,0]';
fixed = zeros(3*numNodes,1);
fext = zeros(3*numNodes,1);
case 'v'
    %%%% initial velocity %%%%
    % enter elements as row: node1 node2 EA Lo
    elements = [ 1 2 EA 1; 2 3 EA 1; 3 4 EA 1; 1 4 EA 1];
    [numElements,junk]=size(elements);
    % enter nodes as row: Xo, Yo, Zo
    nodes = [0,0,0;1,0,0;1,1,0;0,1,0];
    [numNodes,junk]=size(nodes);
    % enter masses as: node(row) mass
    masses=[1;1;1;1];
    % enter ICs as: x-dir, y-dir, z-dir for node1:nodeN
    % in a column vector
    dn = [0,0,0,0,0,0,0,0,0,0,0,0,0]';
    vn = [0.5,0,0,5,0,0,0,0.5,0,0,0,5]';
    fixed = zeros(3*numNodes,1);
    fext = zeros(3*numNodes,1);
case 'uv'
    %%%% initial displ and vel %%%%
    % enter elements as row: node1 node2 EA Lo

```

```

elements = [ 1 2 EA 1; 2 3 EA 1; 3 4 EA 1; 1 4 EA 1];
[numElements,junk]=size(elements);
% enter nodes as row: Xo, Yo, Zo
nodes = [0,0,0;1,0,0;1,1,0;0,1,0];
[numNodes,junk]=size(nodes);
% enter masses as: node(row) mass
masses=[1;1;1;1];
% enter ICs as: x-dir, y-dir, z-dir for node1:nodeN
% in a column vector
dn = [0,0,1,0,0,1,1,0,0,0,0,1]';
vn = [0,0,0,0,0,0,0,0,0,0,0,1]';
fixed = zeros(3*numNodes,1);
fext = zeros(3*numNodes,1);

end

```

```

case 'pend'
    %%% pendulum %%%
    switch ic
        case 'v'
            %%% initial velocity %%%
            % enter elements as row: node1 node2 EA Lo
            elements = [1 2 EA 3.0443];
            [numElements,junk]=size(elements);
            % enter nodes as row: Xo, Yo, Zo
            nodes = [0,0,0;0,-3.0443,0];
            [numNodes,junk]=size(nodes);

```

```
% enter masses as:node(row) mass
masses=[0,10.0005255];
% enter ICs as: x-dir, y-dir, z-dir for node1:nodeN
% in a column vector
dn = [0,0,0,0,0,0]';
vn = [0,0,0,7.725,0,0]';
fixed = [1,1,1,0,0,0];
fext = [0,0,0,0,0,0]';

end

end

return
```

```

function doconvauto(ialgo,rho1,rho2,rho3,convergeType)

%clear all;
%close all;
%clc;

% to get the convergence plot

global ialgo rho1 rho2 rho3 irep t tend ;

format long;

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%% Begin input parameters%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% ialgo = 100;
% rho1 = 0.0;
% rho2 = 0.0;
% rho3 = 0.0;
%rho3 = (1.-rho1*rho2)/(rho1+rho2+2.*rho1*rho2)
irep = 1;
% Converge type: 1=Standard, 2=align at tilda time
% convergeType=2;

% get numerical method parameters
[W,op1,dup1] = gssss (ialgo,rho1,rho2,rho3)

```



```

% if ialgo == 100
%     alpha = (3+rho1+rho2-rho1*rho2)/(2*(1+rho1)*(1+rho2));
% elseif ialgo == 200
%     alpha = 1/(1+rho3);
% end

gamma=W(1)*(1-opl(6));

t = 0;           % initial time
tend = 0.1;     % end time
tol = 1e-14;   % non-linear iteration tolerance
n = [10000, 1000, 500, 250, 125];
%n = [100000, 1000, 500, 250, 150, 100, 50];

for i = 1:5
    if convergeType==2
        timestep(i) = tend/(n(i) + gamma);
    else
        timestep(i) = tend/n(i);
    end
end

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

for i = 1:5
    convresult(i,:) = doconvergence(timestep(i),n(i))

```

```

end

x= timestep(2:5);

% Calculate error for each case, y-axis of convergence plot value

for i=2:5
    errorpos(i-1)=abs((convresult(1,2)-convresult(i,2))/convresult(1,2));
end
for i=2:5
    errorvel(i-1)=abs((convresult(1,3)-convresult(i,3))/convresult(1,3));
end
for i=2:5
    erroracc(i-1)=abs((convresult(1,4)-convresult(i,4))/convresult(1,4));
end

% Print slope of each of the lines, leftmost 2 points used to calculate
slopepos=(log10(errorpos(2))-log10(errorpos(1)))/(log10(x(2))-log10(x(1)))
slopevel=(log10(errorvel(2))-log10(errorvel(1)))/(log10(x(2))-log10(x(1)))
slopeacc=(log10(erroracc(2))-log10(erroracc(1)))/(log10(x(2))-log10(x(1)))

```

```

figure
if convergeType==1
    str=sprintf('tet_%d_%1.2f_%1.2f_%1.2fstandard.eps',ialgo,rho1
    ,rho2,rho3);
    loglog(x,errorpos,'*-',x,errorvel,'*-',x,erroracc,'*-')
    xlabel('Log(\Deltat)')
    ylabel('Log(Error)')
    legend('u','v','a')
end
if convergeType==2
    str=sprintf('tet_%d_%1.2f_%1.2f_%1.2faligned.eps',ialgo,rho1,
    rho2,rho3);
    loglog(x,erroracc,'r*-')
    xlabel('Log(\Deltat)')
    ylabel('Log(Error)')
    legend('a')
end

saveas(gcf,str,'epsc')

```

```

% Subroutine for convergence plot

% Edited 3/14/07 ajh

% This plots the corrected acceleration convergence plot for velo
city based

% scheme based on the nonlinear alpha term

function result = doconvergence(passedh,n)

% declare all the global variables here
global fixed A b W opl dupl h Xo elements nodes numNodes numEleme
nts fext masses tend displacement;
global dp dn vn an dh position velocity totalenergy ke pe linearm
omentum angularmomentum time
global ialgo rho1 rho2 rho3 irep t;

[dontcare1,dontcare2,dontcare3,dontcare4,dontcare5,tol,t,dontcare
6,dontcare7] = inputParams;
% Overwrite h from inputParams
h=passedh;

% formulate the mass matrix
mass = zeros(numNodes*3,numNodes*3);
for i=1:numNodes
    mass(3*i-2,3*i-2)=masses(i);
    mass(3*i-1,3*i-1)=masses(i);
    mass(3*i,3*i)=masses(i);

```

```

end

% initialize strain matrices they use
% A is general to any element
A=1/4*[eye(3,3) -1*eye(3,3);-1*eye(3,3) eye(3,3)];

% b is element specific
for i = 1:numElements
    xo1=nodes(elements(i,1),1);
    yo1=nodes(elements(i,1),2);
    zo1=nodes(elements(i,1),3);
    xo2=nodes(elements(i,2),1);
    yo2=nodes(elements(i,2),2);
    zo2=nodes(elements(i,2),3);
    b(1:6,i)=4/elements(i,4)^2*A*[xo1;yo1;zo1;xo2;yo2;zo2];
end

Xo=zeros(numNodes,1);
for i=1:numNodes
    Xo(3*i-2:3*i)=[nodes(i,1);nodes(i,2);nodes(i,3)];
end

% calculate initial accelerations
fint=zeros(numNodes*3,1);

for i = 1:numElements
    node1=elements(i,1);

```

```

node2=elements(i,2);
un(1:3,1)=dn(node1*3-2:node1*3);
un(4:6,1)=dn(node2*3-2:node2*3);
Lo=elements(i,4);
EA=elements(i,3);
strainn=b(:,i)'un+2/Lo2*un'*A*un;
elementFint=EA*Lo*(b(:,i)+4/Lo2*A*un)*strainn;
fint((node1*3-2):(node1*3),1)=fint((node1*3-2):(node1*3),1)+e
lementFint(1:3,1);
fint((node2*3-2):(node2*3),1)=fint((node2*3-2):(node2*3),1)+e
lementFint(4:6,1);
end

% remove rows/cols for fixed nodes
massSolve=mass;
for i=numNodes*3:-1:1
    if fixed(i)==1
        massSolve(i,:)=[];
        massSolve(:,i)=[];
    end
end
fextSolve=fext;
for i=numNodes*3:-1:1
    if fixed(i)==1
        fextSolve(i)=[];
    end
end
end

```

```

fintSolve=fint;
for i=numNodes*3:-1:1
    if fixed(i)==1
        fintSolve(i)=[];
    end
end

anSolve = inv(massSolve)*(fextSolve - fintSolve);

% put back zeros in an
an=zeros(numNodes,1);
numFixed=0;
for i=1:numNodes*3
    if fixed(i)==0
        an(i)=anSolve(i-numFixed);
    else
        numFixed=numFixed+1;
    end
end

% initialize result vectors
linearmomentum=zeros(1,3);
angularmomentum=zeros(1,3);

% get numerical method parameters
[W,opl,dupl] = gssss (ialgo,rho1,rho2,rho3)

```

```

% get the predictor corrector coeff
[xsip,xsic] = getXsi(W,opl,dupl,h,irep)

istep = 0;

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%% START THE TIME INTEGRATION PROCEDURE %%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

while (istep<n)
    istep = istep + 1;
    t = h*istep

    % cut down dn, vn for fixed nodes (an already proper size
    % from calculating it)
    dnSolve=dn;
    vnSolve=vn;
    anSolve=an;
    for i=numNodes*3:-1:1
        if fixed(i)==1
            dnSolve(i)=[];
            vnSolve(i)=[];
            anSolve(i)=[];
        end
    end

    % Predict    %

```



```

dpSolve = xsip(1,1)*dnSolve + xsip(1,2)*vnSolve + xsip(1,
3)*anSolve;
dhSolve = xsip(2,1)*dnSolve + xsip(2,2)*vnSolve + xsip(2,
3)*anSolve;
vhSolve = xsip(3,1)*dnSolve + xsip(3,2)*vnSolve + xsip(3,
3)*anSolve;
ahSolve = xsip(4,1)*dnSolve + xsip(4,2)*vnSolve + xsip(4,
3)*anSolve;

```

```

% start the N/L Iteration %

```

```

converge = 0;

```

```

nliter=0;

```

```

while (converge ~= 1)

```

```

    nliter=nliter+1;

```

```

    % put back zeros in dh, and dp needed for total ktang
    and fint

```

```

    dh=zeros(numNodes,1);

```

```

    dp=zeros(numNodes,1);

```

```

    numFixed=0;

```

```

    for i=1:numNodes*3

```

```

        if fixed(i)==0

```

```

            dh(i)=dhSolve(i-numFixed);

```

```

            dp(i)=dpSolve(i-numFixed);

```

```

        else

```

```

        numFixed=numFixed+1;
    end
end

% get the tangent stiffness matrix and internal force
[fint, ktang] = getFandK;

ktangSolve=ktang;
for i=numNodes*3:-1:1
    if fixed(i)==1
        ktangSolve(i,:)=[];
        ktangSolve(:,i)=[];
    end
end

fintSolve=fint;
for i=numNodes*3:-1:1
    if fixed(i)==1
        fintSolve(i)=[];
    end
end

