Development of computer software for simulation of transmission line dynamic behavior

Amr Mohamed Abd Elaziz Nafie
Iowa State University

Development of computer software for simulation of transmission line dynamic behavior

by

Amr Mohamed Abd Elaziz Nafie

A dissertation submitted to the graduate faculty

in partial fulfillment of the requirements for the degree of

DOCTOR OF PHILOSOPHY

Major: Civil Engineering (Structural Engineering)

Major Professors: Fouad F. Fanous and Terry J. Wipf

Iowa State University

Ames, Iowa

1997

Graduate College
Iowa State University

This is to certify that the Doctoral dissertation of

Amr Mohamed Abd Elaziz Nafie

has met the dissertation requirements of Iowa State University

Signature was redacted for privacy.

Co-major Professor

Signature was redacted for privacy.

Co-major Professor

Signature was redacted for privacy.

For the Major Program

Signature was redacted for privacy.

For The Graduate College

TABLE OF CONTENTS

ACKNOWLEDGMENTS

I thank God for enabling me to complete this work and attribute any success that I attain in my life to His grace and guidance.

I wish to extend my sincere appreciation to my major professors Dr. Fouad Fanous, and Dr. Terry J. Wipf for their help and support throughout this work. I would also like to thank Dr. Mardith Baenziger, Dr. Kenneth G. McConnell, and Dr. Joseph Gray for serving on my graduate committee, and for their help and valuable advice.

I express my utmost gratitude and deepest appreciation to my parents for their patience, care and advice. I would also like to accord my special thanks to my wife for her extreme patience and major support throughout the period of this work.

I would also like to extend my thanks to the Electric Power Research Center, EPRC, for funding this research.

ABSTRACT

Structural failure of transmission line systems is often attributed to dynamic effects such as a broken conductor, a broken insulator, or conductor galloping. The focus of this research was to develop a computer program, DYNTRN, that can analyze the structural response of a transmission line system due to dynamic effects, and present the response in a graphical form. The program uses the stiffness method to analyze a system consisting of conductors, insulators, and support structures. Four types of elements can be used to model the transmission line components: beam elements, cable elements, truss elements, and spring elements. A dynamic condensation method was introduced to efficiently model cable elements. Geometric nonlinearities were accounted for using the Newton-Raphson method. State-of-the-art software tools and object oriented design were used to develop a program that is modular and interactive. An object oriented method was developed to efficiently store and solve the stiffness matrix of the structure. Results obtained from the program were verified using commercial finite element software. The program was also validated using published experimental work. The final product of this research is a computer program that can graphically simulate dynamic behavior of transmission lines.

CHAPTER 1 - INTRODUCTION

1.1 Background

Structural failure of transmission lines due to extreme weather related conditions is among the major problems facing the power industry. Losses arising from these failures are not only due to the loss of business resulting from electricity outages, but are also due to the cost of repairs.

Designing a transmission line support structure for extreme environmental conditions is a challenging problem due to several reasons. First, the main loads affecting transmission lines, such as wind and ice, are hard to predict. Secondly, most of the critical forces are dynamic in nature, and the response of the line to these forces is complex. A broken conductor, a broken insulator, or conductor galloping are examples of failure conditions that cause these critical forces.

Dynamic behavior of transmission lines has been studied experimentally using full scale lines [1,2], as well as scale models [3,4]. Testing of full scale models is expensive, and can be limited in scope. Moreover, most of the dynamic tests produce inelastic deformations and broken components, making it hard to investigate different loading scenarios. Although scale model testing is less expensive than testing full-scale transmission lines, results produced by scale models are not as accurate. In addition, it is still not very economical to investigate a large number of alternative loading conditions using scale models.

Computer modeling can be an attractive alternative. This is least expensive, and in many cases can provide accurate results, particularly when experimental data can be used to validate the computer model. Computer modeling is also an efficient way for conducting parametric studies that can yield a better understanding of the structural behavior of the line. Results obtained from computer models are also more comprehensive. For example, stress or strain can be checked at any location; in contrast, an experimental model will only provide results at a finite number of locations. Graphical simulation is an added advantage to computer modeling, as the structural behavior of the line can be visualized.

There are a wide variety of structural analysis programs that can perform both static and dynamic analyses, utilizing finite element techniques to solve for unknown structural displacements and stresses. Most of these programs, however, are not adequate for failure analysis of transmission lines. Some programs have been developed specifically for the analysis of transmission lines, such as ETADS [5], CABLE7 [6], and BROKE [7]. ETADS, however, is oriented toward static analysis, and does not account for the dynamic effects occurring within the span of the cable element[1]. CABLE7 is a two-dimensional program, and accounts for the effects of the transmission line support structures by utilizing linear springs. BROKE, on the other hand, models a conductor using a single linear truss element along the span of the conductor. This approximation does not accurately represent the dynamic behavior of the cable within its span. Both CABLE7 and BROKE are limited to specific types of analyses, such as the broken conductor analysis.

1.2 Objective

The objective of the research presented herein was to develop an analytical tool that is capable of analyzing a complete transmission line subjected to different dynamic loading conditions, such as a galloping conductor, a broken conductor or a broken insulator. An additional objective was that the software be capable of presenting the results in a graphical form to achieve better understanding of the structural behavior of the transmission line. To achieve this goal, a software referred to as DYNTRN, which is capable of analyzing the main components of a transmission line such as conductors, support structures, and insulators, was developed.

1.3 Methodology

DYNTRN used the stiffness method to calculate the deformations and forces in the transmission line components. The primary challenge in the development of the software was the representation of the cable element. Due to the nonlinear problem associated with the

[1] This applies to ETADS version 2.15C

dynamic behavior of the cable element, it was necessary to represent the cable element using a number of internal sub-elements, such as axial tension-only elements. A method was developed to extend the static condensation method to dynamic nonlinear problems. This technique allowed the use of special routines that can minimize numerical instabilities within the span of the cable and speed up the convergence process.

Another challenge faced in the development process was the size of the finite element model of the transmission line. A transmission line computer model usually consists of several support structures, with several spans of conductor phases attached. The stiffness matrix produced during this process is usually sparse, i.e., most of its elements are zero. Therefore, a pointer-based method was developed to efficiently store and solve sparse matrices. Pointers are programming variables that store computer memory addresses.

The next issue in the development process involved the means of development, or programming. The main priority was to develop a software that was highly interactive, modular, and possessed good graphical capabilities. Therefore, Object Oriented Programming (OOP) was the programming method of choice. OOP is a programming philosophy that produces a software code that is easy to maintain and modify.

Developing a software to simulate conductor galloping was also another challenge. The objective here was not to develop a galloping model, but rather to document several galloping models, and allow the user to decide which of these to be used. There are several models present in the literature to calculate or predict the amplitude and shape of galloping. Galloping amplitudes provided by any of these galloping models may be used as input to the dynamic program, DYNTRN.

The final step in the development process was the validation of the dynamic analysis program, DYNTRN. This was accomplished by comparing DYNTRN results with closed form solutions, or other finite element programs. In addition, DYNTRN was calibrated using pertinent published experimental data by others.

CHAPTER 2 - LITERATURE REVIEW

As stated in chapter one, the objective of this research was to develop a computer software capable of analyzing a complete transmission line due to dynamic events such as a conductor galloping event, a broken conductor event, or a broken insulator event. Because the cable element, which is used to model a conductor, is of primary importance in any transmission line analysis program, it was necessary to study cable behavior and investigate the different methods developed for modeling this element. Previous studies related to the calculation of the static or dynamic effect of a broken conductor or a broken insulator were also reviewed. Experimental data published in the literature was used to validate the software, DYNTRN.

An extensive review of the different models developed to analyze conductor galloping was also undertaken. A large number of models were found in the literature, ranging from simple models to extremely complex and detailed models. In the opinion of the author of this work, the choice of galloping model depends on the input available to the user, and the accuracy sought from the analysis. The literature review presented herein is intended to inform the user of DYNTRN about the different galloping models published in the literature, since galloping amplitude and frequency represent an essential part of the input for the galloping analysis performed by DYNTRN.

One of the objectives of the work presented herein was to develop a highly interactive program allowing the user to investigate different analysis scenarios. To accomplish this objective, literature related to Object Oriented Programming (OOP) technique was reviewed, and its use in structural analysis programs was investigated.

2.1 The Cable Element

2.1.1 Theoretical Analysis of Cables

Cables are flexible elements that have virtually no bending stiffness. Due to the high flexibility of cable elements, they undergo large deflections. These deflections provide cables with a sufficient stiffness to sustain applied loadings. Due to a uniform load, cables deflect in the form of a catenary as shown in Figure 2.1. Irvine [8] presented the catenary equation of the cable as follows:

$$l = \frac{HL_0}{EA_0} + \frac{HL_0}{W}\left\{\sinh^{-1}\left(\frac{V}{H}\right) - \sinh^{-1}\left(\frac{V-W}{H}\right)\right\}$$

$$h = \frac{WL_0}{EA_0}\left(\frac{V}{W} - \frac{1}{2}\right) + \frac{HL_0}{W}\left[\sqrt{1+\left(\frac{V}{H}\right)^2} - \sqrt{1+\left(\frac{V-W}{H}\right)^2}\right] \qquad 2.1$$

Where,

l = horizontal span,

h = vertical distance between the support as shown in Figure 2.1,

W = the total load on the cable,

L_0 = the original length of the cable,

E = modulus of elasticity,

A_0= cross sectional area of the cable,

H = horizontal tension in the cable,

V = vertical reaction at the left support.

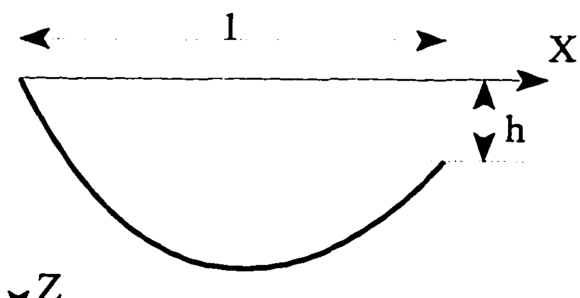


Figure 2.1 Catenary Cable

Irvine [8] also presented the approximation of the catenary formulas using parabolic equations. The parabolic equations were derived by assuming the load to be distributed on the span of the cable rather than the deformed profile. For flat cables, the error due to this assumption is acceptable. In addition, Reference [8] also studied the dynamics of the cable element and the procedure to extract mode shapes for a flat horizontal cable. The mode-shapes of a flat horizontal cable as presented in Reference [8] are listed below.

i) For out-of-plane modes:

$$\omega_n = \frac{n\pi}{l}\sqrt{\frac{H}{m}}, \quad v_n = A_n \sin\left(\frac{n\pi x}{l}\right) \tag{2.2}$$

ii) For anti-symmetric in-plane modes:

$$\omega_n = \frac{n\pi}{l}\sqrt{\frac{H}{m}}, \quad w_n = A_n \sin\left(\frac{n\pi x}{l}\right)$$

$$u_n = -\left(\frac{mgl}{H}\right)A_n\left\{\left(\frac{1}{2}-\frac{x}{l}\right)\sin\left(\frac{n\pi x}{l}\right) + \frac{1-\cos(n\pi x/l)}{n\pi}\right\} \tag{2.3}$$

iii) For symmetric in-plane modes:

$$\tan\frac{\Omega_n}{2} = \frac{\Omega_n}{2} - \frac{4}{\lambda^2}\left(\frac{\Omega_n}{2}\right)^3, \quad \omega_n = \frac{\Omega_n}{l}\sqrt{\frac{H}{m}}$$

$$w_n = A_n\left(1 - \tan\left(\frac{\Omega_n}{2}\right)\sin\left(\Omega_n\frac{x}{l}\right) - \cos\left(\Omega_n\frac{x}{l}\right)\right) \tag{2.4}$$

$$u_n = \left(\frac{mgl}{H}\right) A_n \left[\frac{\Omega_n^2 L_x}{\lambda^2 L_e} - \left(\frac{1}{2} - \frac{x}{l}\right)\left\{1 - \tan\left(\frac{\Omega_n}{2}\right)\sin\left(\Omega_n\frac{x}{l}\right) - \cos\left(\Omega_n\frac{x}{l}\right)\right\} - \frac{1}{\Omega_n}\left\{\Omega_n\frac{x}{l} - \tan\left(\frac{\Omega_n}{2}\right)\left(1 - \cos\left(\Omega_n\frac{x}{l}\right)\right) - \sin\left(\Omega_n\frac{x}{l}\right)\right\}\right]$$

Where,

v_n = mode shape in the out-of-plane direction,

u_n = mode shape in the longitudinal direction,

w_n= mode shape in the vertical direction,

n = mode number,

m = mass per unit length,

g = acceleration of gravity,

x = variable along the x axis,

$\lambda^2 = (mgl/H)^2\, l\,/\,(HL_e/EA_0)$,

$L_e \approx l(1+8\,(d/l)^2)$, where d is the cable sag,

$L_x = l[x/l + (3/8)(mgl/H)^2\,(x/l - 2(x/l)^2 + (4/3)(x/l)^3]$,

A_n = is the scaling variable for the mode.

In contrast to the notation used by Irvine [8], the following notation for n was used in this review: n = one for the first symmetric mode, two for the first anti-symmetric mode, three for the second symmetric mode, four for the second anti-symmetric mode and so on.

As shown above, the in-plane symmetric modes are calculated by solving the first equation in the series of the symmetric in plane mode-shapes equations, (iii), to obtain Ω_n. This equation is a nonlinear equation and can only be solved numerically. Irvine [8] performed a parametric study on the mode shapes of the cable element and found that a mode cross over between the symmetric and anti-symmetric modes occurred at certain values of λ. For example at $\lambda^2 = 4\pi^2$, the first symmetric mode, n=1, and the first anti-symmetric mode, n=2, crossed, i.e., they had the same frequency of vibration. The same situation occurred for the second symmetric mode, n=3, and the second anti-symmetric mode, n=4, at $\lambda^2 = 16\pi^2$. Irvine [8] compared the natural frequencies calculated using the flat-sag formulation to that calculated using deep-profile equations, and found that the error was less than 6% when the sag-to-span ratio was 13%. Cable mode shapes were also presented by West et al. [9] and Nariboli et al. [10].

2.1.2 Numerical Analysis of Cable Elements

Several numerical methods were proposed for solving cable structures. O'Brien and Francis [11] presented an iterative numerical method to solve a two-dimensional cable structure subjected to static concentrated loads. The cable segments were treated as straight links. In subsequent work, O'Brien [12] used the catenary equations to calculate the tension in the cable segments instead of the straight link equations. This modification not only yielded better accuracy, but also allowed for analyzing a combination of uniform and concentrated loadings on the cable. In his work, O'Brien [12] also generalized the method to a three-dimensional static analysis. Similar methods were developed by Skop and Ohara [13,14].

Peyrot et al. [15,16] presented an iterative method for solving static and dynamic problems involving cable elements. The method was based on the catenary cable equations, and was used to develop a numerical algorithm to calculate the stiffness matrix of the cable element, thus allowing it to be used in a direct stiffness program.

Baron and Venkatesan [17] developed the stiffness matrix of a two-nodes truss element in three-dimensional space, including the effect of stress stiffening. They used the direct stiffness method in a nonlinear iterative scheme to solve for several cable structures subjected to static concentrated forces. Similar methods based on two-nodes truss elements were developed by Webster [18], Mitsugi et al. [19] and Broughton et al. [20].

Desai et al. [21] formulated the stiffness matrix of a three-nodes cable element using a parabolic assumed function. A comparison between the three-nodes element and the two-nodes truss element was presented in the reference, and showed that the three-nodes cable element is more accurate in a static analysis [21].

In a static analysis for a cable element subjected to uniform loads, the cable element developed using catenary equations is superior to other elements because of its exact formulation, thus allowing fewer elements to be used in the analysis. For non-uniform static loads, as well as in dynamic analyses, the catenary assumption for the shape of the cable element is no longer valid; therefore, the superiority of one formulation over the other cannot be established without further investigation. In the work presented herein, the cable element was modeled using a variable number of two-nodes tension-only axial elements. The number of sub-elements within the span of the cable is decided by the user.

2.2 Broken Conductor Analysis

2.2.1 Experimental Studies of Broken Conductors

Comellini [22] reported that the probability of occurrence of a broken conductor or insulator string in Italy and France was about $0.66*10^{-5}$ per structure per year. Reference [22] used a reduced scale model to study longitudinal loads and base moments acting on the structures due to a broken conductor. In the study, static and dynamic longitudinal forces were calculated as a function of span, insulator length, and structure stiffness. Comellini [22] found that increasing the structure flexibility decreased the longitudinal forces exerted on the structure. Taking the probability of slippage of a conductor into consideration, Reference [22] calculated the risk of failure of a structure due to a broken conductor.

Govers [4] studied the problem of a broken conductor using a 150-kv transmission line that was retired from service near Amsterdam. This study was complemented with tests performed by Govers on a reduced scale model. The effect of the length and type of insulator, the span, the conductor initial tension, and the type of conductor material on the impact ratio of the longitudinal load due to a broken conductor were considered in that work. Govers [4] also studied the effect of the number of spans included in the model on the results. The study yielded relationships to calculate the impact ratio due to a broken conductor using the span-to-sag ratio and the span to the insulator's length ratio.

Peyrot [1] performed a series of broken conductor and broken insulator tests on a retired eight span 138-KV transmission line in Wisconsin. The effect of the broken conductor on adjacent spans was studied, and the dynamic longitudinal loads were measured.

Mozer et al. [3] studied a model transmission line to investigate the broken conductor effect on transmission structures. In his work, Mozer and his colleagues, recorded the time history of the loads and moments due to a broken conductor. Response spectra curves to calculate the response factor of structures subjected to a broken conductor were also developed. Kempner et al. [23] carried out similar work. Experimental data from

References [1,3] was used to validate DYNTRN simulation capabilities, as presented in Chapter 4.

2.2.2 Analytical Analysis of Broken Conductors

Several analytical methods were presented to calculate the forces generated due to a broken conductor. A review of some of these methods was presented in [6].

Baenziger [6,24] developed a computer program, CABLE7, to calculate the dynamic response of a transmission line due to a broken conductor. The support structures were represented using spring elements. The program calculated the time history of the conductor tension and the insulator tension in the span adjacent to the break. Results were in good agreement with experimental data in the literature. In her work, Baenziger also performed a parametric study to investigate the effect of different line parameters on the impact factor due to a broken conductor. Parameters studied included the insulator length, the span length, the stiffness of the support structures, and the initial conductor tension.

Siddiqui and Fleming [7] developed a broken conductor analysis software, BROKE, using the stiffness method. The cable element was approximated using a single two-nodes truss element to represent the entire length of the conductor. To account for the cable sag, a modified Young's modulus, E_{eq}, was used in Reference [7], as follows:

$$E_{eq} = \frac{E}{1 + \frac{(wl)^2 AE}{12T^3}} \qquad 2.5$$

Where,

E = modulus of elasticity of the cable,

w = weight per unit length of the cable,

A = cross sectional area of the cable,

l = span length,

T = horizontal tension.

The insulator was also approximated to eliminate potential numerical instabilities.

Reference [7] compared the results obtained by BROKE to experimental results reported by Mozer [3]. Loads measured at the arms of the structures fell within 10%. Moments at the base of the model structures were within 20%.

2.3 Broken Insulator Analysis

A broken insulator event refers to an event where the insulator breaks, followed by the free fall of the conductors. The conductors will then fall on the ground, hang on the bracing or the arm of the support structure, or simply hang in the air combining two spans into one. Investigation of transmission line failures documented in References [25,26], showed that a broken insulator was among the probable causes of cascade failures. The broken insulator phenomenon has received little attention from researchers, as compared to the event of a broken conductor.

Comellini [22] studied the broken insulator phenomenon using a reduced scale model. Reference [22] stated that the longitudinal imbalance loads calculated in the final static position, whether hanging in the air or touching the ground, were very small. The peak dynamic transient forces were found negligible when the conductor was touching the ground, but reached relatively higher values in other cases.

Peyrot and Goulois [27] presented an algorithm to analyze a cable partially lying on the ground. Wipf et al. [25] used the software developed by Peyrot et al. [27] to analyze a broken insulator event. In their study, it was found that 92% of the conductor's length was lying on the ground; therefore, the tension in the conductor was reduced considerably, producing a high imbalance longitudinal load.

To the author's knowledge, no published work has presented a dynamic analysis of a broken insulator except the work presented in Reference [26]. Two spans of transmission line conductors where analyzed using ETADS [5]. A fuse element which automatically breaks at a specified time was used as the center insulator. The analysis simulated a real transmission line failure where the insulator broke, causing the conductors to fall and hang on the structure's cross bracing. The static analysis showed virtually no change in the conductor tension as it fell to its new position. Dynamic transient analysis, however, showed a decrease in tension of more than 20% as the conductor fell. An increase of about 10% in the tension was also observed as the conductor vibrated about its new equilibrium position.

2.4 Conductor Galloping

Conductor galloping is defined in the transmission line reference book [28] as a low frequency, high amplitude, primarily vertical motion that is usually caused by a moderately strong steady wind acting on an asymmetrically iced conductor surface. Due to the high amplitude motion associated with galloping, flash overs as well as mechanical failure of conductors or insulators can occur. Any of these failures can lead to a cascade failure of several transmission support structures [25].

2.4.1 The Galloping Phenomena

In 1932, Den Hartog [29] provided an explanation of the galloping phenomena and why it occurred. Reference [29] studied galloping on an elliptic cross-section and showed how the lift forces changing with the angle of attack of wind can reinforce the galloping motion. As the conductor moves vertically, the relative direction of the wind with respect to the conductor, i.e., the wind angle of attack, will change. This change will cause a corresponding change in lift forces that may reinforce or suppress the galloping motion depending on the conductor's cross-section and the initial angle of attack. The study concluded that instability would occur if the negative slope of the lift curve exceeded the damping action due to drag. Namely,

$$\frac{dL}{d\alpha} > D \ ,$$

where

L = the lift force acting on the conductor cross section,

D = the drag force acting on the conductor cross section,

α = the wind angle of attack.

The effect of the conductor torsional motion in the initiation of galloping was later identified by Ruedy [30] and Cheers [31]. Edwards et al. [32] carried out an experimental investigation to study the galloping behavior of conductors by monitoring the translational and torsional amplitudes of several transmission lines. The study also recorded forces on an experimental test line specifically constructed for conductor galloping investigation. The test line was covered with a half-circular wooden airfoils to form a D-Section. Edwards et al. [32] concluded that the torsional motion of the conductor was important in initiating and sometimes sustaining galloping instability. The relation between the galloping frequency of the torsional and translation motions was also studied. Reference [32] found that in most cases of galloping, the frequency of the torsional and vertical vibrations were equal, although the natural frequencies of the respective degrees of freedom were not necessarily the same.

Richardson et al. [33] investigated the effect of the wind angle of attack on the galloping stability of transmission lines by analyzing the dynamic behavior of a conductor using a lumped parameter system, as well as a continuous system. Reference [33] showed that the stability criteria for the continuous system can be solved using an equivalent lumped parameter system. Stability criteria for two and three degrees of freedom configurations, including horizontal, vertical, and torsion, were studied. An experimental verification using wind tunnel showed good correlation with the theoretical model with respect to the stability regions and galloping frequencies. The critical wind speed above which galloping occurred showed some discrepancies between the theoretical and experimental model.

Nigol and Clarke [34] presented an experimental investigation that proved the importance of conductor torsion in initiating galloping. They found that ice usually built up on the upper quarter region of the conductor on the windward side. The orientation of ice caused the conductor to rotate, thus decreasing its effective torsional stiffness. Further reduction of the torsional stiffness resulted if the torsional motion was excited. Due to this reduction, the fundamental torsional frequency would decrease and reach a value comparable to the fundamental vertical frequency. If the value of the torsional frequency was equal to or a multiple of the vertical frequency, locking between the torsional and the vertical modes would occur; thus, galloping in the vertical mode could be initiated. This explanation was verified using a test line consisting of three 800 ft. spans. The conductor was fitted with plastic crescent shapes and additional weight to simulate natural ice accumulation. The artificial ice decreased the torsional fundamental frequency from the initial value of 2.4 Hz. to 0.8 Hz. The fundamental vertical frequency for the conductor was 0.28 Hz. The galloping motion was self excited at average wind speeds ranging from 12 to 25 mph. One, two, and three loop galloping cases were obtained by varying the orientation of the artificial ice. Further reduction of the torsional stiffness, and hence the torsional frequency, was believed to have occurred due to the aerodynamic moment caused by wind.

Nigol and Buchan [35,36] studied the phenomena of galloping using simulated ice shapes. The purpose of the study was to investigate the mechanism of galloping and the effect of torsional motion. A series of wind tunnel tests was performed that included:

1. Static tests to measure the aerodynamic coefficients of four ice shapes.
2. Dynamic tests where the conductors were restrained to move in a pure vertical direction to study Den-Hartog galloping [29].
3. Dynamic tests to investigate self-excited pure torsional motions.
4. Unrestrained tests to study the interaction between torsional and translational motions.

The authors of the study [35,36] reported that no galloping in the pure vertical tests was generated, even in cases where static wind tunnel tests showed instability according to Den-Hartog equation. The study also showed that galloping could occur in regions where the slope of the lift curve is positive if galloping was initiated due to the self-exciting torsional motion.

Jones [37] examined the effect of the horizontal-vertical coupling of the conductor galloping motion by deriving the analytical equations of motion including vertical and horizontal degrees of freedom. Reference [37] documented that the coupling term between the horizontal (out-of-plane) and vertical equations did not allow a vertical motion to exist if the horizontal motion was set to zero. This result might explain why Nigol and Buchan [35] were unable to excite the vertical galloping motion when they restrained the horizontal motion. Furthermore, the study also found that an initial horizontal displacement or velocity could cause vertical galloping. It should be noted, however, that the results presented in the reference [37] were concluded based only on the analytical equations listed in that reference. No experimental data was included to support these findings.

Yu et al. [38,39] developed a three degrees-of-freedom (DOF) model, which included the longitudinal, vertical, and torsional DOF. The references also compared several models with different combinations of DOF included. Models with three DOF, vertical DOF only, vertical-torsional DOF, and vertical-longitudinal DOF were compared. The vertical and vertical-longitudinal models produced results that were more conservative than the 3 DOF model. The results of the vertical-torsional model, however, were close to the 3 DOF model (see reference [39] for details).

From the above review, one can conclude that torsional motion has a significant role in analyzing and modeling conductor galloping. For a more accurate and refined model, horizontal motion should be considered.

2.4.2 Modeling Galloping using Analytical Methods

Galloping is a self-excited motion that is caused by negative aerodynamic damping acting on the iced conductor. The aerodynamic forces acting in the vertical and horizontal directions across the conductor, as well as the aerodynamic torsional moments, depend on the wind angle of attack, which in turn is dependant on the velocity and torsional displacement of the moving conductor. These forces also depend on the ice shape coating the conductor as well as the direction and speed of wind. If the ice shape as well as wind speed and direction is assumed constant, aerodynamic forces will be the only variable. The aerodynamic forces will change according to the changing value of the displacement and velocity of the conductor. Therefore, the general equations of motion describing galloping can be written as:

$$[M]\{\ddot{U}\} + [C]\{\dot{U}\} + [K]\{U\} = \{F(\{\dot{U}\},\{U\})\} \qquad \textbf{2.6}$$

Where [M], [C], and [K] are the mass matrix, the damping matrix, and the stiffness matrix, respectively. $\{\ddot{U}\}$, $\{\dot{U}\}$, and $\{U\}$ are the acceleration vector, the velocity vector, and the displacement vector, respectively. $\{F(\{\dot{U}\},\{U\})\}$ is the aerodynamic force vector acting on the conductor including drag, lift, and moment. The above equation is a nonlinear equation that can be integrated over time.

Desai et. al. [40] developed a finite element model to represent several spans of a transmission line. Reference [40] used a three-node cable element with four DOF at each node: three translational and one torsional. The support structures were considered rigid. The insulators were modeled using equivalent springs. A finite difference time integration scheme was used to solve Equation 2.6. Through this analysis, a time history of the vibrating cable was generated. The method was compared to the experimental work performed by Edwards [32] and Stumpf [41]. Good correlation was found between the analytical and experimental model. An inherent problem with the time integration method was the high computational effort. Reference [40] reported that galloping was simulated for more than two thousand seconds in order to reach a periodic response. Estimating the aerodynamic forces acting on the galloping conductor is another difficulty associated with the method listed in Reference [40], because the ice shape coating the conductor is usually not known.

2.4.2.1 Solving the Galloping Problem

Several methods were introduced to solve the nonlinear galloping equations using nonlinear analysis techniques, such as the perturbation theory. Blevins and Iwan [42] solved a two degrees-of-freedom (torsional and vertical) lumped parameter model using asymptotic techniques. Egbert [43] used the describing function method to solve one degree of freedom vertical galloping. Byun and Egbert [44] refined the describing function method to include torsional motion. Richardson [45,46] used energy methods to solve the galloping problem. Desai et al. [47] used averaging methods to solve the same problem presented by Blevins and Iwan [42]. Yu, Desai, Shah, and Popelwell [38,39] solved a three DOF lumped model using averaging techniques. The model was built by defining the assumed displacement function of the cable to match one of the mode shapes, thus developing the mass, stiffness, and damping matrices associated with the three DOF lumped element. Adjacent spans were modeled using springs. Only one mode shape could be represented at a time, i.e., the mode shape that needed to be investigated. The nonlinear equations were solved using perturbation analysis and averaging techniques. Results were compared to the experimental work by Edwards and Madeyski [32], and Nigol and Clarke [34], and double checked using time integration.

2.4.2.2 Estimating the Aerodynamic Forces

Nigol and Buchan [35] tried to simulate natural ice occurrences by spraying different air-water mixtures on a conductor placed in a walk-in refrigerator. Plastic replicas of the most common ice formations were made and tested in a wind tunnel to develop a relationship between the aerodynamic coefficient used to calculate the aerodynamic forces and the wind angle of attack.

Hunt and Richards [48] used a lift coefficient changing from 0.6 to -0.6 at zero-angle of attack, and a constant drag coefficient of 1.0, with the assumption that these would cause worst case galloping. In their work, an energy balance method was used to obtain the vertical galloping peak-to-peak amplitude, which was estimated to be:

$$Y_{\max} = 0.26\frac{V_w}{f} \qquad 2.7$$

Where, V_w is the wind speed and f is the natural frequency of the conductor.

In the equation presented by Hunt, the galloping amplitude is directly proportional to wind speed, which is usually true for low wind speeds. Reference [48] cautioned, however, that when the value of the wind speed exceeds a certain value, the maximum galloping amplitude will start to decrease. Baenziger et al. [49] modified the equation presented by Hunt and Richard to include a limit on the wind speed beyond which the galloping amplitude will not increase. The modified equation presented in Reference [49] is as follows:

$$\begin{aligned} Y_{\max} &= \frac{0.26V_w}{f} && 0 \le \frac{V_w}{f} \le 125d \\ Y_{\max} &= 33d && \frac{V_w}{f} > 125d \end{aligned} \qquad 2.8$$

Where $Y_{\max}$ is the peak-to-peak galloping amplitude and d is the conductor's diameter.

2.4.3 Modeling Galloping using Statistical Methods

A galloping guideline using galloping observations was established by Davison [50] who developed galloping ellipses which were used widely for design of clearances in transmission lines. Davidson's model has been refined and modified by later researchers [28].

Rawlins [51] used the galloping observations from numerous reports to study the correlation between different parameters of a transmission line and conductor galloping. The galloping data covered a broad range of lines with different design, conductor sizes and geographical areas. One objective of the study was to differentiate between the lines that gallop from these that do not gallop. Another objective was to find a correlation between the line parameters and the number of galloping loops. The final objective was to estimate the galloping amplitude based on the line parameters. Wind speed was eliminated from the study. Rawlins [51] found that one-loop galloping was mostly dominant, followed by two loops, and then three loops, and that higher number of loops occurred in dead-end spans more than in suspension spans. Reference [51] also found that the ratio of the conductor tension, T, to the conductor weight per unit length, w, can be successfully used to distinguish between galloping and non-galloping cases. Higher values of T/w didn't favor galloping. Rawlins [51] also developed a chart to estimate galloping expected maximum amplitudes (see Figure 2.2). The chart calculates the value Y_{max}/S based on the catenary parameter, M', and T/w. M' is expressed as:

$$M' = \begin{cases} 10.67\ D^3/IS^2 \textit{ for suspension spans} \\ 21.3\ D^3/IS^2 \textit{ for semi-suspension spans} \\ 54.2 * 10^6\ D^3/S^4 \textit{ for deadend spans} \end{cases}$$

Where, D is the sag in meters, I is the insulator length in meters and S is the span length in meters. Y_{max} is the peak-to-peak galloping amplitude.

Although Rawlins model was useful in establishing realistic guidelines for estimating galloping amplitudes, in the author's opinion, a larger sample of data, and including weather conditions in the model, would yield more accurate results. In addition, the data used to develop the model was based on observations, and therefore one may argue its accuracy.

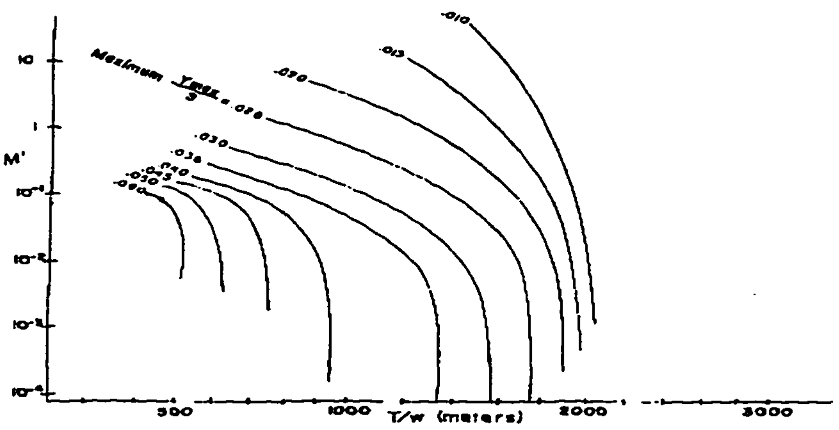


Figure 2.2 Galloping Amplitude - Statistical Methods [51]

2.4.4 Loads Generated by Galloping

Dynamic loads generated by galloping were studied by Krishnasamy [52], who reported a series of measurements on three sites in southern Ontario. Vertical loads as high as twice the conductor weight was reported. Dynamic forces generated due to galloping were also studied by McConnell [2]. The objective of his work was to study the forces generated by galloping on transmission structures, and to investigate the effect of a proposed energy absorber device to be used in controlling galloping and reducing the impact of galloping forces on transmission structures. Galloping forces were measured using force transducers installed between the structure and the insulator. The maximum dynamic force reported by McConnell [2] was about 15% higher than the force due to static loads.

Baenziger et al. [49] developed an analytical model for calculating additional conductor tension caused by galloping. A simplified equation was developed by assuming the mode shape of the conductor to be a simple harmonic shape. To utilize the model, one needs to provide the galloping amplitude as an input to calculate the dynamic galloping forces. The analytical model was checked versus the results of a wind tunnel experiment. The amplitudes used in the analytical model were calculated using the equation developed by Hunt and Richard [48] (see Equation 2.7). Not all of the presented experimental results were very close to the results calculated using the analytical model. Discrepancies ranging from 1.4 to 35.9

percent were reported. Reference [49] also tested the analytical model using field data reported by Krishnasamy [52]. Variation between the analytical and field results were also reported. Maximum variation reported was 30.1 percent. A detailed presentation of the analytical model can be found in [53].

Similar work was done by Wu [54], who developed an analytical model to calculate the conductor end forces caused by a galloping conductor using numerical analysis, and compared it with experimental results. In the experimental model, a support excitation was used to generate galloping in lieu of a wind tunnel test. A variation of 20% between the analytical and experimental model was reported.

2.5 Object Oriented Programming

Schildt [55] defines Object Oriented Programming (OOP) as a new and more efficient way of programming and manipulating complex data. When programming in an object oriented fashion, the programmer decomposes the problem into subgroups of code and data related to the group. These subgroups are translated into self-contained units called objects. Objects contain data pertaining to the object, the code to internally manipulate the data, and the code that interfaces the object with other objects in the program. Objects can be thought of as variables of new types defined by the user.

2.5.1 Benefits of Object Oriented Programming

The major benefit of using OOP in a finite element program is that the code produced is more readable and easy to maintain. Filho et al. [56] have expressed that the revolution in hardware and software in recent years has made software design more complex; therefore, a better way of programming Finite Element Methods (FEM) was needed. Arruda et al. [57] also explained the benefits of using OOP for the development of a structural analysis system under a graphical environment, specifically discussing the C++ language under Microsoft windows. Miller [58] discussed the benefits of OOP in integrating the traditional pre-processors and post-processors to the main module of a structural analysis or design program, thus making the program more interactive and efficient. Reference [58] also discussed the use of OOP to encourage concurrent and distributed processing, through which different people involved in the creation of a structure can interact and communicate. Miller also explained the attractiveness of using object-oriented databases to provide persistent storage for engineering systems. Gajewski [59] discussed the need for a new programming paradigm, such as OOP, to replace the "traditional waterfall algorithm-driven structured programming approach," and stated that the traditional system of programming, and in the majority of cases Fortran as a programming language, is no longer adequate for the increasing size and complexity of finite element programs. Zimmermann et al. [60] discussed specific benefits of OOP, and supported their discussion with examples. These benefits included:

1. Auto-description capability: the ability of the code to describe itself without additional comments.
2. Non-sequential aspect of procedures: viewing the code as a collection of objects responding independently to messages without a constraint to a prescribed sequence of operations.
3. Independent testing capabilities: because objects are developed independently, it is possible to test the internal methods of any object independent of other objects.
4. Automatic storage management capability.

2.5.2 Using Object Oriented Programming in Structural Analysis Software

Although the OOP popularity in the software industry has exponentially increased in the last decade, using OOP for analyzing engineering problems, and specifically structural engineering problems, has been slower. Gajewski [59] attributed the reluctance of engineers to move to the OOP technology to three main reasons:

1. The presence of old systems programmed in a structured traditional way and does not mix well with OOP design.
2. The presence of tools and programs that does not fully support OOP.
3. The investment required to learn OOP.

