TWO-DIMENSIONAL PROGRESSIVE COLLAPSE ANALYSIS ON REINFORCED CONCRETE FRAME STRUCTURES BASED ON THE LUMPED DAMAGED-PLASTICITY METHOD

By

Han Xiao

A thesis submitted in partial fulfillment of

the requirements for the degree of

Master of Science

(CIVIL AND ENVIRONMENTAL ENGINEERING)

at the

UNIVERSITY of WISCONSIN-MADISON

2016

© Copyright by Han Xiao 2016

All Rights Reserved

Contents

Chapter 1. Introduction 1
1.1 General classification of progressive collapse analysis methods 3
1.2 Comparison of alternate load path methods 6
1.2.1 Linear-elastic static 6
1.2.2 Nonlinear Static 9
1.2.3 Dynamic analysis procedures 10
1.2.4 Discussion 10
1.3 Simplifications of plastic behavior 11
1.4 Catenary action 15
1.5 Outline 19
Chapter 2. Lumped damaged-plasticity method 20
2.1 Basic idea of damage variables 20
2.2 Equivalence and constitutive equations 21
2.2.1 Equivalence equations 21
2.2.2 Plasticity evolution laws 23
2.2.3 Damage evolution laws 25
2.2.4 Assumptions of the lumped damaged-plasticity method 27
2.3 Calculation of constants 29
2.4 Catenary action 32
Chapter 3. Flexural model validation and calibration 39
3.1 Description of the specimen 39
3.2 Modeling methods 46
3.2.1 Hand calculations 47
3.2.2 Detailed finite element model with ABAQUS 55
3.2.3 Model with fiber element method 61
3.2.4 Model with the lumped damaged-plasticity method 66
3.3 Comparison of models 71
3.4 Deficiency of the lumped damaged-plasticity model 75
Chapter 4. Comparison to experimental data 77
4.1 Experiment description 77
4.2 Calculation of sectional properties 85
4.2.1 Material properties 85
4.2.2 Mechanical properties of sections 90

4.2.3 Lumped-damaged plasticity elements 98
4.3 Outcomes of the lumped-damaged plasticity model 100
4.4 Nonlinear dynamic analysis 104
Chapter 5. Conclusions and future research 113
5.1 Conclusions and deficiencies 113
5.2 Future research 116
Appendix 121
Appendix 1. Header file 121
Appendix 2. Main .cpp file 124
Appendix 3. .Cpp file to connect editor 186
References 118

LIST OF FIGURES

Figure 1.1 Degrading inelastic element for reinforced concrete beam-columns under biaxial bending and axial load (Lai and Will, 1986) 14

Figure 1.2 Simplified model by Yi et al. (2008) to calculate catenary action 16

Figure 1.3 Simply supported beam with linear elastic axial restraint (Izzuddin, 2005) 17

Figure 1.4 Plastic mechanism with a plastic hinge at midspan (Izzuddin, 2005) 17

Figure 1.5 Plastic interaction of axial force and bending moment (Izzuddin, 2005) 17

Figure 1.6 loading stages of elasto-plastic response (Izzuddin, 2005) 18

Figure 2.1 Transfer of catenary forces through stirrups (Orton, 2007) 33

Figure 2.2 Force-displacement relationship (Sasani and Kropelnicki, 2008) 34

Figure 2.3 Assumption of unbonded length 37

Figure 3.1 Model configuration 39

Figure 3.2 Displacement step applied at the free end of specimen 40

Figure 3.3 Cross-section layout and dimensions (units of inches) 41

Figure 3.4 Hognestad’s stress-strain diagram in flexure (Hognestad, 1951) 42

Figure 3.5 Stress-strain relation of concrete 44

Figure 3.6 Stress-strain relation of longitudinal reinforcement 46

Figure 3.7 Linear strain distribution at first yield point ($\varepsilon_s = \varepsilon_y$) 49

Figure 3.8 Linear strain distribution at ultimate point ($\varepsilon_c = \varepsilon_{cu}$) 51