% form the jacobian matrix %
jac = xsic(4)*massSolve + ktangSolve; % this is follo
wing the 2d code, different from thesis page 68

% get the residual vector %

```

```

res = fextSolve - massSolve*ahSolve - fintSolve;

% solve for the delta    %
del = inv(jac)*res;

atol = norm(del);

% Correct %
dpSolve = dpSolve + xsic(1)*del;
dhSolve = dhSolve + xsic(2)*del;
vhSolve = vhSolve + xsic(3)*del;
ahSolve = ahSolve + xsic(4)*del;

% check for convergence %
if (atol<tol)
    nliter;
    converge = 1;
end

end % end of nonlinear iteration (solution converged)

% need to stick ahSolve into a non-reduced ah for final u
pdates
ah=zeros(numNodes,1);
numFixed=0;
for i=1:numNodes*3

```

```

        if fixed(i)==0
            ah(i)=ahSolve(i-numFixed);
        else
            numFixed=numFixed+1;
        end
    end
end

% Design updates at the end of time step %
ap = an + (ah-an)/opl(6)/W(1);
vp = vn + dupl(4)*an*h + dupl(5)*(ap-an)*h;
dp = dn + dupl(1)*vn*h + dupl(2)*an*h*h + dupl(3)*(ap-an)
*h*h;

% Get ready to the next time step %
dn = dp;
vn = vp;
an = ap;

if n==istep
    curpos = Xo + dp;
    pos2 = (curpos(4)^2 + curpos(5)^2 + curpos(6)^2)^0.5;
    vel2 = (vp(4)^2 + vp(5)^2 + vp(6)^2)^0.5;
    accel2 = (ap(4)^2 + ap(5)^2 + ap(6)^2)^0.5;
    result = [h,pos2,vel2,accel2,t];
end

```

end

```

function fext=getfext(dt,tn1)

fext=zeros(10,1);

tn=tn1-dt;

if tn1<0.25
    fn1=25/0.25*tn1;
elseif tn1<0.5
    fn1=-50/0.5*tn1+50;
else
    fn1=0;
end

if tn<0.25
    fn=25/0.25*tn;
elseif tn<0.5
    fn=-50/0.5*tn+50;
else
    fn=0;
end

fext(10,1)=-((fn+fn1)/2);

function[fint,beta_spring]=getfint_armero_EMM(dn,dn1,Ka,L0)

Rn1=sqrt((dn1(4)-dn1(2))^2+(dn1(3)-dn1(1))^2);

```

```

Rn=sqrt((dn(4)-dn(2))^2+(dn(3)-dn(1))^2);

PEn1=1/2*Ka*(Rn1-L0)^2;
PEn=1/2*Ka*(Rn-L0)^2;

% dir_n=[(dn(3)-dn(1));(dn(4)-dn(2))];
% dir_n1=[(dn1(3)-dn1(1));(dn1(4)-dn1(2))];

r1x = (dn1(3) + dn(3))/2 - (dn1(1) + dn(1))/2;
r1y = (dn1(4) + dn(4))/2 - (dn1(2) + dn(2))/2;

if abs(Rn1-Rn)<1e-14
    Rn_half=(Rn1+Rn)/2;
    beta_spring=0;
    fint(1,1)=Ka*(Rn_half-L0)*(-r1x)/Rn_half;
    fint(2,1)=Ka*(Rn_half-L0)*(-r1y)/Rn_half;
    fint(3,1)=Ka*(Rn_half-L0)*(r1x)/Rn_half;
    fint(4,1)=Ka*(Rn_half-L0)*(r1y)/Rn_half;
else
    %beta_spring=((PEn1-PEn)/(Rn1-Rn)*(-r1x)*2/(Rn1 +
    %Rn))+Ka*L0-Ka*Rn)/(Ka*Rn1-Ka*Rn);
    RHS=(PEn1-PEn)/(Rn1-Rn);
    beta_spring=(-1*Ka*L0+Ka*Rn-RHS)/(Ka*(Rn-Rn1));
    fint(1,1)=(PEn1-PEn)/(Rn1-Rn)*(-r1x)*2/(Rn1 + Rn);
    fint(2,1)=(PEn1-PEn)/(Rn1-Rn)*(-r1y)*2/(Rn1 + Rn);
    fint(3,1)=(PEn1-PEn)/(Rn1-Rn)*(r1x)*2/(Rn1 + Rn);
    fint(4,1)=(PEn1-PEn)/(Rn1-Rn)*(r1y)*2/(Rn1 + Rn);

```

end

```
function[fint,beta_beam]=getfintbeam_armero_EMM(dn,dn1,Cb)
```

```
x1n=dn(1);
```

```
y1n=dn(2);
```

```
x2n=dn(3);
```

```
y2n=dn(4);
```

```
x3n=dn(5);
```

```
y3n=dn(6);
```

```
x1n1=dn1(1);
```

```
y1n1=dn1(2);
```

```
x2n1=dn1(3);
```

```
y2n1=dn1(4);
```

```
x3n1=dn1(5);
```

```
y3n1=dn1(6);
```

```
L1n1=sqrt((y3n1-y1n1)^2+(x3n1-x1n1)^2);
```

```
L2n1=sqrt((y3n1-y2n1)^2+(x3n1-x2n1)^2);
```

```
L1n=sqrt((y3n-y1n)^2+(x3n-x1n)^2);
```

```
L2n=sqrt((y3n-y2n)^2+(x3n-x2n)^2);
```

```
nu_n=L1n*L2n;
```

```
nu_n1=L1n1*L2n1;
```

```
nu_nhalf=(nu_n1+nu_n)/2;
```



```

lambda_n=(x1n-x3n)*(x2n-x3n)+(y1n-y3n)*(y2n-y3n);
lambda_n1=(x1n1-x3n1)*(x2n1-x3n1)+(y1n1-y3n1)*(y2n1-y3n1);
lambda_nhalf=(lambda_n1+lambda_n)/2;

L1bar=(L1n1+L1n)/2;
L2bar=(L2n1+L2n)/2;

r1x=(x1n1+x1n)/2-(x3n1+x3n)/2;
r1y=(y1n1+y1n)/2-(y3n1+y3n)/2;
r2x=(x2n1+x2n)/2-(x3n1+x3n)/2;
r2y=(y2n1+y2n)/2-(y3n1+y3n)/2;

fint=zeros(6,1);

if (abs(lambda_n1-lambda_n)<1e-14 || abs(nu_n1-nu_n)<1e-14)

    dlv=Cb/(2*(nu_nhalf-lambda_nhalf))+Cb*(lambda_nhalf+nu_nhalf)
    /(2*(nu_nhalf-lambda_nhalf)^2);
    dln=Cb/(2*(nu_nhalf-lambda_nhalf))-Cb*(lambda_nhalf+nu_nhalf)
    /(2*(nu_nhalf-lambda_nhalf)^2);

    fint(1,1)=dlv*r2x+dln*(L2bar/L1bar*r1x);
    fint(2,1)=dlv*r2y+dln*(L2bar/L1bar*r1y);
    fint(3,1)=dlv*r1x+dln*(L1bar/L2bar*r2x);
    fint(4,1)=dlv*r1y+dln*(L1bar/L2bar*r2y);
    fint(5,1)=dlv*(-r1x-r2x)+dln*(L2bar/L1bar*-r1x+L1bar/L2bar*-r2x);

```

```

fint(6,1)=dlv*(-r1y-r2y)+dln*(L2bar/L1bar*-r1y+L1bar/L2bar*-r2y);

beta_beam=0;

% disp('fint from singularity')
% fint
% pause

else

V11=1/2*Cb*(nu_n1+lambda_n1)/(nu_n1-lambda_n1);
V00=1/2*Cb*(nu_n+lambda_n)/(nu_n-lambda_n);
V10=1/2*Cb*(nu_n+lambda_n1)/(nu_n-lambda_n1);
V01=1/2*Cb*(nu_n1+lambda_n)/(nu_n1-lambda_n);

fint1a=(1/2*(V11+V10)-1/2*(V01+V00))/(lambda_n1-lambda_n)*r2x;
fint2a=(1/2*(V11+V10)-1/2*(V01+V00))/(lambda_n1-lambda_n)*r2y;
fint3a=(1/2*(V11+V10)-1/2*(V01+V00))/(lambda_n1-lambda_n)*r1x;
fint4a=(1/2*(V11+V10)-1/2*(V01+V00))/(lambda_n1-lambda_n)*r1y;
fint5a=(1/2*(V11+V10)-1/2*(V01+V00))/(lambda_n1-lambda_n)*(-r1x-r2x);
fint6a=(1/2*(V11+V10)-1/2*(V01+V00))/(lambda_n1-lambda_n)*(-r1y-r2y);

fint1b=(1/2*(V11+V01)-1/2*(V10+V00))/(nu_n1-nu_n)*
(L2bar/L1bar*r1x);
fint2b=(1/2*(V11+V01)-1/2*(V10+V00))/(nu_n1-nu_n)*
(L2bar/L1bar*r1y);
fint3b=(1/2*(V11+V01)-1/2*(V10+V00))/(nu_n1-nu_n)*
(L1bar/L2bar*r2x);
fint4b=(1/2*(V11+V01)-1/2*(V10+V00))/(nu_n1-nu_n)*
(L1bar/L2bar*r2y);

```

```

fint5b=(1/2*(V11+V01)-1/2*(V10+V00))/(nu_n1-nu_n)*
(L2bar/L1bar*-r1x+L1bar/L2bar*-r2x);
fint6b=(1/2*(V11+V01)-1/2*(V10+V00))/(nu_n1-nu_n)*
(L2bar/L1bar*-r1y+L1bar/L2bar*-r2y);

fint(1,1)=fint1a+fint1b;
fint(2,1)=fint2a+fint2b;
fint(3,1)=fint3a+fint3b;
fint(4,1)=fint4a+fint4b;
fint(5,1)=fint5a+fint5b;
fint(6,1)=fint6a+fint6b;

% to calc beta
RHS=(1/2*(V11+V10)-1/2*(V01+V00))/(lambda_n1-lambda_n)+
(1/2*(V11+V01)-1/2*(V10+V00))/(nu_n1-nu_n);
beta_beam=(Cb+RHS*lambda_n-RHS*nu_n)/(RHS*
(lambda_n-lambda_n1-nu_n+nu_n1));

end

%function to iterate on the value of c

function[fint,c]=getfintbeam_c_iteration(dn,dn1,Cb)

tol=1e-8;

x1n=dn(1);

```

$$y_{1n} = dn(2);$$

$$x_{2n} = dn(3);$$

$$y_{2n} = dn(4);$$

$$x_{3n} = dn(5);$$

$$y_{3n} = dn(6);$$

$$x_{1n1} = dn1(1);$$

$$y_{1n1} = dn1(2);$$

$$x_{2n1} = dn1(3);$$

$$y_{2n1} = dn1(4);$$

$$x_{3n1} = dn1(5);$$

$$y_{3n1} = dn1(6);$$

$$L_{1n1} = \text{sqrt}((y_{2n1} - y_{1n1})^2 + (x_{2n1} - x_{1n1})^2);$$

$$L_{2n1} = \text{sqrt}((y_{3n1} - y_{2n1})^2 + (x_{3n1} - x_{2n1})^2);$$

$$L_{1n} = \text{sqrt}((y_{2n} - y_{1n})^2 + (x_{2n} - x_{1n})^2);$$

$$L_{2n} = \text{sqrt}((y_{3n} - y_{2n})^2 + (x_{3n} - x_{2n})^2);$$

$$nu_n = L_{1n} * L_{2n};$$

$$nu_n1 = L_{1n1} * L_{2n1};$$

$$lambda_n = (x_{1n} - x_{2n}) * (x_{3n} - x_{2n}) + (y_{1n} - y_{2n}) * (y_{3n} - y_{2n});$$

$$lambda_n1 = (x_{1n1} - x_{2n1}) * (x_{3n1} - x_{2n1}) + (y_{1n1} - y_{2n1}) * (y_{3n1} - y_{2n1});$$

$$PE_{n1} = 1/2 * C_b * (nu_n1 + lambda_n1) / (nu_n1 - lambda_n1);$$

$$PE_n = 1/2 * C_b * (nu_n + lambda_n) / (nu_n - lambda_n);$$