Despite the factors listed above, OOP has recently picked up momentum among structural engineering researchers. Forde et al. [61] presented one of the first attempts to produce an object-oriented finite element program. Reference [61] explained the object oriented concepts and presented the prototypes for the objects used in the development of an isoparametric two-dimensional finite element program. Reference [61] also compared the object-oriented development to a traditional procedural development to demonstrate the advantage of OOP in such issues as development time, code maintenance, and testing. Scholz [62] presented the object-oriented development of a finite element program using the C++ language. The program is capable of solving two, three, and four nodes beam elements based on theory of Timoshenko [63,64]. The reference highlighted the benefits of OOP in improving the readability of programming code. Zimmermann et al. [60,65,66] discussed the theory of object-oriented design, and presented examples to compare object-oriented programs to ones implemented in the traditional procedural approach. References [60,65,66] investigated the advantage of OOP on such issues as code readability, maintainability, ease of programming and debugging, and extendibility. Speed of execution of two finite element programs was also compared, one developed in Fortran, and the other developed in C++ using OOP. The finite element programs consisted of two phases: the assembly phase where the stiffness matrix was computed, and the solution phase where the matrix was solved. In the assembly phase, the procedural implementation using Fortran was about 35% faster than the object oriented implementation using C++. In the solution phase, however, the Fortran speed advantage disappeared. This lag in speed was attributed to the higher level of abstraction associated with the C++ implementation.

Yu et al. [67] developed an object-oriented Enhanced Entity Relationship (EER) data model which relied on an object-oriented class library to perform the basic operations of finite element analysis. The model included Database Management System (DBMS) techniques. The model was used to perform inter-laminar stress analysis of composite structures. Ju et al. [68] demonstrated the use of object-oriented techniques to include sub-structuring in a finite element program. Using this technique, multi-level sub-structuring was achieved. Miki et al. [69] proposed an object oriented approach to perform large displacement analysis on truss structures using the relaxation method. The object oriented design was chosen in

Reference [69] to give a high level of modularity so that the program can be easily extended.

2.5.3 Efficiency Issues in Object Oriented Programs

Object oriented programming uses a high level of abstraction to achieve a highly modular and readable code. This high level of abstraction, however, produces a code that is slower than that produced by a traditional procedural approach. Miller [58] affirmed that there was a speed penalty associated with OOP, but considered it modest and unimportant. Miller reported a speed penalty in one case in the order of 10-15%, but stated that the increasingly interactive programming environments would change the perception and need of speed as known in the batch processing modes. Forde et al. [61] stated that the decrease in execution speed due to OOP is not significant, and argued that in some cases the object oriented design might produce faster applications. Reference [61] added that expensive computational procedures could always be programmed in a native procedural language and connected to the object oriented system to improve execution speed. Devloo [70] discussed efficiency issues in object-oriented design, and stated that the overuse of the object oriented principles might yield a program that is very inefficient with respect to execution time. Reference [70] also showed that an object-oriented program can be comparable to a procedural program in speed if the former was written for maximum efficiency. Devloo discussed how such issues as dynamic memory allocation, functions returning objects by value, overuse of function calls, and using objects out of context could affect the execution efficiency of the program.

CHAPTER 3 - DESIGN OF THE SIMULATION SOFTWARE

3.1 Overview of Object Oriented Programming

Traditionally, programs were focused on procedures which manipulated global data, or data defined in the main program, and passed to procedures as arguments. Object Oriented Programming (OOP) is a software development philosophy which packages related data and functions in entities called objects. In an object-oriented program, any object should belong to an object type, or class. An object class defines the type of data to be stored in the object, the functions which manipulate the stored data, and the functions which provide the interface with other objects. Data and functions defined in a class might be private, i.e., not accessible by other objects, or public, i.e., can be accessed by other objects.

In OOP, after classes are specified, objects can be declared in the main program using the defined classes the same way as other variables are declared. By using objects, a more concise, readable, and modular code can be developed. This can be achieved through three main properties of objects: encapsulation, inheritance, and polymorphism.

Encapsulation is the property of hiding some of the implementation details of an object from other objects in the program. In this way, the internal implementation of an object can be changed without affecting other objects in the program. This property is important to produce a modular code that is easy to modify. For example, one can change the method by which an element calculates its stiffness matrix without changing other parts of the program.

By arranging objects in a hierarchal order, objects can inherit data and code defined for other objects. A beam element and a truss element, for example, possess properties and behaviors that pertain to both of them. By defining an element object that includes the data and implementation needed for most structural elements, one can then inherit different types of elements from the main element object. This will produce a more concise and readable code.

Polymorphism is a property through which a certain procedure can have more than one implementation, depending on the types of the arguments supplied to the procedure. A stiffness procedure, for example, can have different implementations depending on the type of element calling the procedure, allowing different elements to calculate their stiffness matrices using different methods, such as the direct stiffness or the finite element approach. For more discussion about OOP see references [55,71].

3.2 Object Oriented Design of DYNTRN

The simulation software, DYNTRN, consists of a number of objects that communicate with each other to accomplish the different tasks required by the program. Objects can be considered as the building blocks used to construct the overall program. Starting with objects that represent vectors and matrices, one can build more complex objects such as nodes and elements. These objects are combined to build the main object which represents the structure under consideration.

All the objects, used in the program inherit from an object called CObject. CObject is provided by the Visual C++ compiler [72], and gives the inheriting objects access to many services, such as input/output and error checking capabilities. In the next sections different object classes used in the program will be presented, and the relationship between these object classes will be discussed. A complete listing of the class prototypes used in DYNTRN is presented in Appendix A.

3.2.1 Basic Building Blocks

Vectors and matrices are the basic building blocks for most structural analysis programs. Three object classes, VECTOR, MATRIX, and BMATRIX, were developed to represent vectors, matrices, and banded matrices respectively. Routines to perform matrix and vector operations, such as addition, subtraction, multiplication, matrix transpose, and finding the norm of a matrix or a vector, were included in the three object classes. Routines for decomposition and back-substitution were also developed. An object class which represents sparse square matrices, SSMATRIX, was also developed. SSMATRIX is discussed in details in Section 3.2.9. An object class representing geometric vectors, GVECTOR, was also developed to be used in modeling coordinates, forces, and displacements. GVECTOR contains a VECTOR object used to represent the x, y, and z components of a 3-D geometric vector.

3.2.2 Coordinate System Object

A coordinate system object class, CSYS, was constructed as shown in Figure 3.1. It contains four GVECTOR objects to represent the origin of the coordinate system, and the three axes, X, Y, and Z. CSYS also contains a transformation matrix, T, used to transform geometric vectors from the coordinate system represented by the CSYS object to the global coordinate system, and vice versa. T can be written as:

$$[T] = \begin{bmatrix} \alpha_X & \beta_X & \gamma_X \\ \alpha_Y & \beta_Y & \gamma_Y \\ \alpha_Z & \beta_Z & \gamma_Z \end{bmatrix} \qquad 3.1$$

Where, α, β, and γ are the direction cosines of a geometric vector with respect to the global X, Y and Z axes, respectively. The subscripts, X, Y and Z refer to the geometric vectors representing the X, Y, and Z axes of the CSYS object, respectively.

A geometric vector can be transferred from the coordinates represented by the CSYS object to the global coordinates, and vice versa as follows:

$$\begin{aligned} \{V\}_{Global} &= [T]^t \, \{V\}_{CSYS} \\ \{V\}_{CSYS} &= [T] \, \{V\}_{Global} \end{aligned} \qquad 3.2$$

Where,

$\{V\}_{Global}$ is a 3-D geometric vector in the global coordinate system,

$\{V\}_{CSYS}$ is the same 3-D geometric vector transformed to the coordinate system represented by the CSYS object.

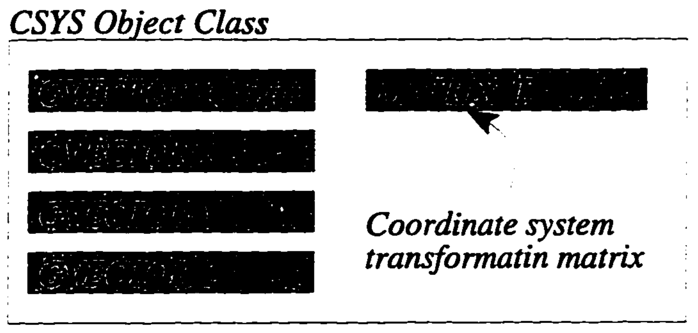


Figure 3.1 - Coordinate System Object Class

3.2.3 Material and Section Properties Object Classes

The material properties object class, MATERIAL, and the section properties object class, ELPROP, were constructed using an array of real values. Every material or section property is referenced using a specific integer, which is the index of the array element where the property is stored. By using the C language "#define" statement, integers could be assigned to names, thus providing an elegant way for retrieving properties. If "Section", for example, is an object of type ELPROP, one can retrieve the area property by writing the statement " A = Section.GetVal(AREA);" instead of the statement "A = Section.GetVal(0);" where the name "AREA" is assigned the integer zero using the statement "#define AREA 0". In this way, the developed code is more readable.

3.2.4 Force and Displacement Object Classes

Forces, moments, displacements, and rotations are represented by the FORCE, MOMENT, DISP, and ROTAT object classes respectively, as shown in Figure 3.2. The FORCE and MOMENT classes inherit from the GVECTOR class, and do not add any new functions to it. Therefore, FORCE and MOMENT classes are practically identical to the GVECTOR class. They were developed, however, for future extendibility, and to improve the readability of the code. The DISP object class also inherits the GVECTOR class, but adds to

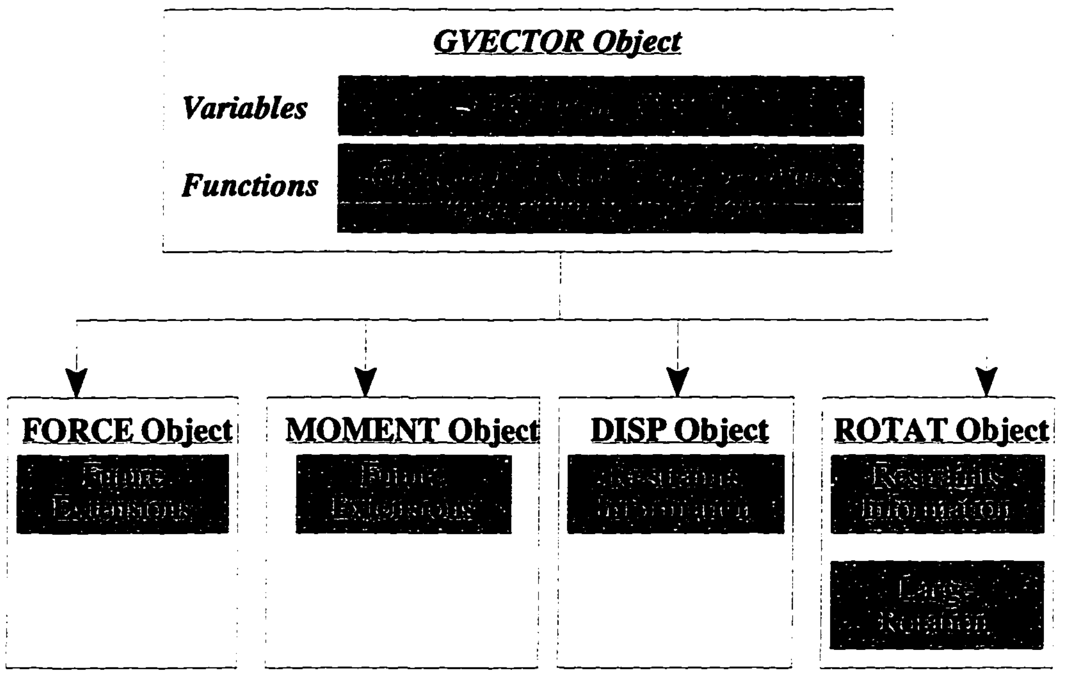


Figure 3.2 - Force and Displacement Classess

it an array of boolean values to indicate whether the degrees of freedom represented by the DISP object are restrained or not. The ROTAT class is identical to the DISP class, with additional capabilities to handle large rotations in analyses with geometric nonlinearity.

3.2.5 Loading Object Classes

Figure 3.3 shows the main structure of the nodal load history object class, NHLOAD, and the element load history object class, ELHLOAD. The load history objects contain variables and functions for defining the time scale, interpolating loads at any time value, and implementing input/output operations. The class contains a time scale array, which contains a series of time values. For each time value, a corresponding load object is defined by the user. If no load object is defined for the time value, a load value is interpolated, as shown in Figure 3.3. Two classes of load objects are defined, nodal load objects, NLOAD, or element load objects, ELLOAD. Each is used with the respective load history object.

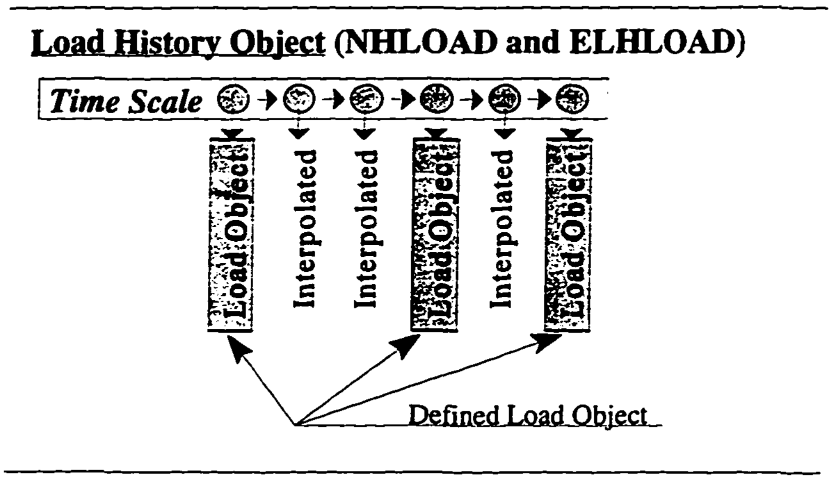


Figure 3.3 - Load History Class

3.2.6 Node Object Class

The node object class, NODE, is designed to contain the information related to the node, including its original coordinates, final coordinates, displacements, rotations, accelerations, velocities, internal forces, and internal moments. It also contains other variables to store initial conditions for a dynamic analysis. In addition, it stores the name and address of the nodal load history object, NHLOAD, which defines the nodal load history for the node. Functions used for input/output, updating the node after a solution, file storage, and mapping the node to its position in the global structure matrices are also defined in the NODE object class.

3.2.7 Element Object Class

The element object class, ELEMENT, includes functions for calculating the element stiffness matrix and transferring it to the global axes, procedures for calculating the internal stresses in the element, and other functions related to input/output operations, such as plotting the deformed shape of the element. In order to perform the tasks listed above, the ELEMENT object stores the following information:

- Name and address of the node objects attached to the element.
- Name and address of the element load history object applied to the element.
- Name and address of the section and material property objects.

The ELEMENT object class also contains other variables to store the element original length, its temperature, and its local coordinate system.

In this work, several element types were inherited from the main element class, ELEMENT (see Figure 3.4). The BEAM object class represents a two-node beam element, the TRUSS class represents an axial two-node element, the RSPRING class represents a linear spring attached to one node along one of the global axes, and the CABLE object class represents an element with a variable number of internal nodes with no flexural stiffness. The CABLE element implements the principles of sub-structuring and matrix condensation to eliminate the degrees-of-freedom associated with the internal nodes. The condensation process is discussed in detail in Section 3.3.4. CABLE2 is a cable element that inherits from CABLE. CABLE2 uses a variable number of axial two-nodes tension-only elements along the span of the cable. The number of cable sub-elements used in CABLE2 is specified by the user. Inheriting the CABLE2 object class from the CABLE object class allowed for future extensions. In this way, cable object classes with alternative types of sub-elements can be defined and inherited from the CABLE class, thus making use of the matrix condensation capabilities present in the CABLE class. For more information about the stiffness matrices of different types of elements see Reference [73].

Although the ELEMENT class contains the basic information needed by the element to calculate its stiffness matrix and perform other tasks, most of the implementation details, such as stiffness matrix calculation, are performed by the inheriting objects. For this reason, the ELEMENT object class was defined as an abstract class, i.e., an object of type ELEMENT can never be used directly in the program. Instead, an object class inheriting from the ELEMENT class, such as a BEAM or a TRUSS class, should always be used.

Although the ELEMENT class cannot be used directly, it is very useful to the program in two ways. First, it reduces the code size, because it defines variables and functions that are common to all elements, such as the section properties and material properties. Secondly, it

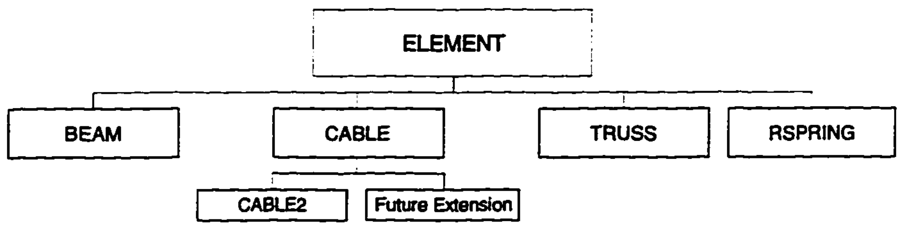


Figure 3.4 - The ELEMENT Class and its Inheritors

helps in producing a simpler and more versatile interface between the program and the different element objects present in the structure, as explained in Section 3.2.8.

3.2.8 CDyntrnDoc Object Class

CDyntrnDoc is the main object class which acts as the container and organizer of other objects. It contains information about nodes, elements, material and section properties, and nodal and element loads existing in the structure. It keeps track of this information using a list class called "CMapStringToOb", which is provided by the Visual C++ [72] compiler, and is used to map a number of objects to corresponding character strings. In other words, it assigns every object a name, and stores the objects in a linked list. In this way, every object, a node for example, will have a name assigned to it. "CMapStringToOb" object can iterate sequentially over the list or can look up a certain object by knowing its name, thus allowing search operations to be performed on the program database.

Figure 3.5 shows the lists included in CDyntrnDoc. It contains a list of nodal load history objects, a list of element load history objects, a list of material property objects, a list of section property objects, a list of node objects, and a list of element objects. Figure 3.5 also shows the relation between the object lists within the CDyntrnDoc object. A node object contains the name and address of a nodal load history object. Similarly, each element object contains pointers to an element load history object, a material property object, a section property object, and a number of node objects. CDyntrnDoc also contains other objects and variables for performing other services, such as data input, data output, controlling solution parameters, performing problem solution, and displaying graphics.

CDyntrnDoc acts as the main interface of the program. It receives messages from the user or other objects in the program to perform certain operations, such as input, output, or a solution operation. Then, it transfers the message to the appropriate object stored in one the lists to perform this operation. For example, in the solution process, the CDyntrnDoc object receives a message to assemble the stiffness matrix. Consequently, it iterates through the list of element objects, sending a message for each element to form its own stiffness matrix and place it in the global stiffness matrix.

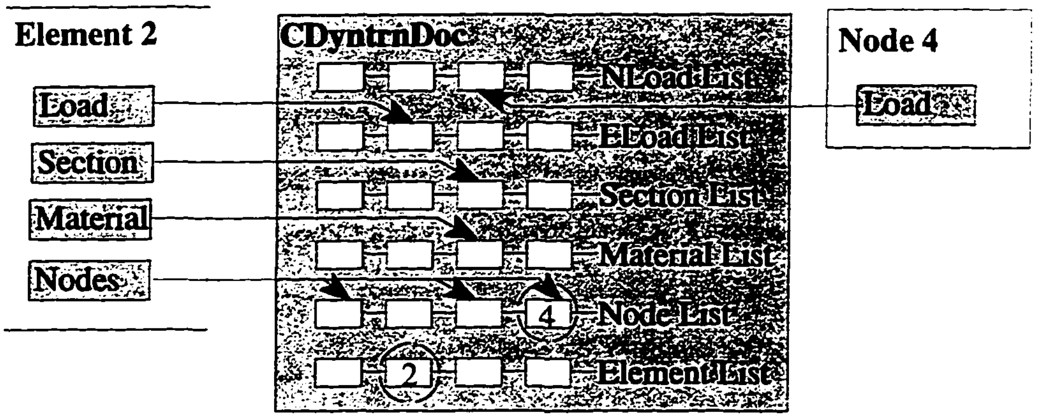


Figure 3.5 - Main Container Object Class, "CDyntrnDoc"

The element list contained in CDyntrnDoc is recognized by the program as a list of ELEMENT objects; however, the list contains no ELEMENT objects. Instead, it contains objects inherited from the ELEMENT class, such as BEAM or TRUSS objects. This is very useful because the container object, CDyntrnDoc, makes no assumptions regarding the type of elements included in the list, and therefore the interface becomes more versatile.

Although the container object doesn't recognize the inheritor class of any element, it has the ability of calling the routines belonging to the inheritor by using special types of functions called virtual functions. A virtual function is defined in both parent and child classes. When a parent class is used to call a virtual function, the function belonging to the actual class of the object is used instead. Figure 3.6 shows an example explaining virtual functions. In the figure, e1 and e2 are pointer variables, both declared as pointers to class ELEMENT, where a pointer in software terminology is a variable that contains the memory address of another variable or object. In the example, e1 is assigned the address of an object created as a BEAM object, and e2 is assigned the address of an object created as a TRUSS object. This assignment is possible because both BEAM and TRUSS are inherited from the ELEMENT class. Although the program recognizes both e1 and e2 as pointers to the ELEMENT class, when the virtual function " CalculateStiffness()" is called by the pointer e1, the BEAM version is used, and when it is invoked through e2, the TRUSS version is used instead. In both cases, the ELEMENT version of the function is not used.

ELEMENT *e1, *e2;	*e1 and e2 declared pointers to ELEMENT*
e1 = new(BEAM);	*e1 assigned a BEAM object address*
e2 = new(TRUSS);	*e2 assigned a TRUSS object address*
e1-> CalculateStiffness();	*The BEAM function is invoked*
e2-> CalculateStiffness();	*The TRUSS function is invoked*

Figure 3.6 Example of Virtual Functions

3.2.9 Sparse Matrix Object Class

The global stiffness matrix of a structural system is usually sparse, i.e., most of the elements of the matrix are zeros. Therefore, it is not efficient to store all the elements of the matrix. Many programs use banded matrices to reduce the space required for storage and the time needed for the solution. Banded matrices, however, still waste space by requiring storage of zero elements within the band width. Some routines have also been developed to automatically renumber the nodes to achieve optimum storage and speed. Other methods have also been developed to arrange the matrix information in a certain way to eliminate the storage of zeros [74].

A new method is developed here to represent square sparse matrices using object-oriented design. The method uses pointers and linked lists to eliminate the storage of zeros. The developed sparse matrix class, SSMATRIX, doesn't store any value that falls below a certain threshold. Unlike banded matrices, the way nodes are numbered doesn't affect the storage efficiency of the SSMATRIX object.

As shown in Figure 3.7 , SSMATRIX class is based on two objects that act as the building blocks for the matrix. The first object is the ROWHEAD object, and the second object is the RELEM object. SSMATRIX contains a variable to store the number of rows in the matrix, which is equal to the total number of degrees of freedom for the structure. SSMATRIX also contains an array of ROWHEAD objects. The number of ROWHEAD objects in the array is equal to the number of rows in the matrix. Therefore, each ROWHEAD object corresponds to a row in the sparse matrix. Each ROWHEAD object stores a number which corresponds to the number of non-zero values in the row represented by this object. It also points to the first RELEM object in the row. Each RELEM object points to the RELEM preceding it, and the one following it, as shown in Figure 3.7. Each RELEM also stores two values: the column number it is representing, and the actual value of the matrix element. In every row, the first and last RELEM object store no values and act as the head and tail of the row respectively. The number of intermediate RELEM objects in any row corresponds to the number of non-zero elements in that row, and should be equal to the number stored in the ROWHEAD object.

Several routines are implemented to perform different matrix operations for SSMATRIX objects, including matrix decomposition and back substitution. During the operation of these routines, an RELEM object is eliminated if the absolute of its stored value falls below the zero threshold. Similarly, a new RELEM object is inserted for zero elements that acquire a non-zero value during these operations. The new object oriented implementation of the sparse matrix saves computer space by reducing the number of values stored. It also increases the speed of matrix operations by enhancing the process of iteration through the matrix. Because every element in the row points to the element following it, forward iteration

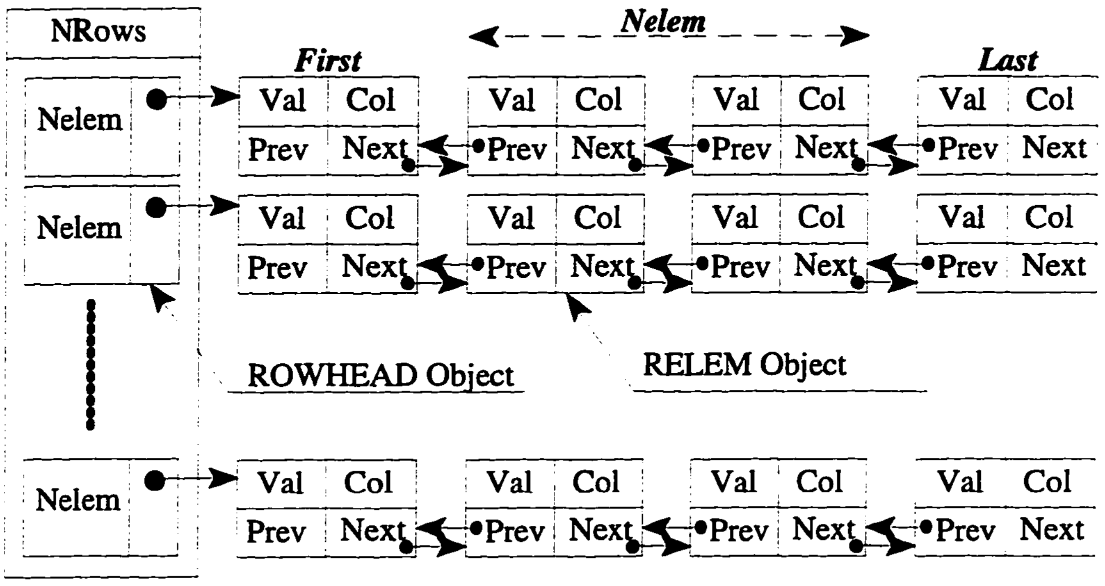


Figure 3.7 Square Sparse Matrix Object (SSMATRIX)

through the row will be very fast. Backward iteration is also improved by using pointers to preceding elements. This improvement in the performance, however, comes at the expense of more storage requirements, since two pointers have to be stored for every non-zero value in the matrix.

3.3 Analysis Procedure

3.3.1 Dynamic Analysis

Due to the highly nonlinear nature of transmission line problems, the step-by-step integration method was chosen for implementation in the dynamic program. Since this analysis is implemented in the time domain, a wide range of loading histories, such as galloping, can easily be applied to the structures. Figure 3.8 shows the force-deformation relation for a large deformation dynamic analysis. As the solution proceeds from Point A, at time t_n, to Point B, at time t_{n+1}, the incremental dynamic equilibrium equation of the structural system for the time step $(t_{n+1} - t_n)$ can be written as:

$$[M]\{\Delta\ddot{U}\}_{n-1} + [C]\{\Delta\dot{U}\}_{n+1} + [K]\{\Delta U\}_{n-1} = \{\Delta F\}_{n+1} \qquad 3.3$$

Where,

[M] = mass matrix,

[C] = damping matrix,

[K] = tangential stiffness matrix,

$\{\Delta F\}_{n+1}$= incremental external force vector,

$\{\Delta\ddot{U}\}_{n+1}$= incremental acceleration vector,

$\{\Delta\dot{U}\}_{n+1}$= incremental velocity vector,

$\{\Delta U\}_{n+1}$= incremental displacement vector.

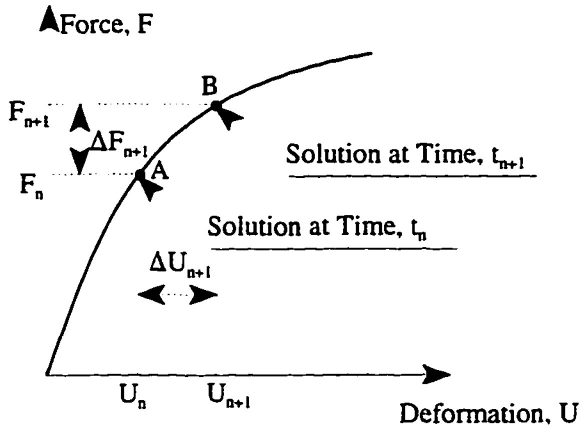


Figure 3.8 Dynamic Solution of Large Deformation Problems

There are several numerical techniques that can be used to solve Equation 3.1 [63]. Among these methods is the Newmark technique. Assuming that the displacement, velocity and acceleration at the beginning of the time step, n, are known, the Newmark technique calculates the incremental acceleration and velocity at the end of the time step, n+1, as:

$$\{\Delta\ddot{U}\}_{n+1} = a_0\{\Delta U\}_{n+1} - a_2\{\dot{U}\}_n - a_3\{\ddot{U}\}_n \qquad 3.4$$

$$\{\Delta\dot{U}\}_{n-1} = a_1\{\Delta U\}_{n+1} - a_4\{\dot{U}\}_n - a_5\{\ddot{U}\}_n \qquad 3.5$$

a_0 to a_5 are defined in Appendix B as functions of two parameters, α and δ. The parameters, α and δ, can be used to adjust the stability and accuracy of the integration method. Substituting Equations 3.4 and 3.5 in the general equation of motion the following equation is derived:

$$([K]+a_0[M]+a_1[C])\{\Delta U\}_{n-1} = \{\Delta F\}_{n-1} + (a_2[M]+a_4[C])\{\dot{U}\}_n + (a_3[M]+a_5[C])\{\ddot{U}\}_n \qquad 3.6$$

Since the acceleration, $\{\ddot{U}\}_n$, and velocity, $\{\dot{U}\}_n$, at the beginning of the time step are known, Equation 3.6 can be solved for the incremental displacement $\{\Delta u\}_{n+1}$. The incremental displacement vector can then be substituted in Equation 3.4 to calculate the incremental acceleration, and in Equation 3.5 to calculate the incremental velocity. The incremental values are used to update the displacements, accelerations, and velocities at the end of the time step, as follows:

$$\begin{aligned} \{\ddot{U}\}_{n-1} &= \{\ddot{U}\}_n + \{\Delta\ddot{U}\}_{n+1} \\ \{\dot{U}\}_{n+1} &= \{\dot{U}\}_n + \{\Delta\dot{U}\}_{n-1} \\ \{U\}_{n-1} &= \{U\}_n + \{\Delta U\}_{n+1} \end{aligned} \qquad 3.7$$

The values of the displacement, velocity, and acceleration at the end of the time step are used as initial values for the time step to follow, and the solution proceeds.

3.3.2 Large Deformation Analysis (Geometric Nonlinearity)

In some structural analysis problems, the deformation (displacements or rotations) of the structural element is large enough, compared to the original dimensions of the element, that the behavior of the element in the deflected position is considerably different from its original behavior. In these cases a nonlinear large deformation analysis is required. The large deformation dynamic analysis method using the Full-Newton-Raphson technique, and the updated Lagrangian formulation [75] as implemented in DYNTRN, is outlined below:

1. Starting with known values for $\{U\}_n$, $\{\dot{U}\}_n$, and $\{\ddot{U}\}_n$, $\{U\}_{n+1}$, $\{\dot{U}\}_{n+1}$, and $\{\ddot{U}\}_{n+1}$ can be calculated as described in Section 3.3.1.
2. The calculated deflection vector, $\{U\}_{n+1}$, can be written as:

$$\{U\}^t_{n+1} = \{U_X\ ,\ U_Y\ ,\ U_Z\ ,\ R_X\ ,\ R_Y\ ,\ R_Z\} \qquad 3.8$$

   Where U_x, U_y, and U_Z are the displacements in the X, Y, and Z directions, and R_x, R_y, and R_Z are the rotations about the global axes, X, Y, and Z. Using the calculated displacements, the geometry of the structural element is updated, as shown in Figure 3.9, and the element coordinate system is updated.
3. The rotations, R_X, R_Y, and R_Z are then transformed to the displaced element coordinate system as R_{XE}, R_{YE}, R_{ZE}. Transferring large rotations form one coordinate system to another is not performed by simply multiplying the rotations by the transformation matrix of the updated coordinate system. The procedure used to transfer a rotation vector from one coordinate system to another is listed in Appendix C. More information about this procedure can be found in [76-78].
4. The axial deformation, $\{U_{XE}\}$, of the element is next calculated as the difference between the updated and original length of the element. In the updated coordinate system, U_{YE} and U_{ZE} will be zero.
5. The internal forces in the displaced coordinate system are calculated as: $\{F_{EI}\} = [K_E]\ \{U_E\}$, where $[K_E]$ is the element tangent stiffness matrix in the displaced coordinate system, $\{U_E\}^t = \{U_{XE},0,0,R_{XE},R_{YE},R_{ZE}\}$.
6. The internal forces are then transformed to the global coordinate system, and the residual forces, $\{F_R\}$ calculated as:

$$\{F_R\} = \{F_E\}\ -\ \{F_I\}\ -\ [M]\{\ddot{U}\}_{n+1}\ -\ [C]\{\dot{U}\}_{n+1} \qquad 3.9$$

Where $\{F_E\}$ and $\{F_I\}$ are the applied force vector, and the internal force vector, respectively, in the global coordinate system.

7. If the L2 norm of the residual force vector $\{F_R\}$ is below the user specified tolerance value, the solution is then converged, and proceeds to the next time step. Otherwise, an updated tangential stiffness matrix, [K], is calculated for the structure, and the deformation vector, $\{U\}_{n+1}$, updated for the next iteration, i+1, using the relation:

$$\{U^{i+1}\}_{n+1} = \{U^{i}\}_{n+1} + \{\delta U^{i+1}\}_{n+1}, \qquad \text{3.10}$$
$$where,\ \{\delta U^{i+1}\}_{n+1} = [K]^{-1}\{F_R\}$$

8. An updated incremental displacement vector, $\{\Delta U^{i+1}\}_{n+1}$, is calculated as:

$$\{\Delta U^{i+1}\}_{n+1} = \{U^{i+1}\}_{n+1} - \{U\}_{n} \qquad \text{3.11}$$

The updated acceleration, and velocity vectors, are then calculated, using Equations 3.2, 3.3 and 3.5, and the solution proceeds to Step 2. The procedure continues until the solution converges as mentioned in Step 7.

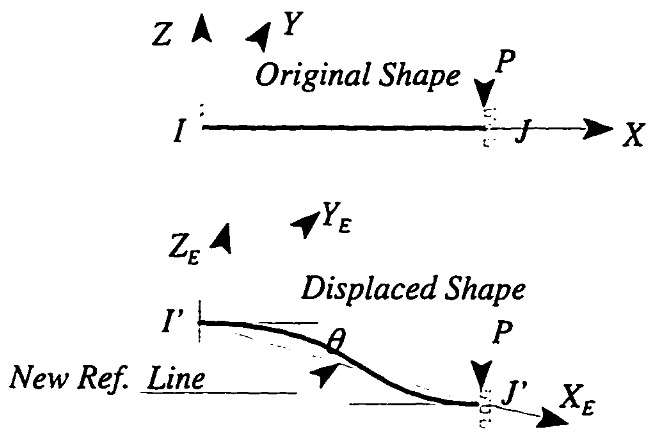


Figure 3.9- Large Deformation Analysis

The above large deformation analysis is implemented in the solution routine performed within the CDyntrnDoc object as presented in Figure 3.10. First, the stiffness matrix is assembled, then the restraint degrees of freedom are eliminated. Next, the incremental joint loads' vector, element loads' vector and restraining forces' vector are formed. The structural equations are then solved, the geometry is updated, and the residual forces are calculated. The residual forces' vector is used to check if a converged solution is achieved.

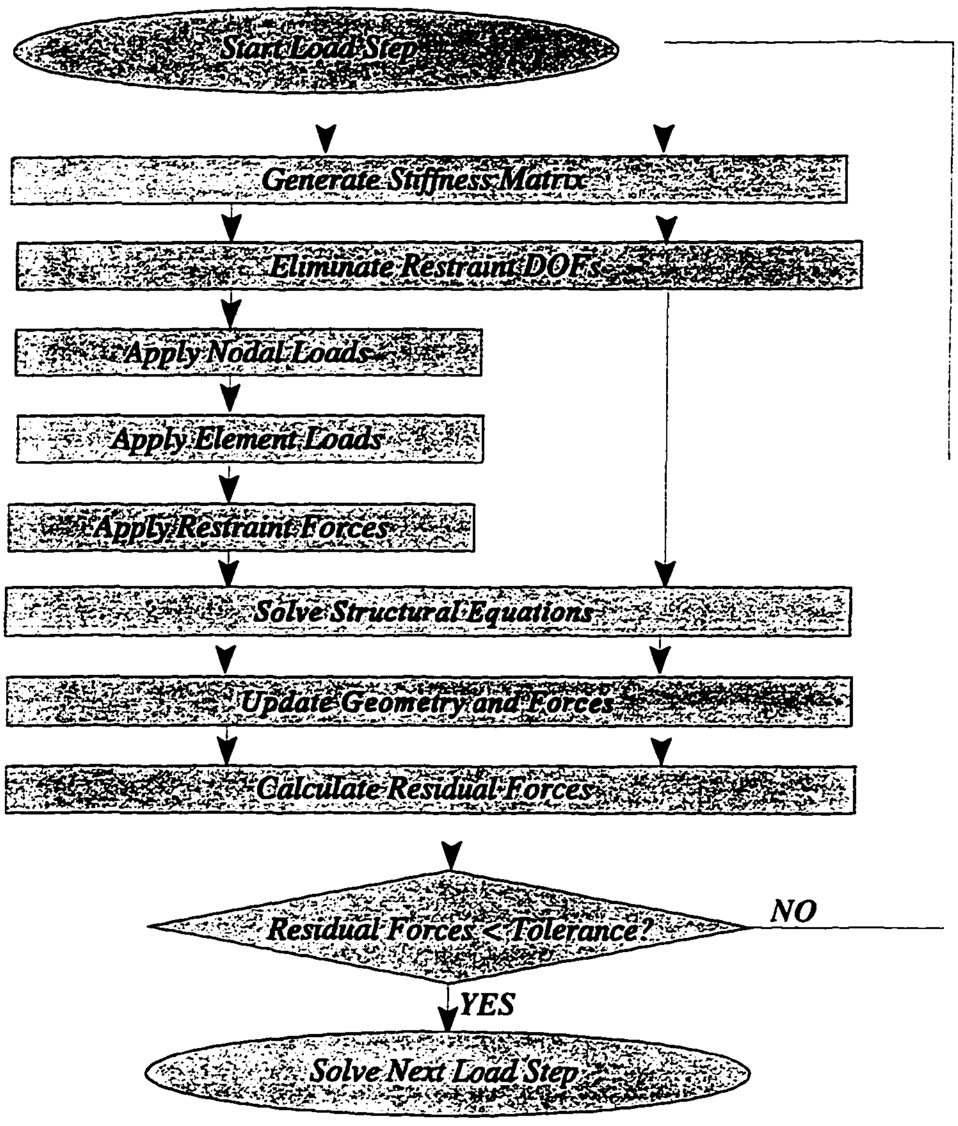

Start Load Step
Generate Stiffness Matrix
Eliminate Restraint DOFs
Apply Nodal Loads
Apply Element Loads
Apply Restraint Forces
Solve Structural Equations
Update Geometry and Forces
Calculate Residual Forces
Residual Forces < Tolerance?
NO
YES
Solve Next Load Step

Figure 3.10 - Nonlinear Solution Procedure

For each step of the solution process, all the information pertaining to the solution is stored in the element and node lists contained in the CDyntrnDoc object. In this way the solution process can be stopped at any time, a parameter, such as the applied load, the material properties, or the cross-sectional properties, can be changed, and then the solution resumed. The user can also remove a component from the program database, such as a cable element, and then resume the analysis. This feature makes a broken component analysis, such as a broken conductor analysis, very simple.