Figure 3.9 Moment-displacement relation from hand calculation 54

Figure 3.10 Yield surfaces in the deviatoric plane, corresponding to different values of K_c (Dassault Systémes, 2012) 56

Figure 3.11 Response of concrete to uniaxial loading in compression (Dassault Systémes, 2012) 58

Figure 3.12 Moment-displacement relation from ABAQUS analysis 60

Figure 3.13 Analysis procedure of fiber element method (Petrangeli et al., 1999) 62

Figure 3.14 Unloading and reloading stages in concrete compression 65

Figure 3.15 Moment-displacement relation from fiber element analysis 66

Figure 3.16 Moment-displacement relation for the lumped damaged-plasticity model 69

Figure 3.17 Moment-damage relation for the lumped damaged-plasticity model 70

Figure 3.18 Moment-plastic rotation relation for the lumped damaged-plasticity model 71

Figure 3.19 Moment-displacement relation for the lumped damaged-plasticity model 72

Figure 4.1 Sketch of original office building (Stinger and Orton, 2013) 78

Figure 4.2 Layout of column cross-sections in original building (Stinger, 2011) 79

Figure 4.3 Layout of beam cross-sections in original building (Stinger, 2011) 79

Figure 4.4 Details of shear reinforcement in original building (Stinger, 2011) 80
Figure 4.5 Layout of column cross-sections in the test frame (Stinger, 2011)............................ 80
Figure 4.6 Layout of beam cross-sections in the test frame (Stinger, 2011) 81
Figure 4.7 Details of shear reinforcement in the test frame (Stinger, 2011) 81
Figure 4.8 Details of longitudinal reinforcement for the continuous test frame (Stinger, 2011) . 82
Figure 4.9 Details of transverse reinforcement for the continuous test frame (Stinger, 2011)..... 82
Figure 4.10 Design of reaction frame (Stinger, 2011) 83
Figure 4.11 Relation of vertical load and displacement at the top of central column (Stinger, 2011) 84
Figure 4.12 Assumed stress-strain relationship for #2 and #3 reinforcing bars 86
Figure 4.13 Stress-strain relationship of concrete... 90
Figure 4.14 Plastic hinge positions A and B.. 92
Figure 4.15 Element categories and lengths 99
Figure 4.16 Comparison of vertical load and displacement at the top of center column between lumped damaged-plasticity model and Stinger (2011) experimental data................................. 101
Figure 4.17 Beams at position B in catenary action (Stinger, 2011) 103
Figure 4.18 Time series for the vertical displacement at the top of center column on second floor (with 5.5-kip load) 106
Figure 4.19 Time series for moments of beams on the second floor at position A (with 5.5-kip load) 107
Figure 4.20 Time series for the vertical displacement at the top of center column on second floor (with 6-kip load) 109
Figure 4.21 Time series for moments of beams on the second floor at positions A (with 6-kip load) 110
Figure 4.22 Time series for axial forces of beams on the second floor (with 6-kip load).......... 111

LIST OF TABLES

Table 1.1 Load dynamic amplification factor for LSP in GSA 2013 7
Table 1.2 Load dynamic amplification factor for NSP in GSA 2013... 9
Table 3.1 Material property of concrete 44
Table 3.2 Material property of reinforcement.. 45
Table 3.3 Mechanical properties of sections in specimen 52
Table 3.4 Values of factors defined in concrete damage plasticity model 57
Table 3.5 Factors input into OpenSees for the lumped damaged-plasticity model 68
Table 3.6 Comparison of different models 73
Table 4.1 Experimental material properties for test frame (Stinger 2011) 85
Table 4.2 Concrete and section properties of beams in lumped damaged-plasticity model......... 89
Table 4.3 Concrete and section properties of columns in lumped damaged-plasticity model 89
Table 4.4 Mechanical properties of sections at A.. 93
Table 4.5 Mechanical properties of sections at B 94
Table 4.6 Mechanical properties of sections for columns 95
Table 4.7 Mechanical properties of beam sections applied with positive moment after column removal 96
Table 4.8 Mechanical properties of beam sections applied with negative moment after column removal 97
Table 4.9 Material properties and mechanical properties of reaction frame steel elements......... 98
Table 4.10 Factors for the lumped damaged-plasticity model... 100
Table 4.11 Data of the experiment and simulation.. 101
Table 4.12 Modeled moments of the reaction frame members 104
Table 4.13 Peak values and final values for 5.5-kip load 107
Table 4.14 Peak values and final values for 6 kips load... 112