```

% Guess that c=0.5 for starting guess
c=0.5;
converged=0;
citer=0;

%%%% Start iteration %%%%

while(converged==0)
    citer=citer+1;

    resC=see: mathetmatica

    if (abs(resC)<tol) %&&abs(delC)<tol)
        converged=1;
    else

        jacC=see: mathematica
        delC=-resC/jacC;
        c=c+delC;

    end

    if mod(citer,1000)==0

```

```

        c=-999;
        fint=[-999 -999 -999 -999 -999 -999]';
        return
    end

end

x1nc=(1-c)*x1n+c*x1n1;
y1nc=(1-c)*y1n+c*y1n1;
x2nc=(1-c)*x2n+c*x2n1;
y2nc=(1-c)*y2n+c*y2n1;
x3nc=(1-c)*x3n+c*x3n1;
y3nc=(1-c)*y3n+c*y3n1;

var1=sqrt((-x2nc+x3nc)^2+(-y2nc+y3nc)^2);
var2=sqrt((-x1nc+x2nc)^2+(-y1nc+y2nc)^2);
fint(1,1)=(Cb*(-x2nc+x3nc-((-x1nc+x2nc)*var1)/var2))/(2*(-((x1nc-x2nc)*(-x2nc+x3nc))-
(y1nc-y2nc)*(-y2nc+y3nc)+var2*var1))-
(Cb*(x2nc-x3nc-((-x1nc+x2nc)*var1)/var2)*((x1nc-x2nc)*(-x2nc+x3nc)+(y1nc-y2nc)*(-y2nc+y3nc)+var2*var1))/
(2*(-((x1nc-x2nc)*(-x2nc+x3nc))-
(y1nc-y2nc)*(-y2nc+y3nc)+var2*var1)^2);
fint(2,1)=(Cb*(-y2nc+y3nc-((-y1nc+y2nc)*var1)/var2))/(2*(-((x1nc-x2nc)*(-x2nc+x3nc))-
(y1nc-y2nc)*(-y2nc+y3nc)+var2*var1))-
(Cb*(y2nc-y3nc-((-y1nc+y2nc)*var1)/var2)*((x1nc-x2nc)*(-x2nc+x3nc)+(y1nc-y2nc)*(-y2nc+y3nc)+var2*var1))/
(2*(-((x1nc-x2nc)*(-x2nc+x3nc))-
(y1nc-y2nc)*(-y2nc+y3nc)+var2*var1)^2);
fint(3,1)=(Cb*(-x1nc+2*x2nc-x3nc-((-x2nc+x3nc)*var2)/var1+((-x1nc+x2nc)

```

```

*var1)/var2))/(2*(-((x1nc-x2nc)*(-x2nc+x3nc))-(y1nc-y2nc)*(-y2nc+y3nc)
+var2*var1))-(Cb*(x1nc-2*x2nc+x3nc-((-x2nc+x3nc)*var2)/var1+((-x1nc+x2nc)
*var1)/var2)*((x1nc-x2nc)*(-x2nc+x3nc)+(y1nc-y2nc)*(-y2nc+y3nc)+var2*var1
))/((2*(-((x1nc-x2nc)*(-x2nc+x3nc))-(y1nc-y2nc)*(-y2nc+y3nc)+var2*var1)^2);
fint(4,1)=(Cb*(-y1nc+2*y2nc-y3nc-(var2*(-y2nc+y3nc))/var1+((-y1nc+y2nc)
*var1)/var2))/(2*(-((x1nc-x2nc)*(-x2nc+x3nc))-(y1nc-y2nc)*(-y2nc+y3nc)
+var2*var1))-(Cb*(y1nc-2*y2nc+y3nc-(var2*(-y2nc+y3nc))/var1+((-y1nc+
y2ncvar1)/var2)*((x1nc-x2nc)*(-x2nc+x3nc)+(y1nc-y2nc)*(-y2nc+y3nc)+
var2*var1))/((2*(-((x1nc-x2nc)*(-x2nc+x3nc))-(y1nc-y2nc)*(-y2nc+y3nc)
+var2*var1)^2);
fint(5,1)=(Cb*(x1nc-x2nc+((-x2nc+x3nc)*var2)/var1))/(2*(-((x1nc-x2nc)
*(-x2nc+x3nc))-(y1nc-y2nc)*(-y2nc+y3nc)+var2*var1))-(Cb*(-x1nc+x2nc+
((-x2nc+x3nc)*var2)/var1)*((x1nc-x2nc)*(-x2nc+x3nc)+(y1nc-y2nc)*(-
-y2nc+y3nc)+var2*var1))/((2*(-((x1nc-x2nc)*(-x2nc+x3nc))-(y1nc-y2nc)
*(-y2nc+y3nc)+var2*var1)^2);
fint(6,1)=(Cb*(y1nc-y2nc+(var2*(-y2nc+y3nc))/var1))/(2*(-((x1nc-x2nc)
*(-x2nc+x3nc))-(y1nc-y2nc)*(-y2nc+y3nc)+var2*var1))-(Cb*(-y1nc+y2nc
+(var2*(-y2nc+y3nc))/var1)*((x1nc-x2nc)*(-x2nc+x3nc)+(y1nc-y2nc)*
(-y2nc+y3nc)+var2*var1))/((2*(-((x1nc-x2nc)*(-x2nc+x3nc))-(y1nc-y2nc)
*(-y2nc+y3nc)+var2*var1)^2);

```

```
function[fint]=getfint_beam_gs4(dh,Cb)
```

```
x1h=dh(1);
```

```
y1h=dh(2);
```

```
x2h=dh(3);
```

```
y2h=dh(4);
```

```

x3h=dh(5);
y3h=dh(6);

L1h=sqrt((y3h-y1h)^2+(x3h-x1h)^2);
L2h=sqrt((y3h-y2h)^2+(x3h-x2h)^2);

nu_h=L1h*L2h;

lambda_h=(x1h-x3h)*(x2h-x3h)+(y1h-y3h)*(y2h-y3h);

r1x = x1h - x3h;
r1y = y1h - y3h;
r2x = x2h - x3h;
r2y = y2h - y3h;

fint=zeros(6,1);

dlv=Cb/(2*(nu_h-lambda_h))+Cb*(lambda_h+nu_h)/(2*(nu_h-lambda_h)^2);
dln=Cb/(2*(nu_h-lambda_h))-Cb*(lambda_h+nu_h)/(2*(nu_h-lambda_h)^2);

fint(1,1)=dlv*r2x+dln*(L2h/L1h*r1x);
fint(2,1)=dlv*r2y+dln*(L2h/L1h*r1y);
fint(3,1)=dlv*r1x+dln*(L1h/L2h*r2x);
fint(4,1)=dlv*r1y+dln*(L1h/L2h*r2y);
fint(5,1)=dlv*(-r1x-r2x)+dln*(L2h/L1h*-r1x+L1h/L2h*-r2x);
fint(6,1)=dlv*(-r1y-r2y)+dln*(L2h/L1h*-r1y+L1h/L2h*-r2y);

```



```

%function to iterate on the value of c

function[fint,c]=getfint_c_iteration(dn,dn1,Ka,L0)

tol=1e-12;

Rn1=sqrt((dn1(4)-dn1(2))^2+(dn1(3)-dn1(1))^2);
Rn=sqrt((dn(4)-dn(2))^2+(dn(3)-dn(1))^2);

PEn1=1/2*Ka*(Rn1-L0)^2;
PEn=1/2*Ka*(Rn-L0)^2;

x1n=dn(1);
y1n=dn(2);
x2n=dn(3);
y2n=dn(4);

x1n1=dn1(1);
y1n1=dn1(2);
x2n1=dn1(3);
y2n1=dn1(4);

% Guess that c=0.5 for starting guess
c=0.5;
converged=0;
citer=0;

```

```

%%%%% Start iteration %%%%%

while(converged==0)
    citer=citer+1;
    if (abs(resC)<tol) %&&abs(delC)<tol)
        converged=1;
    else

        jacC=see: mathematica
        delC=-resC/jacC;
        c=c+delC;

    end

    if mod(citer,1000)==0

        c=-999;
        fint=['-999 -999 -999 -999 -999 -999'];
        return
    end

end

end

xnc1=(1-c)*x1n+c*x1n1;

```

```

ync1=(1-c)*y1n+c*y1n1;
xnc2=(1-c)*x2n+c*x2n1;
ync2=(1-c)*y2n+c*y2n1;
Rnc=sqrt((xnc2-xnc1)^2+(ync2-ync1)^2);
fint(1,1)=-Ka*(xnc2-xnc1)*(Rnc-L0)/Rnc;
fint(2,1)=-Ka*(ync2-ync1)*(Rnc-L0)/Rnc;

fint(3,1)=Ka*(xnc2-xnc1)*(Rnc-L0)/Rnc;
fint(4,1)=Ka*(ync2-ync1)*(Rnc-L0)/Rnc;

function[fint]=getfint_spring_gs4(dh,Ka,L0)

L=sqrt((dh(4)-dh(2))^2+(dh(3)-dh(1))^2);

r1x = dh(3) - dh(1);
r1y = dh(4) - dh(2);

fint(1,1)=Ka*(L-L0)*(-r1x)/L;
fint(2,1)=Ka*(L-L0)*(-r1y)/L;
fint(3,1)=Ka*(L-L0)*(r1x)/L;
fint(4,1)=Ka*(L-L0)*(r1y)/L;

function J=getJ_armero_EMM(dn,dn1,dt,Ka,L0,M)

x1n1=dn1(1);
y1n1=dn1(2);
x2n1=dn1(3);

```

```

y2n1=dn1(4);

x1n=dn(1);
y1n=dn(2);
x2n=dn(3);
y2n=dn(4);

Rn1=sqrt((dn1(4)-dn1(2))^2+(dn1(3)-dn1(1))^2);
Rn=sqrt((dn(4)-dn(2))^2+(dn(3)-dn(1))^2);

if abs(Rn1-Rn)<1e-8

    var1=sqrt(((x1n-x1n1)/2+(x2n+x2n1)/2)^2+((-y1n-y1n1)/2+(y2n+
    y2n1)/2)^2);
    var2=sqrt((-x1n+x2n)^2+(-y1n+y2n)^2);
    var3=sqrt((-x1n1+x2n1)^2+(-y1n1+y2n1)^2);