3.3.3 Matrix Condensation and Sub-Structuring

Matrix condensation in linear static analysis is a well-known concept discussed in most structural analysis text books. Little work, however, has been done to extend the method to dynamic analysis. Guyan [79] proposed a method to evaluate an approximate condensed mass matrix using energy equations. The method was used by ANSYS [76] and extended to calculate condensed damping matrices.

An exact method for condensing dynamic matrices was developed herein, and used in DYNTRN. The method used a simplified form similar to the static condensation equations, and was extended to nonlinear analysis.

3.3.3.1 Matrix Condensation Process in Dynamic Analyses

The Newmark technique previously presented needs to be modified to be used in cable elements which consist of several sub-elements, with internal degrees-of-freedom. In this case, the problem is solved in two phases. In the first phase, the internal degrees of freedom for each element are condensed, and accounted for by a modified stiffness matrix and a modified force vector. By eliminating the internal degrees of freedom, the solution of the problem can proceed as shown in the Section 3.3.2. In the second phase the displacements and forces of the internal nodes are updated using the values associated with the external degrees of freedom, which were obtained in the first phase. The dynamic condensation is explained in details next.

The condensed equation of motion can be written as:

$$([K_{22}]_D - [K_{21}]_D([K_{11}])_D^{-1}[K_{12}]_D)\ \{\Delta U_2\}_{n+1} = \{\Delta F_2\}_D - [K_{21}]_D\ ([K_{11}])_D^{-1}\ \{\Delta F_1\}_D \qquad 3.12$$

Where the subscripts 1 and 2 refer to the internal and external degrees-of-freedom, respectively. The complete derivation of Equation 3.12 , and symbol definitions can be found in Appendix B. Equations for calculating $[K_{ij}]_D$ and $\{\Delta F_i\}_D$ are also listed in Appendix B.

Equation 3.12 can be expressed in the form:

$$[K_{22}]_{eq}\{\Delta U_2\}_{n+1} = \{\Delta F_{2eq}\}_{n+1} \qquad 3.13$$

where:

$$\begin{aligned} [K_{22}]_{eq} &= [K_{22}]_D - [K_{21}]_D([K_{11}])_D^{-1}[K_{12}]_D \\ \{\Delta F_2\}_{eq} &= \{\Delta F_2\}_D - [K_{21}]_D\ ([K_{11}])_D^{-1}\ \{\Delta F_1\}_D \end{aligned} \qquad 3.14$$

Assuming the initial velocity and acceleration to be defined for both the internal and external loads, both $[K_{22}]_{eq}$ and $\{\Delta F_{2eq}\}_{n+1}$ can be calculated. By adding $[K_{22}]_{eq}$ to the total structure stiffness matrix, and adding $\{\Delta F_{2eq}\}_{n+1}$ to the total structure load vector, one can solve for the incremental displacements $\{\Delta U_2\}_{n+1}$. One can then substitute into equations 3.4 and 3.5 to obtain the incremental acceleration and velocity at the external nodes.

The incremental displacements for the internal nodes, $\{\Delta U_1\}_{n+1}$, can then be calculated using the following equation:

$$\begin{aligned} &([K_{11}]+a_0[M_{11}]+a_1[C_{11}])\{\Delta U_1\}_{n+1} + ([K_{12}]+a_0[M_{12}]+a_1[C_{12}])\{\Delta U_2\}_{n+1} = \\ &\{\Delta F_1\}_{n+1} + [M_{11}](a_2\{\dot{U}_1\}_n + a_3\{\ddot{U}_1\}_n) + [C_{11}](a_4\{\dot{U}_1\}_n + a_5\{\ddot{U}_1\}_n) \\ &\qquad + [M_{12}](a_2\{\dot{U}_2\}_n + a_3\{\ddot{U}_2\}_n) + [C_{12}](a_4\{\dot{U}_2\}_n + a_5\{\ddot{U}_2\}_n) \end{aligned} \qquad 3.15$$

The incremental acceleration and velocity of the internal nodes can also be calculated using Equation 3.4 and 3.5. Using these incremental values, the displacement, velocity and acceleration of the internal nodes can be updated to reflect the values at the end of the time step, using Equation 3.7. This process can then be repeated for the following time steps.

3.3.3.2 Adapting the Condensation Process to Nonlinear Analysis

The condensation process as described above is suitable for linear analysis. In order to use the same procedure for nonlinear analysis some modifications have to be introduced.

The structural equations of motion of the structure considering nonlinear effects can be written as:

$$\begin{aligned}\{FS_1\}_{n+1} + [M_{11}]\{\ddot{U}_1\}_{n+1} + [M_{12}]\{\ddot{U}_2\}_{n+1} \\ + [C_{11}]\{\dot{U}_1\}_{n+1} + [C_{12}]\{\dot{U}_2\}_{n+1} = \{F_1\}_{n+1} + \{RF_1\}\end{aligned} \tag{3.16}$$

$$\begin{aligned}\{FS_2\}_{n+1} + [M_{22}]\{\ddot{U}_2\}_{n+1} + [M_{21}]\{\ddot{U}_1\}_{n+1} \\ + [C_{22}]\{\dot{U}_2\}_{n+1} + [C_{21}]\{\dot{U}_1\}_{n+1} = \{F_2\}_{n+1} + \{RF_2\}\end{aligned} \tag{3.17}$$

Where, {FS} is the straining force vector, and is calculated by evaluating the total strains on elements, {RF} is the residual force vector, and F is the external load vector.

In a nonlinear analysis, the ultimate goal is to reduce the residual forces {RF} below a specific tolerance value. In order to minimize the residual forces associated with the internal as well as the external degrees of freedom, iterations are performed on the internal degrees of freedom, until the internal residual forces, $\{RF_1\}$ are nearly eliminated for each iteration of the external degrees of the freedom.

3.4 Development of the Graphical User Interface

In order provide good graphical capabilities in the dynamic simulation program, DYNTRN, the Graphical User Interface, GUI, of DYNTRN was developed to produce the following features:

- Simplified input/output.
- User-friendly and highly interactive interface.

- Integrity and error checking.
- Graphical output of geometry and data.

In order to develop the above features, it was found to be more efficient to use a software other than Visual C++ [72] for this purpose. Delphi [80], an Object Pascal development suite, was found superior to the Visual C++ since it included more than 70 graphical objects ready to use and modify. It was necessary, however, to communicate between the main program written in the Visual C++ environment and the GUI developed in Delphi. This was done by converting the GUI to a Dynamic Link Library, DLL. As the name implies, the DLL is a library that is linked to the main program during execution. It basically contained the visual functions used to draw and implement the GUI. The DLL gave the main program, access to the services provided by the GUI developed using Delphi.

There was a problem of back communication, however, from the main program to the GUI library, i.e., the GUI library needed to access some of the services provided by the main program. For example, if the user clicks a menu item indicating a request to input nodal coordinates, the GUI will display the Nodal coordinates dialog box, and will need the services of the main program to retrieve information about the nodes that already existed in the database. Back communication was achieved through two channels: callback functions and messages. Callback functions are functions defined in the main program, and at the same time declared in the DLL, i.e., the DLL only knows the address of the function, but the actual implementation is included in the main program. Broadcasting messages is another way of communication, where a program or a library sends a message to another program. The topic of message broadcasting is beyond the scope of this work, and is specific to Microsoft Windows environment. More discussion about this issue can be found in [81]. The relationship between the GUI library and the main program is shown in Figure 3.11. The details of the different objects and functions used to develop the GUI library are outside the scope of this presentation, and therefore will not be discussed.

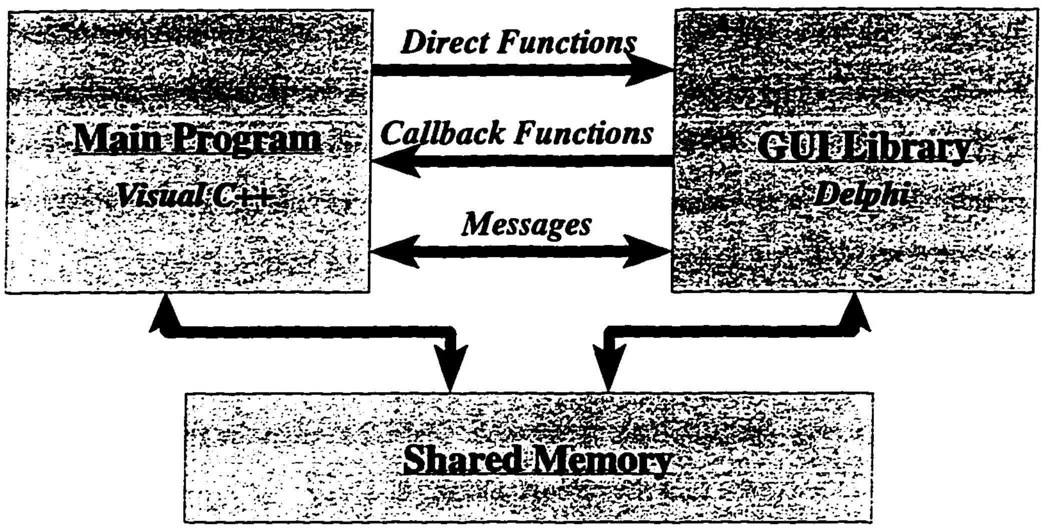


Figure 3.11- Relation between Main Program and GUI Library

3.5 Using the Dynamic Simulation Software, DYNTRN

In most structural analysis programs, the analysis process consists of three sequential phases: The modeling phase where geometry and loads are specified, the solution phase, where the equilibrium equations are solved, and the output phase where the results are obtained. The relation between these phases, however, possesses more flexibility in DYNTRN than in conventional structural analysis programs. In DYNTRN, for example, the solution can be stopped at any time, certain aspects of the model can be changed and then the solution can be resumed. Results can also be viewed as the solution proceeds.

Figure 3.12 shows the relation between the main components of the structural model. Each element possesses information about the attached nodes, the applied element loads, and the material and section-properties specific to the element. Each node contains information about the nodal loads applied to it. DYNTRN identifies loads, properties, nodes and elements using names specified by the user.

The relationship between the components of the structural model requires a special sequence in constructing the model. For example, an element cannot be defined if the attached nodes are not defined first. Similar precautions have to be observed when altering the model by removing an element or a node from the model. A node, for example cannot be removed, if an existing element is still attached to the node. In the following sections, different steps involved in the modeling and solution processes using DYNTRN will be explained.

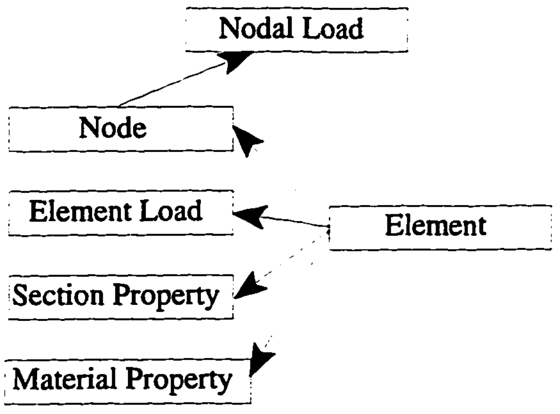


Figure 3.12 Relation between the Components of the Structural Model

3.5.1 Defining Loads

Figure 3.13 shows the load time-history used by DYNTRN. A time scale has to be defined by the user. The time scale specifies the main time points used to define the load history. In Figure 3.13, four time points, (t_0 - t_3), were specified, and corresponding load values, (F_0 - F_3), were defined. The load, F_i, applied to the structure at any time, t_i, can be calculated by interpolating between the defined loads. At least two time points have to be specified for the load history. The user can define two types of loads, nodal loads and element loads.

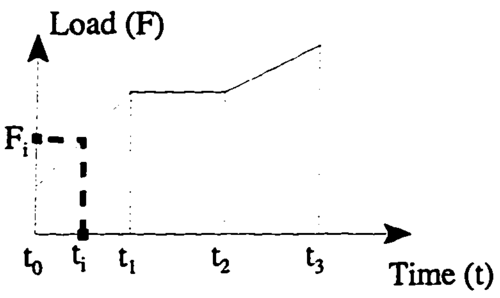


Figure 3.13 Load History as defined in DYNTRN

Figure 3.14 shows the dialog box used to input the nodal load time-history. For each defined time point, nodal loads, i.e., forces and moments, and nodal restraints are specified in the global coordinate system. Figure 3.15 shows the window used to define the element loads. The current version of DYNTRN only support uniform loads along the full span of the element. Element loads can be defined in the global coordinate system, or in the element local coordinate system.

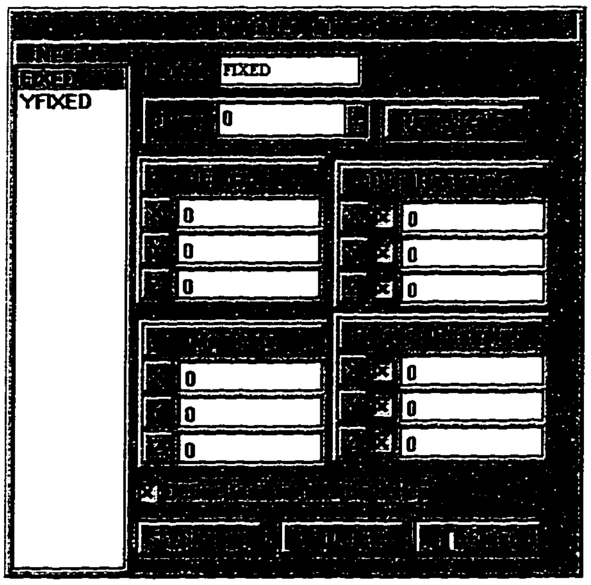
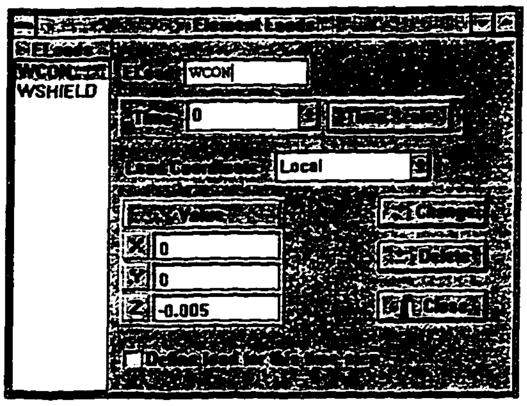


Figure 3.14 Nodal Load Input Window



Element Load

ELoads

WCON

WSHIELD

ELoad: WCON

Time: 0

Load Coordinates: Local

X: 0

Y: 0

Z: -0.005

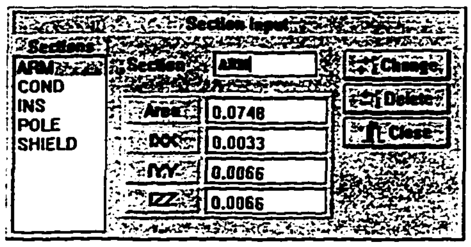
Figure 3.15 Element Load Input Window

3.5.2 Defining Properties

Figure 3.16 shows the window used to define the section properties for the element. Values for the cross-sectional area, A, the moments of inertia, I_{YY}, and I_{ZZ} and the polar moment of inertia, I_{XX}, are specified for each defined section. Figure 3.17 shows the window used to define the material properties used for the element. Young's modulus of elasticity, poisson ratio, mass density, coefficient of thermal expansion, and the parameters, α and β, are specified for each defined material property. The parameters, α and β, are used to calculate the damping matrix for the element as follows:

$$[C] = \alpha[M] + \beta[K] \qquad 3.18$$

Where, [M],[K] and [C] are the mass, stiffness, and damping matrices, respectively.



Section Input

Sections

ARM

COND

INS

POLE

SHIELD

Section: ARM

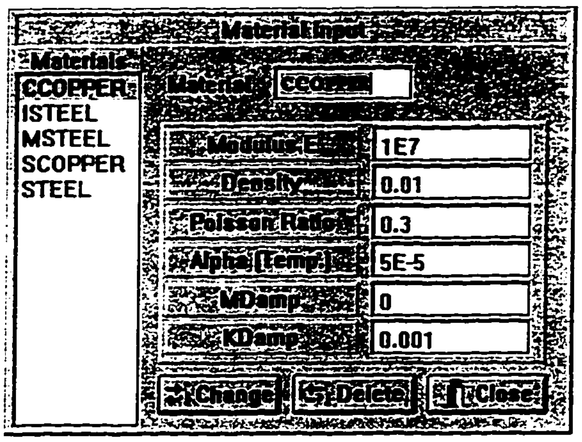
Area: 0.0748

IXX: 0.0033

IYY: 0.0066

IZZ: 0.0066

Figure 3.16 Section Properties Input Window



Materials

CCOPPER
ISTEEL
MSTEEL
SCOPPER
STEEL

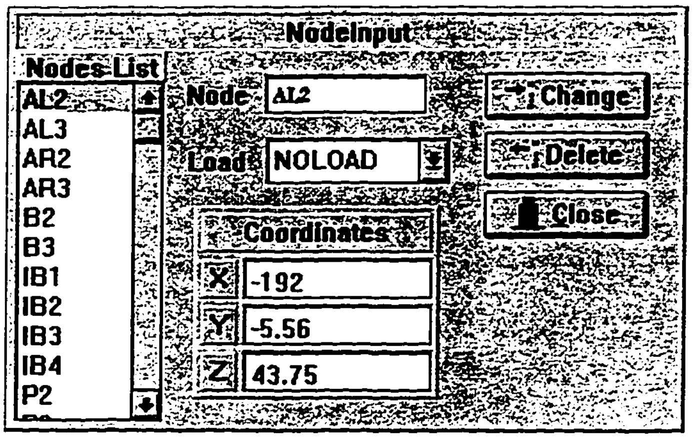
Material: CCOPPER

Modulus E: 1E7
Density: 0.01
Poisson Ratio: 0.3
Alpha (Temp.): 5E-5
MDamp: 0
KDamp: 0.001

Figure 3.17 Material Properties Input Window

3.5.3 Defining Nodes

Figure 3.18 shows the dialog box used for defining nodal data. The user supplies the coordinates of the node in the global coordinate system, and the name of the applied nodal load time-history, if any.



Nodes List

AL2
AL3
AR2
AR3
B2
B3
IB1
IB2
IB3
IB4
P2

Node: AL2

Load: NOLOAD

Coordinates

X: -192
Y: -5.56
Z: 43.75

Figure 3.18 Node Input Data

3.5.4 Defining Elements

The current version of DYNTRN is capable of analyzing four types of elements: 3-D beam elements, 3-D cable elements, 3-D truss elements, and spring elements. In the cable and truss elements, the internal forces are calculated using large displacement theory. In the beam element, both the large displacement and large rotation theories are applied. The spring element is a linear element. A description of the different elements used in DYNTRN is given next.

3.5.4.1 The Beam Element

The beam element is defined using three nodes. The first two nodes define the two ends of the beam element. The third node is used to define the local coordinate system of the beam element as shown in Figure 3.19. In the current version of DYNTRN, element loads cannot be applied on beam elements.

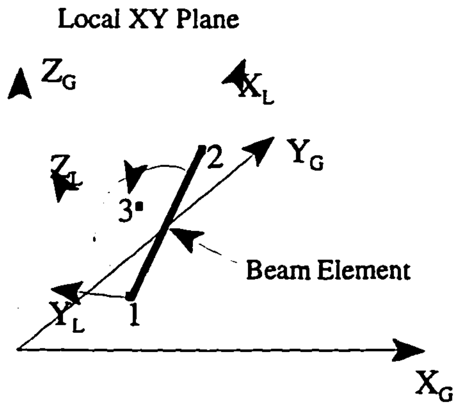
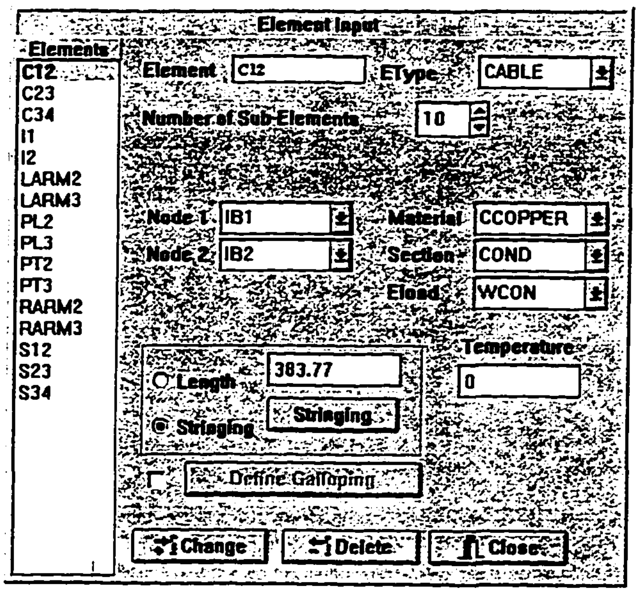


Figure 3.19 Orientation of the Beam Element

3.5.4.2 Cable Elements

DYNTRN models the cable element using a number of two-nodes tension-only axial sub-elements. The number of sub-elements included is chosen by the user. A minimum of five elements, and a maximum of 100 elements can be used. The choice of the number of sub-elements to be used depends on the type of analysis to be performed. Generally, as the contribution of higher mode shapes increases in the solution, more sub-elements have to be used to accurately model the cable element. A sensitivity study is usually a good means of assessing the effect of the number of sub-elements used to model the cable element on the accuracy of the results.

Figure 3.20 shows the input window for a cable element. The element is defined using two nodes. The two nodes define the end points of the cable element. The local axes of the cable element are oriented so that the local x-axis is along the span of the cable, and the local x-z plane is normal to the global x-y plane. The un-stretched length of the cable is either specified directly, or calculated using stringing tension information. An element load time-history can be applied to the cable element,or galloping motion can be applied instead. The galloping motion and the load time history cannot be applied simultaneously on the same element. A galloping motion cannot be applied on a cable element oriented parallel to the global z-axis. For the case of a galloping cable, the weight of the cable is assumed to act in the direction of the negative global z-axis.



Elements: C12, C23, C34, I1, I2, LARM2, LARM3, PL2, PL3, PT2, PT3, RARM2, RARM3, S12, S23, S34

Element: C12 EType: CABLE

Number of Sub-Elements: 10

Node 1: IB1 Material: CCOPPER

Node 2: IB2 Section: COND

Eload: WCON

○ Length: 383.77 Temperature: 0

● Stringing: Stringing

Figure 3.20 Cable Element Input Data

Figure 3.21 shows the window used to define the galloping motion for the cable element. The weight of the galloping cable, the amplitude of the in-plane and out-of-plane galloping, the initial phase of the galloping conductor, and the galloping mode are required as input for the galloping motion. The frequency of the galloping conductor is either supplied directly by the user, or specified as the natural vibration of the specified galloping mode. In that case, it is calculated by the program. The mode shapes used to calculate the galloping motion of the conductor are calculated using the formulation developed by Irvine [8], which is explained in Chapter 2. This formulation assumes the conductor to be flat. As stated in Chapter 2, the difference between the natural frequencies calculated using the flat-sag formulation and that obtained using the deep-profile formulation was found by Irvine [8] to be less than 6% when the sag-to-span ratio was 13%. The error increased to 20% for a sag-to-span ratio of 23%. Therefore, for cables with deep profiles the results of the galloping analysis using DYNTRN should be checked carefully.

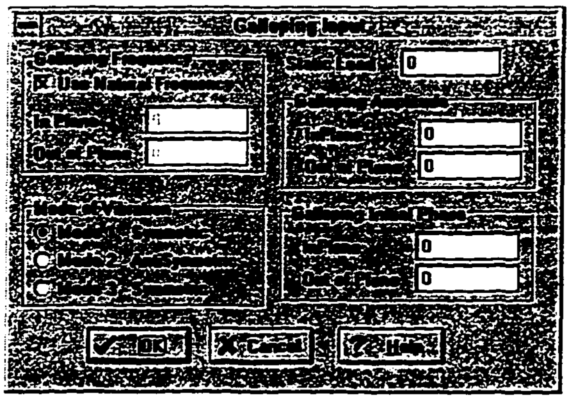


Figure 3.21 Galloping Vibration Input Data

3.5.4.3 Truss Element

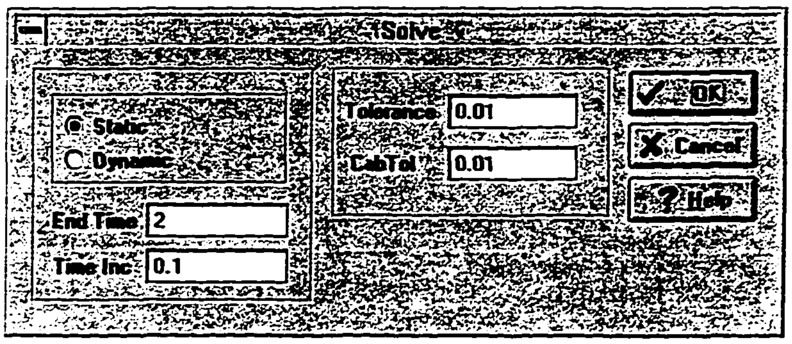
The truss element is a two-nodes axial element. The element is defined using two nodes. The orientation of the local axes is calculated similar to the cable element discussed in Section 3.5.4.2. Element loads cannot be applied to the truss element. Truss elements can be defined by the user to be tension-only, i.e., the stiffness of the element becomes zero, if the element is subjected to a compression force.

3.5.4.4 Spring Element

The spring element is a linear element with one degree-of-freedom. The element is defined using one node, and is oriented in the direction of one of the global axes. The stiffness of the spring element is required as input by the user. The spring element can be used as a simplified representation of the transmission line support structures. Temporary spring elements can also be used to help in achieving a converged solution, as explained in Section 3.5.5.

3.5.5 Solving the Problem

Both dynamic and static analysis can be performed in DYNTRN. Figure 3.22 shows the solution dialog box. The user supplies the end time of the solution, and the solution time step. The solution starts at time zero, or at the end time of the previous solution phase. The choice of the time step size is important to obtain an accurate solution. As a general guideline, the time step should be less than 1/10 of the highest mode shape which significantly participates in the solution. Sometimes, a smaller time step might be needed to achieve a converged solution. Sensitivity studies should be conducted to evaluate the effect of the time step size on the accuracy of the results.



[✓ Static] Dynamic

End Time: 2

Time Inc: 0.1

Tolerance: 0.01

CabTol: 0.01

Figure 3.22 Solution Input Data

At any time, the solution process can be stopped and the structural model can be altered, by deleting an element, for example, or changing the material properties. The solution can then be resumed. The time at which the previous solution phase stopped will be used as the start time for the new solution phase. This process of stopping and resuming the solution allows for performing failure analysis, where an element can be removed, such as a cable element, to simulate a broken conductor event.

In some cases, the solution process encounters degrees-of-freedom with low stiffness values at certain points on the nonlinear path of the solution. A vertical truss element, for example, will posses very little stiffness to resist a lateral horizontal force, until the orientation of the element is changed to an inclined position. In the inclined position, the horizontal component of the element axial stiffness is used to resist the lateral horizontal force. If a DOF with very low stiffness exists in the structure at any time during the solution, unrealistic large deformations may occur, thus, causing the solution to diverge. To solve this divergence problem, a temporary spring may be placed at that DOF to artificially increase its stiffness. When the geometry of the structure is updated, causing the stiffness value of the DOF to increase, the solution can be stopped, the spring element removed, and then the solution can be resumed.

3.5.6 Checking the Results

Figure 3.23 shows the main window of the DYNTRN program. A deformed plot of the transmission line as well as a parametric plot of user-defined variables is shown in the figure. Using user-defined variables, nodal displacements, rotations, velocities, accelerations, forces and moments can be checked. The plots are updated as the solution proceeds, thus allowing the user to check and respond to the results during the solution phase. During the solution phase, the analysis results are stored at the end of every n^{th} time step, where n is a number defined by the user. After the solution is complete, a complete printout of the results can be obtained for any time value, for which the solution results were stored.

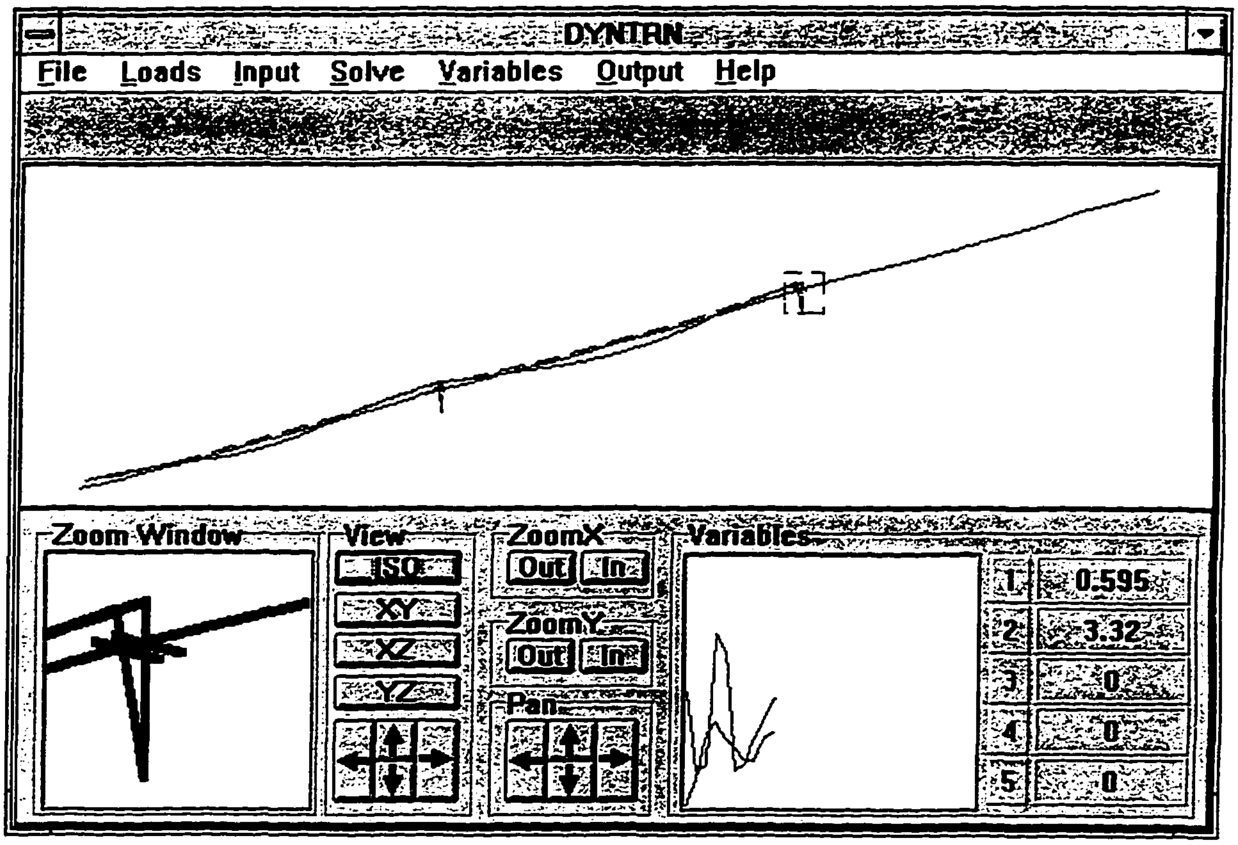


Figure 3.23 DYNTRN Main Window

CHAPTER 4 - SOFTWARE VERIFICATION

Testing the validity of the dynamic analysis program (DYNTRN) results, was done in two phases. In the first phase, the results produced by DYNTRN were verified using analytical results produced by ANSYS [76], a commercial finite element program. When a theoretical solution of the problem was known, the theoretical results were used for comparison instead of ANSYS. In the second phase, DYNTRN was calibrated using published experimental data. Four types of dynamic simulations were conducted and compared to experimental results: a broken insulator analysis, a broken conductor analysis, a broken shield wire analysis, and a conductor galloping analysis.

4.1 Analytical Verification

The main objective of the analytical verification was to verify that DYNTRN was producing accurate and reliable results. In addition, it was performed to eliminate any mistake from the software. Therefore, this verification was performed simultaneously with the development process. The major steps in the analytical verification process can be summarized as follows:

- Verification of the large deflection solution of all the elements included in DYNTRN, due to static loads. Examples 1 and 2 in Appendix D show the analytical verification for a 3-D beam element and a 3-D truss element, respectively. The results of these two examples were compared to ANSYS. Examples 3 and 4 in the appendix present the validation for the spring, and cable elements, respectively, as compared to closed form solutions.
- Verification of the time integration scheme used in the dynamic analysis, as shown in Example 5 in Appendix D.
- Verification of the dynamic condensation method developed in this study and presented in Chapter 3. Example 6 in Appendix D compares the results obtained using DYNTRN, with that obtained using ANSYS. Unlike ANSYS, DYNTRN used the dynamic condensation method to represent the cable element.

- Verification of the conductor galloping routine for a single span conductor fixed at both ends, as shown in Example 7 in Appendix D. The conductor galloping results obtained by DYNTRN were compared to results calculated analytically using the closed form formulation developed in Appendix E.
- The final stage was to verify a problem which modeled a real transmission line. The dimensions and properties of a real line were obtained from Reference [3], and were used to construct a three-span transmission line computer model (see Example 8 in Appendix D). The problem was solved using both DYNTRN and ANSYS due to static and dynamic loads. Comparison of the results is shown in Appendix D.

Analytical verification of DYNTRN showed good agreement with theory and ANSYS. This verification, however, only showed that DYNTRN produced reliable results to the analysis of the finite element model. It did not, however, show the adequacy of the finite element model in representing a real transmission line. To prove the adequacy of DYNTRN in simulating real transmission lines, comparison with published experimental data was performed. The experimental verification of DYNTRN is presented in the next section.

4.2 Experimental Verification

4.2.1 Broken Insulator Analysis

4.2.1.1 Description of the Problem

A broken insulator analysis refers to a situation where an insulator is broken followed by the free fall of the conductor. DYNTRN simulates a broken insulator analysis by removing the element representing insulator from the structure database. Therefore the conductor will start to fall under its loads, and its response can be tracked.

4.2.1.2 Background on Experimental Data

A series of broken insulator experimental tests were performed on an eight-span segment of a retired 138-KV transmission line in Wisconsin [1]. The line consisted of six intact spans and two end spans anchored to the ground. The support structures were square base lattice steel towers with six conductor phases. A schematic elevation view of the structure is shown in Figure 4.1. For detailed information about the experimental data, the reader is referred to Appendix F. The phase labeled R2 in Figure 4.1 and Appendix F was the only phase modeled and simulated using DYNTRN. In the broken insulator experiment using phase R2, Reference [1] reported that the falling conductors did not reach the ground level. This information was important in the verification process, because the current version of DYNTRN doesn't have the capability of analyzing a cable that hits the ground.

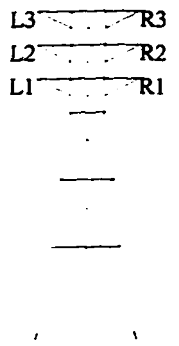


Figure 4.1- A Sketch of the Lattice Structures Tested in Reference [1].

<u>4.2.1.3 Analytical Model</u>

The computer model developed to represent phase R2 of the transmission line is shown in Figure 4.2. Conductors were modeled as cable elements, while insulators were modeled as truss elements. The properties of the conductors and insulators on phase R2 are listed in Table 4.1. The cross section area and modulus of elasticity of the insulator were not listed in the reference [1], and thus realistic values were assumed. Performing a parametric study on the value EA (young modulus * cross section area of the insulator) showed that the solution is not sensitive to the variation in this value. However, a smaller value of EA yielded better numerical stability for the solution. The above conclusion, regarding EA sensitivity, is in accordance with the results found in [6].

The support structures were modeled as linear springs, with stiffnesses calculated from force-displacement data of the support structures presented in the test report [1]. The stiffness data were calculated by performing linear static analysis on a single support structure.

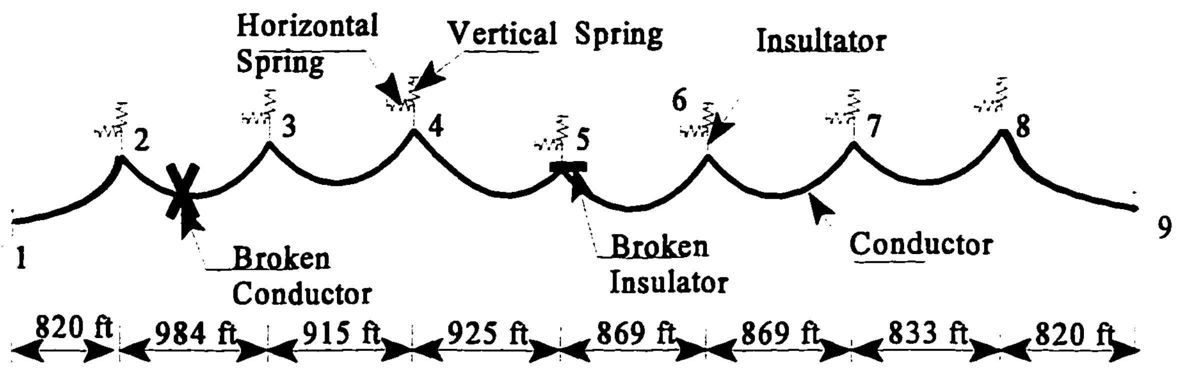


Figure 4.2. Computer Model for Broken Insulator and Broken Conductor Tests

Table 4.1- Conductor and Insulator Properties - Broken Insulator/Conductor Model [1]

Cable Properties		Insulator Properties	
Type	471A copper/bronze	Type	Single string porcelain bells
Area	0.3 in.^{2}	Area	Assumed (1 in.^{2})
Young modulus	15,000,000 psi	Young modulus	Assumed (2,900,000 psi)
Weight / length	0.073 lb/in	Weight	Assumed (67 lb)
Stringing tension	4305 lb	Length	87 in.