ABSTRACT

Progressive collapse, a chain reaction or propagation of failures following damage to a relatively small portion of a structure, has drawn attention from researchers and the public because it leads to severe loss of life and property. Collapse of the Ronan Point apartment in 1968, terrorist attacks at the Murrah Federal Office building in 1995, and the tragedy of World Trade Center on September 11th, 2001 are examples of progressive collapse. However, the current procedures of progressive collapse analysis are limited by types of buildings or calculation efficiencies. This paper introduces the lumped damaged-plasticity method as an efficient and accurate method to check the resistance of two-dimensional (2-D) reinforced concrete frame structures in progressive collapse analysis.

The lumped damaged-plasticity method is based on a simplification of continuum damage mechanics, which introduces a damage variable. This damage variable abstracts damage at the microscale, such as the density of microcracks and microvoids, and its influence on the behavior of the material at the continuum level. Unlike theories of continuum damage mechanics, in the following model all the damage and plasticity in a beam or column element are concentrated at both ends of the element. A yield surface and a damage surface are developed to describe the evolution of plastic rotation and damage. Catenary action is included into this method to account for the main resistance mechanism after flexural failure.

A reinforced concrete cantilever beam and a 2-D frame structure were used to demonstrate static analysis by the lumped damaged-plasticity method. Results are compared with other models for the cantilever beam, and with experimental data for the frame structure. Dynamic progressive collapse analysis results on the frame structure are also shown. While further research is needed,

the lumped damaged-plasticity method is shown to be an efficient and accurate method to check resistance of 2-D RC structures in progressive collapse analysis.

Chapter 1. Introduction

Progressive collapse, a chain reaction or propagation of failures following damage to a relatively small portion of a structure, has drawn attention from researchers and the public because it leads to severe loss of life and property. One early and infamous event of progressive collapse was the collapse of Ronan Point apartment in England, 1968. This collapse was initiated by a gas explosion in a corner kitchen on the 18th floor, which blew out the exterior wall and the corner bay. The southeast corner of the apartment completely collapsed, killing 4 of the 260 residents and injuring 17 others. This event led England to address progressive collapse in the Building (Fifth Amendment) Regulations (1970), and spurred interest in collapse research. The high interest in progressive collapse now follows the terrorist attacks on Murrah Federal Office building in 1995 and ones at the World Trade Center on September 11th, 2001. Both events caused severe damage and loss of life.

To save lives and improve structural safety, progressive collapse analysis has become important to civil engineers. To indirectly improve collapse resistance, ACI-318 (2014) recommends improving integrity and robustness of buildings. For example, section 9.7.7 in ACI 318-14 notes detailing requirements of beam reinforcement and connections to improve redundancy and ductility. Design guidelines for progressive collapse analysis are also available. The General Services Administration (GSA) published design guidelines (in 2003 and amended in 2013) for progressive collapse resistance and provided detailed analysis recommendations, which are discussed in the literature review below. In 2009, the United States Department of Defense (DoD) published Unified Facilities Criteria (UFC) addressing progressive collapse resistance and analysis. Although design guidelines have provided suggestions on progressive collapse analysis,

development of simplified methodology able to accurately and efficiently simulate collapse of structures is still needed in research and design.

The alternate load path method is included in GSA (2003; 2013) and UFC (Department of Defense 2009) provisions, and is widely used in progressive collapse simulation. The premise of the alternate load path method is to remove vulnerable members in structures to check behaviors of remaining members in the post-damage state after load redistribution. Progressive collapse happens with increasing numbers of failed members. A level of initial damage is prescribed, and the alternate load path method is assumed to be independent of the event that initiates the damage. Calculating resistances and reactions for specific events like bomb or gas explosions is complicated.