    J=see: mathematica
%    disp('used singular one')
else

    var1=sqrt((-x1n1+x2n1)^2+(-y1n1+y2n1)^2);
    var2=sqrt((-x1n+x2n)^2+(-y1n+y2n)^2);

    J=see: mathematica
end

```

```

function J=getJbeam_armero_EMM(dn,dn1,dt,Cb,M)

x1n=dn(1);
y1n=dn(2);
x2n=dn(3);
y2n=dn(4);
x3n=dn(5);
y3n=dn(6);

x1n1=dn1(1);
y1n1=dn1(2);
x2n1=dn1(3);
y2n1=dn1(4);
x3n1=dn1(5);
y3n1=dn1(6);

L1n=sqrt((y3n-y1n)^2+(x3n-x1n)^2);
L2n=sqrt((y3n-y2n)^2+(x3n-x2n)^2);

L1n1=sqrt((y3n1-y1n1)^2+(x3n1-x1n1)^2);
L2n1=sqrt((y3n1-y2n1)^2+(x3n1-x2n1)^2);

nu_n=L1n*L2n;
nu_n1=L1n1*L2n1;

lambda_n=(x1n-x3n)*(x2n-x3n)+(y1n-y3n)*(y2n-y3n);

```

```
lambda_n1=(x1n1-x3n1)*(x2n1-x3n1)+(y1n1-y3n1)*(y2n1-y3n1);
```

```
if (abs(lambda_n1-lambda_n)<1e-8 || abs(nu_n1-nu_n)<1e-8)
```

```
    var1=sqrt((-x2n1+x3n1)^2+(-y2n1+y3n1)^2);
```

```
    var2=sqrt((-x1n1+x3n1)^2+(-y1n1+y3n1)^2);
```

```
    var3=sqrt((-x2n+x3n)^2+(-y2n+y3n)^2);
```

```
    var4=sqrt((-x1n+x3n)^2+(-y1n+y3n)^2);
```

```
    J=see: mathematica
```

```
else
```

```
    var1=sqrt((-x2n1+x3n1)^2+(-y2n1+y3n1)^2);
```

```
    var2=sqrt((-x1n1+x3n1)^2+(-y1n1+y3n1)^2);
```

```
    var3=sqrt((-x1n+x3n)^2+(-y1n+y3n)^2);
```

```
    var4=sqrt((-x2n+x3n)^2+(-y2n+y3n)^2);
```

```
    J=see: mathematica
```

```
end
```

```
function J=getJbeam_c_iteration(dn,dn1,dt,c,Cb,M)
```

```
x1n=dn(1);
```

```
y1n=dn(2);
```

```
x2n=dn(3);
```

```
y2n=dn(4);
```

```
x3n=dn(5);
```

```
y3n=dn(6);
```

```
x1n1=dn1(1);
```

```
y1n1=dn1(2);
```

```
x2n1=dn1(3);
```

```
y2n1=dn1(4);
```

```
x3n1=dn1(5);
```

```
y3n1=dn1(6);
```

```
J=see: mathematica
```

```
function J=getJ_beam_gs4(dn,dn1,vn,an,dt,Cb,W,opl,dupl)
```

```
x1n1=dn1(1);
```

```
y1n1=dn1(2);
```

```
x2n1=dn1(3);
```

```
y2n1=dn1(4);
```

```
x3n1=dn1(5);
```

```
y3n1=dn1(6);
```

```
x1n=dn(1);
```

```
y1n=dn(2);
```

```
x2n=dn(3);
```

```
y2n=dn(4);
```

x3n=dn(5);

y3n=dn(6);

v1xn=vn(1);

v1yn=vn(2);

v2xn=vn(3);

v2yn=vn(4);

v3xn=vn(5);

v3yn=vn(6);

a1xn=an(1);

a1yn=an(2);

a2xn=an(3);

a2yn=an(4);

a3xn=an(5);

a3yn=an(6);

W1=W(1);

W2=W(2);

W3=W(3);

L1=opl(1);

L2=opl(2);

L3=opl(3);

l1=dupl(1);

l2=dupl(2);


```
l3=dupl(3);
```

```
J=see: mathematica
```

```
function J=getJ_c_iteration(dn,dn1,dt,c,Ka,L0,M)
```

```
x1n1=dn1(1);
```

```
y1n1=dn1(2);
```

```
x2n1=dn1(3);
```

```
y2n1=dn1(4);
```

```
x1n=dn(1);
```

```
y1n=dn(2);
```

```
x2n=dn(3);
```

```
y2n=dn(4);
```

```
J=see: mathematica
```

```
function J=getJ_spring_gs4(dn,dn1,vn,an,dt,K,L0,W,opl,dupl)
```

```
W1=W(1);
```

```
W2=W(2);
```

```
W3=W(3);
```

```
L1=opl(1);
```

```
L2=opl(2);
```

```
L3=op1(3);
% L4=op1(4);
% L5=op1(5);
% L6=op1(6);

l1=dup1(1);
l2=dup1(2);
l3=dup1(3);
% l4=dup1(4);
% l5=dup1(5);

x1n1=dn1(1);
y1n1=dn1(2);
x2n1=dn1(3);
y2n1=dn1(4);

x1n=dn(1);
y1n=dn(2);
x2n=dn(3);
y2n=dn(4);

v1xn=vn(1);
v1yn=vn(2);
v2xn=vn(3);
v2yn=vn(4);

a1xn=an(1);
```

```

a1yn=an(2);
a2xn=an(3);
a2yn=an(4);

J=see: mathematica

end

function [PE axial bending soft_bar rigid_bar]=getPE(d,elements)

[numElm,cols]=size(elements);

PE=0;
axial=0;
bending=0;
rigid_bar=0;
soft_bar=0;
for i=1:numElm
    % spring potential
    if elements(i,3)==-1
        node_i=elements(i,1);
        node_j=elements(i,2);
        L=sqrt((d(node_j*2-1)-d(node_i*2-1))^2+(d(node_j*2)-
        d(node_i*2))^2);
        PE=PE+1/2*elements(i,5)*(L-elements(i,4))^2;
        axial=axial+1/2*elements(i,5)*(L-elements(i,4))^2;
    end
end

```

```

if elements(i,5) > 500
    rigid_bar=rigid_bar+1/2*elements(i,5)*(L-elements(i,4))^2;
else
    soft_bar=soft_bar+1/2*elements(i,5)*(L-elements(i,4))^2;
end
% beam potential
else
    node_i=elements(i,1);
    node_j=elements(i,2);
    node_k=elements(i,3);
    x1=d(node_i*2-1);
    y1=d(node_i*2);
    x2=d(node_j*2-1);
    y2=d(node_j*2);
    x3=d(node_k*2-1);
    y3=d(node_k*2);

    L2=sqrt((y3-y2)^2+(x3-x2)^2);
    L1=sqrt((y3-y1)^2+(x3-x1)^2);

    nu=L1*L2;
    lambda=(x1-x3)*(x2-x3)+(y1-y3)*(y2-y3);

    PE=PE+1/2*elements(i,5)*(nu+lambda)/(nu-lambda);
    bending=bending+1/2*elements(i,5)*(nu+lambda)/(nu-lambda);

end

```

```

end

function [nodes,elements,dt,tend,rho1,rho2,rho3,tol,ialgo,problem
        , method]=inputParams

global initial_stiffness

%%%%% Input values %%%%%
problem = 'armero'; % either 'pend', 'beam', or 'armero'
dt = 0.05;
tend = 20;
rho1=0.9; % of course only applicable if using GS4 options (2 or 3)
rho2=1;
rho3=0.9;
ialgo=100;
tol=1e-6;
initial_stiffness=0; % 1 means use initial J, 0 means always update J
method = 1; % 1=greenspan only, 2=gs4 only, 3=hybrid of the two,
% 4=c iteration
%%%%% End input values %%%%%

% No need to change anything below here
switch problem
    % Enter nodes as row vector [x_pos, y_pos, x_vel, y_vel, mass, fixed_x,
    % fixed_y (1=fixed 0=free)] Entere elements as: Spring: [node_i node_j
    % -1 L0 Ka] Beam: [node_i node_j noke_k -1 Cb] if node_k = -1, spring
    % element, otherwise assumed beam

```

```

case 'armero'
    % Armero's problem %
    nodes(1,:)= [0 0 0 0 2 1 1];
    nodes(2,:)= [0 1 0 0 2 0 0];
    nodes(3,:)= [0.25 1 0 0 0.2 0 0];
    nodes(4,:)= [0.5 1 0 0 0.2 0 0];
    nodes(5,:)= [0.75 1 0 0 0.2 0 0];
    nodes(6,:)= [1 1 0 0 2 0 0];
    nodes(7,:)= [2 1 0 0 2 1 1];
    elements(1,:)= [1 2 -1 1 1e6];
    elements(2,:)= [2 3 -1 0.25 500];
    elements(3,:)= [3 4 -1 0.25 500];
    elements(4,:)= [4 5 -1 0.25 500];
    elements(5,:)= [5 6 -1 0.25 500];
    elements(6,:)= [6 7 -1 1 1e6];
    elements(7,:)= [2 4 3 -1 10];
    elements(8,:)= [3 5 4 -1 10];
    elements(9,:)= [4 6 5 -1 10];

case 'pend'
    % Pendulum problem %
    nodes(1,:)= [10 0 0 0 2 1 1];
    nodes(2,:)= [10 10 -10 0 2 0 0];
    elements(1,:)= [1 2 -1 10 15e5];

case 'beam'
    % simple beam test problem %
    nodes(1,:)= [11 50 0 0 2 0 0];
    nodes(2,:)= [10 40 0 0 2 1 1];

```

```

        nodes(3,:)= [11 30 0 0 2 0 0];
        elements(1,:)= [1 3 2 -1 1000];
        elements(2,:)= [1 2 -1 sqrt(101) 500];
        elements(3,:)= [2 3 -1 sqrt(101) 500];

end

function main_hybrid

global initial_stiffness

[nodes,elements,dt,tend,rho1,rho2,rho3,tol,ialgo,problem,method]
=inputParams;

% count nodes/elements
[numNodes cols]=size(nodes);
[numElm cols]=size(elements);

% form the fixed vector
fixed=zeros(numNodes*2,1);
for i=1:numNodes
    fixed(2*i-1)=nodes(i,6);
    fixed(2*i)=nodes(i,7);
end

% form the mass matrix, diagonal
M=zeros(numNodes);

```

```

for i=1:numNodes
M(2*i-1,2*i-1)=nodes(i,5);
M(2*i,2*i)=nodes(i,5);
end

%%%% Initialization %%%%
%Load u0 and v0
u0=zeros(numNodes*2,1);
v0=zeros(numNodes*2,1);
for i=1:numNodes
u0(2*i-1)=nodes(i,1);
u0(2*i)=nodes(i,2);
v0(2*i-1)=nodes(i,3);
v0(2*i)=nodes(i,4);
end

%%%% Time %%%%
istep = 1;
tstart = 0.0;

%%%% Initialization %%%%
dlmwrite('nodes.dat',nodes, 'delimiter', ',', 'precision', 15);
dlmwrite('elements.dat',elements, 'delimiter', ',', 'precision', 15);
t = tstart;
time(istep) = t;
velocity(:,istep) = v0;
displacement(:,istep) = u0;

```