Therefore, the stiffness values may be underestimated because the effect of the attached conductors and shield wires were not included. The stiffness used for the vertical and horizontal directions were 6030 lb/in. and 2670 lb/in., respectively.

4.2.1.4 Sensitivity Studies

In order to acquire confidence in the model, sensitivity studies were performed on the dynamic time step, as well as the number of cable elements used per span. Changing the size of the time step from 0.10 sec to 0.05 sec produced approximately the same response. Similarly, the number of cable elements per span had a negligible effect on the response.

Comparing five, ten, and twenty elements, the insulator force next to the break differed by less than 2%. Therefore, for the broken insulator simulation, ten cable elements per span and a time step of 0.05 seconds were used. A small value for the time step size was used (1% of the period) in order to capture the details of the time-response of the results.

4.2.1.5 Analytical Simulation

The broken insulator incident was simulated in the computer model in two steps. First, a static analysis was performed due to conductor self weight. Then, the insulator at Location 5 was removed from the program database (see Figure 4.2). Since the damping of both the insulators and the conductors were not known, a range of realistic values was studied. The results illustrated that the solution was not very sensitive to the value of damping of the insulators. The solution, however, was affected by the value of damping used for the conductors. The damping was applied to the conductors as a ratio of the cable stiffness. The damping matrix, [C]= β*[K], where [K] is the stiffness matrix, and β is a user-defined ratio.

4.2.1.6 Results and Discussion

Figure 4.3-A shows the insulator force to the left of the broken insulator, at Location 4, for two β values, 0.01 and 0.04. These values correspond to damping ratios, η, of 0.5%, and 2% respectively, and are calculated using the following equation:

$$\eta = \frac{\beta\omega}{2} \qquad \mathbf{4.1}$$

Where ω is the natural frequency in rad/sec of the falling conductor first mode of vibration.

Figure 4.3-B shows the horizontal component of the insulator force for the same damping values. The value of the peak tension in the insulator changed from 2320 lb to 2120 lb (about 9.4%) as the damping ratio increased from 0.5% to 2%. The peak horizontal components of the insulator force were 896 lb and 809 lb, for 0.5% and 2% damping, respectively. The published experimental values for the insulator force and its horizontal components were 1960 lb and 891 lb, respectively.

Table 4.2 shows a comparison between the analytical and experimental values of the peak insulator force at Location 4, after an insulator broke at Location 5. Also listed is the percent error in the analytical results when compared to the published experimental data. From the results listed in the table, it can be seen that the conductors' damping properties affect the results of the computer simulation. Unfortunately, the damping properties of the conductors were not listed in the experimental report [1].

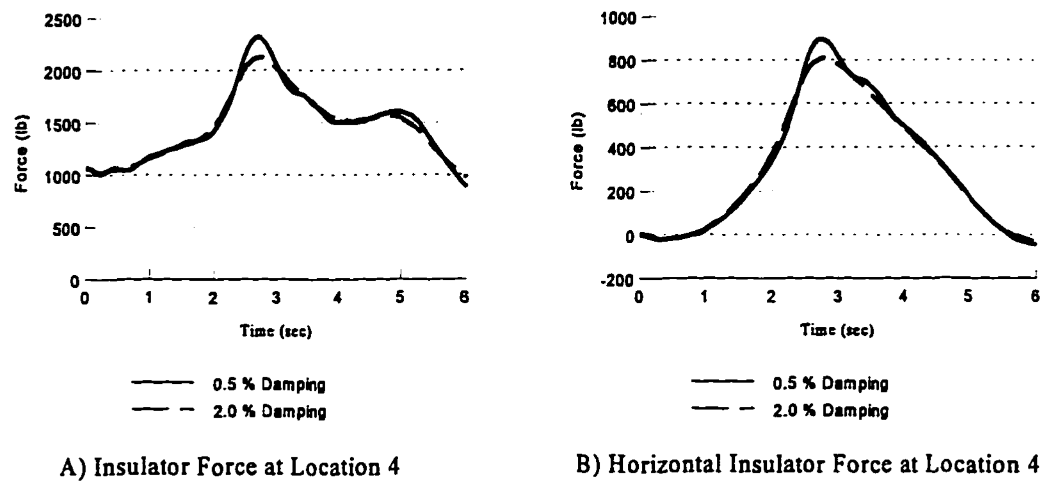


Figure 4.3 Broken Insulator Results using DYNTRN

Table 4.2 Insulator Force at Location 4 - Analytical vs. Experimental

	Insulator Force		Insulator Force (Longitudinal Component)	
	Value	Error[1]	Value	Error[1]
Analytical (0.5% damping)	2,320 lb	18%	896 lb	0.5%
Analytical (2.0% damping)	2,120 lb	8%	809 lb	-9.2%
Experimental	1,960 lb	N/A	891 lb	N/A

[1] Error = ((analytical value- experimental value) / experimental value) x 100

4.2.2 Broken Conductor Analysis

4.2.2.1 Description of the Problem

In a broken conductor analysis, the effect of a broken conductor on the adjacent spans of the transmission line is investigated. A broken conductor problem is simulated in DYNTRN by removing the cable element representing the broken conductor from the program database.

4.2.2.2 Background on Experimental Data

DYNTRN was also validated using the broken conductor results presented in reference [1]. A series of broken conductor tests were conducted on the same line mentioned the section 4.2.1. The test chosen for verification used the same conductor phase, R2, used in the broken insulator analysis presented earlier (see Figure 4.2). For more information on the experimental data used in this analysis, refer to Appendix F.

4.2.2.3 Analytical Model

The computer model was exactly the same as the one used for the broken insulator analysis (see Figure 4.2). The properties of the model are listed in Table 4.1.

4.2.2.4 Sensitivity Studies

A sensitivity study was performed to find the effect of the number of cable elements per span on the analysis results. Figure 4.4 shows the insulator force at Location 3, to the left of the broken conductor. The figure shows the results of three analyses, using 10, 20 and 30 cable elements per span. The response of the first analysis using 10 elements was approximately 10% higher than the second analysis using 20 elements. The latter was approximately 2.5% higher than the third analysis using 30 elements. Based on these results, it was decided to use 20 elements per span for the broken conductor problem presented herein.

In order to reach convergence in the broken conductor problem, it was necessary to use time step size of 0.001 sec. In some cases, it was even necessary to use a value as small as 0.0001 sec during parts of the analysis where the cable was slack, i.e., had zero tension. These values of the time step were considerably smaller than the value used in the broken insulator analysis, and values recommended in the literature [6]. Therefore, no further reduction in the size of the time step was conducted to investigate its effect on the results.

4.2.2.5 Analytical Simulation

The broken conductor analysis was conducted in two steps. First, a static analysis was conducted due to the conductors' self weight. Then, the cable element representing the conductor between Locations 2 and 3 was removed from the program database, and a dynamic time integration analysis was performed.

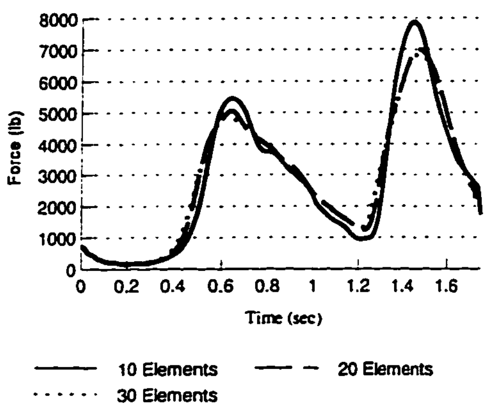


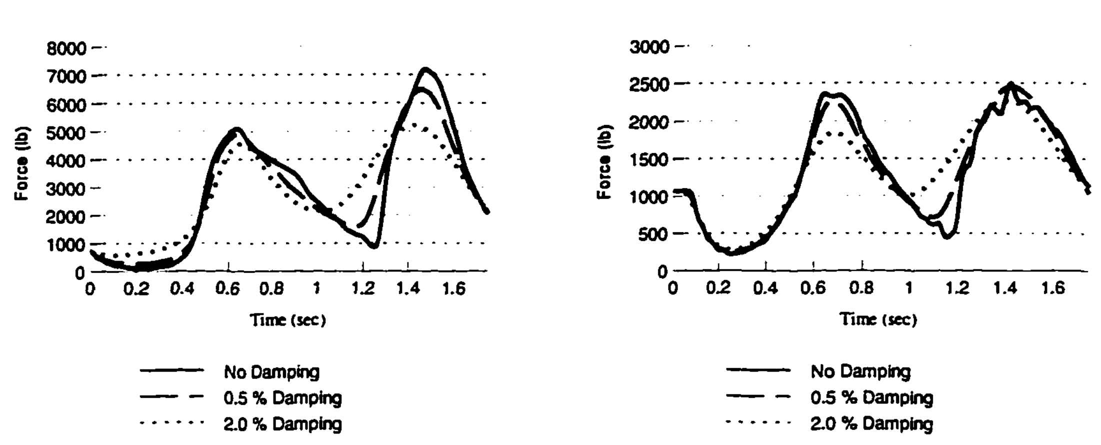
Figure 4.4 Broken Conductor Analysis - Sensitivity Study

4.2.2.6 Results and Discussion

Figure 4.5 presents the computer analysis results of the broken conductor problem. Figure 4.5-A shows the force in the insulator at Location 3, to the left of the broken conductor. Figure 4.5-B shows the force in the insulator at Location 4. The results are plotted for three values of the damping parameter, β, of the cable element: 0, 0.004 and 0.015. These values correspond to damping ratios, η, of zero, 0.5%, and 2%, respectively. Damping ratios are calculated as shown in Equation 4.1.

As seen in Figure 4.5, the response consists of two peaks occurring at approximately 0.65 sec and 1.5 sec. The figure also shows that the results are very sensitive to the conductor damping ratio used in the analysis. The peak force increased by approximately 25% as the damping decreased from 2% to 0.5%.

Table 4.3 shows a comparison between the results obtained from the computer model, and the published experimental results. The times listed are the times at which the peaks occurred. In the computer model, the time at which the first peak of the insulator force occurred was 0.15 sec higher than the time reported in the experimental report. Also listed is the percent error in the analytical results when compared to the published experimental data. For a damping ratio of 2%, the difference between the analytical and experimental results was more than 30%. The difference decreased to approximately 20% as the damping ratio decreased to 0.5%. With no damping present in the conductor, the difference decreased to 8% for the force at Location 3, and 16% for that at Location 4.



A) Insulator Force at Location 3 B) Insulator Force at Location 4

Figure 4.5 Broken Conductor Analysis- Analytical Results

Two conclusions can be drawn from the results in Table 4.3. First, the broken conductor analysis is more sensitive to the damping value, β, than the broken insulator analysis, presented in Section 4.2.1. This may be due to the participation of higher modes in the broken conductor case, where a very small value of β is needed to ensure that the higher modes of vibration are not damped out. Secondly, the results obtained by the computer program, DYNTRN, are under-conservative compared to the experimental results. The analysis showed, however, that using 10 cable elements per span instead of 20 to model the conductors yielded results that are closer to the experimental results. The first peak for the

Table 4.3 Broken Conductor Results - Analytical vs. Experimental

Insulator Force at Location 3 (Next to break)						
	First Peak			Second Peak		
	Value	Error[1]	Time[2]	Value	Error[1]	Time[2]
Analytical (No damping)	5,100 lb	-8.1%	0.64 sec	7,200 lb	-8.1%	1.48 sec
Analytical (0.5% damping)	4,910 lb	-11.5%	0.64 sec	6,480 lb	-17.3%	1.44 sec
Analytical (2.0% damping)	4,550 lb	-18%	0.66 sec	5,210 lb	-33.5%	1.41 sec
Experimental	5,547 lb	N/A	0.50 sec	7,832 lb	N/A	1.5 sec
Insulator Force at Location 4 (One span away from break)						
	First Peak			Second Peak		
	Value	Error[1]	Time[2]	Value	Error[1]	Time[2]
Analytical (No damping)	2,370 lb	-16.3%	0.64 sec	2,270 lb	-11.5%	1.48 sec
Analytical (0.5% damping)	2,270 lb	-19.8%	0.67 sec	2,450 lb	-4.5%	1.41 sec
Analytical (2.0% damping)	1,850 lb	-34.7%	0.68 sec	2,280 lb	-11.1%	1.39 sec
Experimental	2,832 lb	N/A	0.47 sec	2,566 lb	N/A	1.3 sec

[1] Error = ((analytical value- experimental value) / experimental value) x 100

[2] Time at which the peak in the force value occurs

insulator force at Location 3, for example, was less than the experimental value by about 1.5 percent, when no cable damping was used. From a theoretical point of view, however, modeling the conductor with 20 elements per span should yield a more accurate result.

The conclusions drawn above are based on a single experiment, and cannot be generalized. Therefore, an extensive experimental program is needed to produce general guidelines about the results of the computer simulation program, DYNTRN, with respect to the broken conductor analysis. This, however, is not within the scope of this report. The discussion in this section showed that the computer simulation produced results that are close to the experimental results, when small or no cable damping is used.

4.2.3 Broken Shield Wire Analysis

4.2.3.1 Description of the Problem

In a broken shield wire analysis, the effect of a broken shield wire on the supporting structure and adjacent spans of the transmission line is investigated. A broken shield wire problem is simulated in DYNTRN by removing the cable element representing the broken shield wire from the program database.

4.2.3.2 Background on Experimental Data

Mozer et al. [3] conducted a series of broken conductor and broken shield wire tests on a scale model to study the behavior of transmission lines due to these loading conditions. The test model consisted of two scale-model structures and three 32 ft spans of conductors and shield wires, as shown in Figure 4.6. Conductors and shield wires were modeled using 18 and 24 gauge copper wires, respectively. Lead weights were attached to the wires to increase their weight. A schematic of the model structure is shown in Figure 4.7. Detailed information about the experimental data is listed in Appendix F.

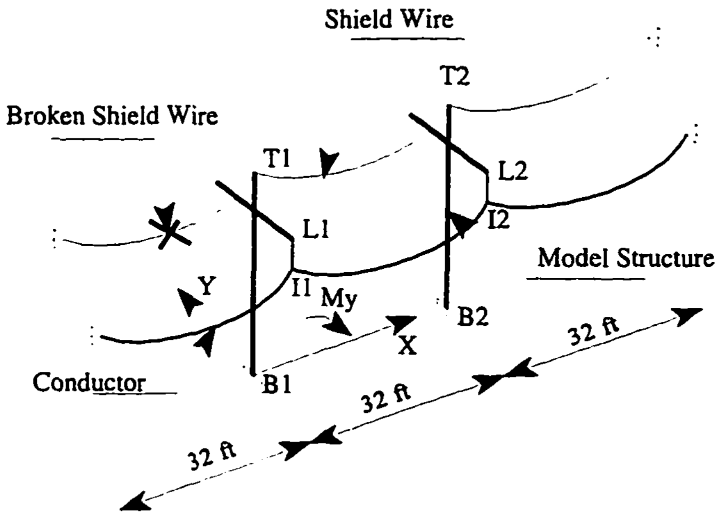
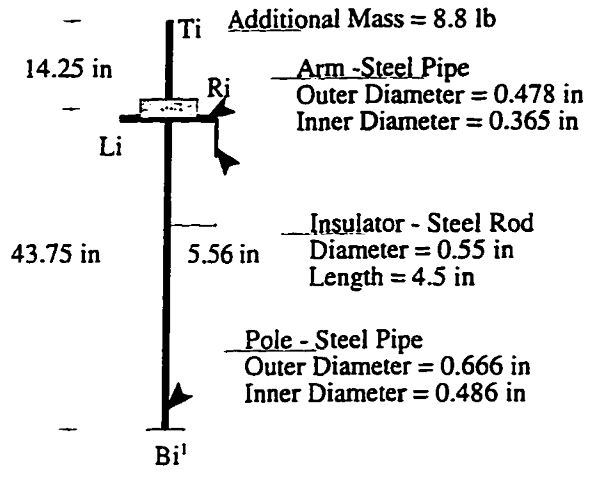


Figure 4.6 Analytical Model used in the Broken Shield Wire Verification



[1] Subscript "i" refers to the structure number

Figure 4.7 Scale Model of Transmission Line Structure [3]

4.2.3.3 Analytical Model

The computer model used to represent the experimental scale model is shown in Figures 4.6 and 4.7. Beam elements were used to represent support structures, truss elements were used to represent insulators, and cable elements were used to represent conductors and shield wires. The cross sectional properties of the support structures, and the insulators are shown in Figure 4.7. Properties of the conductors and shield wires are listed in Table 4.4.

Table 4.4 Conductor and Shield Wire Properties - Broken Shield Wire Analysis

Property Name	Conductor	Shield Wire
Type	Copper 18 gauge	Copper 24 gauge
Area * Young modulus (EA)	12100 lb	3300 lb
Weight / length	0.0597 lb/ft	0.0132 lb/ft
Stringing tension	15.3 lb	1.51 lb

4.2.3.4 Sensitivity Studies

A sensitivity study was performed to assess the effect of damping and other parameters on the moment at the base of the model structure. Both the conductor damping and the shield wire damping ratios had negligible effect on the results. The structure damping, however, had a small effect on the results. Increasing the damping ratio from 1% to 5% decreased the value of the first peak from 130 lb-in. to 123 lb-in.; about 5%. A sensitivity study on the number of cable elements per span showed that using 10 cable elements per span yielded the same results as 20 elements for the broken shield wire analysis.

4.2.3.5 Analytical Simulation

The transmission line was analyzed first due to static loads listed in Table 4.4. Then the broken shield wire event was simulated by removing the cable element representing the broken shield wire as shown in Figure 4.6. The simulation was performed using a value of damping of 0.5% and 1% for the cables (conductors and shield wires) and the structures, respectively. A time step size of 0.01 sec was used in the simulation, which is about 2.5% of the shield wire fundamental period of vibration. A value of a time step smaller than that recommended in the literature [6] was used to capture the details of the time response of the results.

4.2.3.6 Results and Conclusion

A comparison between the analytical and experimental solution is shown in Figure 4.8. The figure shows the moment at the base of the model structure at Location B1. The moment plotted is the moment bending the structure in the direction of the line, M_y. The figure shows good agreement between the computer simulation and the experimental results.

4.2.4 Conductor Galloping Analysis

4.2.4.1 Description of the Problem

Conductor galloping refers to the low frequency, high amplitude vibration occurring to conductors due to the action of wind on iced conductors. DYNTRN simulates the galloping action of the conductor by imposing a harmonic displacement on the cable with an amplitude that represents the amplitude of galloping. Therefore, to utilize this method, one needs to

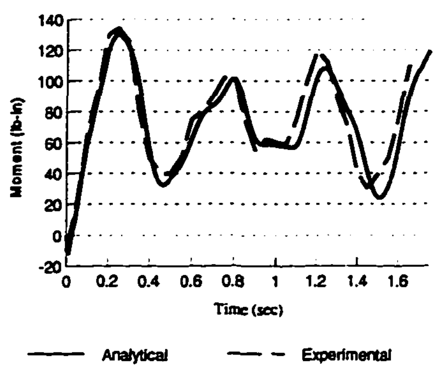


Figure 4.8 Broken Shield Wire - Moment at B1

define the amplitude and frequency of the galloping motion. Several galloping models exist in the literature that can be used to obtain these values. Some galloping models can also provide the mode shape in which the conductor will gallop (see Chapter 2 for a review of galloping models).

As reported in the literature, a stable galloping vibration will occur in one of the dynamic mode shapes of the conductor [38,39]. Therefore, the shape of the imposed displacements on the cable element should be selected to match one of the conductor's vibration modes. This process is shown schematically in Figure 4.9. The galloping simulation procedure can be summarized as follows:

1. At the beginning of a time step, the profile of the galloping conductor, X_g, can be expressed as:

$$X_g = X_s + A_g MS_i \sin(\omega_g t) \qquad 4.2$$

Where X_s is the static profile of the cable resulting from the static load supplied by the user, A_g is the galloping amplitude defined by the user, MS_i is the normalized mode shape for mode i. ω_g is the frequency of the galloping vibration in rad/sec, and t is the time in seconds. The frequency of galloping is either provided by the user or selected as the natural frequency of mode i. The user also selects which vibration mode to be used in the galloping analysis.

2. After calculating the position of the cable using Equation 4.2, the stiffness and cable end forces are calculated. The cable stiffness matrix is used in the assembly of the global stiffness matrix of the structure. The end forces are added to the global internal force vector of the structure, and the global residual force vector is calculated as explained in Chapter 3.
3. If the L2 norm of the residual force vector is smaller than the tolerance value chosen by the user, the analysis has converged, and proceeds to the next time step. Otherwise, the position of the end nodes of the cable element is updated, and a new static profile and mode shape are calculated for the updated position of the cable. The galloping profile of the cable is then recalculated according to Equation 4.2, and the analysis proceeds to Step 2.

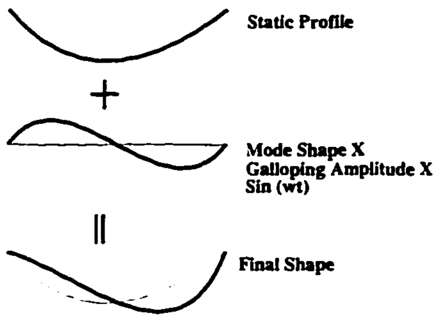


Figure 4.9 Conductor Galloping Analysis

4.2.4.2 Background on Experimental Work

In order to validate the galloping procedure discussed above, it was necessary to find experimental work on galloping that gives information on both the amplitudes of the galloping motion and its effect on the transmission line components, such as the forces in the attached insulators, or the stresses in the supporting structures. Unfortunately, in most of the reviewed experimental work related to galloping, the galloping amplitude and shape of vibration were the major parameters studied, and no information on the galloping forces was published. In few situations, such as Reference [2], galloping forces were the main focus of investigation, but the corresponding galloping amplitudes were not mentioned. Reference [2] was used to verify the galloping method. However, no galloping amplitudes were reported in the experimental work. There was also no information given about the ice shape accumulated on the conductor, and therefore the galloping amplitude could not be calculated using the galloping models discussed in Chapter 2. Therefore, the data was only used to compare the shape and frequency of the response. The comparison, however, does not prove the accuracy of the results as far as values are concerned. In Reference [2], the insulator force response was published for two cases of real galloping situations. The conductor used was Grosbeak 636. For more information about the experimental data published in Reference [2], see Appendix F.

4.2.4.3 Analytical Model

Figure 4.10 shows the computer model used to represent the transmission line. Four conductor spans were included in the model. The support structures were considered rigid, and were replaced by fixed supports, since no information about the supporting structures was given in the reference. This assumption was justified because the reported magnitude of galloping forces was not high enough to cause considerable deformations in the structures. The conductors were assumed to be on the same elevation, since no information about the elevations was reported. Conductors were represented using cable elements with 10 sub-elements per span. Insulators were modeled using truss elements.

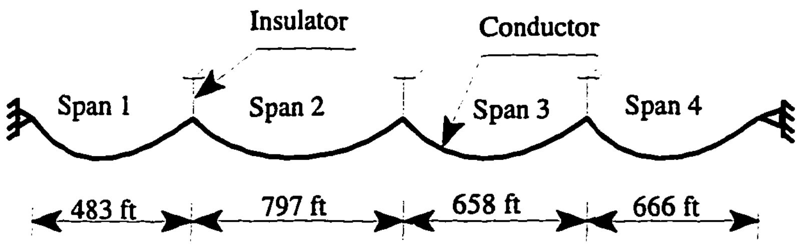


Figure 4.10. Computer Model used in the Conductor Galloping Analysis Example [2]

4.2.4.4 Analytical Simulation

A static analysis was performed due to a uniform load of 0.875 lb/ft applied on the cables. Although the magnitude of the load was not listed in the reference [2], it was calculated using the listed insulator tension data. Galloping amplitudes of nine inches and ten inches were then imposed on span 2 and span 3, respectively. Span 2 was forced to vibrate in the first anti-symmetric mode, while span 3 vibrated in the first symmetric mode. The frequency of the galloping vibration was chosen to be 0.46 Hz and 0.28 Hz for span 2 and 3, respectively. No galloping amplitudes were imposed on spans 1 and 4. The galloping amplitudes, frequencies and modes of vibration used in the analytical model were chosen to give a response similar in magnitude and frequency to that obtained in the experimental report by McConnell [2]. In order to perform the comparison between the analytical and experimental results, a frequency analysis of the analytical response was performed and compared to the results of the frequency analysis performed in the experimental report [2], which showed that the most dominant frequencies were 0.28 Hz and 0.46 Hz with corresponding amplitudes of 55.4 lb and 36 lb, respectively.

4.2.4.5 Results and Discussion

Figure 4.11 shows the force in the insulator between spans 2 and 3, obtained from the computer model as well as the experimental data. The experimental data were obtained at a wind speed of 8.1 mph. It can be seen that both responses are comparable. The discrepancies between the two responses might be due to the higher modes of vibration detected in the experimental report, but was not modeled in the analytical solution. Only the most two dominant modes of vibration were modeled analytically, because in DYNTRN, more than one mode of galloping vibration cannot be imposed on the cable element simultaneously.

As stated earlier, the comparison presented herein only proves that the analytical results are similar in nature to the measured results, but in no way does it prove its accuracy as far as values are concerned. It should be noted, however, that the relation between the galloping amplitude and the corresponding conductor tension was verified using theoretical formulas for the case of a single span conductor fixed at both ends (see Appendices D and E).

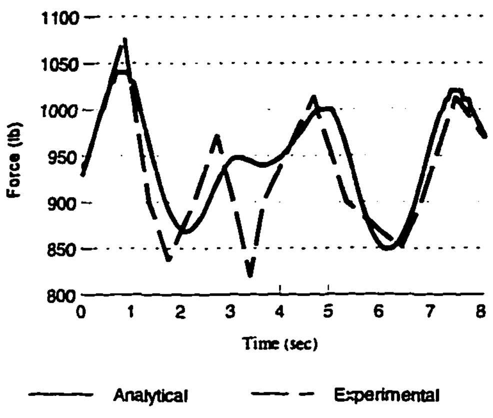


Figure 4.11 Galloping Conductor Analysis - Analytical vs. Experimental

CHAPTER 5 - SUMMARY, CONCLUSION, AND RECOMMENDATION

5.1 Summary

A graphical computer simulation tool, DYNTRN [82], was developed for analyzing complete transmission lines including support structures, insulators, and conductors due to dynamic loading conditions, such as a broken conductor, a broken insulator, or conductor galloping.

A literature review was conducted to study previous pertinent research. The cable element used to simulate conductors was studied in detail. Research performed in the area of broken conductor and broken insulator analyses was also reviewed. Several galloping models presented in the literature were also documented. Output obtained from these models can be used as input to DYNTRN. Object Oriented Programming (OOP) was used as the development tool for DYNTRN. Therefore, principles of OOP, as well as previous research using OOP in structural analysis programs, were reviewed.

The design of the analytical software using OOP was explained, and different objects used to develop the program were presented. The time integration procedure used to solve the dynamic equations of motion was explained. A dynamic condensation method used to condense the internal degrees-of-freedom of a cable element was developed and documented. A new pointer-based method was developed to efficiently store and solve large sparse square matrices. The procedure for solving large deflection and large rotation problems was also explained.

Finally, analytical as well as experimental verification of the simulation software, DYNTRN, was conducted. Four experimental case studies, including a broken insulator analysis, a broken conductor analysis, a broken shield wire analysis, and conductor galloping analysis, were presented.

5.2 Conclusion

This research produced a computer analysis tool that is capable of simulating the response of a complete transmission line system in 3-D. The main features of the computer program, DYNTRN [82], are:

- The capability to analyze transmission line support structures, conductors, and insulators.
- The ability to solve for different types of static loadings, including loads that are not readily solvable using general purpose finite element programs due to convergence problems, such as large horizontal imbalance loads caused by differential ice loading on transmission line conductors.
- The ability to solve for different dynamic conditions including conditions that are specific to transmission lines, such as a broken conductor event, a broken insulator event, or a conductor galloping event.
- An interactive and user friendly graphical user interface, advanced graphical capabilities including 3-D plots, and real-time parametric graphs that are graphically updated as the solution proceeds.
- The capability to use names instead of numbers to model nodes and elements of the transmission line, making large problems easier to model.
- An interactive analysis solution, making it possible to change loads or properties or remove elements in the middle of an analysis.
- The ability, through the use of OOP, to allow for future extensions and modifications.
- An efficient pointer-based equation solver that is especially useful for large problems.
- An innovative method for modeling cable elements using dynamic condensation.

DYNTRN was tested for several dynamic loadings including a broken insulator, a broken conductor, a broken shield wire, and a galloping conductor analysis. The results showed reasonable agreement with experimental work published in the literature.

5.3 Recommendation

Subsequent research on the work presented herein should focus on two directions. The first direction is the continuous enhancement of the capabilities and efficiency of the program. The second direction relates to experimental testing.

DYNTRN was designed to allow for future extensions. This was made possible through the modular design of the program using OOP. Enhancements to DYNTRN could include new solution capabilities, improved solution techniques, or a more efficient program interface. The following future additions are proposed for DYNTRN:

- Adding the capability to analyze a case with the conductor partially lying on the ground.
- Integrating into DYNTRN one or more of the galloping models reviewed in Chapter 2. Thus, the galloping amplitude provided by the particular galloping model can be automatically included as input to DYNTRN. Currently, the user must input data from a separate calculation to DYNTRN.
- Initiating a research program to develop and investigate an efficient method for simulating the cable element and calculating its stiffness matrix.
- Developing an interface program for automatic generation of transmission line models. Location of support structures, geometry of the support structure, and the number of conductor phases are examples of typical input data to the interface program. The interface program could also include a database of common conductor types as well as common support structure types used in transmission lines.

As shown in Chapter 4, only a limited validation of DYNTRN could be performed due to the limited amount of available experimental data. In order to further validate DYNTRN, an extensive experimental program is needed to:

- Validate the stress and strain in the different components of the transmission line.
- Establish guidelines for acceptable ranges of input data such as damping, and evaluate the sensitivity of the results to these input values.
- Provide guidelines on modeling issues, such as the number of elements required to represent the conductor and the adequate time step for different types of analysis.

APPENDIX A - DYNTRN OBJECT PROTOTYPES

A.1 Matrix and Vector Classes

Matrix Class

```
class MATRIX:public CObject
{
/* Variables */
protected:
    double *beg;
public:
    int row,col,band;
/*Functions*/
    MATRIX (): beg(NULL) { }
    MATRIX (int r,int c);
    ~MATRIX();
    void Create(int r, int c);
    void CreateSBand(int r,int b);
    void Destroy();
    DECLARE_SERIAL(MATRIX)
    virtual void Serialize(CArchive& archive);
    MATRIX (const MATRIX& MatSrc);
    MATRIX &operator = (const MATRIX &M2);
    MATRIX &operator = (double *di);
    MATRIX &operator += (const MATRIX &M1);
    MATRIX &operator -= (const MATRIX &M1);
    MATRIX &operator *= (const MATRIX &M1);
    MATRIX &operator *= (double d);
    VECTOR operator [] (int c) const;
    BOOL IsEmpty() const;
    MATRIX operator - ();
    void Add(const MATRIX &M1,const MATRIX &M2);
    void Subtract(const MATRIX &M1,const MATRIX &M2);
    void Multiply(const MATRIX &M1,const MATRIX &M2);
    void Multiply(const MATRIX &M1,double d);
    void Clear();
    BOOL operator == (const MATRIX &M)  const;
    BOOL operator != (const MATRIX &M)  const;
    void IDMATRIX();
    void MatProduct(VECTOR &V);
    double GetIJ(int irow,int icol);
    double& PutIJ(int irow,int icol);  };
```

```
// Other MATRIX Operations not memebers of the MATRIX class
MATRIX  operator + (const MATRIX &M1, const MATRIX &M2);
MATRIX  operator - (const MATRIX &M1, const MATRIX &M2);
MATRIX  operator * (double d,const MATRIX &M1);
MATRIX  operator * (const MATRIX &M1,double d);
MATRIX  operator * (const MATRIX &M1,const MATRIX &M2);
VECTOR  operator * (const MATRIX &M1,const VECTOR &V1);
void Insert(MATRIX &Main,const MATRIX &Part,int Istart,int Jstart);
void Insert(MATRIX &Main,const VECTOR &Part,int Istart,int Jstart);
void AddInsert(MATRIX &Main,const MATRIX &Part,int Istart,int Jstart);
void Extract(const MATRIX& Main,MATRIX& Part,int Istart,int Jstart);
void Extract(const MATRIX& Main,VECTOR& Part,int Istart,int Jstart);
double  Trace(const MATRIX &M);
double  Trace2(const MATRIX &M);
double  Norm(const MATRIX &M);
MATRIX Transp(const MATRIX &M2);
```

Square Banded Matrix Class

```
class BMATRIX:public CObject
{
/* Variables */
protected:
    double *beg;
public:
    int row,band;
/* Functions */
    BMATRIX (): beg(NULL) { }
    BMATRIX (int r,int b);
    ~BMATRIX();
    void Create(int r, int b);
    void Destroy();
    DECLARE_SERIAL(BMATRIX)
    virtual void Serialize(CArchive& archive);
    BMATRIX (const BMATRIX& MatSrc);
    BMATRIX &operator = (const BMATRIX &M2);
    BMATRIX &operator += (const BMATRIX &M1);
    BMATRIX &operator -= (const BMATRIX &M1);
    BMATRIX &operator *= (double d);
    BOOL IsEmpty() const;
    BMATRIX operator - ();
    void Add(const BMATRIX &M1,const BMATRIX &M2);
    void Subtract(const BMATRIX &M1,const BMATRIX &M2);
    void Multiply(const BMATRIX &M1,double d);
    void Clear();
    void IDMATRIX();
    int GetStart(int irow) const;
    double GetNIJ(int irow,int icol) const;
    double &PutNIJ(int irow,int icol);
    double GetIJ(int irow,int iband) const;
    double& PutIJ(int irow,int iband);
};

// Other BMATRIX Operations not members of class BMATRIX
BMATRIX  operator + (const BMATRIX &M1, const BMATRIX &M2);
BMATRIX  operator - (const BMATRIX &M1, const BMATRIX &M2);
BMATRIX  operator * (double d,const BMATRIX &M1);
BMATRIX  operator * (const BMATRIX &M1,double d);
VECTOR operator * (const BMATRIX &M1, const VECTOR &V1);
void Insert(BMATRIX &Main,const MATRIX &Part,int Istart,int Jstart);
void AddInsert(BMATRIX &Main,const MATRIX &Part,int Istart,int Jstart);
void Extract(const BMATRIX& Main,MATRIX& Part,int Istart,int Jstart);
void Extract(const BMATRIX& Main,BMATRIX& Part,int Istart,int Jstart);
```

<u>**Vector Class**</u>

```
class VECTOR:public CObject
{
/* Variables */
protected:
    double *beg;
public:
    int row;
/* Functions */
    VECTOR (): beg(NULL) { }
    VECTOR (int r);
    ~VECTOR();
    virtual void Create(int r);
    void Destroy();
    DECLARE_SERIAL(VECTOR)
    virtual void Serialize(CArchive& archive);
    VECTOR (const VECTOR& VecSrc);
    BOOL IsEmpty() const;
    VECTOR &operator = (const VECTOR &V2);
    VECTOR &operator = (double *di);
    VECTOR &operator += (const VECTOR &V1);
    VECTOR &operator -= (const VECTOR &V1);
    VECTOR &operator *= (double d);
    VECTOR operator - ();
    void Add(const VECTOR &V1,const VECTOR &V2);
    void Subtract(const VECTOR &V1, const VECTOR &V2);
    double  GetI(int irow) const;
    double& PutI(int irow);
    void Clear();
    BOOL operator == (const VECTOR &V) const;
    BOOL operator != (const VECTOR &V) const;
 };

// Other vector operations not members of VECTOR class
VECTOR  operator + (const VECTOR &V1,const VECTOR &V2);
VECTOR  operator - (const VECTOR &V1,const VECTOR &V2);
VECTOR  operator * (double d,const VECTOR &V1);
VECTOR  operator * (const VECTOR &V1,double d);
double  operator * (const VECTOR &V1,const VECTOR &V2); //SCALAR PRODUCT
VECTOR  Mult(const VECTOR &V1, const VECTOR &V2);
void Insert(VECTOR &Main,const VECTOR &Part,int Istart);
void AddInsert(VECTOR &Main,const VECTOR &Part,int Istart);
void Extract(VECTOR &Main,VECTOR &Part,int Istart);
double   Norm(const VECTOR &V);
double   INorm(const VECTOR &V);
```

<u>3D Vector Class</u>

```
class GVECTOR : public CObject
{
// Variables
protected:
    VECTOR   m_XYZ;        // Vector containing the coordinates
public:
// Functions
    GVECTOR() : m_XYZ(3) {}
    GVECTOR (const GVECTOR& GVecSrc);
    ~GVECTOR() {}
    DECLARE_SERIAL(GVECTOR)
    virtual void Serialize(CArchive& archive);
    void PutGVECTOR(double *xyzinp);
    void AddGVECTOR(double *xyzinp);
    void GetGVECTOR(double *xyzout);
    friend GVECTOR  operator ^ (const GVECTOR &V1,const GVECTOR &V2);
    void GetDirCos(CSYS* pcs,double &c1, double &c2, double &c3);
    // Transfer from one Csys to another
    GVECTOR TransfCsys(CSYS *pOld, CSYS *pNew);
    GVECTOR FTransfCsys(CSYS *pOld, CSYS *pNew); //Force version
    // Transfer to Global System
    GVECTOR TransfToGlobal(CSYS* pOld);
    GVECTOR FTransfToGlobal(CSYS* pOld); // Force version
    // Transfer From Global System
    GVECTOR TransfFromGlobal(CSYS* pNew);
    GVECTOR FTransfFromGlobal(CSYS* pNew); // Force version
    operator VECTOR () const;
    GVECTOR& operator = (const VECTOR &V1);
    GVECTOR& operator = (const GVECTOR &G1);
    GVECTOR& operator -= (const GVECTOR &G1);
    GVECTOR& operator += (const GVECTOR &G1);
    GVECTOR& operator -= (const VECTOR &V1);
    GVECTOR& operator += (const VECTOR &V1);
    void Subtract(const GVECTOR &G1,const GVECTOR &G2);
    void Add(const GVECTOR &G1,const GVECTOR &G2);
    void Subtract(const GVECTOR &G1,const VECTOR &V2);
    void Add(const GVECTOR &G1,const VECTOR &V2);
    double  GetI(int irow) const;
    double& PutI(int irow);
    void Zero();
    // Change a GVECTOR to a device context point
    void ToTVECTOR(TVECTOR &t1);
    void TextSerialize(fstream & FIO,int RW) ;
};
```