Detailed finite element analysis, fiber models, and lumped plasticity models are three common methods to introduce plasticity and damage into concrete structures. Detailed finite element analysis gives the most robust results, though it is time consuming for nonlinear dynamic progressive collapse analysis. Fiber models are mesh dependent and limited to modeling normal stresses in the longitudinal direction. The lumped plasticity model is efficient, but damage is ignored. The lumped damaged-plasticity model is an improved lumped plasticity method, and includes both damage and plasticity at both ends of an element while the rest of the element remains elastic. Flow rules are used as evolution laws for plasticity and damage.

In addition to flexural resistance and damage, catenary action should also be considered in progressive collapse. Catenary action states the beam can provides extra resistance to external vertical loadings relying on the tensile capacity of the beam and large displacements. After flexural failure, catenary action becomes the main resistance source for moment-resisting frame.

The focus of this research is to develop a simplified lumped-damaged-plasticity method with consideration of catenary action, implement this method to simulate resistance mechanisms in progressive collapse, and examine whether this method retains acceptable accuracy of collapse phenomena while significantly decreasing computational time. In this report, the developed lumped damaged-plasticity model is applied to progressive collapse analysis on 2-D reinforcement frame structures. The lumped damaged-plasticity model is verified and validated using a detailed ABAQUS model, a fiber element model, and experimental data in progressive collapse analysis. The proposed methodology is intended to provide an efficient and accurate method in progressive collapse analysis for researchers and designers, and guide further research to expand the simplified method to complicated 3-D structures.

1.1 General classification of progressive collapse analysis methods

During past decades, many research methods of progressive collapse have been proposed. Generally, research methods of progressive collapse are divided into direct and indirect method. Leyendecker and Ellingwood (1977) reviewed different methods of progressive collapse analysis, and proposed a probabilistic analysis method based on statistics and the Monte Carlo method. The authors defined direct design as consideration of resistance to progressive collapse, and defined indirect design as specification of minimum levels of strength, continuity, and ductility.

Direct methods are ways to model the loading scenario directly during collapse. These methods can be used to quantify the behavior of buildings during extreme events, and then to decide whether the structure is resilient towards progressive collapse. Direct methods are separated into local resistance methods and collapse simulation. The goal of the local resistance method is to prevent the initial damage from happening for a given hazard. This can stop disproportionate failures due

to the removal of collapse triggers. However, since magnitudes and locations of extreme forces (e.g., gas explosion, collisions, and bomb explosions) are rarely well known, local resistance methods are limited to examination of specific scenarios and are not always practical.

Indirect methods are methods whereby some metric of structural behavior (for example, ductility, or the presence of tie forces) is calculated. This method provides some designing details and criteria (for example, provide a minimum reinforcement ratio or give a minimum strength requirement for vulnerable elements) to increase robustness of buildings. If the structure passes these checks, then no direct analysis is needed; the structure is assumed to be resilient with respect to collapse. Byfield (2006) investigated progressive collapse robustness via indirect methods. He investigated the resiliency of buildings designed with conventional nonseismic design methods and subjected to bomb explosions in World War II. He noted that weak beam-column connections was the main reason for progressive collapse, and strengthening beam-column connections near the probable locations of first damage led to better resistance to progressive collapse.

The most common direct method for collapse simulation is the alternate load path method, which has been widely used in recent research (e.g., Le and Xue, 2014; Bao et al., 2008; Ruth et al., 2006), and is also recommended in the present guidelines (GSA, 2003, 2013; Department of Defense, 2009). The alternate load path method is performed by removing one or more major bearing elements, and analyzing the remaining structure to determine if this initiating damage propagates. This method is event independent, such that the solution is adequate for any hazard that could cause the prescribed member loss. General Services Administration (2003; 2013) recommended the alternate load path method, including linear static, nonlinear static, linear dynamic, and nonlinear dynamic analysis methods. Marjanishvili (2004) compared the different alternate load