```

KE(istep)=1/2*v0'*M*v0;
[PE(istep) axial(istep) bending(istep) soft_bar(istep) rigid_bar
(istep)]=getPE(u0,elements);
TE(istep)=PE(istep)+KE(istep);
accel(istep)=0;
time(istep)=t;
cSave(:,istep)=ones(numElm,1).*0.5;
dn=u0;
vn=v0;
%NOTICE THAT A_0 is hardcoded to zeros
[rows cols]=size(vn);
an=zeros(rows,cols);
dt_original=dt;
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%% Start time integration %%%%%

while (t<tend)
switch method
case 1
converged=0;
while converged==0
    % EMM only
    [dn1,vn1]=newton_EMM(numNodes,numElm, elements,
dn, vn, dn, M, dt, (t+dt),tol,problem,fixed);

    if dn1(1)==-999

```

```

        % This section is for the adaptive timestepping
        dt=dt/10;
        output=sprintf('Convergence failed, lowering dt to %f \n',dt);
        disp(output)
        %pause
    else
        converged=1;
    end

end

an1=an; % ignored, just needs a value

case 2
% GS4 only
[dn1,vn1,an1]=newton_gs4(numNodes,numElm, elements, dn, vn, an, M
, dt, (t+dt),tol,problem,fixed,ialgo,rho1,rho2,rho3);

case 3
% Hybrid GS4/EMM. Get converged values at n+1 from gs4,
% use as initial guess in EMM
[dn1,vn1,an1]=newton_gs4(numNodes,numElm, elements, dn, vn, an, M
, dt, (t+dt),tol,problem,fixed,ialgo,rho1,rho2,rho3);

disp('Finished gs4 timestep, passing initial guess to EMM')
[dn dn1]

```

```

% Now get converged solution with EMM
[dn1,vn1]=newton_EMM(numNodes,numElm, elements, dn, vn, dn1, M,
dt, (t+dt),tol,problem,fixed);

if dn1(1)==-999
    [dn1,vn1]=newton_EMM_visualize(numNodes,numElm, elements, dn,
    vn, dn1, M, dt, (t+dt),tol,problem,fixed);
end

case 4
% c iteration method
[dn1,vn1,c_elm]=newton_c_iteration(numNodes,numElm, elements, dn,
vn, M, dt, (t+dt),tol,problem,fixed);
an1=an; % ignored, just needs a value
cSave(:,istep)=c_elm;
end

t = t+dt;
output=sprintf('Just converged at time t=%f',t);
disp(output)
dt=dt_original;
% timestep over
vn=vn1;
dn=dn1;
an=an1;
istep=istep+1;
displacement(:,istep)=dn1;

```

```

velocity(:,istep)=vn1;

KE(istep)=1/2*vn1'*M*vn1;
[PE(istep) axial(istep) bending(istep) soft_bar(istep) rigid_bar
 (istep)]=getPE(dn1,elements);
TE(istep)=PE(istep)+KE(istep);
time(istep)=t;

if mod(istep,1)==0
dmlwrite('time.dat',time, 'delimiter', ',', 'precision', 15);
dmlwrite('displacement.dat',displacement, 'delimiter', ',', 'precision', 15);
dmlwrite('velocity.dat',velocity, 'delimiter', ',', 'precision', 15);
dmlwrite('PE.dat',PE, 'delimiter', ',', 'precision', 15);
dmlwrite('KE.dat',KE, 'delimiter', ',', 'precision', 15);
dmlwrite('TE.dat',TE, 'delimiter', ',', 'precision', 15);
dmlwrite('accel.dat',accel, 'delimiter', ',', 'precision', 15);
dmlwrite('axial.dat',axial, 'delimiter', ',', 'precision', 15);
dmlwrite('bending.dat',bending, 'delimiter', ',', 'precision', 15);
dmlwrite('soft_bar.dat',soft_bar, 'delimiter', ',', 'precision', 15);
dmlwrite('rigid_bar.dat',rigid_bar, 'delimiter', ',', 'precision', 15);
if method == 4
dmlwrite('c.dat',cSave, 'delimiter', ',', 'precision', 15);
end
end

end

dmlwrite('time.dat',time, 'delimiter', ',', 'precision', 15);

```

```

dlmwrite('displacement.dat',displacement, 'delimiter', ',', 'precision', 15);
dlmwrite('velocity.dat',velocity, 'delimiter', ',', 'precision', 15);
dlmwrite('PE.dat',PE, 'delimiter', ',', 'precision', 15);
dlmwrite('KE.dat',KE, 'delimiter', ',', 'precision', 15);
dlmwrite('TE.dat',TE, 'delimiter', ',', 'precision', 15);
dlmwrite('accel.dat',accel, 'delimiter', ',', 'precision', 15);
dlmwrite('axial.dat',axial, 'delimiter', ',', 'precision', 15);
dlmwrite('bending.dat',bending, 'delimiter', ',', 'precision', 15);
dlmwrite('soft_bar.dat',soft_bar, 'delimiter', ',', 'precision', 15);
dlmwrite('rigid_bar.dat',rigid_bar, 'delimiter', ',', 'precision', 15);
if method == 4
dlmwrite('c.dat',cSave, 'delimiter', ',', 'precision', 15);
end

```

```

function [dn1,vn1,c_elm]=newton_c_iteration(numNodes,numElm, elements, dn, vn, M, dt, t,tol,problem,fixed)

```

```

% Take in values at n, return values at n+1

```

```

% cut down dn, vn for fixed nodes

```

```

dnSolve=dn;

```

```

vnSolve=vn;

```

```

for i=numNodes*2:-1:1

```

```

if fixed(i)==1

```

```

dnSolve(i)=[];

```

```

vnSolve(i)=[];

```

```

end

```

```

end

% Initial guesses
dn1=dn;
dn1Solve=dnSolve;
converged=0;
nliter=0;

while (converged==0)
nliter=nliter+1;

%%% Begin newton iteration %%%

% Loop over elements and form fint
fint=zeros(numNodes*2,1);
c_elm=zeros(numElm,1);
for i=1:numElm
% spring elements
if elements(i,3)==-1
start1=elements(i,1)*2-1;
start2=elements(i,2)*2-1;
[fintelement,c_elm(i)]=getfint_c_iteration([dn(start1)
dn(start1+1) dn(start2) dn(start2+1)], [dn1(start1)
dn1(start1+1) dn1(start2) dn1(start2+1)],elements(i,5),elements(i,4));
if c_elm(i)==-999
disp('spring element trying to use bisection, quit')
pause

```

```

else
fint(start1:start1+1,1)=fint(start1:start1+1,1)+
fintelelement(1:2);
fint(start2:start2+1,1)=fint(start2:start2+1,1)
+fintelelement(3:4);
end
% beam elements
else
start1=elements(i,1)*2-1;
start2=elements(i,3)*2-1;
start3=elements(i,2)*2-1;
[fintelelement,c_elm(i)]=getfintbeam_c_iteration([dn(start1) dn(start1+1) dn(start1+2) dn(start2) dn(start2+1) dn(start2+2) dn(start3) dn(start3+1) dn(start3+2)]);
if c_elm(i)==-999
disp('beam element trying to use bisection, quit')
pause
%[fintelelement,c_elm(i)]=n_particle_getfint_bisection([dn(start1) dn(start1+1) dn(start1+2) dn(start2) dn(start2+1) dn(start2+2) dn(start3) dn(start3+1) dn(start3+2)]);
%fint(start1:start1+2,1)=fint(start1:start1+2,1)+fintelelement(1:3);
%fint(start2:start2+2,1)=fint(start2:start2+2,1)+fintelelement(4:6);
%visualize_fint([dn(start1) dn(start1+1) dn(start1+2) dn(start2) dn(start2+1) dn(start2+2) dn(start3) dn(start3+1) dn(start3+2)]);
%return
else
fint(start1:start1+1,1)=fint(start1:start1+1,1)+fintelelement(1:2);
fint(start2:start2+1,1)=fint(start2:start2+1,1)+fintelelement(3:4);
fint(start3:start3+1,1)=fint(start3:start3+1,1)+fintelelement(5:6);
end
end
end

```

```

massSolve=M;
fintSolve=fint;
for i=numNodes*2:-1:1
if fixed(i)==1
massSolve(i,:)=[];
massSolve(:,i)=[];
fintSolve(i)=[];
end
end

% get external force
switch problem
case 'armero'
[fext]=getfext(dt,t);
res=massSolve*((-2*vnSolve)/dt + (2*(-dnSolve+dn1Solve))/dt^2) + fintSolve - fe
otherwise
res=massSolve*((-2*vnSolve)/dt + (2*(-dnSolve+dn1Solve))/dt^2) + fintSolve;
end

% if (norm(res)<tol && norm(del)<tol)
if (norm(res)<tol)
converged=1;
vn1Solve=-vnSolve+(2/dt)*(dn1Solve-dnSolve);

% Fill dn1 and vn1 with fixed zeros for full vector
vn1=zeros(numNodes*2,1);

```



```

numFixed=0;
for i=1:numNodes*2
if fixed(i)==0
dn1(i)=dn1Solve(i-numFixed);
vn1(i)=vn1Solve(i-numFixed);
else
numFixed=numFixed+1;
end
end

else
% Loop over elements and form J
J=zeros(numNodes*2,numNodes*2);
for i=1:numElm
% spring elements
if elements(i,3)==-1
start1=elements(i,1)*2-1;
start2=elements(i,2)*2-1;
[Jelm]=getJ_c_iteration([dn(start1) dn(start1+1) dn(start2) dn
(start2+1)], [dn1(start1) dn1(start1+1) dn1(start2) dn1(start2
+1)], dt, c_elm(i), elements(i,5), elements(i,4), [M(start1, start1)
M(start1+1, start1+1) M(start2, start2) M(start2+1, start2+1)]];
J(start1:start1+1, start1:start1+1)=J(start1:start1+1, start1:start1+1)
+Jelm(1:2, 1:2);
J(start1:start1+1, start2:start2+1)=J(start1:start1+1, start2:start2+1)
+Jelm(1:2, 3:4);
J(start2:start2+1, start1:start1+1)=J(start2:start2+1, start1:start1+1)

```

```

+Jelm(3:4,1:2);
J(start2:start2+1,start2:start2+1)=J(start2:start2+1,start2:start2+1)
+Jelm(3:4,3:4);
%disp('J coming from a spring element')
%Jelm
% beam elements
else
start1=elements(i,1)*2-1;
start2=elements(i,3)*2-1;
start3=elements(i,2)*2-1;
[Jelm]=getJbeam_c_iteration([dn(start1) dn(start1+1) dn(start2)
    dn(start2+1) dn(start3) dn(start3+1)],[dn1(start1) dn1(start1+1)
    dn1(start2) dn1(start2+1) dn1(start3) dn1(start3+1)],dt,c_elm(i)
,elements(i,5),[M(start1,start1) M(start1+1,start1+1) M(start2,start2)
    M(start2+1,start2+1) M(start3,start3) M(start3+1,start3+1)]];
J(start1:start1+1,start1:start1+1)=J(start1:start1+1,start1:start1+1)
+Jelm(1:2,1:2);
J(start1:start1+1,start2:start2+1)=J(start1:start1+1,start2:start2+1)
+Jelm(1:2,3:4);
J(start1:start1+1,start3:start3+1)=J(start1:start1+1,start3:start3+1)
+Jelm(1:2,5:6);