A.2 Coordinate System Class

Coordinate Sytem Class

```
// All Coordinate Systems are referenced
// internally to the Global Coordinate system
class CSYS : public CObject
{
/* Variables */
public:
   GVECTOR X;
   GVECTOR Y;
   GVECTOR Z;
```

```
    GVECTOR origin;
    MATRIX TRANS;
public:
/* Functions */
    CSYS();
    ~CSYS();
    DECLARE_SERIAL(CSYS)
    virtual void Serialize(CArchive& archive);
    // Different ways to define the goordinate system
    void GenCSYS(GPOINT &po,GVECTOR &ax,GVECTOR &ay,GVECTOR &az);
    void GenCSYS_O_Px_Pxy(GPOINT &po,GPOINT &px,GPOINT &pxy);
    void GenCSYS_O_Px_Pz(GPOINT &po,GPOINT &px,GPOINT &pz);
    void GenCSYS_O_Px_Ay(GPOINT &po,GPOINT &px,GVECTOR &ay);
    MATRIX& GetTransf() {return TRANS;}
};
```

A.3 Displacement, Rotation, Force and Moment Classes

Displacement Class

```
class DISP :public GVECTOR
{
/* Variables */
public:
    BOOL FixCode[3];
public:
/* Functions */
    DISP();
    ~DISP() {}
    DECLARE_SERIAL(DISP)
    virtual void Serialize(CArchive& archive);
    void GetRDISP(double* rdisp);
    DISP& operator = (const VECTOR &V);
    DISP& operator = (const DISP &G);
    void TextSerialize(fstream & FIO,int RW) ;
};
```

Rotation Class

```
class ROTAT :public GVECTOR
{
/* Variables */
public:
     BOOL FixCode[3];
/* Functions */
     void OutOmega(VECTOR &Omega) const;
     void InOmega(const VECTOR &Omega);
     void OutAux(MATRIX &A);
     void InAux(const MATRIX &A);
     void InTrans(const MATRIX &T);
     void OutTrans(MATRIX &T);
     ROTAT();
     ~ROTAT() {}
     DECLARE_SERIAL(ROTAT)
     virtual void Serialize(CArchive& archive);
     void FDump(ofstream &FOut) const;
     void GetRROTAT(double *jrest);
     void ChangeCoord(ROTAT& nwrotat, CSYS* Old,CSYS* New);
     ROTAT& operator = (const VECTOR &V);
     ROTAT& operator = (const ROTAT &G);
     friend ROTAT operator + (const ROTAT &R1,const ROTAT &R2);
     void TextSerialize(fstream & FIO,int RW) ;
};

// Other functions related to the rotation class, but not members of the class
     void OutAuxO(MATRIX &A,const VECTOR &w);
     void InAuxO(const MATRIX &A,VECTOR &w);
```

<u>**Force Class**</u>

```
class FORCE :public GVECTOR
{
/* Functions */
public:
     FORCE() {}
     ~FORCE() {}
     DECLARE_SERIAL(FORCE)
     virtual void Serialize(CArchive& archive);
     void GetFORCE(double *jloads);
     FORCE& operator = (const VECTOR &V);
     FORCE& operator = (const FORCE &G);
     void TextSerialize(fstream & FIO,int RW) ;
};
```

Moment Class

```
class MOMENT :public GVECTOR
{
/* Functions */
public:
    MOMENT() { }
    ~MOMENT() { }
    DECLARE_SERIAL(MOMENT)
    virtual void Serialize(CArchive& archive);
    void GetMOMENT(double *jloads);
    MOMENT& operator = (const VECTOR &V);
    MOMENT& operator = (const MOMENT &G);
    void TextSerialize(fstream & FIO,int RW) ;
};
```

A.4 Load and Load History Classes

Nodal Load Class

```
class NLOAD : public CObject
{
/* Variables */
public:
    FORCE m_force;
    MOMENT m_moment;
    DISP m_disp;
    ROTAT m_rotat;
    int m_Defined;
/* Functions */
    NLOAD();
    NLOAD (const NLOAD& NLDSrc);
    ~NLOAD() { }
    DECLARE_SERIAL(NLOAD)
    virtual void Serialize(CArchive& archive);
    NLOAD &operator = (const NLOAD &NL1);
    friend NLOAD  operator - (const NLOAD &ld1, const NLOAD &ld2);
    friend NLOAD  operator + (const NLOAD &ld1, const NLOAD &ld2);
    friend NLOAD  operator * (double d1, const NLOAD &ld2);
    void TextSerialize(fstream & FIO,int RW) ;
};
```

Nodal Load History Class

```
class NHLOAD : public CObject
{
/* Variables */
public:
    CString m_Name;
    CObArray m_Loads;
    double m_STime;
    double m_ETime;
    double m_ITime;
/* Functions */
    NHLOAD();
    ~NHLOAD();
    DECLARE_SERIAL(NHLOAD)
    virtual void Serialize(CArchive& archive);
    void SetLoad(int Index, NLOAD nload);
    NLOAD GetLoad(double tm);
    NLOAD InterpolateLd(int pInd,int nInd,double tm);
    void AdjustTScale(double STime,double ETime,double ITime);
    void ClearArray();
    NHLOAD &operator = (const NHLOAD &NL1);
    void TextSerialize(fstream & FIO,int RW) ;
};
```

Element Load Class

```
class ELLOAD : public CObject
{
/* Variables */
public:
    // Force values to discribe the element forces
    GVECTOR m_Val[2];
    int m_Defined;
/* Functions */
    ELLOAD();
    ELLOAD (const ELLOAD& ELDSrc);
    ~ELLOAD() { }
    DECLARE_SERIAL(ELLOAD)
    virtual void Serialize(CArchive& archive);
    ELLOAD &operator = (const ELLOAD &EL1);
    void TextSerialize(fstream & FIO,int RW) ;
};
```

Element Load History Class

```
class ELHLOAD : public CObject
{
/* Variables */
public:
    CString m_Name;
    // An Array to describe the loads
    CObArray m_Loads;
    // Variables to describe the time scale
    double m_STime;
    double m_ETime;
    double m_ITime;
    int m_LCoord; // Local or Global
/* Functions */
    ELHLOAD();
    ~ELHLOAD();
    DECLARE_SERIAL(ELHLOAD)
    virtual void Serialize(CArchive& archive);
    void SetLoad(int Index, ELLOAD elload);
    ELLOAD GetLoad(double tm);
    ELLOAD InterpolateLd(int pInd,int nInd,double tm);
    void AdjustTScale(double STime,double ETime,double ITime);
    void ClearArray();
    ELHLOAD &operator = (const ELHLOAD &EL1);
    void TextSerialize(fstream & FIO,int RW) ;
};
```

A.5 Node Class

Node Class

```
class NODE : public CObject
{
/* Variables */
    int DOFMap[6];
    int FORMap[6];
    public:
    CString m_Name;
    // Drawing VECTORS
    TVECTOR DR_Ocoord;
    TVECTOR DR_Dcoord;
    // Nodal Load
    CString nhload;
    NHLOAD* pnhload;
    // Coordinates
    GVECTOR m_nOcoord;
    GVECTOR m_ncoord;
    // Force and moment
    FORCE  m_nforce;
    MOMENT  m_nmoment;
    FORCE  m_nFForce;
    FORCE  m_nINForce;
    MOMENT  m_nFMoment;
    MOMENT  m_nINMoment;
    // Displacements and rotations
    ROTAT  m_nrotat;
    DISP   m_ndisp;
    DISP   m_nSdisp;
    ROTAT  m_nINIrotat;
    DISP   m_nINIdisp;
    // Iterational values - used for non-linear iterations
    ROTAT  m_nITRrotat;
    DISP   m_nITRdisp;
    // Translational velocity and acceleration
    GVECTOR m_nDVel;
    GVECTOR m_nDAcc;
    // Angular velocity and Acceleration
    GVECTOR m_nRVel;
    GVECTOR m_nRAcc;
/* Functions */
    NODE() {}
    ~NODE() {}
    DECLARE_SERIAL(NODE)
    virtual void Serialize(CArchive& archive);
    NODE& operator = (const NODE &N);
    void GetJloads(double* jloads, double Ctm);
    BOOL IsFixed(int i);
    void GetJrest(double* jrest,double* ijrest, double Ctm);
    void InputMap(int* dof,int* forc);
    void GetMap(int* dof,int* forc);
    void UpdateDisp(VECTOR& V);
    void UpdateDynNode();
    void RecoverNode();
    void OutDisp(ofstream& O,int spc,int wd);
    GVECTOR GetDispCoord(int DType);
    void PutTVECTOR(int DType,TVECTOR &p1);
    void ResolvePointers();
    void CalcNodeDrawExtents();
    double GetNodeVal(int PType,int PDir);
    void TextSerialize(fstream & FIO,int RW) ;
    void SaveNodeResults(ofstream &FOut);
    void LoadNodeResults(ifstream &FIn, int Ref);
};
```

A.6 Material and Section Properties Classes

Material Class

```
class MATERIAL : public CObject
{
/* Variables */
    double Val[20];
    public:
    CString m_Name;
/* Functions */
    MATERIAL() { }
    ~MATERIAL() { }
    DECLARE_SERIAL(MATERIAL)
    virtual void Serialize(CArchive& archive);
    double GetVal(int Lab);
    double& PutVal(int Lab);
    void TextSerialize(fstream & FIO,int RW) ;
};
```

Element Section Properties Class

```
class ELPROP : public CObject
{
/* Variables */
  double Val[20];
  public:
  CString m_Name;
/* Functions */
    ELPROP() { }
    ~ELPROP() { }
    DECLARE_SERIAL(ELPROP)
    virtual void Serialize(CArchive& archive);
    double GetVal(int Lab);
    double& PutVal(int Lab);
    void TextSerialize(fstream & FIO,int RW) ;
};
```

A.7 Classes to Represent Arrays

A Class to Represent an Array of Nodes

```
class CNodeArray : public CObject
{
/* Variables */
    NODE* m_pNode;
    int m_nNodes;
    public:
/* Functions */
    CNodeArray();
    ~CNodeArray();
    int GetNum();
    void PutNodes(int NNums);
    void ClearNodes();
    NODE*  GetNode(int pos) {return m_pNode+pos;}
    DECLARE_SERIAL(CNodeArray)
    void Assign(const CNodeArray &ANode);
    virtual void Serialize(CArchive& archive);
};
```

A Class to Represent an Array of Geometric Vectors

```
class CGVectorArray : public CObject
{
/* Variables */
    GVECTOR* m_pGV;
    int m_nGV;
public:
/* Functions */
    CGVectorArray();
    ~CGVectorArray();
    int GetNum();
    void Initiate(int NNums);
    void Clear();
    GVECTOR*  Get(int pos) {return m_pGV+pos;}
    DECLARE_SERIAL(CGVectorArray)
    virtual void Serialize(CArchive& archive);
};
```

A Class to Represent an Array of Real Numbers

```
class CDoubleArray : public CObject
{
/* Variables */
    double* m_pd;
    int m_nd;
    public:
/* Functions */
    CDoubleArray();
    ~CDoubleArray();
    int GetNum();
    void Initiate(int NNums);
    void Clear();
    double  Get(int pos) {return *(m_pd+pos);}
    double& Put(int pos) {return *(m_pd+pos);}
    DECLARE_SERIAL(CDoubleArray)
    // Override the Serialize function
    virtual void Serialize(CArchive& archive);
    void Assign(const CDoubleArray &DArray);
};
```

A.8 Classes Representing Structural Elements

Structural Element Class (Abstract Class)

```
class ELEMENT : public CObject
{
/* Variables */
    public:
    // Name and Type
    double ftemp[5];
    int itemp[5];
    CString m_Name;
    int m_EType;
    CStringArray ElNodes;          // Node Names
    NODE* pElNodes[3];             // Node Pointers
    CString ElMat;                 // Material Name
    MATERIAL *pElMat;              // Material Pointer
    CString ElProp;                // Section Property Name
    ELPROP *pElProp;               // Section Property Pointer
    CString ElHload;              // Load History Name
    ELHLOAD *pElHload;            // Load History Pointer
    CSYS ElLocCsys;               // Local Coordinate System
    double m_Length;                    // Element Length
    double m_CurTemp;             // Current Temperature
    int m_nSElem;                 // Number of Internal Elements
    int m_SLoc;                   // Load Start Location
    int m_ELoc;                   // Load End Location
    double MDamp,KDamp;
    public:
/* Functions */
    ELEMENT();
    ~ELEMENT() { }
    DECLARE_SERIAL(ELEMENT)
    virtual void Serialize(CArchive& archive);
    protected:
    virtual void Stiff(MATRIX &K,MATRIX& M,MATRIX &C) { }
    virtual void GlobStiff(MATRIX &K,MATRIX& M,MATRIX &C) { }
    virtual void CalculateGlobalForces(VECTOR &TotalF) { }
    public:
    virtual void Initiate();
    virtual void ResolvePointers();
    virtual void SetELoad(double Ctm) { }
    virtual void PutElemMatrices() { }
    virtual void UpdateElement() { }
    virtual void UpdateDynElem() { }
    virtual void RecoverElem() { }
    virtual void OutELEM() { }
    virtual void DrawElem(int DType) { }
    virtual void CalcElemDrawExtents() { }
    virtual void PrepareDrawElem(int DType) { }
    virtual void OutDisp(ofstream& O) { }
    virtual void OutForc(ofstream& O) { }
    virtual double GetElemVal(int PType,int PDir,int NNum) {return 0;}
    virtual void TextSerialize(fstream & FIO,int RW) ;
    virtual void SaveElmResults(ofstream &FOut) { }
    virtual void LoadElmResults(ifstream &FIn, int Ref) { } };
```

Beam Element Class

```
class BEAM1:public ELEMENT
{
/* Variables */
    public:
```

```
    NODE End1;
    NODE End2;
    CSYS ESys;
    double FinLength;
/* Functions */
    BEAM1();
    ~BEAM1() { }
    DECLARE_SERIAL(BEAM1)
    virtual void Serialize(CArchive& archive);
    protected:
    virtual void Stiff(MATRIX &K,MATRIX& M,MATRIX &C);
    virtual void GlobStiff(MATRIX &K,MATRIX& M,MATRIX &C);
    virtual void CalculateGlobalForces(VECTOR &TotalF);
    void ObtainLocRotTrans(MATRIX &TL1,MATRIX &TL2);
    void UpdateElemCoord(const MATRIX &TL1,const MATRIX &TL2);
    void CalcDeform(VECTOR &U1,VECTOR &U2);
    void CalcForceNL();
    void CalcForceL();
    public:
    virtual void Initiate();
    virtual void ResolvePointers();
    virtual void SetELoad(double Ctm);
    virtual void PutElemMatrices();
    virtual void UpdateElement();
    virtual void UpdateDynElem();
    virtual void RecoverElem();
    virtual void OutELEM() { }
    virtual void DrawElem(int DType);
    virtual void CalcElemDrawExtents();
    virtual void PrepareDrawElem(int DType);
    virtual void OutDisp(ofstream& O);
    virtual void OutForc(ofstream& O);
    virtual double GetElemVal(int PType,int PDir,int NNum);
    virtual void TextSerialize(fstream & FIO,int RW) ;
    virtual void SaveElmResults(ofstream &FOut);
    virtual void LoadElmResults(ifstream &FIn, int Ref);     };
```

Galloping Information Structure

```
typedef struct
{
    int IsGalloping;              // Is Galloping Activated
    int GallopMode;               // Which Galloping Mode
    double HTens;                 // Horizontal Tension
    double GallopWt;              // Weight / unit length
    double Le,Lambda2;            // Mode spahes' parameters
    // Arrays, [0]= In-plane, [1]= Out-of-plane
    double IPD_NatFreq;           // Normalized In-plane Nat. Frequency
    double NatFreq[2];            // Natural Frequencies
    double GallopFreq[2];         // Galloping frequencies
    double IPhase[2];             // Galloping Initial phases
    double GAmp[2];               // Galloping Amplitudes
} SGallopInf;
```

Cable Element Class

```
class CABLE : public ELEMENT
{
/* Variables */
    protected:
    CNodeArray IntNod;                  // Internal Nodes
    CDoubleArray T;                     // Internal Tensions
    CDoubleArray TInd;                  // Internal Tension Indicators
    CGVectorArray ModeRatio;            // Mode Multipliers
    FORCE m_IEnd1,m_FEnd1;
    FORCE m_IEnd2,m_FEnd2;
    // Matrices: 1= Internal DOF, 2=External DOF
    BMATRIX K11,M11,C11;                // Internal DOF
    MATRIX K12,M12,C12;                 // Internal - External DOF
    MATRIX K22,M22,C22;                 // External DOF
    VECTOR DcDiag;                      // Diagonal of Decomposed Matrix
    CSYS GravityCsys;                   // Gravity Coordinates
    int m_Solve;
    public:
    int m_IsStringing;                  // Stringing Information or Original Length
    double m_STension;                  // Stringing Tension
    double m_STemp;                     // Stringing Temperature
    double m_SLoad;                     // Stringing Load
    double m_Span,m_HSpan,m_VSpan;      // Spans, Hz. Vt. and Inclined
    int IntDOF;                         // Internal DOF
    SGallopInf m_GallopInf;             // Galloping Information
    public:
/* Functions */
    CABLE();
    ~CABLE() { }
    DECLARE_SERIAL(CABLE)
    virtual void Serialize(CArchive& archive);
```

```
protected:
virtual void Stiff(MATRIX &K,MATRIX& M,MATRIX &C);
virtual void GlobStiff(MATRIX &K,MATRIX& M,MATRIX &C);
virtual void CalculateGlobalForces(VECTOR &TotalF);
virtual void GetIntForce(VECTOR& IntForce);
virtual void CalcMass();
void CalcDamp();
virtual void CabSolve(VECTOR& EForce);
virtual void CabStiff();
void UpdateIncForceDyn(VECTOR &F);
void UpdateTotalForceDyn(VECTOR &F);
void UpdateIncDyn21(VECTOR &F);
void GenIntNodes();
virtual void CalcIntStiff();
void ObtainElemLoad(VECTOR &F);
void ObtainInertiaForce(VECTOR &FAcc);
void ObtainDampForce(VECTOR &FVel);
void ObtainIncElemLoad(VECTOR &F);
void ObtainInitialIntVectors(VECTOR &I_Disp1,VECTOR &I_Acc1,VECTOR
&I_Vel1);
void ObtainInitialExtVectors(VECTOR &I_Disp2,VECTOR &I_Acc2,VECTOR
&I_Vel2);
void ResolveInternalPointers();
void CalculateNatFreq();
void CalculateModeDisp();
void CalcStaticDisp(double sld);
void CalcCatFunc(VECTOR& ZF,MATRIX &Jac,double a_Wt,
                        double a_L0,double a_H,double a_V,int a_mode);
void CalcHVFromL(double a_L0,double a_Wt,double* a_HV);
double CalcLFromH(double a_H,double a_Wt);
void CalcGalopDisp(double Ctm);
virtual void CabGallop();
void CreateInternalMatrices();
void DestroyInternalMatrices();
friend double GallopFreqEq(double a_freq, double* AuxVal, int AuxNum);
friend double AppHEq(double a_H, double* AuxVal, int AuxNum);
public:
virtual void Initiate();
virtual void ResolvePointers();
virtual void SetELoad(double Ctm);
virtual void PutElemMatrices();
virtual void UpdateElement();
virtual void UpdateDynElem();
```

```
    virtual void RecoverElem();
    virtual void OutELEM() { }
    virtual void DrawElem(int DType);
    virtual void PrepareDrawElem(int DType);
    virtual void CalcElemDrawExtents();
    virtual void OutDisp(ofstream& O);
    virtual void OutForc(ofstream& O);
    virtual double GetElemVal(int PType,int PDir,int NNum);
    virtual void TextSerialize(fstream & FIO,int RW) ;
    virtual void SaveElmResults(ofstream &FOut);
    virtual void LoadElmResults(ifstream &FIn, int Ref);
};
```

Two-Nodes Cable Class

```
class CABLE2 : public CABLE
{
/* Functions */
    public:
    CABLE2();
    ~CABLE2() { }
    DECLARE_SERIAL(CABLE2)
    virtual void Serialize(CArchive& archive);
    protected:
    virtual void CalcMass();
    virtual void CabSolve(VECTOR& EForce);
    virtual void CabGallop();
    virtual void CabStiff();
    virtual void CalcIntStiff();
    virtual void SetELoad(double Ctm);
    virtual void DrawElem(int DType);
    virtual void PrepareDrawElem(int DType);
    virtual void CalcElemDrawExtents();
};
```

Cable Type Class (Future Extension)

```
class CABLE3 : public CABLE
{
/* Functions */
    public:
    CABLE3();
    ~CABLE3() { }
    DECLARE_SERIAL(CABLE3)
    virtual void Serialize(CArchive& archive);
    protected:
    virtual void CalcMass();
    virtual void CabSolve(VECTOR& EForce);
    virtual void CabGallop();
    virtual void CabStiff();
    virtual void CalcIntStiff();
    virtual void SetELoad(double Ctm);
    virtual void DrawElem(int DType);
    virtual void PrepareDrawElem(int DType);
    virtual void CalcElemDrawExtents();
};
```

TRUSS Element Class

```
class TRUSS : public ELEMENT
{
/* Variables */
    double T;           // Tension
    public:
    int m_TOnly;
/* Functions */
    TRUSS();
    ~TRUSS() { }
    DECLARE_SERIAL(TRUSS)
    virtual void Serialize(CArchive& archive);
    protected:
    virtual void Stiff(MATRIX &K,MATRIX& M,MATRIX &C);
    virtual void GlobStiff(MATRIX &K,MATRIX& M,MATRIX &C);
    virtual void CalculateGlobalForces(VECTOR &TotalF);
    public:
    virtual void Initiate();
    virtual void ResolvePointers();
    virtual void SetELoad(double Ctm);
    virtual void PutElemMatrices();
    virtual void UpdateElement();
    virtual void UpdateDynElem() { }
    virtual void RecoverElem() { }
    virtual void OutELEM() { }
    virtual void DrawElem(int DType);
    virtual void PrepareDrawElem(int DType);
    virtual void CalcElemDrawExtents();
    virtual void OutDisp(ofstream& O);
    virtual void OutForc(ofstream& O);
    virtual double GetElemVal(int PType,int PDir,int NNum);
    virtual void TextSerialize(fstream & FIO,int RW) ;
    virtual void SaveElmResults(ofstream &FOut);
    virtual void LoadElmResults(ifstream &FIn, int Ref);
};
```

Spring Element Class

```
class RSPRING : public ELEMENT
{
/* Variables */
    public:
    double m_Stiff;
    int m_RIter;
    double T;
    int m_DIRECTION;
/* Functions */
    RSPRING();
    ~RSPRING() {}
    DECLARE_SERIAL(RSPRING)
    virtual void Serialize(CArchive& archive);
    protected:
    virtual void Stiff(MATRIX &K,MATRIX& M,MATRIX &C);
    virtual void GlobStiff(MATRIX &K,MATRIX& M,MATRIX &C);
    virtual void CalculateGlobalForces(VECTOR &TotalF);
    public:
    virtual void Initiate();
    virtual void ResolvePointers();
    virtual void SetELoad(double Ctm);
    virtual void PutElemMatrices();
    virtual void UpdateElement();
    virtual void UpdateDynElem() {}
    virtual void RecoverElem() {}
    virtual void OutELEM() {}
    virtual void DrawElem(int DType);
    virtual void PrepareDrawElem(int DType);
    virtual void CalcElemDrawExtents();
    virtual void OutDisp(ofstream& O);
    virtual void OutForc(ofstream& O);
    virtual double GetElemVal(int PType,int PDir,int NNum) {return 0;}
    virtual void TextSerialize(fstream & FIO,int RW) ;
    virtual void SaveElmResults(ofstream &FOut);
    virtual void LoadElmResults(ifstream &FIn, int Ref);
};
```

A.9 A Class Representing the Global Structure, CDyntrnDoc

Main Structure Class

```
class CDyntrnDoc : public CDocument
{
    public:
    CDyntrnDoc();
    ~CDyntrnDoc();
     DECLARE_SERIAL(CDyntrnDoc)
    public:
    CString m_Name;
    OUTLIST m_ResList;
    ROOpt m_OutOption;
    RGOpt m_GOption;
    int m_OutCounter;
    protected:
    ofstream out;
    public:
    ofstream dbout;
    ofstream outD1;
    ofstream outD2;
    UINT SolMsg;
    UINT DrawMsg;
    UINT GraphMsg;
    public:
    // Dynamic Parameters
    BOOL Dynamic;
    BOOL NonLinear;
    double Delta_t;
    double Alpha;    // Newmark parameters Alpha and Delta
    double Beta;
    double Delta;
    double ITime;
    double CTime;
    double FTime;
    double CTol;
    double m_CabTol;
double Gravity;
double IntCoeff[6]; // Dynamic Coefficients
public:
CSYS *pglob;
ELLOAD *pel;
public:
int NNodes;
// Object Lists
CMapStringToOb Materials;
CMapStringToOb ElemProperties;
CMapStringToOb Nodes;
CMapStringToOb Elements;
CMapStringToOb Elloads;
CMapStringToOb Nloads;
// Temporary objects to carry time steps until the load transactions are confirmed
ELHLOAD tmpehld;
NHLOAD tmpnhld;
// Drawing options
public:
CMetaFileDC *pMFile;
HMETAFILE   HMFile;
float m_DxMax;
float m_DyMax;
float m_DxMin;
float m_DyMin;
double m_DAngX,m_DAngZ;
double m_DAngXInc,m_DAngZInc;
double Sin_AngX,Sin_AngZ;
double Cos_AngX,Cos_AngZ;
double m_DScale;
int m_DWidth;
int m_DHeight;
// Solution Attributes
public:
int NIter;
VECTOR IntForceVector;
VECTOR ResForceVector;
VECTOR EqForceVector;
VECTOR INForceVector;
VECTOR FNForceVector;
VECTOR ElForceVector;
VECTOR ForceVector;
VECTOR RestrVector;
VECTOR TotalVector;
VECTOR SolutionVector;
VECTOR IDispVector;
VECTOR FDispVector;
VECTOR VelVector;
VECTOR AccVector;
SSMATRIX GlobalStiff;
SSMATRIX GlobalMass;
SSMATRIX GlobalDamp;
//Solution Operations
public:
void CreateSolutionVectors(int NNum);
void MapNodes();
void CalcDynamicCoeff();
void BuildElementMatrices();
void CalculateEqStiff();
void CalculateEqForces();
void ApplyIncNodalLoads();
void ApplyNodalRestr();
void SolveStiff(VECTOR &RHVector);
void UpdateStructure();
void UpdateDynamic();
void RecoverSolution();
void ObtainResidual();
void SetElementLoads(double Ctm);
void Solve(void* Ptr,HWND PWnd);
public:
BOOL MaterialLookup(const char *key, MATERIAL*& mat);
BOOL PropertiesLookup(const char *key, ELPROP*& prop);
BOOL NodesLookup(const char *key, NODE*& node);
BOOL ElementsLookup(const char *key, ELEMENT*& element);
BOOL ElloadsLookup(const char *key, ELHLOAD*& elload);
BOOL NloadsLookup(const char *key, NHLOAD*& nload);
virtual void Serialize(CArchive& ar);
virtual void DeleteContents();
// Drawing Functions and output
public:
void OpenMemoryMetaFile();
void CloseMemoryMetaFile();
void CalcDrawExtents();
void PrepareDraw(int DType);
void Draw(HWND PWnd,int DType);
void AdjustView(double AngX,double AngZ);
```

```
    void ClearDraw(HWND Pwnd);
    void OutNodDisp();
    void OutElemRes();
    void OutDynRes(ofstream &OutD);
    void TextExport(fstream & FIO);
    void TextImport(fstream & FIO);
    void SaveResults(ofstream &resfile);
    void LoadResults(ifstream &resfile);
    // Friend Functions belonging to Dialogs
    public:
    friend afx_msg void OnUpdateMatLabel();
    public:
    virtual BOOL  OnNewDocument();
    virtual BOOL  OpenDocument(const char* pszPathName,HWND Pwnd);
    virtual BOOL  OnSaveDocument(const char* pszPathName);
    public:
    void Destroy(); // delete all the document entities
};
```

A.10 Sparse Matrix Classes

<u>A Class to Represent an Element of a Row of a Sparse Matrix</u>

```
class RELEM: public CObject
{
/* Variables */
    public:
    int m_col;
    double m_val;
    RELEM* m_next;
    RELEM* m_prev;
};
```

<u>A ROWHEAD Class for Manging Rows of a Sparse Square Matrix SSMATRIX</u>

```
class ROWHEAD: public CObject
{
/* Variables */
    protected:
    int m_nelem;
    RELEM* m_beg;
    public:
/* Functions */
    ROWHEAD();
    void Destroy();
    ~ROWHEAD();
    int IsEmpty() const;
    RELEM* SearchCol(RELEM*& prev,RELEM*& next,int col) const;
    void Put(const int col,const double val);
    void Mul(const int col,const double val);
    void Add(const int col,const double val);
    void Get(const int col,double& val) const;
    void Del(const int col);
    void Del();
    void AddTo(ROWHEAD &R1,double m1);
    double Mult(const VECTOR &V1);
    RELEM* GetFirst() {return m_beg;}
    RELEM* GetNext(RELEM *elem) { return elem->m_next;}
    RELEM* GetPrev(RELEM *elem) { return elem->m_prev;}
};
```

<u>**A Sparse Square Matrix Class**</u>

```
class SSMATRIX: public CObject
{
/* Variables */
    int m_row;
    ROWHEAD* m_rowhead;
/* Functions */
    public:
    SSMATRIX();
    ~SSMATRIX();
    void Create(int row);
    void Destroy();
    void Clear();
    void PutIJ(const int row,const int col,const double val);
    void AddIJ(const int row,const int col,const double val);
    void SubIJ(const int row,const int col,const double val);
    void MulIJ(const int row,const int col,const double val);
    double  GetIJ(const int row,const int col) const;
    void  DelIJ(const int row,const int col);
    void  DelRow(const int i) { m_rowhead[i].Del();}
    void AddTo(SSMATRIX &SM1,double m1);
    friend VECTOR  operator * (const SSMATRIX &SM1,const VECTOR &V1);
    void CholeskyDecompose(VECTOR& p);
    // a = bx
    void CholeskyBackSub(VECTOR& p,VECTOR& b, VECTOR& x);
};
```

A.11 Classes Performing Output Operations

<u>**Two-Dimensional Vector Class**</u>

```
class TVECTOR
{
/* Variables */
    public:
    float x;
    float y;
/* Functions */
    TVECTOR() {x=0; y=0;}
    void ToPoint(CPoint &p1);
};
```

<u>**A Class to Represent a Defined Output Variable**</u>

```
class OUTVAR : public CObject
{
/* Variables */
    public:
    int m_Type;
    CString m_Name;
    int m_NNum;
    int m_Par;
    int m_Dir;
    public:
/* Functions */
    OUTVAR();
    ~OUTVAR() {}
    DECLARE_SERIAL(OUTVAR)
    virtual void Serialize(CArchive& archive);
};
```

<u>**A Class to Handle Output Storage**</u>

```
class OUTLIST: public CObject
{
/* Variables */
    long m_pos[MAXOUT];
    public:
    int max_index;
    OUTVAR m_var[VMAX];
    CString m_ResName;
    CString m_VarName;
    OUTLIST();
    ~OUTLIST() {}
    DECLARE_SERIAL(OUTLIST)
    float GetOutput(int index);
    void PutOutput();
    virtual void Serialize(CArchive& archive);
    void LoadVariables();
    void StoreCurVariables(float tm);
    void PutTimesInMem(void __far *MemPtr);
};
```

APPENDIX B - DYNAMIC STRUCTURAL EQUATIONS

B.1 General Dynamic Equation

The incremental form of the dynamic equation of motion, from the beginning of a time step, n, to the end of the time step, n+1, can be expressed as :

$$[M]\{\Delta\ddot{U}\}_{n+1}+[C]\{\Delta\dot{U}\}_{n+1}+[K]\{\Delta U\}_{n+1}=\{\Delta F\}_{n+1} \qquad \textbf{B.1}$$

where ,

[M] = mass matrix,

[C] = damping matrix,

[K] = tangent stiffness matrix,

$\{\Delta\ddot{U}\}$= incremental acceleration vector,

$\{\Delta\dot{U}\}$= incremental velocity vector,

$\{\Delta U\}$= incremental deformation vector,

$\{\Delta F\}$= incremental force vector.

The acceleration and velocity at the end of the time step, n+1, can be defined in terms of the acceleration, velocity, and displacement at beginning of the time step, n, using Newmark method [63], as follows:

$$\begin{aligned}\{\ddot{U}\}_{n+1}&=\frac{1}{\alpha(\Delta t)^2}(\{U\}_{n+1}-\{U\}_n)-\frac{1}{\alpha(\Delta t)}\{\dot{U}\}_n-(\frac{1}{2\alpha}-1)\{\ddot{U}\}_n\\ \{\dot{U}\}_{n+1}&=\{\dot{U}\}_n+(\Delta t)(1-\delta)\{\ddot{U}\}_n+\delta(\Delta t)\{\ddot{U}\}_{n+1}\end{aligned} \qquad \textbf{B.2}$$

where,

Δt is the time increment, α and δ are integration parameters

Newmark Equation B.2 can be written in the incremental form as follows:

$$\{\Delta\ddot{U}\}_{n+1} = \frac{1}{\alpha(\Delta t)^2}\{\Delta U\}_{n+1} - \frac{1}{\alpha(\Delta t)}\{\dot{U}\}_n - \frac{1}{2\alpha}\{\ddot{U}\}_n$$
$$\{\Delta\dot{U}\}_{n+1} = \delta(\Delta t)\{\Delta\ddot{U}\}_{n+1} + (\Delta t)\{\ddot{U}\}_n \qquad \textbf{B.3}$$

Equation B.3 can be written as:

$$\{\Delta\ddot{U}\}_{n+1} = a_0\{\Delta U\}_{n+1} - a_2\{\dot{U}\}_n - a_3\{\ddot{U}\}_n$$
$$\{\Delta\dot{U}\}_{n+1} = a_1\{\Delta U\}_{n+1} - a_4\{\dot{U}\}_n - a_5\{\ddot{U}\}_n \qquad \textbf{B.4}$$

where,

$$a_0 = \frac{1}{\alpha(\Delta t)^2}\ ,\ a_1 = \frac{\delta}{\alpha(\Delta t)}\ ,\ a_2 = \frac{1}{\alpha(\Delta t)}\ ,\ a_3 = \frac{1}{2\alpha}\ ,\ a_4 = \frac{\delta}{\alpha}\ ,\ a_5 = (\Delta t)(\frac{\delta}{2\alpha} - 1)$$

By substituting Equation B.4 into B.1 one can obtain the following equation:

$$(a_0[M] + a_1[C] + [K])\ \{\Delta U\}_{n+1} = \{\Delta F\}_{n+1} + (a_2\{\dot{U}\}_n + a_3\{\ddot{U}\}_n)\ [M] + (a_4\{\dot{U}\}_n + a_5\{\ddot{U}\}_n)\ [C] \qquad \textbf{B.5}$$

Equation B.5 Can be written in the form $[K]_{eq}\{\Delta U\}_{n+1} = \{\Delta F_{eq}\}_{n+1}$ where,

$$[K]_{eq} = ([K] + a_0[M] + a_1[C])$$
$$\{\Delta F_{eq}\}_{n+1} = \{\Delta F\}_{n+1} + (a_2\{\dot{U}\}_n + a_3\{\ddot{U}\}_n)\ [M] + (a_4\{\dot{U}\}_n + a_5\{\ddot{U}\}_n)\ [C]$$

B.2 Incremental Newmark Method using Dynamic Condensation

Dynamic condensation is used to eliminate the dynamic degrees of freedom associated with the internal nodes of a structural element. The cable element shown in Figure B.1 is used to illustrate the condensation technique. The cable element consists of 11 nodes, two external nodes designated by the subscript 2 and nine internal nodes referenced by the subscript 1.

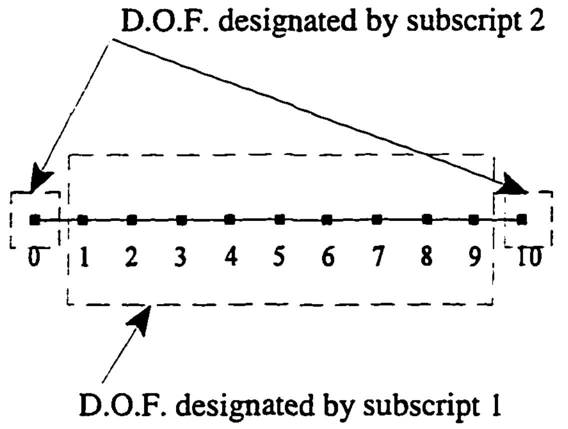

D.O.F. designated by subscript 2
0
1
2
3
4
5
6
7
8
9
10
D.O.F. designated by subscript 1

Figure B.1 Dynamic Matrix Condensation

The purpose of the condensation technique is to eliminate the internal degrees of freedom, DOF of the cable element, so that only the external nodes are included in the analysis. In this way the overall efficiency of the solution can be improved, by decreasing the total degrees of freedom of the structure.