path methods in GSA (2003) provisions, and recommended performing progressive collapse analysis starting from static methodology to methods of increasing complexity until the probability of progressive collapse was low enough or all methods were exhausted. Fu (2009) used ABAQUS to model a 20-story building with the finite element method. His model method was validated by comparing modeling outcomes and experimental data of a two-story composite steel frame. A nonlinear dynamic analysis for progressive collapse following GSA (2003) provisions was done, and Fu noted the dynamic response of the structure was mainly related to the affected loading area after the column removal. Kokot et al. (2012) applied progressive collapse analysis using the linear static procedure, linear dynamic procedure and nonlinear dynamic procedure with a 3-story reinforced concrete flat-slab frame. Simulation results using ABAQUS were compared with data from a real-scale experiment of the same building. They concluded that the static procedures were more conservative than the dynamic procedures. Because of its wide use and acceptance by the civil structural collapse community, the alternate load path method was chosen for investigation of collapse for this research report.

To analyze progressive collapse using alternate load path analysis, the trigger or initial damage needs to be simulated properly. A common technique for simulating damage is to remove from the structure the most vulnerable elements or the most probable places for abnormal sabotage. In performing alternate load path analysis, Murtha-Smith (1988) considered locations that held the highest loads for removal. According to Chapter 3.2.9.2 of the GSA Guidelines (2013), as a minimum, corner and middle load bearing elements along sides should be first considered for removal in progressive collapse. This GSA removal criterion is commonly applied by many researchers (e.g., Fu, 2009; Kokot et al., 2012; Kwasniewski, 2010).

1.2 Comparison of alternate load path methods

Within the alternate load path method, there are four main analysis procedures based on GSA (2003) and UFC (Department of Defense, 2009) provisions: linear-elastic static (LSP), nonlinear static (NSP), linear-elastic dynamic (LDP), nonlinear dynamic (NDP) procedures. The accuracy, calculation time, and difficulties in modeling increase from LSP to NDP. GSA (2013) deleted the LDP procedure, as the extra efforts in performing the dynamic analysis was not believed to offer significant improvements in accuracy above linear or nonlinear static methods.

GSA (2013) is the latest design guidance for progressive collapse. Compared to GSA provisions in 2003, GSA (2013) provides detailed guidance and clear definitions for collapse simulation using the alternate load path method. Therefore, GSA (2013) is addressed in this section.

1.2.1 Linear-elastic static

The linear-elastic static procedure (LSP) models structures with purely linear materials analyzed in static loading scenarios. In GSA (2013) provisions, the use of LSP is limited to structures that are ten stories or fewer. Vertical and lateral stability of elements needs to be considered, though P-Δ effects can be ignored because of small deformations. Dynamic amplification factors are used to magnify the load and compensate for dynamic behaviors in static analysis. A demand-resistance ratio (DRR), also known as the demand-capacity ratio (DCR), is used to decide whether LSP is allowable in design. Reinforced concrete buildings and steel structures have different dynamic behaviors and material properties, and hence DRR and dynamic amplification factors are different for each case.

The dynamic amplification factor is the factor used to increase loads to compensate for dynamic effects in static analysis procedures. In GSA (2013), the dynamic amplification factor is applied only to gravity loads for floor areas above the removed column or wall. The value of the dynamic amplification factor is shown in Table 1.1.

Table 1.1 Load dynamic amplification factor for LSP in GSA 2013

Material	Structure type	Deformation-controlled	Force-controlled
Steel	Framed	$0.9\ m_{LIF} + 1.1$	2.0
Reinforced concrete	Framed	$1.2\ m_{LIF} + 0.80$	2.0
	Load-bearing wall	$2.0\ m_{LIF}$	2.0
Masonry	Load-bearing wall	$2.0\ m_{LIF}$	2.0
Wood	Load-bearing wall	$2.0\ m_{LIF}$	2.0
Cold-formed steel	Load-bearing wall	$2.0\ m_{LIF}$	2.0

In Table 1.1, m_{LIF} is the smallest m of any primary beam, girder, spandrel or wall element that is directly connected to the columns or walls directly above the column or wall removal location, where m is the component or element demand modifier defined in GSA (2013) based on different categories of members and connections. The value for the dynamic amplification factor originates from the idea that the dynamic displacement of an instantaneously applied constant load in linear analysis is twice the displacement achieved when the load is applied statically (McKay et al. 2012).