J(start2:start2+1,start1:start1+1)=J(start2:start2+1,start1:start1+1)
+Jelm(3:4,1:2);
J(start2:start2+1,start2:start2+1)=J(start2:start2+1,start2:start2+1)
+Jelm(3:4,3:4);
J(start2:start2+1,start3:start3+1)=J(start2:start2+1,start3:start3+1)

```

```

+Jelm(3:4,5:6);

J(start3:start3+1,start1:start1+1)=J(start3:start3+1,start1:start1+1)
+Jelm(5:6,1:2);
J(start3:start3+1,start2:start2+1)=J(start3:start3+1,start2:start2+1)
+Jelm(5:6,3:4);
J(start3:start3+1,start3:start3+1)=J(start3:start3+1,start3:start3+1)
+Jelm(5:6,5:6);
%disp('J coming from a beam element')
%Jelm

end

end

JSolve=J;
for i=numNodes*2:-1:1
if fixed(i)==1
JSolve(i,:)=[];
JSolve(:,i)=[];
end
end

del=-1*inv(JSolve)*res;
dn1Solve=dn1Solve+del;

% put back zeros in dn1 fint
numFixed=0;

```

```

for i=1:numNodes*2
if fixed(i)==0
dn1(i)=dn1Solve(i-numFixed);
else
numFixed=numFixed+1;
end
end
end

if mod(nliter,1000)==0
disp('eom residual not converging, stop')
pause
end
end

function [dn1,vn1]=newton_EMM(numNodes,numElm, elements, dn, vn,
dn1, M, dt, t,tol,problem,fixed)

global initial_stiffness

% Take in values at n, return values at n+1

dnSolve=dn;
vnSolve=vn;
dn1Solve=dn1;
for i=numNodes*2:-1:1
if fixed(i)==1

```

```

dnSolve(i)=[];
vnSolve(i)=[];
dn1Solve(i)=[];
end
end

converged=0;
nliter=0;
del=dn1Solve.*0;

while (converged==0)
nliter=nliter+1;

%%% Begin newton iteration %%%%

% Loop over elements and form fint
fint=zeros(numNodes*2,1);
for i=1:numElm
% spring elements
if elements(i,3)==-1
start1=elements(i,1)*2-1;
start2=elements(i,2)*2-1;
[fintelement,beta]=getfint_armero_EMM([dn(start1) dn(start1+1)
    dn(start2) dn(start2+1)], [dn1(start1) dn1(start1+1) dn1(start2)
    dn1(start2+1)], elements(i,5), elements(i,4));
fint(start1:start1+1,1)=fint(start1:start1+1,1)+fintelement(1:2);
fint(start2:start2+1,1)=fint(start2:start2+1,1)+fintelement(3:4);

```

```

beta_save(i,nlitter)=beta;

% beam elements
else
start1=elements(i,1)*2-1;
start2=elements(i,2)*2-1;
start3=elements(i,3)*2-1;
[fintelement,beta]=getfintbeam_armero_EMM([dn(start1) dn(start1+1)
      dn(start2) dn(start2+1) dn(start3) dn(start3+1)], [dn1(start1)
      dn1(start1+1) dn1(start2) dn1(start2+1) dn1(start3) dn1(start3+1)]
,elements(i,5));
fint(start1:start1+1,1)=fint(start1:start1+1,1)+fintelement(1:2);
fint(start2:start2+1,1)=fint(start2:start2+1,1)+fintelement(3:4);
fint(start3:start3+1,1)=fint(start3:start3+1,1)+fintelement(5:6);
beta_save(i,nlitter)=beta;
end
end

massSolve=M;
fintSolve=fint;
for i=numNodes*2:-1:1
if fixed(i)==1
massSolve(i,:)=[];
massSolve(:,i)=[];
fintSolve(i)=[];
end
end

```

```

% get external force
switch problem
case 'armero'
[fext]=getfext(dt,t);
%           fext=fext*0; % for initial velocity only
%           fext=fext/5;
%           % for incremental load stepping
%           if nliter < 100
%               fext=fext*(nliter/100)^2;
%           end
% if nliter< 11
%     fext=fext*10;
% end

res=massSolve*((-2*vnSolve)/dt + (2*(-dnSolve+dn1Solve))/dt^2) +
fintSolve - fext;

otherwise
res=massSolve*((-2*vnSolve)/dt + (2*(-dnSolve+dn1Solve))/dt^2) +
fintSolve;

end

if (norm(res)<tol && norm(del)<tol)
converged=1;
vn1Solve=-vnSolve+(2/dt)*(dn1Solve-dnSolve);
output=sprintf('N/L iterations: %d',nliter);
disp(output)

```

```

% Fill dn1 and vn1 with fixed zeros for full vector
vn1=zeros(numNodes*2,1);
numFixed=0;
for i=1:numNodes*2
if fixed(i)==0
dn1(i)=dn1Solve(i-numFixed);
vn1(i)=vn1Solve(i-numFixed);
else
numFixed=numFixed+1;
end
end

else
% Loop over elements and form J
J=zeros(numNodes*2,numNodes*2);

for i=1:numElm
% spring elements
if elements(i,3)==-1
start1=elements(i,1)*2-1;
start2=elements(i,2)*2-1;
[Jelm]=getJ_armero_EMM([dn(start1) dn(start1+1) dn(start2)
    dn(start2+1)], [dn1(start1) dn1(start1+1) dn1(start2)
    dn1(start2+1)], dt, elements(i,5), elements(i,4), [M(start1, start1)
    M(start1+1, start1+1) M(start2, start2) M(start2+1, start2+1)]);
J(start1:start1+1, start1:start1+1)=J(start1:start1+1, start1:start1+1)

```



```

+Jelm(1:2,1:2);
J(start1:start1+1,start2:start2+1)=J(start1:start1+1,start2:start2+1)
+Jelm(1:2,3:4);
J(start2:start2+1,start1:start1+1)=J(start2:start2+1,start1:start1+1)
+Jelm(3:4,1:2);
J(start2:start2+1,start2:start2+1)=J(start2:start2+1,start2:start2+1)
+Jelm(3:4,3:4);
%
%           disp('J coming from a spring element')
%
%           Jelm
%
%           beam elements
else
start1=elements(i,1)*2-1;
start2=elements(i,2)*2-1;
start3=elements(i,3)*2-1;
[Jelm]=getJbeam_armero_EMM([dn(start1) dn(start1+1) dn(start2)
    dn(start2+1) dn(start3) dn(start3+1)],[dn1(start1) dn1(start1+1)
    dn1(start2) dn1(start2+1) dn1(start3) dn1(start3+1)],
dt,elements(i,5),[M(start1,start1) M(start1+1,start1+1)
    M(start2,start2) M(start2+1,start2+1) M(start3,start3)
    M(start3+1,start3+1)]);
J(start1:start1+1,start1:start1+1)=J(start1:start1+1,start1:start1+1)
+Jelm(1:2,1:2);
J(start1:start1+1,start2:start2+1)=J(start1:start1+1,start2:start2+1)
+Jelm(1:2,3:4);
J(start1:start1+1,start3:start3+1)=J(start1:start1+1,start3:start3+1)
+Jelm(1:2,5:6);

```

```

J(start2:start2+1,start1:start1+1)=J(start2:start2+1,start1:start1+1)
+Jelm(3:4,1:2);
J(start2:start2+1,start2:start2+1)=J(start2:start2+1,start2:start2+1)
+Jelm(3:4,3:4);
J(start2:start2+1,start3:start3+1)=J(start2:start2+1,start3:start3+1)
+Jelm(3:4,5:6);

J(start3:start3+1,start1:start1+1)=J(start3:start3+1,start1:start1+1)
+Jelm(5:6,1:2);
J(start3:start3+1,start2:start2+1)=J(start3:start3+1,start2:start2+1)
+Jelm(5:6,3:4);
J(start3:start3+1,start3:start3+1)=J(start3:start3+1,start3:start3+1)
+Jelm(5:6,5:6);

%           disp('J coming from a beam element')
%           Jelm

end

end

JSolve=J;

for i=numNodes*2:-1:1
if fixed(i)==1
JSolve(i,:)=[];
JSolve(:,i)=[];
end
end

```

```

if nliter==1
JinitialSolve=JSolve;
end

if initial_stiffness==1
del=-1*inv(JinitialSolve)*res;
else
del=-1*inv(JSolve)*res;
%[v,d]=eig(JSolve,massSolve)
%pause
end
dn1Solve=dn1Solve+del;

% put back zeros in dn1 fint
numFixed=0;
for i=1:numNodes*2
if fixed(i)==0
dn1(i)=dn1Solve(i-numFixed);
else
numFixed=numFixed+1;
end
end
end

if mod(nliter,1000)==0

```

```

%%%%% Section for normal use %%%%%
%disp('EMM eom residual not converging')
clear dn1
clear vn1
dn1=-999;
vn1=-999;
return
%%%%% End section for normal use %%%%%

end

end

function [dp,vp,ap]=newton_gs4(numNodes,numElm, elements, dn, vn,
an, M, dt, t,tol,problem,fixed,ialgo,rho1,rho2,rho3)

% Take in values at n, return values at n+1

% get numerical method parameters
[W,opl,dupl] = gssss (ialgo,rho1,rho2,rho3);

% get the predictor corrector coeff
[xsip,xsic] = getXsi(W,opl,dupl,dt,1);

% cut down dn, vn, an for fixed nodes
dnSolve=dn;
vnSolve=vn;

```

```

anSolve=an;
for i=numNodes*2:-1:1
if fixed(i)==1
dnSolve(i)=[];
vnSolve(i)=[];
anSolve(i)=[];
end
end

% Initial guesses
dpSolve = xsip(1,1)*dnSolve + xsip(1,2)*vnSolve + xsip(1,3)*anSolve;
dhSolve = xsip(2,1)*dnSolve + xsip(2,2)*vnSolve + xsip(2,3)*anSolve;
vhSolve = xsip(3,1)*dnSolve + xsip(3,2)*vnSolve + xsip(3,3)*anSolve;
ahSolve = xsip(4,1)*dnSolve + xsip(4,2)*vnSolve + xsip(4,3)*anSolve;

converged=0;
nliter=0;

while (converged==0)
nliter=nliter+1;

% put back values in dh, and dp needed for total ktang and fint
dh=zeros(numNodes*2,1);
dp=zeros(numNodes*2,1);
numFixed=0;
for i=1:numNodes*2
if fixed(i)==0

```

```

dh(i)=dhSolve(i-numFixed);
dp(i)=dpSolve(i-numFixed);
else
numFixed=numFixed+1;
dh(i)=dn(i);
dp(i)=dn(i);
end
end

%%%% Begin newton iteration %%%%%

% Loop over elements and form fint
fint=zeros(numNodes*2,1);
for i=1:numElm
% spring elements
if elements(i,3)==-1
start1=elements(i,1)*2-1;
start2=elements(i,2)*2-1;

[fintelement]=getfint_spring_gs4([dh(start1)
dh(start1+1) dh(start2) dh(start2+1)],elements(i,5),elements(i,4));
fint(start1:start1+1,1)=fint(start1:start1+1,1)+fintelement(1:2);
fint(start2:start2+1,1)=fint(start2:start2+1,1)+fintelement(3:4);