The dynamic equilibrium equations for the internal and external DOF. can be written as follows:

$$\begin{bmatrix} M_{11} & M_{12} \\ M_{21} & M_{22} \end{bmatrix} \begin{Bmatrix} \Delta \ddot{U}_1 \\ \Delta \ddot{U}_2 \end{Bmatrix}_{n+1} + \begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{bmatrix} \begin{Bmatrix} \Delta \dot{U}_1 \\ \Delta \dot{U}_2 \end{Bmatrix}_{n+1} + \begin{bmatrix} K_{11} & K_{12} \\ K_{21} & K_{22} \end{bmatrix} \begin{Bmatrix} \Delta U_1 \\ \Delta U_2 \end{Bmatrix}_{n+1} = \begin{Bmatrix} \Delta F_1 \\ \Delta F_2 \end{Bmatrix}_{n+1} \quad \text{B.5}$$

$$[M_{11}]\{\Delta\ddot{U}_1\}_{n+1}+[M_{12}]\{\Delta\ddot{U}_2\}_{n+1}+[C_{11}]\{\Delta\dot{U}_1\}_{n+1}+[C_{12}]\{\Delta\dot{U}_2\}_{n+1}+ \\ [K_{11}]\{\Delta U_1\}_{n+1}+[K_{12}]\{\Delta U_2\}_{n+1}= \{\Delta F_1\}_{n+1} \qquad \textbf{B.6}$$

$$[M_{22}]\{\Delta\ddot{U}_2\}_{n+1}+[M_{21}]\{\Delta\ddot{U}_1\}_{n+1}+[C_{22}]\{\Delta\dot{U}_2\}_{n+1}+[C_{21}]\{\Delta\dot{U}_1\}_{n+1}+ \\ [K_{21}]\{\Delta U_1\}_{n+1}+[K_{22}]\{\Delta U_2\}_{n+1}= \{\Delta F_2\}_{n+1} \qquad \textbf{B.7}$$

The incremental acceleration and velocity at time t_{n+1} can be eliminated from Equations B.6 and B.7 using Newmark incremental relations derived earlier (see Equation B.4). The resulting equations can be expressed as follows:

$$([K_{11}]+a_0[M_{11}]+a_1[C_{11}])\{\Delta U_1\}_{n+1} + ([K_{12}]+a_0[M_{12}]+a_1[C_{12}])\{\Delta U_2\}_{n+1}= \\ \{\Delta F_1\}_{n+1} + [M_{11}](a_2\{\dot{U}_1\}_n+a_3\{\ddot{U}_1\}_n) + [C_{11}](a_4\{\dot{U}_1\}_n+a_5\{\ddot{U}_1\}_n) \\ + [M_{12}](a_2\{\dot{U}_2\}_n+a_3\{\ddot{U}_2\}_n) + [C_{12}](a_4\{\dot{U}_2\}_n+a_5\{\ddot{U}_2\}_n) \qquad \textbf{B.8}$$

$$([K_{22}]+a_0[M_{22}]+a_1[C_{22}])\{\Delta U_2\}_{n+1} + ([K_{21}]+a_0[M_{21}]+a_1[C_{21}])\{\Delta U_1\}_{n+1}= \\ \{\Delta F_2\}_{n+1} + [M_{22}](a_2\{\dot{U}_2\}_n+a_3\{\ddot{U}_2\}_n) + [C_{22}](a_4\{\dot{U}_2\}_n+a_5\{\ddot{U}_2\}_n) \\ + [M_{21}](a_2\{\dot{U}_1\}_n+a_3\{\ddot{U}_1\}_n) + [C_{21}](a_4\{\dot{U}_1\}_n+a_5\{\ddot{U}_1\}_n) \qquad \textbf{B.9}$$

By substituting Equation B.8 into Equation B.9 , the following equation is developed.

$$[K_{22}]_{eq}\{\Delta U_2\}_{n+1}=\{\Delta F_{2eq}\}_{n+1} \qquad \textbf{B.10}$$

where,

$$[K_{22}]_{eq} = [K_{22}]_D - [K_{21}]_D([K_{11}])_D^{-1}[K_{12}]_D \\ \{\Delta F_2\}_{eq}= \{\Delta F_2\}_D -[K_{21}]_D\ ([K_{11}])_D^{-1}\ \{\Delta F_1\}_D \qquad \textbf{B.11}$$

where,

$$[K_{ij}]_D = a_0[M_{ij}] + a_1[C_{ij}] + [K_{ij}]$$

$$\begin{aligned}\{\Delta F_i\}_D = \{\Delta F_i\} &+ (a_2[M_{ii}]+a_4[C_{ii}])\{\dot{U}_i\} + (a_2[M_{ik}]+a_4[C_{ik}])\{\dot{U}_k\} \\ &+ (a_3[M_{ii}]+a_5[C_{ii}])\{\ddot{U}_i\} + (a_3[M_{ik}]+a_5[C_{ik}])\{\ddot{U}_k\}\end{aligned} \qquad \textbf{B.12}$$

$$i = 1,2 \quad j = 1,2 \quad k = \begin{cases} 2 \ \mathit{if}\ i = 1 \\ 1 \ \mathit{if}\ i = 2 \end{cases}$$

Notice that Equation B.11 is similar to the static condensation equation found in many text books. The only difference is the subscript D, which stands for dynamic, i.e. the dynamic equivalent.

APPENDIX C - TRANSFORMING ROTATION VECTORS

To transform a rotation vector $\{\theta\}_A$ from a coordinate system A to a new coordinate system B, one needs to perform the following steps:

- Assume the rotation $\{\theta\}_A = |\theta_A|\, \hat{e}_A$, where $\hat{e}$ is the unit vector in the direction $\{\theta\}_A$, and $|\theta|$ is the norm of the vector $\{\theta\}_A$.
- Calculate another vector $\{\omega\}_A$, such that $\{\omega\}_A = \tan(|\theta_A|/2) * \hat{e}_A$.
- Calculate a matrix, Ω_A, where, $\Omega_A = \begin{bmatrix} 0 & -\omega_{zA} & \omega_{yA} \\ \omega_{zA} & 0 & -\omega_{xA} \\ -\omega_{yA} & \omega_{xA} & 0 \end{bmatrix}$, *and* $\omega_A = \begin{Bmatrix} \omega_{xA} \\ \omega_{yA} \\ \omega_{zA} \end{Bmatrix}$
- Calculate a rotation matrix, $T_A = I_3 + 2\dfrac{(\Omega_A + \Omega_A^2)}{(1 + |\omega_A|^2)}$, where I_3 is the 3X3 identity

  matrix. T_A represents the rotation matrix for the old coordinate system, A.
- Transform T_A to the new coordinate system B, by calculating $T_B = [E]^t\, T_A\, [E]$, where [E] is the transformation matrix from B to A.
- Using T_B, calculate the matrix Ω_B, and the vector ω_B using the equation:

$$\Omega_B = \frac{(T_B - T_B^t)}{(1 + TRACE(T_B))}, \quad \omega_B = \begin{Bmatrix} \omega_{XB} \\ \omega_{YB} \\ \omega_{ZB} \end{Bmatrix}$$

- Finally, the rotation vector in the new coordinate system, $\{\theta\}_B$, can be calculated as:

$$\{\theta\}_B = 2\, atan(|\omega_B|)\, \hat{e}_B, \text{ where } \hat{e}_B = \frac{\{\omega\}_B}{|\omega_B|}$$

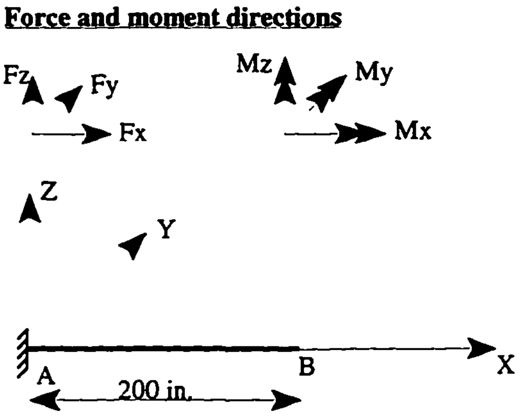
APPENDIX D - ANALYTICAL VERIFICATION EXAMPLES

D.1 Example 1: Verification of the Beam Element

Purpose: To verify the large displacement solution of a 3-D beam element subjected to static concentrated loads

Loads: Concentrated forces and moments applied at point B.

Fx	10,000 lb
Fy	4,000 lb
Fz	-5,000 lb
Mx	1000 lb-in
My	1000 lb-in
Mz	1000 lb-in



Properties:

A	2.0 in^2
Ixx	100 in^4
Iyy	50 in^4
Izz	50 in^4

E	29,000,000 psi
μ	0.3
ρ	0.0007 lb.sec^2/in^4

Results: Internal forces (lb) and moments (lb-in) at point A:

Method	Fx	Fy	Fz	Mx	My	Mz
DYNTRN	10,000	4,000	5,000	999	915500	732600
ANSYS	10,000	4,000	5,000	986	935570	748630

Displacements (in) at point B:

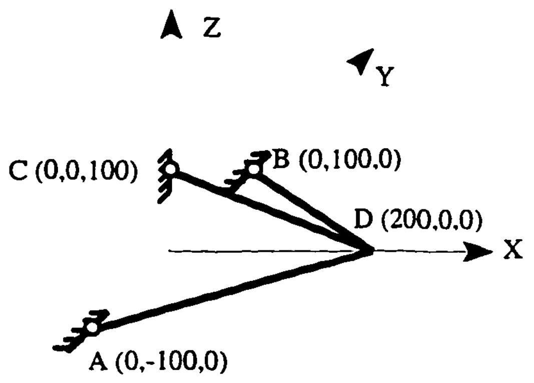
Method	Ux	Uy	Uz	Rotx	Roty	Rotz
DYNTRN	-0.255	6.741	-8.426	0.0018	0.0632	0.0506
ANSYS	-0.249	6.887	-8.6053	0.0017	0.0645	0.0517

D.2 Example 2: Verification of the Truss Element

Purpose: To verify the large displacement solution of a 3-D truss element subjected to static concentrated loads

Loads: Concentrated forces applied at point D.

Fx	100,000 lb
Fy	400,000 lb
Fz	-500,000 lb



Properties:

A	2.0 in^2
E	29E6 psi

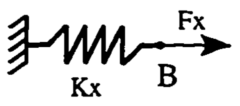
Results:

	Member forces (lb) (positive = tension)			Displacements at D (in)		
Method	AD	BD	CD	Ux	Uy	Uz
DYNTRN	-4.923E4	-9.248E5	1.124E6	-2.598	3.681	-14.082
ANSYS	-4.962E4	-9.261E5	1.126E6	-2.627	3.746	-13.972

D.3 Example 3: Verification of the Spring Element

Purpose: To verify the solution of a linear spring element subjected to static concentrated loads.

Loads: Concentrated forces applied at point B, Fx = 10,000 lb.



Properties: Stiffness, Kx = 1000 lb/in

Results: Displacement, Ux, at B (in)

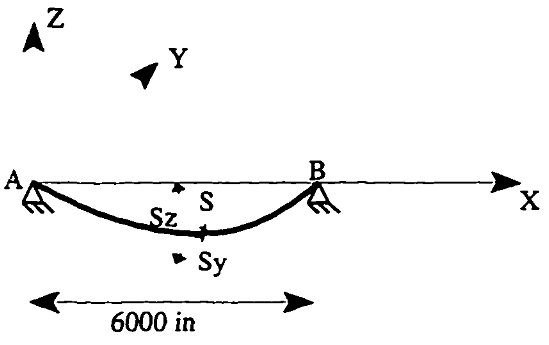
DYNTRN	10
Theory	10

D.4 Example 4: Verification of the Cable Element

Purpose: To verify the large displacement solution of a cable element subjected to uniformly distributed loads.

Loads: Uniformly distributed forces applied on the cable element.

w_y	0.3 lb/in
w_z	0.4 lb/in



Properties:

A	1 in^2
E	10E6 psi
L_0	5999 in

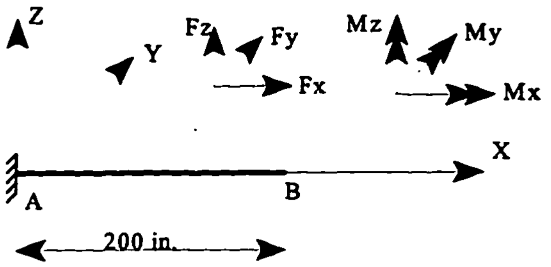
Results: Sags Sz, Sy, and S and cable tension at point A.

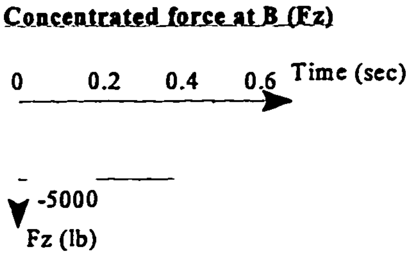
Method	Sags			Tension
	Sz	Sy	S	T
DYNTRN	112.195	84.147	140.24	16,030
Theory [8]	111.77	83.827	139.712	16,160

D.5 Example 5: Verification of the Dynamic Time Integration

Purpose: To verify the large displacement dynamic solution, using time integration.

Loads: Concentrated force, Fz applied at point B as shown in the following figure.



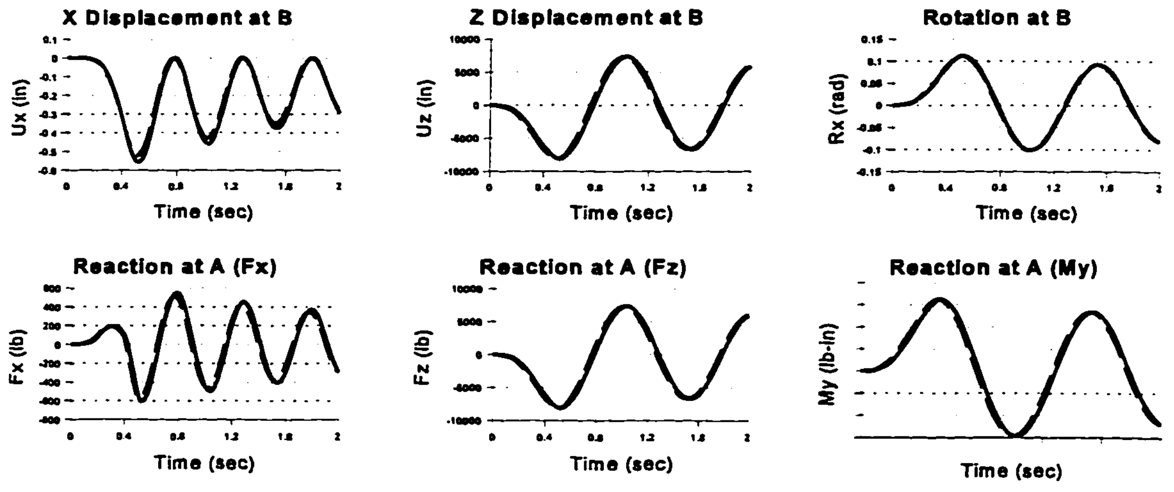


Properties:

A	2.0 in^2
Ixx	100 in^4
Iyy	50 in^4
Izz	50 in^4

E	29,000,000 psi
μ	0.3
ρ	0.07 lb.sec^2/in^4
β	0.01

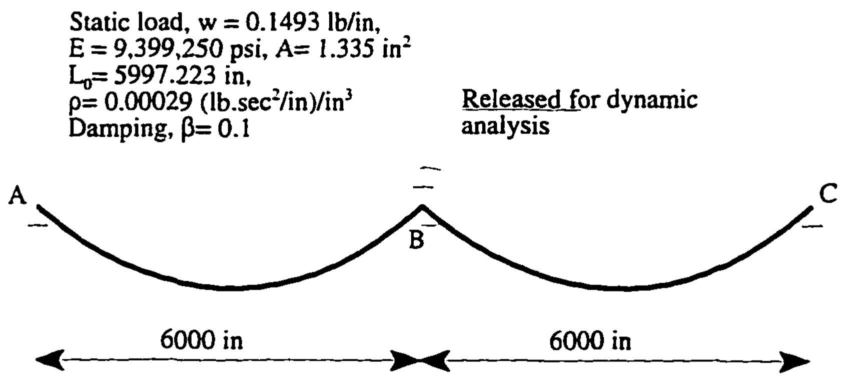
Results:



D.6 Example 6: Verification of the Dynamic Condensation Method

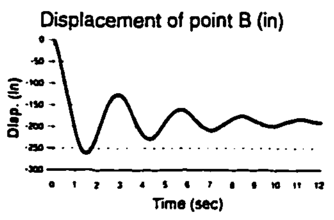
Purpose: To verify the method of dynamic condensation presented in Chapter 3. A two span cable was analyzed due to static load. The intermediate node between the two cables was then released, and the falling cable analyzed using time integration. The problem was solved using DYNTRN, as well as ANSYS. In both cases, 10 axial elements were used to represent each span of the cable. Unlike ANSYS, in the DYNTRN analysis, the dynamic condensation method was used. In order to avoid numerical instabilities generated by the cable structure in ANSYS, the final coordinates of the cable due to static load were input in ANSYS as the initial coordinates. The tension in the static profile of the cable was accounted for in ANSYS using initial strain values.

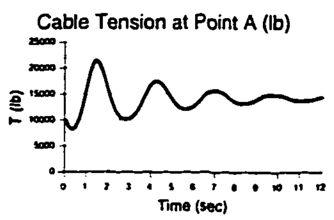
Loads: Uniform load of 0.1493 lb/in.
Released restraint at intermediate node.


Static load, w = 0.1493 lb/in,
E = 9,399,250 psi, A= 1.335 in^2
L_0= 5997.223 in,
ρ= 0.00029 (lb.sec^2/in)/in^3
Damping, β= 0.1
Released for dynamic analysis
A
B
C
6000 in
6000 in

Properties: As shown in Figure.

Results:





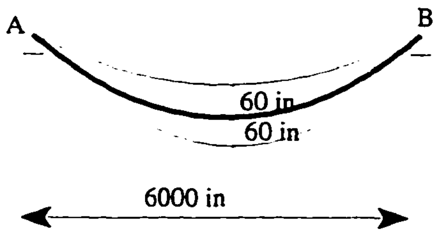
——— DYNTRN ----- ANSYS

D.7 Example 7: Galloping Forces Verification

Purpose: To verify the galloping method explained in Chapter 4 for a single span cable fixed at both ends.
The cable is subjected to a galloping amplitude of 60 inches in the first mode of vibration. The solution is compared to theory. Theoretical solution of the problem is presented in Appendix F.

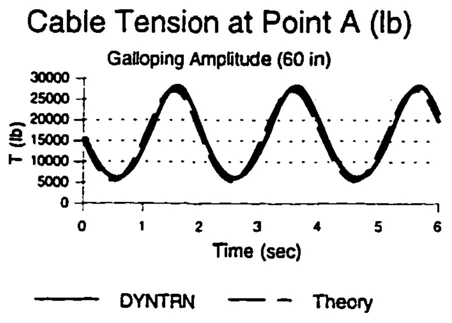
Loads: Uniform static load of 0.5 lb/sec.
Galloping displacement in the shape and frequency of the first symmetric mode, with amplitude of 60 inches.

Static load, w = 0.5 lb/in,
$E = 10^7$ psi, A= 1 in^2
L_0= 6000 in,
ρ= 1.295E-3 (lb.sec^2/in)/in^3



Properties: As shown in Figure.

Results:



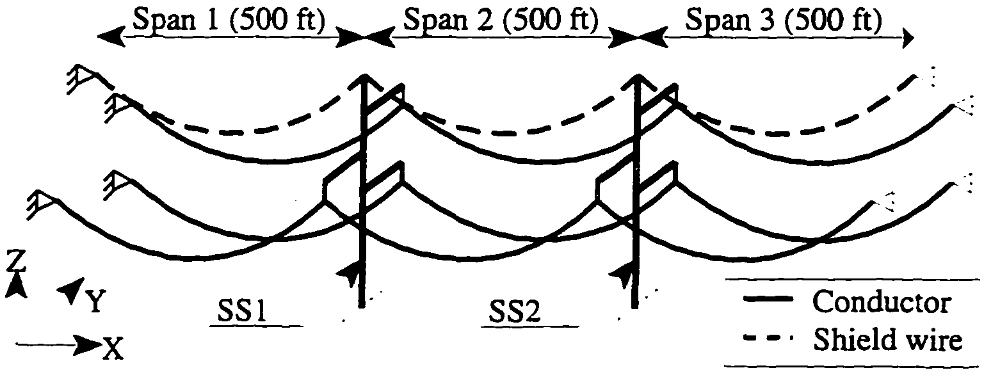
D.8 Example 8: Verification of a Real Transmission Line

Purpose: To verify the results of static and dynamic analyses of a complete transmission line with dimensions and properties comparable to actual transmission lines.
For this purpose, three spans of an actual transmission line described in Reference [3] were analyzed using DYNTRN. The results were checked using ANSYS. In order to avoid numerical instabilities caused by cables in ANSYS, initial coordinates of the cable nodes that are close to the final static profile had to be used as input. The cable un-stretched length was accounted for in ANSYS using initial strains in the axial elements used to represent the cable. Some modifications were made to the data obtained from Reference [3], to make the problem easier to model, such as using a constant cross-section for the poles, instead of a tapered one.

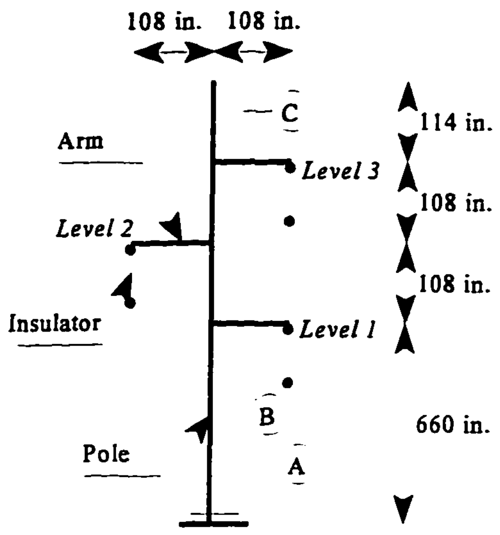
Loads: Static Load:

	Span 1 and Span 2	Span 3
Conductors	Self weight (0.1493 lb/in.)	0.5 in. ice (0.2493 lb/in.)
Shield wires	Self weight (0.0431 lb/in.)	0.25 in. ice (0.0631 lb/in.)

Dynamic load:
The ice on span 3 was suddenly removed, and a dynamic analysis was performed.



Geometry: Two identical support structures, SS1 and SS2, located at the same vertical elevation, as shown in the figure.



Properties:

	Pole	Arm
A	19.625 in.^{2}	4.12 in.^{2}
Iy,Iz	1533.2 in.^{4}	25.24 in.^{4}
Ix	3066.4 in.^{4}	50.48 in.^{4}
E	29E6 psi	29E6 psi
μ	0.3	0.3
ρ	7.32E-4 lb.sec^2/in.^{4}	7.32E-4 lb.sec^2/in.^{4}

	Insulator	Conductor	Shield wire
Type		Lapwing ACSR 45/7	0.5 in. EHS
A	1.0 in.^{2}	1.335 in.^{2}	0.1497 in.^{2}
E	2,900,000 psi	9,399,250 psi	29,000,000 psi
μ	0.3	0.3	0.3
ρ	0.0068 lb.sec^2/in.^{4}	0.00029 lb.sec^2/in.^{4}	0.00075 lb.sec^2/in.^{4}
L_0	84 in.	5997.223 in.	5997.709 in.

Results:

Static Results:

Deformation at Point C, Structure SS2, in global axes						
	Ux	Uy	Uz	Rx	Ry	Rz
DYNTRN	0.965 in.	1.306 in.	-6.89E-3 in.	-2.21E-3 rad	1.42E-3 rad	-4.29E-5 rad
ANSYS	0.963 in.	1.307 in.	-6.89E-3 in.	-2.21E-3 rad	1.42E-3 rad	-4.29E-5 rad

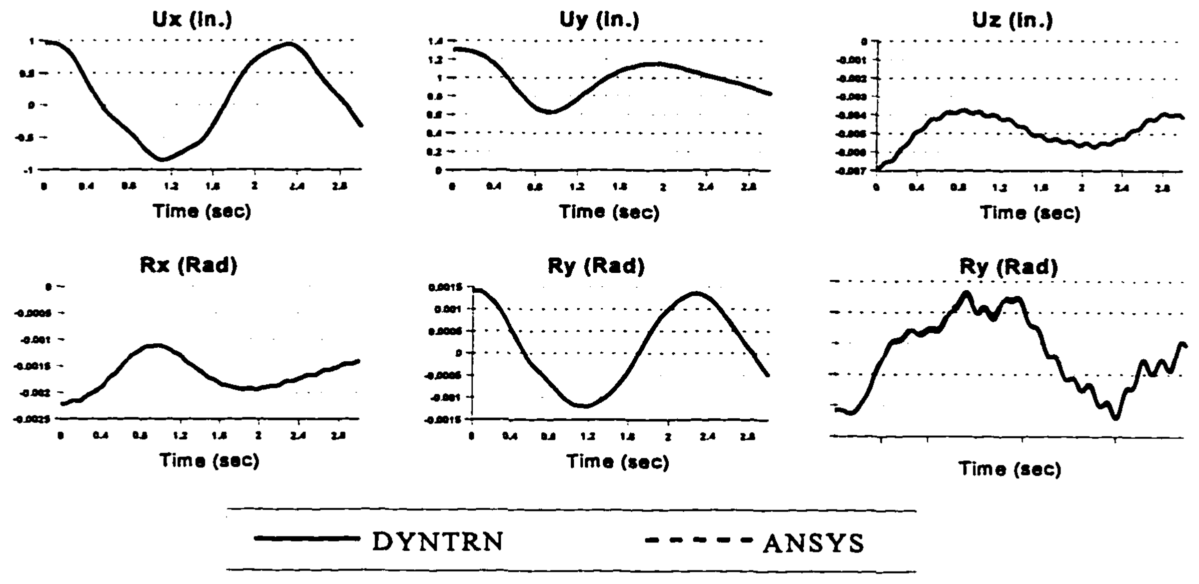
Forces and moments at Section A, Structure SS2, in the global axes						
	Fx	Fy	Fz	Mx	My	Mz
DYNTRN	147.8 lb	-5.798 lb	-3,899 lb	-127,400 lb-in.	137,000 lb-in.	-2,008 lb-in.
ANSYS	147.8 lb	-5.809 lb	-3,901 lb	-127,500 lb-in.	136,660 lb-in.	-2,014 lb-in

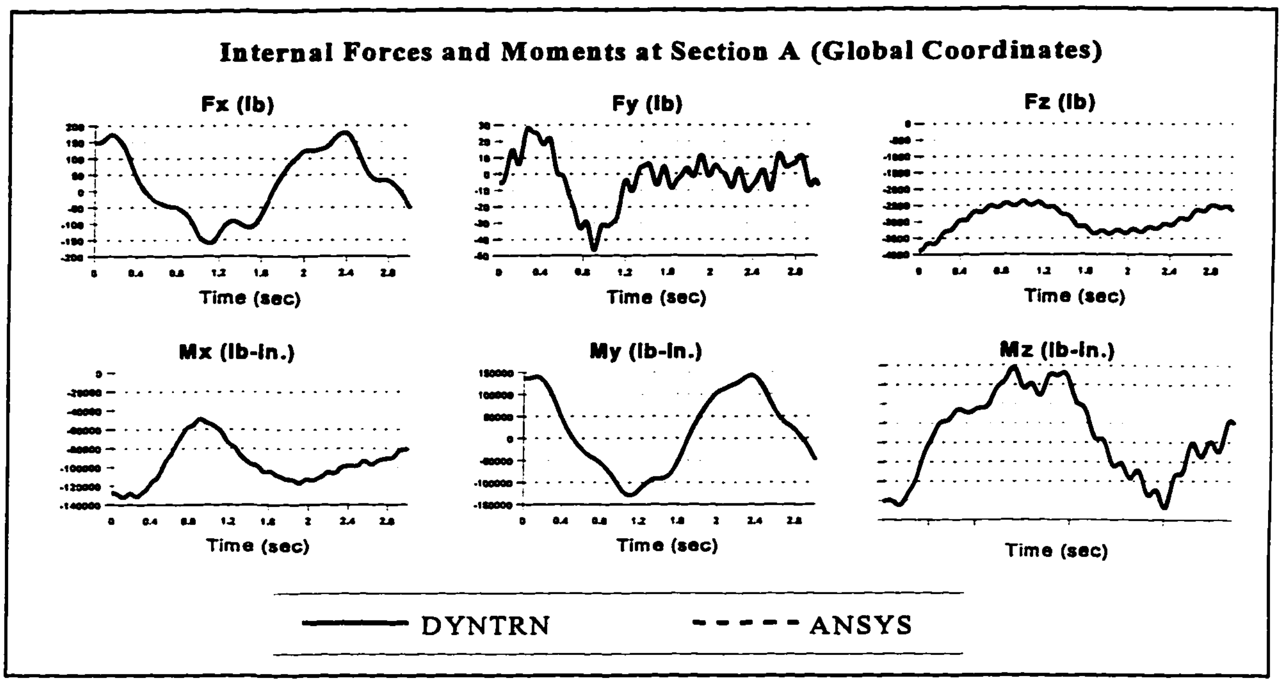
Forces and moments at Section B, Structure SS2, in the global axes						
	Fx	Fy	Fz	Mx	My	Mz
DYNTRN	19.33 lb	-1.218 lb	1,193 lb	-128,900 lb-in.	0.771 lb-in.	-2,087 lb-in.
ANSYS	19.28 lb	-1.221 lb	1,194 lb	-128,940 lb-in.	0.771 lb-in	-2,082 lb-in

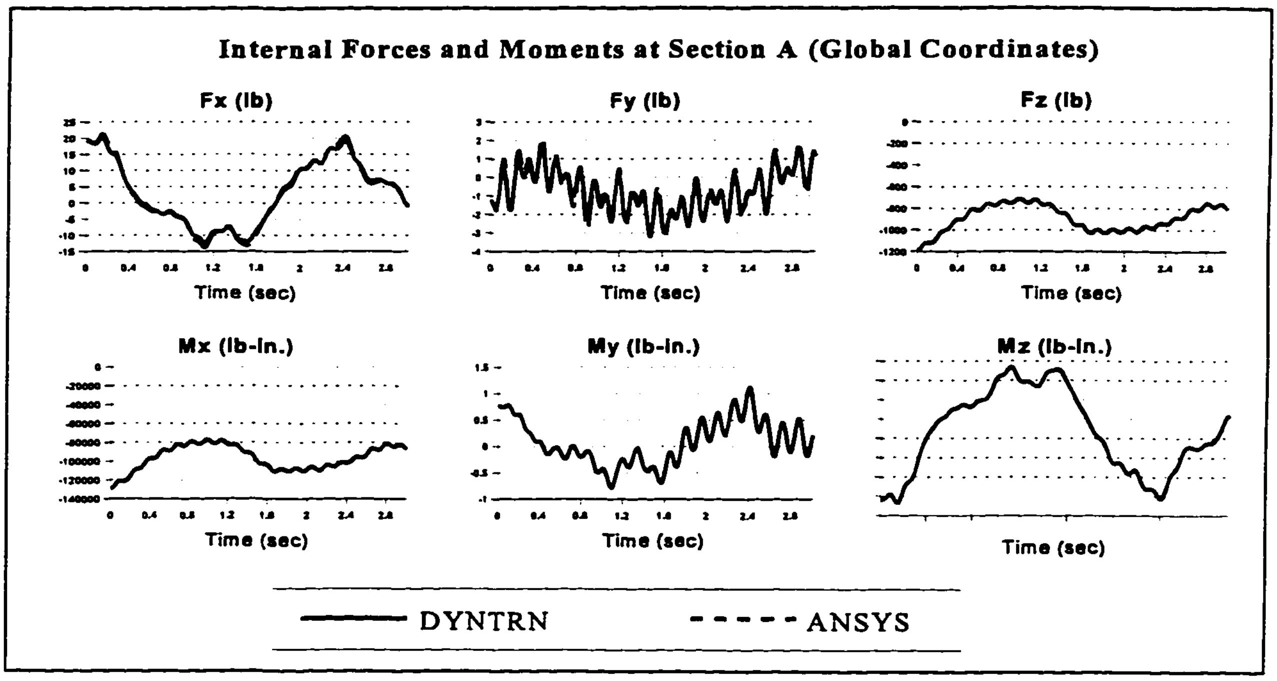
	Conductor end tension in Span 3				Insulator tension at SS2		
	Level 1	Level 2	Level 3	Shield wire	Level 1	Level 2	Level 3
DYNTRN	11,170 lb	11,170 lb	11,190 lb	3,311 lb	1,193 lb	1,194 lb	1,193 lb
ANSYS	11,193 lb	11,192 lb	11,191 lb	3,315 lb	1,194 lb	1,195 lb	1,194 lb

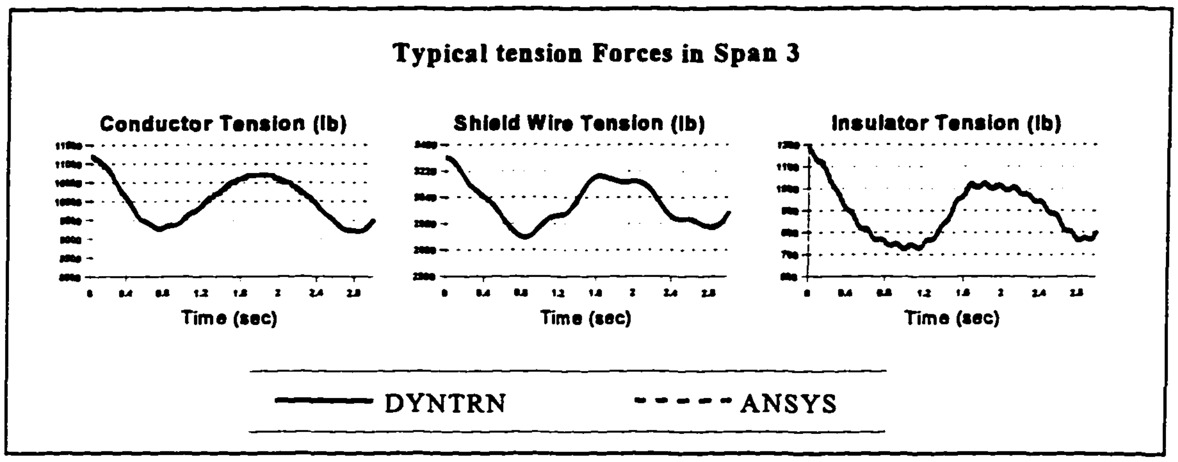
Dynamic Results:

Displacements and Rotations at Point C (Global Coordinates)




Internal Forces and Moments at Section A (Global Coordinates)
Fx (lb)
Fy (lb)
Fz (lb)
Mx (lb-in.)
My (lb-in.)
Mz (lb-in.)
Time (sec)
DYNTRN
ANSYS


Internal Forces and Moments at Section A (Global Coordinates)
Fx (lb)
Fy (lb)
Fz (lb)
Mx (lb-in.)
My (lb-in.)
Mz (lb-in.)
Time (sec)
DYNTRN
ANSYS


Typical tension Forces in Span 3
Conductor Tension (lb)
Shield Wire Tension (lb)
Insulator Tension (lb)
Time (sec)
Time (sec)
Time (sec)
DYNTRN
ANSYS

APPENDIX E - TENSION FORCES DUE TO A GALLOPING CONDUCTOR: THEORETICAL DEVELOPMENT

Consider a conductor subjected to a general dynamic displacement, as shown in Figure E.1. The conductor is fixed at both ends, with a horizontal span, l. Both ends of the cable are assumed at the same level. Studying a small portion of the cable, AB, the length of this portion due to the dynamic displacement is dP_d as shown in Figure E.1. According to the principle of elasticity, the tension due to the dynamic displacement can be calculated as:

$$T_d(S) = E\,A \left(\left. \frac{dP_d}{dS} \right|_S - 1 \right) \qquad \textbf{E.1}$$

where,

E = modulus of elasticity of the cable,

A = cross sectional area of the cable,

$T_d(S)$ = tension at distance S from the origin, due to the dynamic displacement,

S = distance of the portion AB from the origin, measured along the un-stretched length of the cable,

dS = un-stretched length of the portion AB of the cable.

Using the relation between dP_d, dx, du, dz and dw shown in Figure E.1, Equation E.1 can be written as:

$$T_d(S) = E\,A \left(\left. \sqrt{\left(\frac{dx}{dS} + \frac{du}{dS} \right)^2 + \left(\frac{dz}{dS} + \frac{dw}{dS} \right)^2} \right|_S - 1 \right) \qquad \textbf{E.2}$$

Where x and z are the static coordinates of the cable in the direction of the X and Z axes, respectively. u and w are the dynamic displacements in the direction X and Z, respectively.

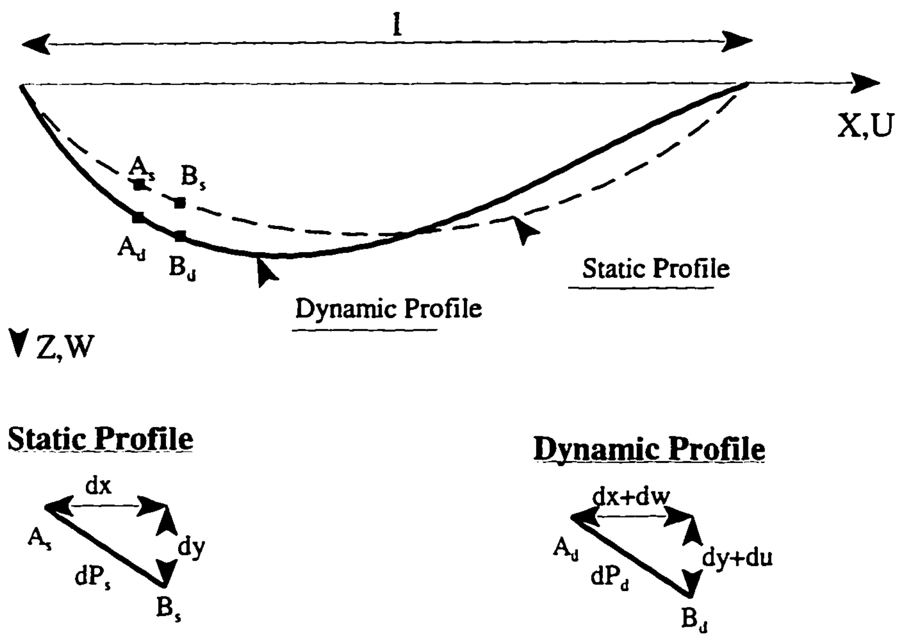


Figure E.1 A conductor subjected to a general dynamic displacement

Equation E.2 can be written as:

$$T_d(S) = E\,A\left(\sqrt{\left(\frac{dx}{dS} + \frac{du}{dx}\frac{dx}{dS}\right)^2 + \left(\frac{dz}{dS} + \frac{dw}{dx}\frac{dx}{dS}\right)^2}\,\Bigg|_S - 1\right) \qquad \text{E.3}$$

For a conductor galloping in one of the mode shapes of the cable, the quantities in Equation E.3 can be calculated as discussed next.

The dynamic displacements, u and v, for a galloping conductor can be calculated as:

$$\begin{aligned} u(x) &= A_g\ M_h(x)\ \sin(\omega t) \\ w(x) &= A_g\ M_v(x)\ \sin(\omega t) \end{aligned} \qquad \text{E.4}$$

where,

A_g = galloping amplitude,

$M_h(x)$ = normalized mode shape in the longitudinal, X, direction,

$M_v(x)$ = normalized mode shape in the vertical, Z, direction,

ω = frequency of the galloping motion in radian/seconds,

t = time in seconds.

The equations used to calculate the mode shapes, $M_h(x)$ and $M_v(x)$, are listed in Chapter 2. The equations are normalized, so that the maximum value of the resulting mode shape is equal to one.

Using Equations E.4, the quantities, du/dx and dw/dx can be calculated, by differentiating the equations with respect to the variable x. Both du/dx and dw/dx are evaluated at a value of x = x(S), calculated as follows [8],

$$x(S) = \frac{HS}{EA} + \frac{H}{mg}\left(\operatorname{asinh}\left(\frac{mgL_0}{2H}\right) - \operatorname{asinh}\left(\frac{mgL_0}{2H} - \frac{mgS}{H}\right)\right) \qquad \text{E.5}$$

where,

H = static horizontal tension in the conductor,

m = mass per unit length of the loaded conductor,

g = acceleration of gravity,

L_0 = un-stretched length of the conductor.

Finally, the quantities, dx/dS and dz/dS, can be calculated as [8]:

$$\frac{dx}{dS} = \frac{H}{EA} + \frac{H}{\sqrt{H^2 + \left(\frac{mgL_0}{2} - mgS\right)^2}}$$

$$\frac{dz}{dS} = \frac{mgL_0}{EA}\left(0.5 - \frac{S}{2L_0}\right) - \frac{0.5mgS}{EA} + \frac{1}{\sqrt{H^2 + \left(\frac{mgL_0}{2} - mgS\right)^2}}\left(\frac{mgL_0}{2} - mgS\right) \qquad \text{E.6}$$

Using Equations E.4, E.5, E.6, Equation E.3 can be evaluated for any time t. In this way, the tension in the galloping conductor can be calculated.

The tension in the conductor due to static load alone can be calculated as [8]:

$$T_s(S) = \sqrt{H^2 + \left(\frac{mgL_0}{2} - mgS \right)^2} \qquad \textbf{E.7}$$

APPENDIX F: EXPERIMENTAL DATA

This appendix contains information about the experimental data used to verify DYNTRN as explained in Chapter 4. The data presented in this appendix is obtained using direct scans from References [1,2,3].

F.1 Experimental Data for the Broken Insulator/ Broken Conductor Analysis [1]

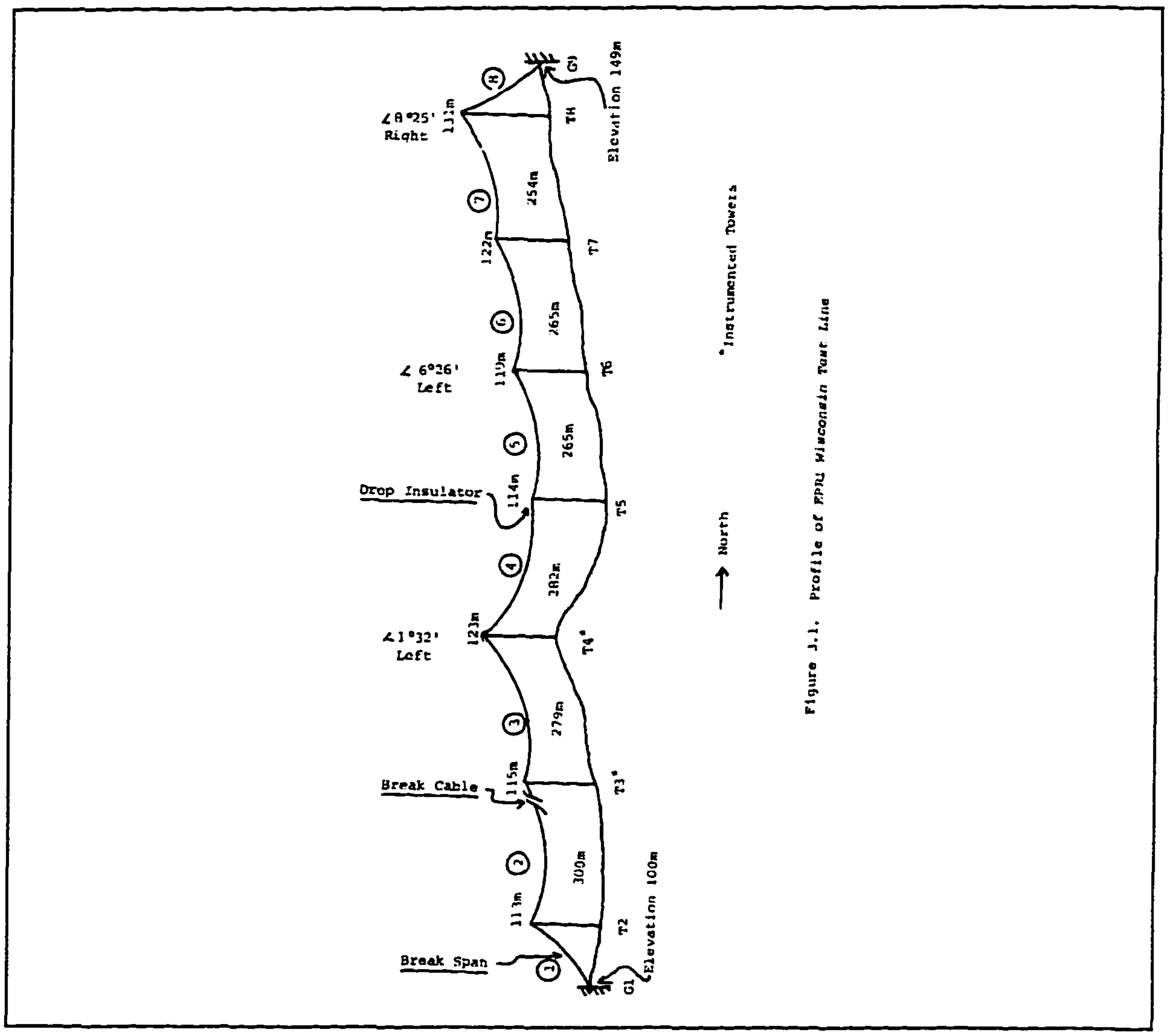
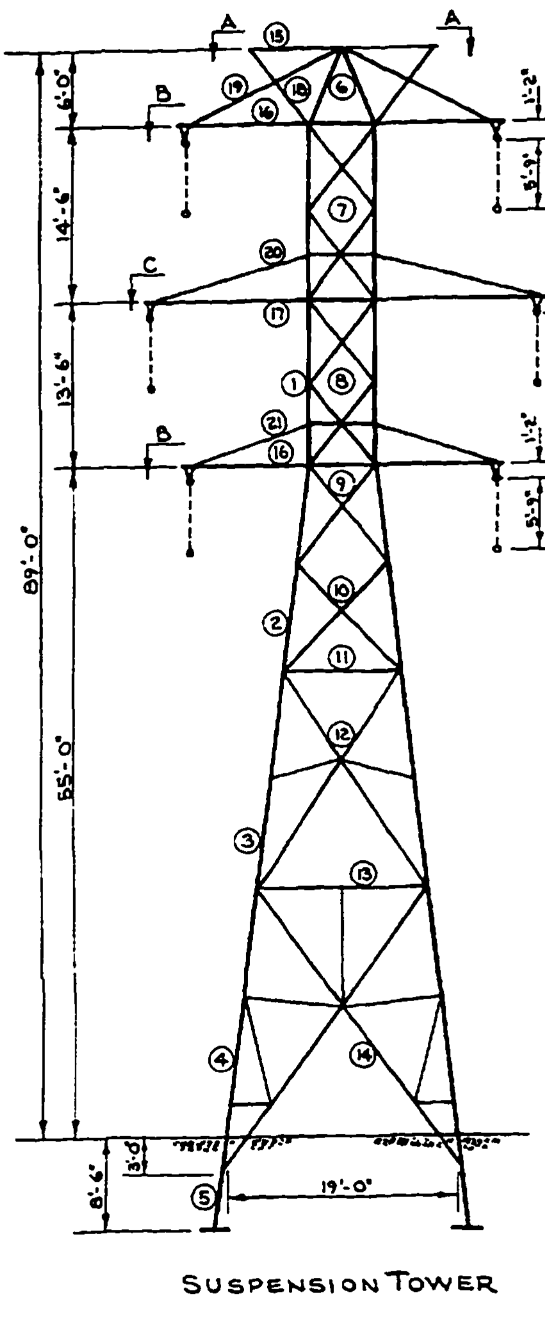
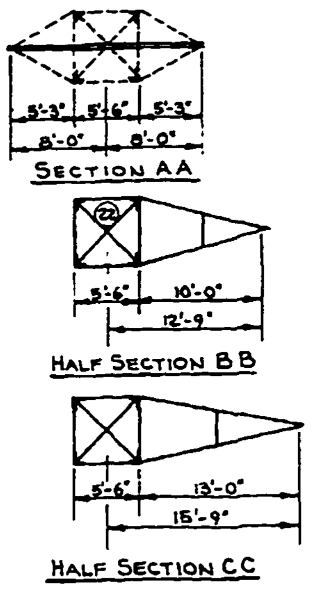


Figure F.1 Transmission Line Tested in Reference [1]





Member	Section			
1	L 3-1/2	x 3-1/2	x 1/4	
2	L 5	x 5	x 5/16	
3		do		
4		do		
5		do		
6	L 2-1/2	x 2-1/2	x 3/16	
7	L 2-1/2	x 2	x 3/16	
8	L 2-1/2	x 2-1/2	x 3/16	
9		do		
10		do		
11	L 3	x 3	x 3/16	
12	L 1-1/2	x 1-1/2	x 3/16	
13	L 3	x 3	x 3/16	
14	L 1-1/2	x 1-1/2	x 3/16	
15	2 Ls3	x 2-1/2	x 3/16	
16	L 3-1/2	x 3-1/2	x 3/16	
17	L 4	x 3	x 1/4	
18	L 3	x 3	x 3/16	
19	L 2	x 2	x 3/16	
20	L 2	x 2	x 3/16	
21	L 2	x 2	x 3/16	

Figure 3-2. Typical Tangent Tower
(Towers T2 through T8)

Figure F.2 Support Structure used in the Broken Conductor/Insulator Analysis [1]

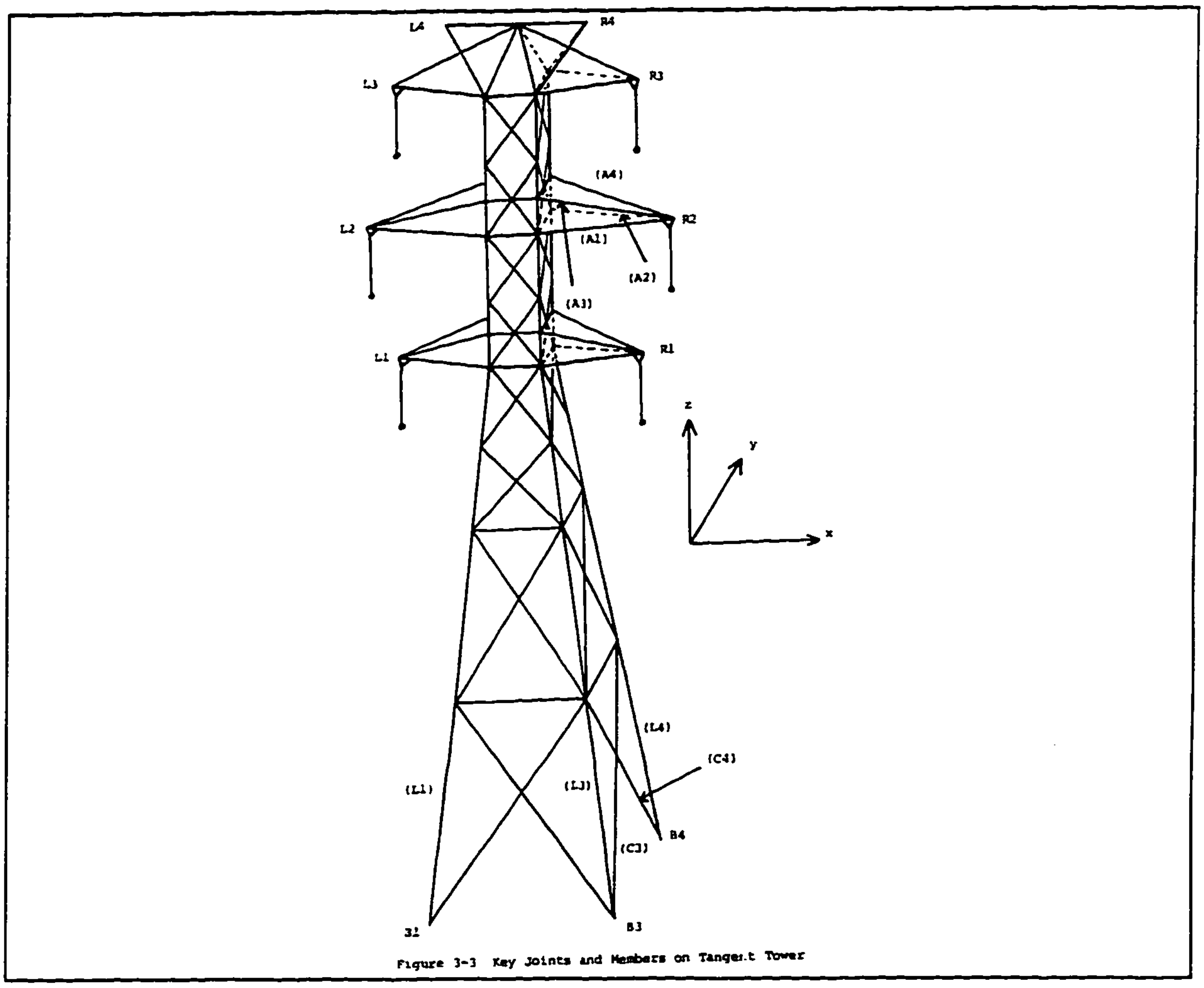
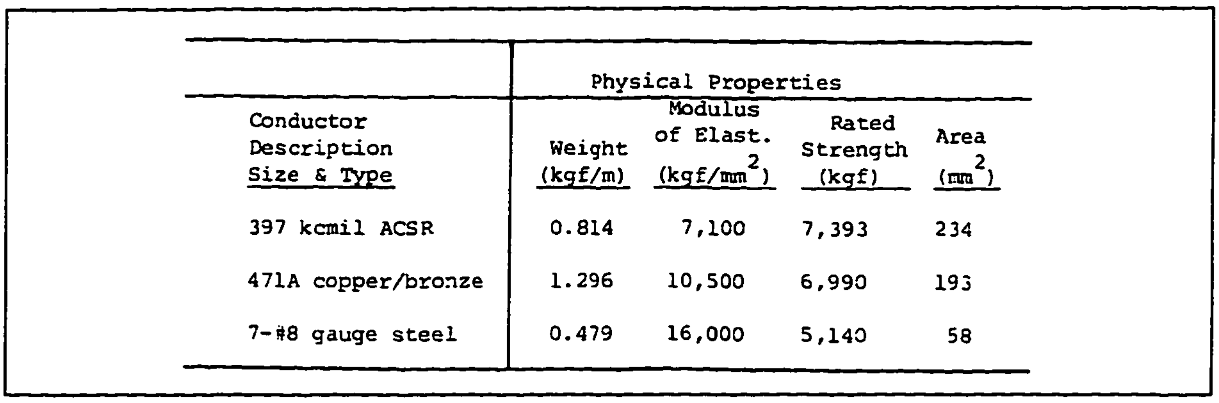


Figure F.3 3-D View of the Support Structures in Reference [1]



	Physical Properties			
Conductor Description Size & Type	Weight (kgf/m)	Modulus of Elast. (kgf/mm^2)	Rated Strength (kgf)	Area (mm^2)
397 kcmil ACSR	0.814	7,100	7,393	234
471A copper/bronze	1.296	10,500	6,990	193
7-#8 gauge steel	0.479	16,000	5,140	58

Figure F.4 Conductor and Shield Wire Properties [1]

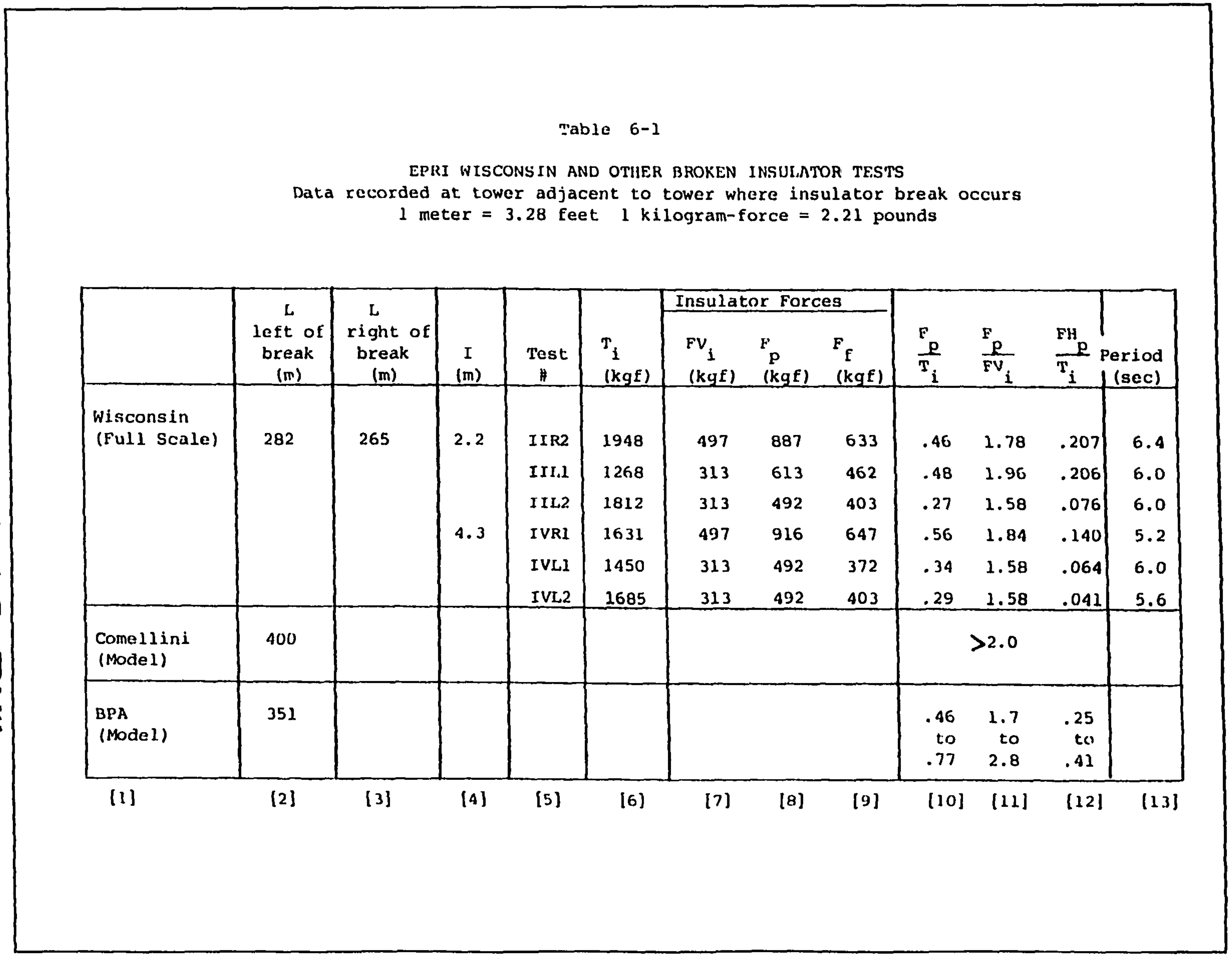
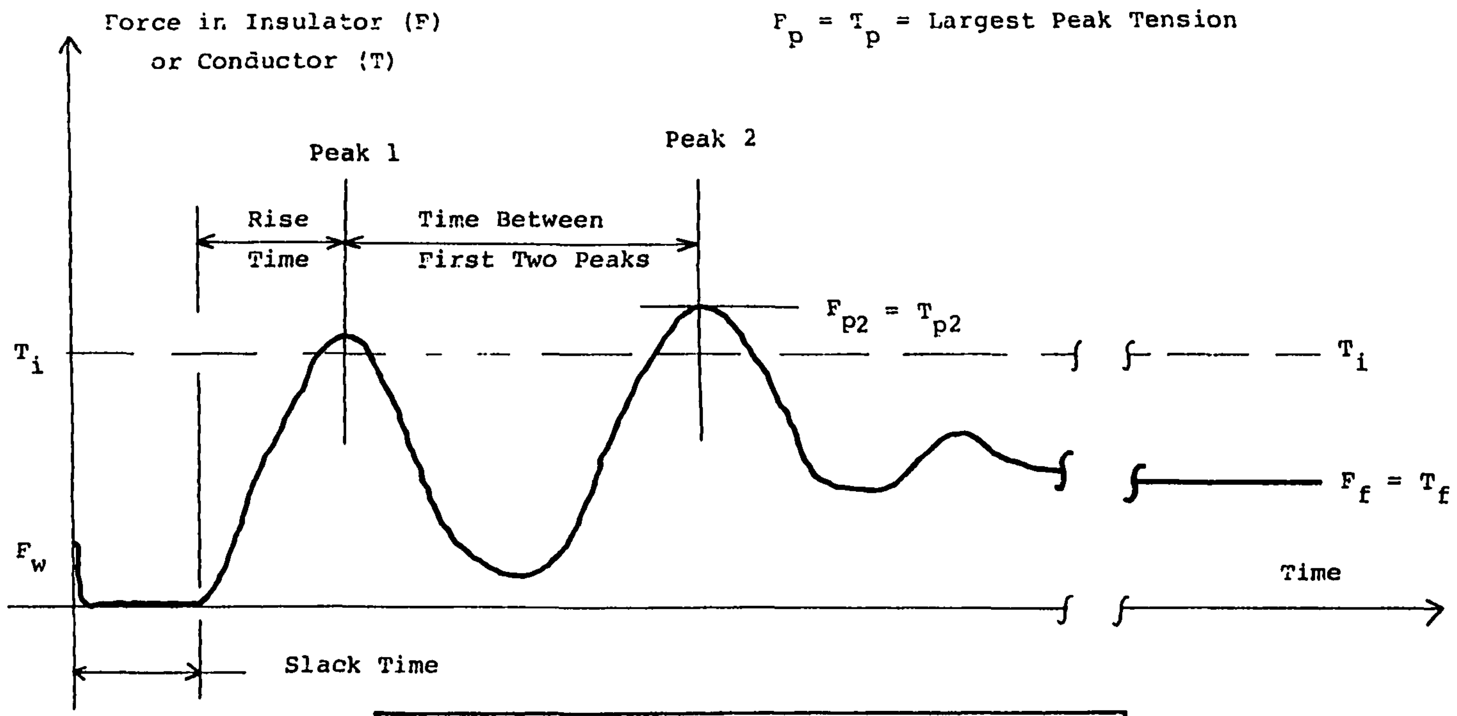


Table 6-1

EPRI WISCONSIN AND OTHER BROKEN INSULATOR TESTS
Data recorded at tower adjacent to tower where insulator break occurs
1 meter = 3.28 feet 1 kilogram-force = 2.21 pounds

	L left of break (m)	L right of break (m)	I (m)	Test #	T_i (kgf)	Insulator Forces FV_i (kgf)	F_p (kgf)	F_f (kgf)	$\frac{F_p}{T_i}$	$\frac{F_p}{FV_i}$	$\frac{FH_p}{T_i}$	Period (sec)
Wisconsin (Full Scale)	282	265	2.2	IIR2	1948	497	887	633	.46	1.78	.207	6.4
				IIL1	1268	313	613	462	.48	1.96	.206	6.0
				IIL2	1812	313	492	403	.27	1.58	.076	6.0
			4.3	IVR1	1631	497	916	647	.56	1.84	.140	5.2
				IVL1	1450	313	492	372	.34	1.58	.064	6.0
				IVL2	1685	313	492	403	.29	1.58	.041	5.6
Comellini (Model)	400									>2.0		
BPA (Model)	351								.46 to .77	1.7 to 2.8	.25 to .41	
[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]	[11]	[12]	[13]

Figure F.5 Broken Insulator Results Test IIR2 [1]



Definitions of Impact Factors	$IFI(T) = \frac{T_p}{T_i}$	$IFF(T) = \frac{T_p}{T_f}$

Figure 6.3. Broken conductor phenomenon.
Thick line shows tension variation in insulator.

Figure F.6 Typical Broken Conductor Response [1]

Table 6-2

EPRI WISCONSIN BROKEN CONDUCTOR TESTS - CONDUCTOR TENSION IN SPAN 3 (NEXT TO BREAK)
1 kilogram-force = 2.21 pounds 1 meter = 3.28 feet

Test #	I (m)	T_i (kgf)	T_{p1}	T_{p2}	T_{p3}	T_f (kgf)	IFF(T)			IFI(T)			Slack time	Time to 1st peak	Time to 2nd peak
							Data	Govers	Energy	Data	Govers	Borges			
IIIR1	2.2	1903	2510	3249*	2067	1122	2.89	3.35	3.04	1.71	1.42	1.70	.27	.50	1.46
IIIR2	2.2	1948	2510	3544*	2659	1123	3.15	3.36	3.05	1.82	1.41	1.70	.27	.50	1.50
IIIR3	2.2						Broken arm								
IIIL1	2.2	1268	1525	2067*	1181	710	2.91	3.40	3.09	1.63	1.39	1.65	.30	.57	1.28
IIIL2	2.2	1812	2067	2510*	1743	886	2.83	>3.6	3.34	1.38	<1.35	1.35	.27	.47	1.28
IIIL3	2.2	2174	2510*	2215	2067	951	2.64	>3.6	3.40	1.15	<1.35	1.15	.21	.39	1.20
VR1	4.3	1631	2749	3044*		770	3.95	4.02	3.12	1.87	1.60	-	.42	.87	1.68
VR2	4.3	1857	2584	2289	2953*	811	3.64	4.19	3.22	1.59	1.48	-	.47	.80	1.63
VR3	4.3						Broken arm								
VL1	4.3	1450	1477*	1329	738	521	2.83	>4.6	3.40	1.02	<1.38	-	.48	.78	1.59
VL2	4.3	1685		*		544	No load cell data					-	.50	.80	1.70
VL3	3.7	2401	2584	2655*		738	3.60	>4.6	3.87	1.11	<1.38	-	.25	.57	1.40
[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]	[11]	[12]	[13]	[14]	[15]	[16]

* indicates largest peak force
Note: Symbols > or < in columns [9] and [12] mean "larger than" or "less than". Exact values could not be determined from Gover's charts.

Figure F.7 Force in Insulator next to the Broken Conductor - Test IIIR2 [1]

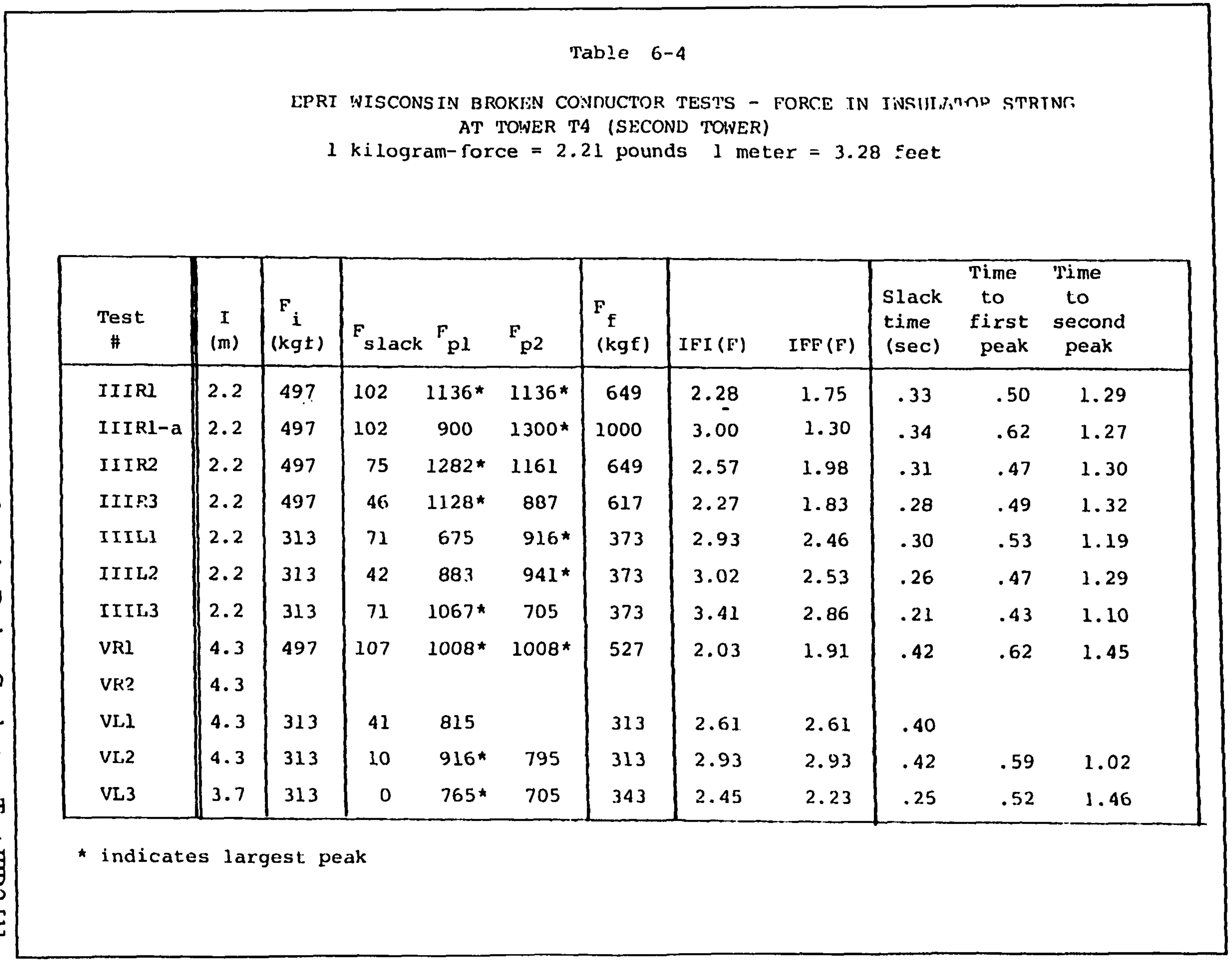


Table 6-4

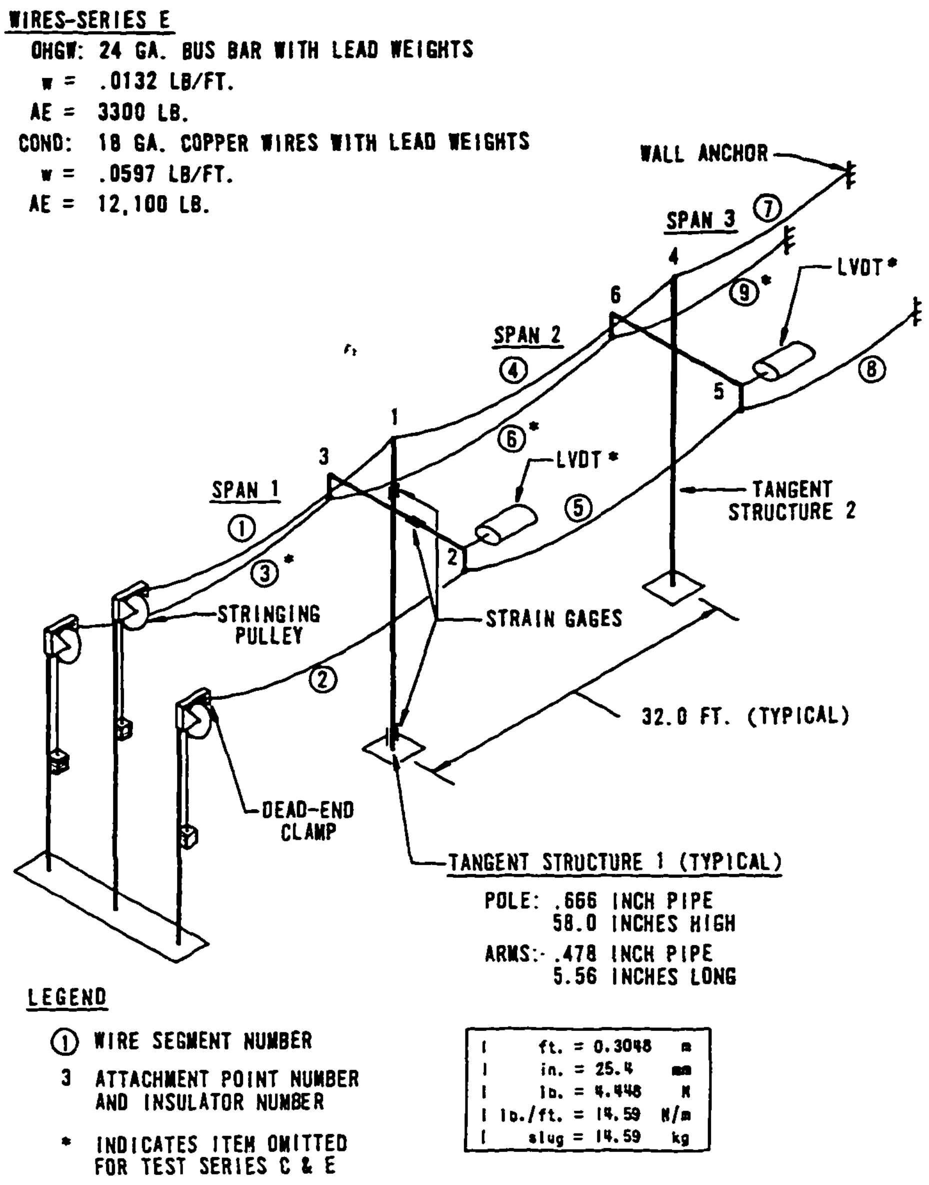
EPRI WISCONSIN BROKEN CONDUCTOR TESTS - FORCE IN INSULATOR STRING
AT TOWER T4 (SECOND TOWER)
1 kilogram-force = 2.21 pounds 1 meter = 3.28 feet

Test #	I (m)	F_i (kgf)	F_{slack}	F_{p1}	F_{p2}	F_f (kgf)	IFI(F)	IFF(F)	Slack time (sec)	Time to first peak	Time to second peak
IIIR1	2.2	497	102	1136*	1136*	649	2.28	1.75	.33	.50	1.29
IIIR1-a	2.2	497	102	900	1300*	1000	3.00	1.30	.34	.62	1.27
IIIR2	2.2	497	75	1282*	1161	649	2.57	1.98	.31	.47	1.30
IIIR3	2.2	497	46	1128*	887	617	2.27	1.83	.28	.49	1.32
IIIL1	2.2	313	71	675	916*	373	2.93	2.46	.30	.53	1.19
IIIL2	2.2	313	42	883	941*	373	3.02	2.53	.26	.47	1.29
IIIL3	2.2	313	71	1067*	705	373	3.41	2.86	.21	.43	1.10
VR1	4.3	497	107	1008*	1008*	527	2.03	1.91	.42	.62	1.45
VR2	4.3										
VL1	4.3	313	41	815		313	2.61	2.61	.40		
VL2	4.3	313	10	916*	795	313	2.93	2.93	.42	.59	1.02
VL3	3.7	313	0	765*	705	343	2.45	2.23	.25	.52	1.46

* indicates largest peak

Figure F.8 Force in Insulator One Span away from the Broken Conductor - Test IIIR2 [1]

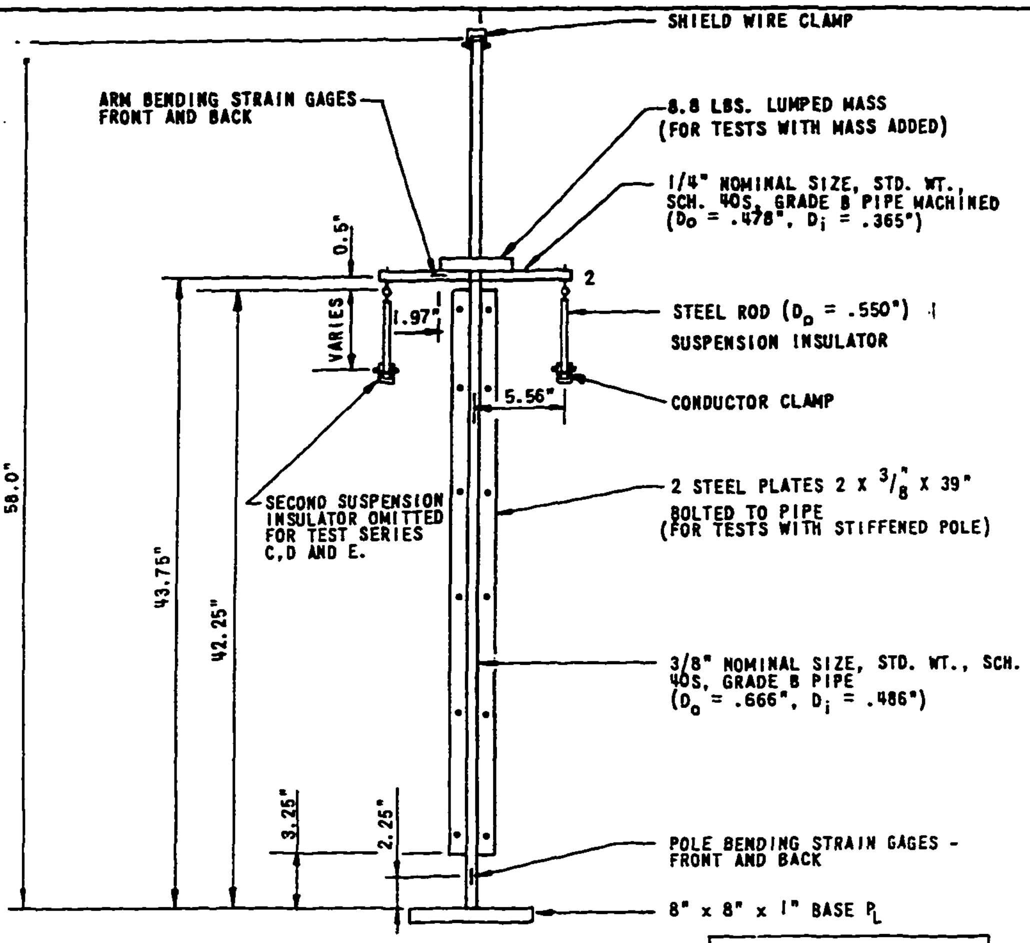
F.2 Experimental Data for the Broken Shield Wire Analysis [3]



1 ft. =	0.3048	m
1 in. =	25.4	mm
1 lb. =	4.448	N
1 lb./ft. =	14.59	N/m
1 slug =	14.59	kg

Figure F-1. Schematic of Test Setup

Figure F.9 Test Setup for the Broken Shield Wire Case Study [3]