% beam elements
else
start1=elements(i,1)*2-1;
start2=elements(i,2)*2-1;

```

```

start3=elements(i,3)*2-1;
[fintelement]=getfint_beam_gs4([dh(start1) dh(start1+1) dh(start2)
    dh(start2+1) dh(start3) dh(start3+1)],elements(i,5));
fint(start1:start1+1,1)=fint(start1:start1+1,1)+fintelement(1:2);
fint(start2:start2+1,1)=fint(start2:start2+1,1)+fintelement(3:4);
fint(start3:start3+1,1)=fint(start3:start3+1,1)+fintelement(5:6);
end
end

massSolve=M;
fintSolve=fint;
for i=numNodes*2:-1:1
if fixed(i)==1
massSolve(i,:)=[];
massSolve(:,i)=[];
fintSolve(i)=[];
end
end

% get external force
switch problem
case 'armero'
[fext]=getfext(dt,t);
res = massSolve*ahSolve + fintSolve-fext;
otherwise
res = massSolve*ahSolve + fintSolve;
end

```

```

if (norm(res)<tol && norm(del)<tol)
converged=1;
output=sprintf('N/L iterations: %d',nliter);
disp(output)

% need to stick ahSolve into a non-reduced ah for final updates
ah=zeros(numNodes*2,1);
numFixed=0;
for i=1:numNodes*2
if fixed(i)==0
ah(i)=ahSolve(i-numFixed);
else
numFixed=numFixed+1;
end
end

% Design updates at the end of time step %
ap = an + (ah-an)/opl(6)/W(1);
vp = vn + dupl(4)*an*dt + dupl(5)*(ap-an)*dt;
dp = dn + dupl(1)*vn*dt + dupl(2)*an*dt*dt + dupl(3)*(ap-an)*dt*dt;

else

% Loop over elements and form J
J=zeros(numNodes*2,numNodes*2);

for i=1:numElm

```



```

% spring elements
if elements(i,3)==-1
start1=elements(i,1)*2-1;
start2=elements(i,2)*2-1;
[Jelm]=getJ_spring_gs4([dn(start1) dn(start1+1) dn(start2)
    dn(start2+1)], [dp(start1) dp(start1+1) dp(start2) dp(start2+1)]
, [vn(start1) vn(start1+1) vn(start2) vn(start2+1)], [an(start1)
    an(start1+1) an(start2) an(start2+1)], dt, elements(i,5),
elements(i,4), W, opl, dupl);
J(start1:start1+1, start1:start1+1)=J(start1:start1+1, start1:start1+1)
+Jelm(1:2, 1:2);
J(start1:start1+1, start2:start2+1)=J(start1:start1+1, start2:start2+1)
+Jelm(1:2, 3:4);
J(start2:start2+1, start1:start1+1)=J(start2:start2+1, start1:start1+1)
+Jelm(3:4, 1:2);
J(start2:start2+1, start2:start2+1)=J(start2:start2+1, start2:start2+1)
+Jelm(3:4, 3:4);
else
start1=elements(i,1)*2-1;
start2=elements(i,2)*2-1;
start3=elements(i,3)*2-1;
[Jelm]=getJ_beam_gs4([dn(start1) dn(start1+1) dn(start2) dn(start2+1)
    dn(start3) dn(start3+1)], [dp(start1) dp(start1+1) dp(start2)
    dp(start2+1) dp(start3) dp(start3+1)], [vn(start1) vn(start1+1)
    vn(start2) vn(start2+1) vn(start3) vn(start3+1)], [an(start1)
    an(start1+1) an(start2) an(start2+1) an(start3) an(start3+1)],
dt, elements(i,5), W, opl, dupl);

```

```

J(start1:start1+1,start1:start1+1)=J(start1:start1+1,start1:start1+1)
+Jelm(1:2,1:2);
J(start1:start1+1,start2:start2+1)=J(start1:start1+1,start2:start2+1)
+Jelm(1:2,3:4);
J(start1:start1+1,start3:start3+1)=J(start1:start1+1,start3:start3+1)
+Jelm(1:2,5:6);

J(start2:start2+1,start1:start1+1)=J(start2:start2+1,start1:start1+1)
+Jelm(3:4,1:2);
J(start2:start2+1,start2:start2+1)=J(start2:start2+1,start2:start2+1)
+Jelm(3:4,3:4);
J(start2:start2+1,start3:start3+1)=J(start2:start2+1,start3:start3+1)
+Jelm(3:4,5:6);

J(start3:start3+1,start1:start1+1)=J(start3:start3+1,start1:start1+1)
+Jelm(5:6,1:2);
J(start3:start3+1,start2:start2+1)=J(start3:start3+1,start2:start2+1)
+Jelm(5:6,3:4);
J(start3:start3+1,start3:start3+1)=J(start3:start3+1,start3:start3+1)
+Jelm(5:6,5:6);

end

end

JSolve=J;
for i=numNodes*2:-1:1
if fixed(i)==1
JSolve(i,:)=[];

```

```

JSolve(:,i)=[];
end
end

JSolve = xsic(4)*massSolve + JSolve;

del=-1*inv(JSolve)*res;

% Correct %
dpSolve = dpSolve + xsic(1)*del;
dhSolve = dhSolve + xsic(2)*del;
vhSolve = vhSolve + xsic(3)*del;
ahSolve = ahSolve + xsic(4)*del;

end

if mod(nliter,1000)==0
disp('GS4 eom residual not converging')
end
end

function results

tol=1e-8;

time=csvread('time.dat');
displacement=csvread('displacement.dat');

```

```

velocity=csvread('velocity.dat');
PE=csvread('PE.dat');
KE=csvread('KE.dat');
TE=csvread('TE.dat');
accel=csvread('accel.dat');
axial=csvread('axial.dat');
bending=csvread('bending.dat');
soft_bar=csvread('soft_bar.dat');
rigid_bar=csvread('rigid_bar.dat');

[numNodes,istep]=size(displacement);
numNodes=numNodes/2;

figure()
plot(time,PE,'*-',time,KE,'*-',time,TE,'*-')
legend('PE','KE','TE');
xlabel('Time')
ylabel('Energy')

soft_bar=soft_bar./PE;
rigid_bar=rigid_bar./PE;
bending=bending./PE;

figure()
plot(time,soft_bar,'-',time,bending,'--',time,rigid_bar,'-')
%plot(time,rigid_bar,'-')
legend('Spring','Bending','Rigid');

```