SHIELD WIRE CLAMP
ARM BENDING STRAIN GAGES FRONT AND BACK
8.8 LBS. LUMPED MASS (FOR TESTS WITH MASS ADDED)
1/4" NOMINAL SIZE, STD. WT., SCH. 40S, GRADE B PIPE MACHINED (Do = .478", Di = .365")
0.5"
2
VARIES
1.97"
STEEL ROD (Do = .550") ⊣ SUSPENSION INSULATOR
5.56"
CONDUCTOR CLAMP
58.0"
SECOND SUSPENSION INSULATOR OMITTED FOR TEST SERIES C,D AND E.
2 STEEL PLATES 2 X 3/8" X 39" BOLTED TO PIPE (FOR TESTS WITH STIFFENED POLE)
43.75"
42.25"
3/8" NOMINAL SIZE, STD. WT., SCH. 40S, GRADE B PIPE (Do = .666", Di = .486")
3.25"
2.25"
POLE BENDING STRAIN GAGES - FRONT AND BACK
8" x 8" x 1" BASE PL

NOTES:

1. CENTERLINE OF POLE TO ARM END MEASURES 5.7 INCHES.
2. STRAIN GAGES ARE ON TOWER ONE ONLY.

1 ft.	= 0.3048 m
1 in.	= 25.4 mm
1 lb.	= 4.448 N
1 ft./lb.	= 0.0685 m/N

A. AS-BUILT MODEL SUPPORT STRUCTURE - SERIES E

$$\underset{\sim}{F_m} = \begin{bmatrix} 0.271 & 0.173 \\ 0.173 & 0.126 \end{bmatrix} \frac{\text{in.}}{\text{lb.}} \quad \text{UNSTIFFENED} \qquad \underset{\sim}{F_m} = \begin{bmatrix} 0.070 & 0.046 \\ 0.046 & 0.043 \end{bmatrix} \frac{\text{in.}}{\text{lb.}} \quad \text{STIFFENED}$$

B. AS-BUILT FLEXIBILITY MATRICES

Figure F-2. As-Built Model Support Structure and Flexibility Matrices

Figure F.10 Scale-Model for the Support Structures used in Reference [3]

Table F-2

(Continued)

Test	Test Type	Pole Type	Ins. Length I (in)	Initial S. W. Tension T_a (lb)	Conductor Tension			Ground Line Moment		Impact Factor		Structure Response Factor
					Initial T_c (lb)	Peak Dynamic T_d (lb)	Residual Static T_r (lb)	Peak M_d (in-lb)	Residual M_r (in-lb)	Arm $IF_A = T_d/T_r$	Structure $IF_S = M_d/M_r$	$SRF = IF_S/IF_A$
E45	Broken conductor	Braced	4.5	-	9.9	16.10	4.20	-	-	3.83	-	-
E46	Broken conductor	Braced	4.5	-	15.2	19.65	4.90	-	-	4.01	-	-
E47	Broken conductor	Braced	4.5	-	20.4	21.10	5.13	-	-	4.11	-	-
E48	Broken conductor	Braced	7.0	-	5.0	10.05	3.00	-	-	3.35	-	-
E49	Broken conductor	Braced	7.0	-	10.2	15.15	3.60	-	-	4.21	-	-
E50	Broken conductor	Braced	7.0	-	15.8	18.20	3.95	-	-	4.61	-	-
E51	Broken conductor	Braced	7.0	-	20.7	18.90	4.00	-	-	4.72	-	-
E52	Broken conductor	Unstiffened w/mass 1/2" S. W. Link	4.5	1.46	15.3	15.45	4.60	564	175	3.36	3.22	0.96
E53	Broken conductor	Unstiffened w/mass 1/2" S. W. Link	4.5	2.63	15.4	16.30	4.63	414	139	3.52	2.98	0.85
E54	Broken shield wire	Unstiffened w/mass	4.5	1.51	15.3	-	-	134	75	-	1.79	-
E55	Broken shield wire	Unstiffened w/mass	4.5	3.14	15.2	-	-	204	113	-	1.80	-
E56	Broken shield wire	Unstiffened w/mass	4.5	5.20	15.1	-	-	244	135	-	1.81	-
E57	Dropped weight	Unstiffened w/mass	4.5	-	8.6	3.50	1.30	152	52	2.69	2.92	1.09

1 lb = 4.448 N
1 in = 25.4 mm
1 in-lb = 0.1130 N-m

Figure F.11 Broken Shield Wire Results - Test E54 [3]

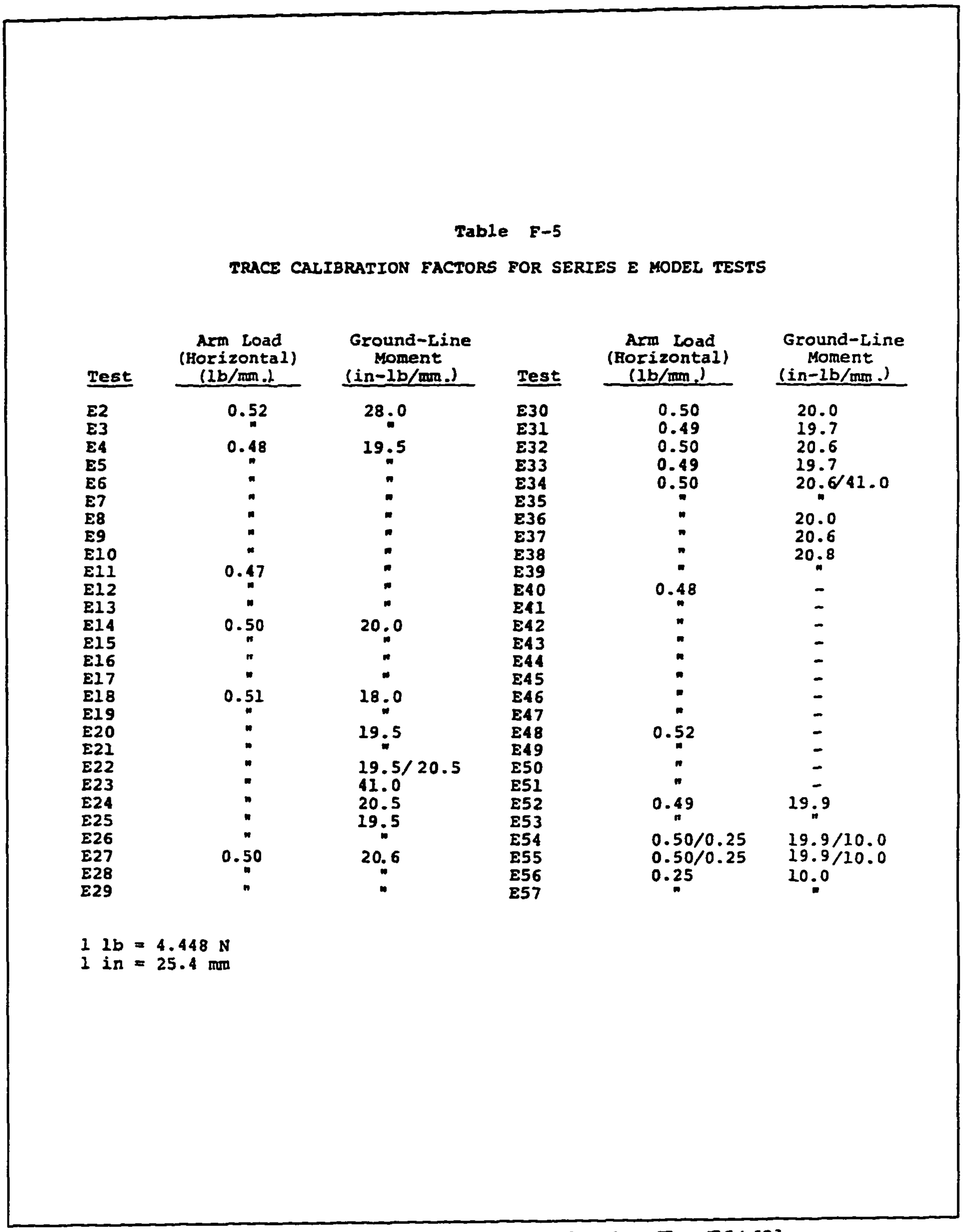


Table F-5

TRACE CALIBRATION FACTORS FOR SERIES E MODEL TESTS

Test	Arm Load (Horizontal) (lb/mm.)	Ground-Line Moment (in-lb/mm.)	Test	Arm Load (Horizontal) (lb/mm.)	Ground-Line Moment (in-lb/mm.)
E2	0.52	28.0	E30	0.50	20.0
E3	"	"	E31	0.49	19.7
E4	0.48	19.5	E32	0.50	20.6
E5	"	"	E33	0.49	19.7
E6	"	"	E34	0.50	20.6/41.0
E7	"	"	E35	"	"
E8	"	"	E36	"	20.0
E9	"	"	E37	"	20.6
E10	"	"	E38	"	20.8
E11	0.47	"	E39	"	"
E12	"	"	E40	0.48	-
E13	"	"	E41	"	-
E14	0.50	20.0	E42	"	-
E15	"	"	E43	"	-
E16	"	"	E44	"	-
E17	"	"	E45	"	-
E18	0.51	18.0	E46	"	-
E19	"	"	E47	"	-
E20	"	19.5	E48	0.52	-
E21	"	"	E49	"	-
E22	"	19.5/20.5	E50	"	-
E23	"	41.0	E51	"	-
E24	"	20.5	E52	0.49	19.9
E25	"	19.5	E53	"	"
E26	"	"	E54	0.50/0.25	19.9/10.0
E27	0.50	20.6	E55	0.50/0.25	19.9/10.0
E28	"	"	E56	0.25	10.0
E29	"	"	E57	"	"

1 lb = 4.448 N
1 in = 25.4 mm

Figure F.12 Scale Factors for Result Graphs - Test E54 [3]

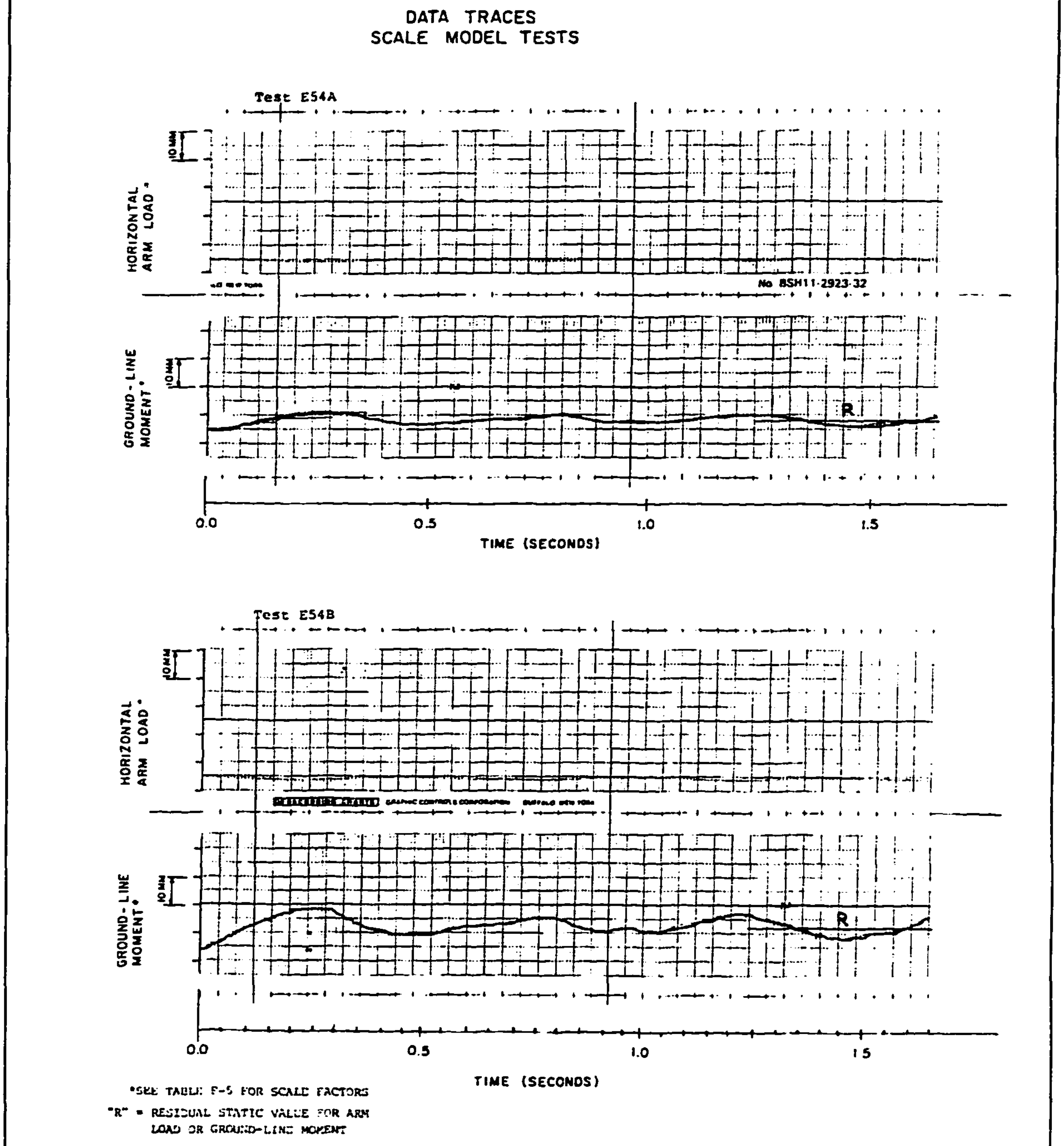


Figure F.13 Broken Shield Wire Test Results - Test E54 [3]

F.3 Experimental Data for the Conductor Galloping Analysis [2]

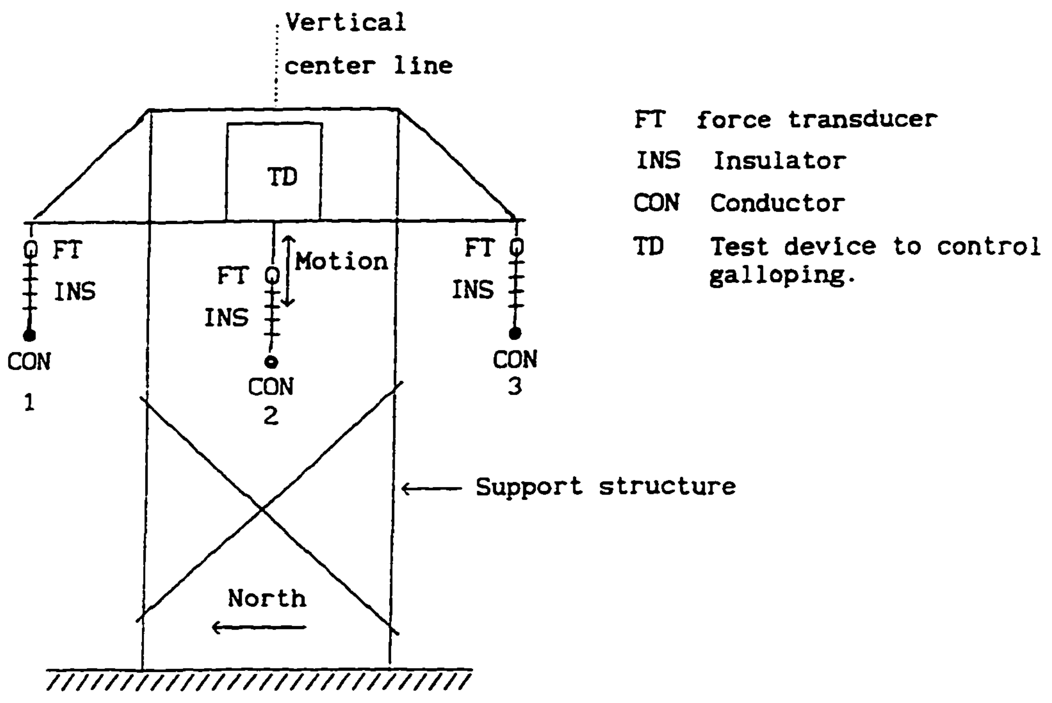
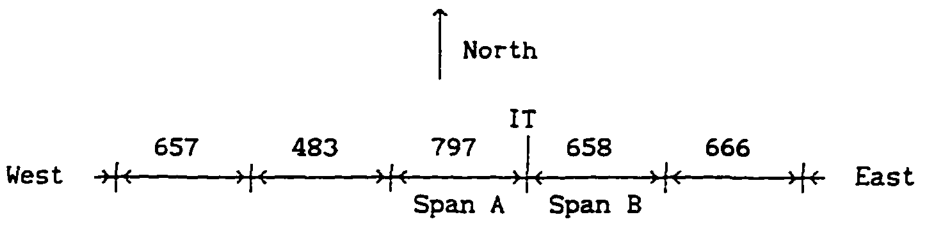


Fig. 3. Schematic of test device on power pole with force and motion sensors indicated.

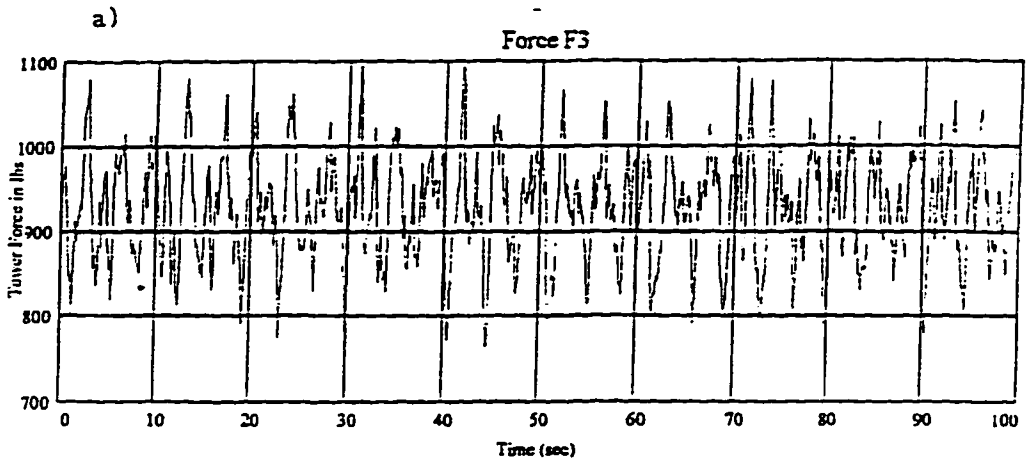


IT instrumented tower.

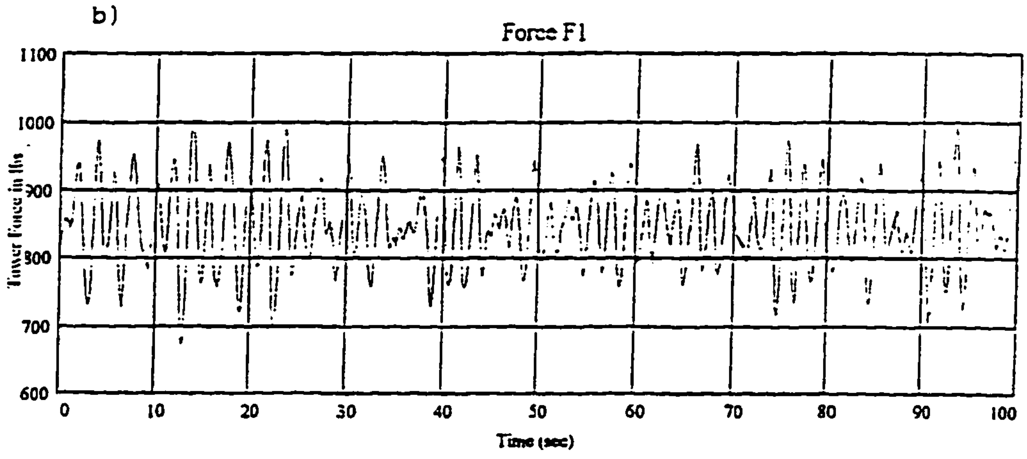
Fig. 4. Support tower spacing in vicinity of test sight.

Figure F.14 Transmission Line Tested in Reference [2]

a)



b)



8. Largest force time histories for a) line 3 when 8.1 mph wind (40 degrees relative to the line) is blowing on 14 February from 22:10 hrs data set and b) line 1 when 21.1 mph wind (42 degrees relative to the line) is blowing on 15 February from 11:30 hrs data set.

Figure F.15 Real Galloping Data: Insulator Force between Spans A and B [2]

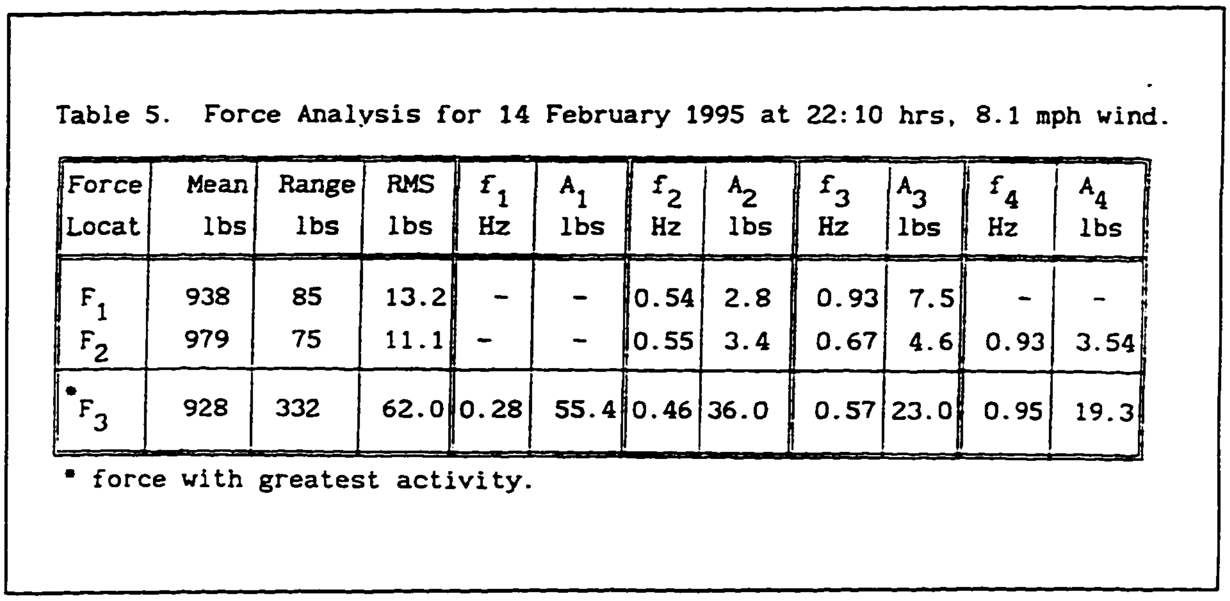


Table 5. Force Analysis for 14 February 1995 at 22:10 hrs, 8.1 mph wind.

Force Locat	Mean lbs	Range lbs	RMS lbs	f_1 Hz	A_1 lbs	f_2 Hz	A_2 lbs	f_3 Hz	A_3 lbs	f_4 Hz	A_4 lbs
F_1	938	85	13.2	-	-	0.54	2.8	0.93	7.5	-	-
F_2	979	75	11.1	-	-	0.55	3.4	0.67	4.6	0.93	3.54
* F_3	928	332	62.0	0.28	55.4	0.46	36.0	0.57	23.0	0.95	19.3

* force with greatest activity.

Figure F.16 Frequency Analysis for Galloping Force Data [2]

REFERENCES

1. A. H. Peyrot, R. O. Kluge and J. W. Lee, *Longitudinal Loading Tests on a Transmission Line*, Final Report Project 1096-1. Palo Alto, California: Electric Power Research Institute, September 1978.
2. K. G. McConnell, *A Vibration Damping Power Line Support*, Final Report. Kansas City, Missouri: Federal Emergency Management Agency, 1995.
3. J. D. Mozer, A. W. Wood and J. A. Hribar, *Longitudinal Unbalanced Loads on Transmission Line Structures*, Final Report Project #561. Palo Alto, California: Electric Power Research Institute, 1978.
4. A. Govers, "On the Impact of Uni-directional Forces on High-voltage Towers Following Conductor-breakage," *International Conference on Large Electric Systems* (CIRGE), Paper No. 22-03, Paris, 1970.
5. Electric Power Research Institute, *TLWorkstation Code ETADS User's Manual.* Palo Alto, California: EPRI, vol. 23, 1990.
6. M. Baenziger, *Broken Conductor Loads on Transmission Line Structures*, Ph. D. Dissertation. Madison, Wisconsin: University of Wisconsin, 1981.
7. F. M. Siddiqui and J. F. Fleming, "Broken wire analysis of transmission line systems," *Computers and Structures*, vol. 18, no. 6, 1984, pp. 1077-1085.
8. H. M. Irvine, *Cable Structures*. London, England: The MIT Press, 1981.
9. H. H. West, L. F. Geschwinder and J. E. Suhoski, "Natural vibrations of suspension cables," *Journal of the Structural Division*, Proceedings of the American Society of Civil Engineers, ASCE, vol. 101, no. ST11, November 1975, pp. 2277-2291.
10. A. G. Nariboli and K. G. McConnell, "Curvature coupling of catenary cable equations," *Journal of Modal Analysis*, April 1988, pp. 49-56.
11. W. T. O'Brien and A. J. Francis, "Cable movements under two-dimensional loads," *Journal of the Structural Division*, Proceedings of the American Society of Civil Engineers, ASCE, vol. 90, no. ST3, June 1964, pp. 89-123.

12. W. T. O'Brien, "General solution of suspended cable problems," *Journal of the Structural Division*, Proceedings of the American Society of Civil Engineers, ASCE, vol. 93, no. ST1, February 1967, pp. 1-26.

13. R. A. Skop and G. J. O'Hara, "The method of imaginary reactions. A new technique for analyzing structural cable systems," *Marine Technology Society Journal*, vol. 4, no. 1, 1970, pp. 21-30.

14. R. A. Skop and G. J. O'Hara, "A method of analysis of internally redundant structural cable arrays," *Marine Technology Society Journal*, vol. 6, no. 1, 1972, pp. 6-18.

15. A. H. Peyrot and A. M. Goulois, " Analysis of cable structures," *Computers and Structures*, vol. 10, 1979, pp. 805-813.

16. A. H. Peyrot and A. M. Goulois," Marine cable structures," *Journal of the Structural Division*, Proceedings of the American Society of Civil Engineers, ASCE, vol. 106, no. ST3, December 1980, pp. 2391-2404.

17. F. Baron and M. S. Venkatesan, "Nonlinear analysis of cable and truss structures," *Journal of the Structural Division*, Proceedings of the American Society of Civil Engineers, ASCE, vol. 97, no. ST2, February 1971, pp. 679-710.

18. R. L. Webster, "Nonlinear static and dynamic response of underwater cable structures using the finite element method," *Offshore Technology Conference*, Paper No. OTC2322, Dallas, Texas, 1975, pp. 753-764.

19. J. Mitsugi and T. Yasaka, "Nonlinear static and dynamic analysis method of cable structures," *American Institute of Aeronautics and Astronautics Journal*, vol. 29, no. 1, January 1991, pp. 150-152.

20. P. Broughton and P. Ndumbaro, *The Analysis of Cable and Catenary Structures*. London, England: Thomas Telford Services Ltd., 1994.

21. Y. M. Desai, N. Popplewell, A. H. Shah and D. N. Buragohain, "Geometric nonlinear static analysis of cable supported structures," *Computers and Structures*, vol. 29, no. 6, 1988, pp. 1001-1009.

22. E. Comellini and C. Manuzio, "Rational determination of design loadings for overhead line towers," *International Conference on Large Electric Systems* (CIRGE), Paper No. 23-08, Paris, 1968.

23. L. Kempner Jr., W. H. Mueller and E. H. Bennett, "Longitudinal impact loading of transmission towers," *Proceedings of the Sessions Related to Steel Structures at Structures Congress `89*, New York, New York: American Society of Civil Engineers, 1989, pp. 248-257.

24. M. B. Thomas and A. H. Peyrot, "Dynamic response of ruptured conductors in transmission lines," *IEEE Transactions on Power Apparatus and Systems*, vol. PAS-101, no. 9, September 1982, pp. 3022-3027.

25. T. J. Wipf, F. Fanous, M. Baenziger, S. Gupta and R. Anjam, *Ice Storm Damage Assessment of the Lehigh-Sycamore 345 KV Transmission Line*. Ames, Iowa: Electric Power Research Center, July 1991.

26. A. M. Nafie, *Failure Analysis of Transmission Line Structures*, Thesis. Ames, Iowa: Iowa State University, 1993.

27. A. H. Peyrot and A. M. Goulois, "Analysis of flexible transmission lines.", *Journal of the Structural Division, Proceedings of the American Society of Civil Engineers*, vol. 104, no. ST5, May 1978, pp. 763-779.

28. Electric Power Research Institute, *Transmission Line Reference Book, Wind-Induced Conductor Motion*. Palo Alto, California: EPRI, 1979.

29. J. P. Den Hartog, "Transmission line vibration due to sleet," *AIEE Trans.* , vol. 51, no. 12, 1932, pp. 1074-1076.

30. R. Ruedy, *Galloping of Transmission Lines*, NRC Report No. 1751. Ottawa, Ontario: National Research Council of Canada, 1948.

31. F. Cheers, *A Note on Galloping Conductors*, NRC Report No. MT14. Ontario: National Research Council of Canada, 1950.

32. A. T. Edwards and A. Madeyski, "Progress report on the investigation of galloping of transmission line conductors," *Trans. AIEE*, vol. 75, no. 3, 1956, pp. 666-686.

33. A. S. Richardson Jr. , J. R. Martuccelli and W. S. Price, "Research study on galloping of electric power transmission lines," *Proceeding of the First International Conference on Wind Effects on Buildings Structures*, NPL, Teddington, vol. II, 1965, pp. 612-686.

34. O. Nigol and G. J. Clarke, " Conductor galloping and control based on torsional mechanism," IEEE Power Engineering Society Winter Meeting, Paper No. C74-0162, New York, NY, 1974.

35. O. Nigol and P. G. Buchan, "Conductor galloping part I - Den Hartog mechanism," *IEEE Transactions on Power Apparatus and Systems*, vol. PAS-100, no. 2, February 1981, pp. 699-707.

36. O. Nigol and P. G. Buchan "Conductor galloping part II - torsional mechanism," *IEEE Transactions on Power Apparatus and Systems*, vol. PAS-100, no. 2, February 1981, pp. 708-720.

37. K. F. Jones, "Coupled vertical and horizontal galloping," *Journal of Engineering Mechanics*, vol. 118, no. 1, January 1992, pp. 92-106.

38. P. Yu, Y. M. Desai, A. H. Shah and N. Popplewell, "Three-degree of freedom model for galloping. Part I: formulation." *Journal of Engineering Mechanics*, vol. 119, no. 12, December 1993, pp. 2404-2425.

39. P. Yu, Y. M. Desai, N. Popplewell and A. H. Shah, "Three-degree of freedom model for galloping. Part II: solutions." *Journal of Engineering Mechanics*, vol. 119, no. 12, December 1993, pp. 2426-2448.

40. Y. M. Desai, P. Yu, N. Popplewell and A. H. Shah, "Finite element modeling of transmission line galloping," *Computers and Structures*, vol. 57, no. 3, 1995, pp. 407-420.

41. P. Stumpf, and H. C. M. Ng, *Investigation of Aerodynamic Stability for Selected Inclined Cables and Conductor Cables*. B. Sc. Thesis, Manitoba: University of Manitoba, 1990.

42. R. D. Blevins and W. D. Iwan, "The galloping response of a two-degree of freedom system," *Journal of Applied Mechanics*, vol. 41, December 1974, pp. 1113-1118.

43. R. I. Egbert, "Estimation of maximum amplitudes of conductor galloping by describing function analysis," *IEEE Transactions on Power Delivery*, vol. PWRD-1, no. 1, January 1986, pp. 251-257.

44. G. S. Byun and R. I. Egbert, "Two-degree-of-freedom analysis of power line galloping by describing function methods." *Electric Power Systems Research*, vol. 21, 1991, pp. 187-193.

45. A. S. Richardson Jr., "Predicting galloping amplitudes," *Journal of Engineering Mechanics*, vol. 114, no. 4, April 1988, pp. 716-723.

46. A. S. Richardson Jr., "Predicting galloping amplitudes: II," *Journal of Engineering Mechanics*, vol. 114, no. 11, November 1988, 1945-1951.

47. Y. M. Desai, A. H. Shah and N. Popplewell, "Galloping analysis for two-degree of-freedom oscillator," *Journal of Engineering Mechanics*, vol. 116, no. 12, December 1990, pp. 2583-2602.

48. J. C. R. Hunt and D. J. W. Richard, "Overhead-line oscillations and the effect of aerodynamic dampers," *The Institute of Electrical Engineers*, vol. 116, no. 11, November 1969, pp. 1869-1874.

49. M. A. Baenziger, W. D. James, B. Wouters and L. Li, "Dynamic loads on transmission line structures due to galloping conductors," *IEEE Transactions on Power Delivery*, vol. 9, no. 1, January 1994, pp. 40-47.

50. A. E. Davison, "Ice coated electrical conductors," *Bulletin, Hydro-Electric Power Commission of Ontario*, vol. 26, no. 9, September 1939, pp. 271-280.

51. C. B. Rawlins, "Analysis of conductor galloping field observations - single conductors," *IEEE transactions on Power Apparatus and Systems*, vol. PAS-100, no. 8, August 1981, pp. 3744-3753.

52. S. Krishnasamy, "Wind and ice loads on overhead transmission lines," *Ontario Hydro Research Review*, vol. 3, June 1981, pp. 11-18.

53. L. Li, *Dynamic Loads on Pole Transmission Line Structures from Galloping Conductors*, M.S. Thesis. Ames, Iowa: Iowa State University, 1990.

54. M. Wu, *Experimental and Analytical Study of Galloping Forces on Support Structures*, Ph. D. Dissertation. Ames, Iowa: Iowa State University, 1996.

55. H. Schildt, *C++ the Complete Reference*. California: Osborne McGraw-Hill, U.S.A., 1991.

56. J. S. R. Filho and P. R. B. Devloo, "Object oriented programming in scientific computations: the beginning of a new era," *Engineering Computations*, vol. 8, 1991, pp. 81-87.

57. R. S. Arruda, L. Landau and N. F. F. Ebecken, "Object-oriented structural analysis in a graphical environment," *Artificial Intelligence and Object Oriented Approaches for Structural Engineering*, Edinburgh, Scotland: Civil-Comp Ltd., 1994, pp. 129-138.

58. G. R. Miller, "An object oriented approach to structural analysis and design," *Comp. and Struct.*, vol. 40, no. 1, 1991, pp. 75-82.

59. R. R. Gajewski, "An object oriented approach to finite element programming," *Artificial Intelligence and Object Oriented Approaches for Structural Engineering*, Edinburgh, Scotland: Civil-Comp Ltd., 1994, pp. 107-113.

60. T. Zimmermann, Y. Dubois-Pelerin and P. Bomme, "Object-oriented finite element programming: I. Governing principles," *Comp. Meth. In Applied Mech. and Eng.*, vol. 98, 1992, pp. 291-303.

61. B. W. Forde, R. B. Foschi and S. F. Stiemer, "Object oriented finite element analysis," *Comp. And Struct.*, vol. 34, 1990, pp. 355-374.

62. S. P. Scholz, "Elements of an object oriented FEM++ program in C++," *Comp. and Struct.*, vol. 43, no. 3, 1992, pp. 517-529.

63. K. J. Bathe, *Finite Element Methods*. Berlin: Springer, 1990.

64. S. P. Timoshenko, " On the correction for shear of the differential equation for transverse vibration of prismatic bars," *The London, Edinburgh and Dublin Philosophical Magazine and Journal of Science*, vol. 41, 1921, pp. 744-746.

65. Dubois-Pelerin, T. Zimmermann and P. Bomme, "Object-oriented finite element programming: II. A prototype program in smalltalk," *Comp. Meth.In Applied Mech. And Eng.*, vol. 98, 1992, pp. 361-397.

66. Dubois-Pelerin and T. Zimmermann, "Object-oriented finite element programming: III. An efficient implementation in C++," *Comp. Meth. In Applied Mech. And Eng.*, vol. 108, 1993, pp. 165-183.

67. G. Yu and H. Adeli, "Object-oriented finite element analysis using EER model," *Journal of Structural Engineering*, vol. 119, no. 9, September 1993, pp. 2763-2781.

68. J. Ju and M. U. Hosain, "Substructuring using an object oriented approach," *Artificial Intelligence and Object Oriented Approaches for Structural Engineering*, Edinburgh, Scotland: Civil-Comp Ltd., 1994, pp. 115-120.

69. M. Miki and Y. Murotsu, "Object-oriented approach to modeling and analysis of truss structures," *AIAA Journal*, vol. 33, no. 2, February 1995, pp. 348-354.

70. P. R. B. Devloo, "Efficiency issues in an object oriented programming environment," *Artificial Intelligence and Object Oriented Approaches for Structural Engineering*, Edinburgh, Scotland: Civil-Comp Ltd., 1994, pp. 147-151.

71. R. B. Murray, *C++ Strategies and Tactics*. Reading, Massachusetts: Addison-Wesley Publishing Company, 1993.

72. Microsoft Corporation, *Visual C++ User's Guide*. Redmond, Washington, 1993.

73. W. Weaver Jr. and J. M. Gere, *Matrix Analysis of Framed Structures*. New York, New York: D. Van Nostrand Company, 1980.

74. W. H. Press, S. A. Teukolsky, W. T. Vetterling and B. P. Flannery, *Numerical Recipes in C, The Art of Scientific Computing*. New York, New York: Cambridge University Press, 1992.

75. R. D. Cook, *Concepts and Applications of Finite Element Analysis*. New York: John Wiley and Sons, Inc., 1981.

76. Swanson Analysis Systems, Inc., *ANSYS User's Manual, Volume IV, Theory*. Houston, PA 1992.

77. CC. Rankin and F. A. Brogan, " An element independent corotational procedure for the treatment of large rotations," *Journal of Pressure Vessels*, vol. 108, May 1986, pp. 165-174.

78. J. Argyris, "An excursion into large rotations," *Computer Methods in Applied Mechanics and Engineering*, vol. 32, North-Holland Publishing Company, 1982, pp. 85-155.

79. R. J. Guyan, "Reduction of stiffness and mass matrices," *AIAA Journal*, vol. 3, no. 2, February 1965.

80. Borland International Inc., *Borland Delphi for Windows User's Guide*. Scotts Valley, California, 1995.

81. Microsoft Corporation, *Microsoft Windows 3.1 Programmer's Reference Library*. Redmond, Washington: Microsoft Press.

82. Electric Power Research Center, *DYNTRN - Structural Analysis Software for Graphical Simulation of Transmission Line Dynamic Behavior*. Ames, Iowa: EPRC, 1997.