```
xlabel('Time')  
ylabel('Energy contribution')
```

(* Get the C iteration equations *)

(* Generic potential and length *)

$$L1nc = \text{Sqrt}[(x2nc - x1nc)^2 + (y2nc - y1nc)^2]$$

$$L2nc = \text{Sqrt}[(x3nc - x2nc)^2 + (y3nc - y2nc)^2]$$

$$\nu nc = L1nc * L2nc$$

$$\lambda nc = (x1nc - x2nc) * (x3nc - x2nc) +$$

$$(y1nc - y2nc) * (y3nc - y2nc)$$

$$PEnc = 1/2 * Cb * (\nu nc + \lambda nc) / (\nu nc - \lambda nc)$$

(* Residual equation for solving for C *)

res =

$$-PEnc + PEn+$$

$$((\partial_{x1nc} PEnc) * (x1n1 - x1n) +$$

$$(\partial_{y1nc} PEnc) * (y1n1 - y1n) +$$

$$(\partial_{x2nc} PEnc) * (x2n1 - x2n) +$$

$$(\partial_{y2nc} PEnc) * (y2n1 - y2n) +$$

$$(\partial_{x3nc} PEnc) * (x3n1 - x3n) +$$

$$(\partial_{y3nc} PEnc) * (y3n1 - y3n)) /$$

$$\{x1nc \rightarrow (1 - c) * x1n + c * x1n1,$$

$$y1nc \rightarrow (1 - c) * y1n + c * y1n1,$$

$$x2nc \rightarrow (1 - c) * x2n + c * x2n1,$$

$$y2nc \rightarrow (1 - c) * y2n + c * y2n1,$$

$$x3nc \rightarrow (1 - c) * x3n + c * x3n1,$$

$$y3nc \rightarrow (1 - c) * y3n + c * y3n1\}$$

(* Residual for pasting into matlab *)

%//InputForm

(* Get jacobian for solving for C *)

∂_c %

(* Jacobian for pasting into matlab *)

%//InputForm

$$L1n = \text{Sqrt}[(x3n - x1n)^2 + (y3n - y1n)^2]$$

$$L2n = \text{Sqrt}[(x3n - x2n)^2 + (y3n - y2n)^2]$$

$$L1n1 =$$

$$\text{Sqrt}[(x3n1 - x1n1)^2 + (y3n1 - y1n1)^2]$$

$$L2n1 =$$

$$\text{Sqrt}[(x3n1 - x2n1)^2 + (y3n1 - y2n1)^2]$$

$$\nu n = L1n * L2n$$

$$\lambda n = (x1n - x3n) * (x2n - x3n) +$$

$$(y1n - y3n) * (y2n - y3n)$$

$$\nu n1 = L1n1 * L2n1$$

$$\lambda n1 = (x1n1 - x3n1) * (x2n1 - x3n1) +$$

$$(y1n1 - y3n1) * (y2n1 - y3n1)$$

$$L1bar = (L1n1 + L1n)/2$$

$$L2bar = (L2n1 + L2n)/2$$

$$V11 = 1/2 * Cb * (\nu n1 + \lambda n1)/(\nu n1 - \lambda n1)$$

$$V10 = 1/2 * Cb * (\nu n + \lambda n1)/(\nu n - \lambda n1)$$

$$V01 = 1/2 * Cb * (\nu n1 + \lambda n)/(\nu n1 - \lambda n)$$

$$V00 = 1/2 * Cb * (\nu n + \lambda n)/(\nu n - \lambda n)$$

$$r1x = (x1n1 + x1n)/2 - (x3n1 + x3n)/2$$

$$r1y = (y1n1 + y1n)/2 - (y3n1 + y3n)/2$$

$$r2x = (x2n1 + x2n)/2 - (x3n1 + x3n)/2$$

$$r2y = (y2n1 + y2n)/2 - (y3n1 + y3n)/2$$

$$fint1a =$$

$$(1/2 * (V11 + V10) - 1/2 * (V01 + V00))/$$

$$(\lambda n1 - \lambda n) * r2x$$

$$fint2a =$$

$$\frac{(1/2 * (V11 + V10) - 1/2 * (V01 + V00))}{(\lambda n1 - \lambda n) * r2y}$$

fint3a =

$$\frac{(1/2 * (V11 + V10) - 1/2 * (V01 + V00))}{(\lambda n1 - \lambda n) * r1x}$$

fint4a =

$$\frac{(1/2 * (V11 + V10) - 1/2 * (V01 + V00))}{(\lambda n1 - \lambda n) * r1y}$$

fint5a =

$$\frac{(1/2 * (V11 + V10) - 1/2 * (V01 + V00))}{(\lambda n1 - \lambda n) * (-r1x - r2x)}$$

fint6a =

$$\frac{(1/2 * (V11 + V10) - 1/2 * (V01 + V00))}{(\lambda n1 - \lambda n) * (-r1y - r2y)}$$

fint1b =

$$\frac{(1/2 * (V11 + V01) - 1/2 * (V10 + V00))}{(\nu n1 - \nu n) * (L2bar/L1bar * r1x)}$$

fint2b =

$$\frac{(1/2 * (V11 + V01) - 1/2 * (V10 + V00))}{(\nu n1 - \nu n) * (L2bar/L1bar * r1y)}$$

fint3b =

$$\frac{(1/2 * (V11 + V01) - 1/2 * (V10 + V00))}{(\nu n1 - \nu n) * (L1bar/L2bar * r2x)}$$

fint4b =

$$\frac{(1/2 * (V11 + V01) - 1/2 * (V10 + V00))}{(\nu n1 - \nu n) * (L1bar/L2bar * r2y)}$$

$$\begin{aligned} \text{fint5b} = & \\ & (1/2 * (V11 + V01) - 1/2 * (V10 + V00))/ \\ & (\nu_{n1} - \nu_n) * \\ & (L2\text{bar}/L1\text{bar} * -r1x + \\ & L1\text{bar}/L2\text{bar} * -r2x) \end{aligned}$$

$$\begin{aligned} \text{res1} = & \\ & M * (2/\text{dt}^2 * (x1_{n1} - x1_n) - \\ & (2/\text{dt}) * v1_{xn}) + \text{fint1a} + \text{fint1b} \end{aligned}$$

$$\begin{aligned} \text{res2} = & \\ & M * (2/\text{dt}^2 * (y1_{n1} - y1_n) - \\ & (2/\text{dt}) * v1_{yn}) + \text{fint2a} + \text{fint2b} \end{aligned}$$

$$\begin{aligned} \text{res3} = & \\ & M * (2/\text{dt}^2 * (x2_{n1} - x2_n) - \\ & (2/\text{dt}) * v2_{xn}) + \text{fint3a} + \text{fint3b} \end{aligned}$$

$$\begin{aligned} \text{res4} = & \\ & M * (2/\text{dt}^2 * (y2_{n1} - y2_n) - \\ & (2/\text{dt}) * v2_{yn}) + \text{fint4a} + \text{fint4b} \end{aligned}$$

$$\begin{aligned} \text{res5} = & \\ & M * (2/\text{dt}^2 * (x3_{n1} - x3_n) - \\ & (2/\text{dt}) * v3_{xn}) + \text{fint5a} + \text{fint5b} \end{aligned}$$

$$\begin{aligned} \text{res6} = & \\ & M * (2/\text{dt}^2 * (y3_{n1} - y3_n) - \\ & (2/\text{dt}) * v3_{yn}) + \text{fint6a} + \text{fint6b} \end{aligned}$$

$$J11 = \partial_{x1_{n1}} \text{res1}$$

$$J12 = \partial_{y1_{n1}} \text{res1}$$

$$J13 = \partial_{x2_{n1}} \text{res1}$$

$$J14 = \partial_{y2n1} \text{res1}$$

$$J15 = \partial_{x3n1} \text{res1}$$

$$J16 = \partial_{y3n1} \text{res1}$$

$$J21 = \partial_{x1n1} \text{res2}$$

$$J22 = \partial_{y1n1} \text{res2}$$

$$J23 = \partial_{x2n1} \text{res2}$$

$$J24 = \partial_{y2n1} \text{res2}$$

$$J25 = \partial_{x3n1} \text{res2}$$

$$J26 = \partial_{y3n1} \text{res2}$$

$$J31 = \partial_{x1n1} \text{res3}$$

$$J32 = \partial_{y1n1} \text{res3}$$

$$J33 = \partial_{x2n1} \text{res3}$$

$$J34 = \partial_{y2n1} \text{res3}$$

$$J35 = \partial_{x3n1} \text{res3}$$

$$J36 = \partial_{y3n1} \text{res3}$$

$$J41 = \partial_{x1n1} \text{res4}$$

$$J42 = \partial_{y1n1} \text{res4}$$

$$J43 = \partial_{x2n1} \text{res4}$$

$$J44 = \partial_{y2n1} \text{res4}$$

$$J45 = \partial_{x3n1} \text{res4}$$

$$J46 = \partial_{y3n1} \text{res4}$$

$$J51 = \partial_{x_{1n1}} \text{res5}$$

$$J52 = \partial_{y_{1n1}} \text{res5}$$

$$J53 = \partial_{x_{2n1}} \text{res5}$$

$$J54 = \partial_{y_{2n1}} \text{res5}$$

J11//InputForm

J12//InputForm

J13//InputForm

J14//InputForm

J15//InputForm

J16//InputForm

J21//InputForm

J22//InputForm

J23//InputForm

J24//InputForm

J25//InputForm

J26//InputForm

J31//InputForm

J32//InputForm

J33//InputForm

J34//InputForm

J35//InputForm

J36//InputForm

J41//InputForm

J42//InputForm

J43//InputForm

J44//InputForm

J45//InputForm

J46//InputForm

J51//InputForm

J52//InputForm

J53//InputForm

J54//InputForm

J55//InputForm

J56//InputForm

J61//InputForm

J62//InputForm

J63//InputForm

J64//InputForm

J65//InputForm

J66//InputForm

$$\nu_{nh} = L1_{nh} * L2_{nh}$$

$$\lambda_{nh} = (x1_{nh} - x2_{nh}) * (x3_{nh} - x2_{nh}) + (y1_{nh} - y2_{nh}) * (y3_{nh} - y2_{nh})$$

$$PE_{nh} = 1/2 * Cb * (\nu_{nh} + \lambda_{nh}) / (\nu_{nh} - \lambda_{nh})$$

$$fint1 = \partial_{x1_{nh}} PE_{nh}$$

$$fint2 = \partial_{y1_{nh}} PE_{nh}$$

$$fint3 = \partial_{x2_{nh}} PE_{nh}$$

$$fint4 = \partial_{y2_{nh}} PE_{nh}$$

$$fint5 = \partial_{x3_{nh}} PE_{nh}$$

$$fint6 = \partial_{y3_{nh}} PE_{nh}$$

(* For pasting into function to calc fint *)

fint1//InputForm

fint2//InputForm

fint3//InputForm

fint4//InputForm

fint5//InputForm

fint6//InputForm

$$x1_{nh} =$$

$$x1_n + vx1_n * (dt * L1 * W1 - (dt * l1 * L3 * W3)/l3) + ax1_n * (dt^2 * L2 * W2 - (dt^2 * l2 * L3 * W3)/l3) + (L3 * W3 * (x1_p - x1_n))/l3$$

$$y1_{nh} =$$

$$y1_n + vy1_n * (dt * L1 * W1 - (dt * l1 * L3 * W3)/l3) +$$

$$ay1n * (dt^2 * L2 * W2 - (dt^2 * l2 * L3 * W3)/l3) +$$

$$(L3 * W3 * (y1p - y1n))/l3$$

$$x2nh =$$

$$x2n + vx2n * (dt * L1 * W1 - (dt * l1 * L3 * W3)/l3) +$$

$$ax2n * (dt^2 * L2 * W2 - (dt^2 * l2 * L3 * W3)/l3) +$$

$$(L3 * W3 * (x2p - x2n))/l3$$

$$y2nh =$$

$$y2n + vy2n * (dt * L1 * W1 - (dt * l1 * L3 * W3)/l3) +$$

$$ay2n * (dt^2 * L2 * W2 - (dt^2 * l2 * L3 * W3)/l3) +$$

$$(L3 * W3 * (y2p - y2n))/l3$$

$$x3nh =$$

$$x3n + vx3n * (dt * L1 * W1 - (dt * l1 * L3 * W3)/l3) +$$

$$ax3n * (dt^2 * L2 * W2 - (dt^2 * l2 * L3 * W3)/l3) +$$

$$(L3 * W3 * (x3p - x3n))/l3$$

$$y3nh =$$

$$y3n + vy3n * (dt * L1 * W1 - (dt * l1 * L3 * W3)/l3) +$$

$$ay3n * (dt^2 * L2 * W2 - (dt^2 * l2 * L3 * W3)/l3) +$$

$$(L3 * W3 * (y3p - y3n))/l3$$

$$res1 = M * (2/dt^2 * (x1n1 - x1n) - (2/dt) * v1xn) +$$

$$fint1$$

$$res2 = M * (2/dt^2 * (y1n1 - y1n) - (2/dt) * v1yn) +$$

$$fint2$$

$$res3 = M * (2/dt^2 * (x2n1 - x2n) - (2/dt) * v2xn) +$$

$$fint3$$

$$res4 = M * (2/dt^2 * (y2n1 - y2n) - (2/dt) * v2yn) +$$

$$fint4$$

$$res5 = M * (2/dt^2 * (x3n1 - x3n) - (2/dt) * v3xn) +$$

fint5

$$\text{res6} = M * (2/\text{dt}^2 * (y_{3n1} - y_{3n}) - (2/\text{dt}) * v_{3yn}) +$$

fint6

$$J_{11} = \partial_{x_{1n1}} \text{res1}$$

$$J_{12} = \partial_{y_{1n1}} \text{res1}$$

$$J_{13} = \partial_{x_{2n1}} \text{res1}$$

$$J_{14} = \partial_{y_{2n1}} \text{res1}$$

$$J_{15} = \partial_{x_{3n1}} \text{res1}$$

$$J_{16} = \partial_{y_{3n1}} \text{res1}$$

$$J_{21} = \partial_{x_{1n1}} \text{res2}$$

$$J_{22} = \partial_{y_{1n1}} \text{res2}$$

$$J_{23} = \partial_{x_{2n1}} \text{res2}$$

$$J_{24} = \partial_{y_{2n1}} \text{res2}$$

$$J_{25} = \partial_{x_{3n1}} \text{res2}$$

$$J_{26} = \partial_{y_{3n1}} \text{res2}$$

$$J_{31} = \partial_{x_{1n1}} \text{res3}$$

$$J_{32} = \partial_{y_{1n1}} \text{res3}$$

$$J_{33} = \partial_{x_{2n1}} \text{res3}$$

$$J_{34} = \partial_{y_{2n1}} \text{res3}$$

$$J_{35} = \partial_{x_{3n1}} \text{res3}$$

$$J_{36} = \partial_{y_{3n1}} \text{res3}$$

$$J41 = \partial_{x1n1} \text{res4}$$

$$J42 = \partial_{y1n1} \text{res4}$$

$$J43 = \partial_{x2n1} \text{res4}$$

$$J44 = \partial_{y2n1} \text{res4}$$

$$J45 = \partial_{x3n1} \text{res4}$$

$$J46 = \partial_{y3n1} \text{res4}$$

$$J51 = \partial_{x1n1} \text{res5}$$

$$J52 = \partial_{y1n1} \text{res5}$$

$$J53 = \partial_{x2n1} \text{res5}$$

$$J54 = \partial_{y2n1} \text{res5}$$

J11//InputForm

J12//InputForm

J13//InputForm

J14//InputForm

J15//InputForm

J16//InputForm

J21//InputForm

J22//InputForm

J23//InputForm

J24//InputForm

J25//InputForm

J26//InputForm

J31//InputForm

J32//InputForm

J33//InputForm

J34//InputForm

J35//InputForm

J36//InputForm

J41//InputForm

J42//InputForm

J43//InputForm

J44//InputForm

J45//InputForm

J46//InputForm

J51//InputForm

J52//InputForm

J53//InputForm

J54//InputForm

J55//InputForm

J56//InputForm

J61//InputForm

J62//InputForm

J63//InputForm

J64//InputForm

J65//InputForm

J66//InputForm