Following removal of the damaged component, forces in every member are computed. The DRR limit is used as the criterion to decide whether LSP is acceptable in progressive collapse analysis. According to Kokot et al. (2012), the DRR is defined in equation (1.1).

$$\text{DRR} = \begin{cases} \frac{M_{max}}{M_r} & \text{in beams (bending moment only)} \\ \frac{N_{max}}{N_r} & \text{in bars (axial force only)} \\ \frac{M_{max}}{M_r(N)} & \text{in columns (combined bending moment and axial force)} \end{cases} \tag{1.1}$$

M_{max} and N_{max} are the maximum internal moment and axial force, respectively, and M_r and N_r are the moment and axial force resistance, respectively. If the largest DRR for any reinforced concrete member is greater than 2 or the structure is irregular, LSP cannot be used (GSA, 2013). The limit 2 for DRR is to account for moment and force redistribution capacity of buildings.

If LSP can be used, the strength of each member should be checked. The structure is not susceptible to progressive collapse provided all members pass a strength check. The strength check is performed separately for force-controlled and displacement-controlled actions as follows:

Force-controlled actions:

$$\Phi Q_{CL} \geq Q_{UF} \tag{1.2}$$

Q_{UF} is the reaction of a component or element for force-controlled action. Q_{CL} is the lower-bound strength of the component or element. Φ is the strength reduction factor from the appropriate material-specific code.

Displacement-controlled actions:

$$\Phi m Q_{CE} \geq Q_{UD} \tag{1.3}$$

Q_{UD} is the reaction of a component or element for displacement-controlled action. Q_{CE} is the expected strength of the component or element. Factor Φ is the strength reduction factor from the appropriate material-specific code. Factor m is the component or element demand modifier, defined in ASCE 41-06.

1.2.2 Nonlinear Static

Nonlinear static (NSP) analysis is identical to linear-elastic static analysis, except that material and geometric nonlinearities are allowed.

For nonlinear static analysis, GSA 2013 guidelines specify different amplification factors for different members as shown in Table 1.2. θ_{pra} is the plastic rotation angle given in the acceptance criteria tables in ASCE 41-06 and GSA 2013 for the appropriate structural response level (Collapse Prevention or Life Safety) for the particular element, component or connection; θ_y is the yield rotation. The load amplification factor is applied only to increase gravity loads for floor areas above the removed column or wall.

Table 1.2 Load dynamic amplification factor for NSP in GSA 2013

Material	Structure type	Load amplification factor
Steel	Framed	$1.08 + 0.76/(\theta_{pra}/\theta_y + 0.83)$
Reinforced concrete	Framed	$1.04 + 0.45/(\theta_{pra}/\theta_y + 0.48)$
	Load-bearing wall	2.0
Masonry	Load-bearing wall	2.0
Wood	Load-bearing wall	2.0
Cold-formed steel	Load-bearing wall	2.0

In GSA 2013, the acceptance criteria of NSP for force-controlled actions and displacement-controlled actions are the same as ones for LSP, which are shown in Equation (1.2) and Equation (1.3).

1.2.3 Dynamic analysis procedures

The nonlinear dynamic procedure (NDP), though the most complicated one among all alternate load path procedures, provides the most accurate simulations. Firstly, the structure is loaded statically with gravity prior to initiation of damage to compute reactions. Then a corresponding model is built in which the damaged member is removed and replaced by the corresponding reactions. The damaged member reactions are then instantaneously removed, and the dynamic analysis is conducted. A quasi-static removal of the reaction forces over a duration less than 1/10 of the period of the structure is also acceptable. Compared with LSP and NSP, no dynamic amplification factor or DRR is needed. The acceptance criteria of NDP for force-controlled actions and displacement-controlled actions are the same as ones for LSP, which are shown in Equation (1.2) and Equation (1.3).

1.2.4 Discussion

Static analysis can be easily applied using software with short calculation time to get approximate results. However, applying a dynamic amplification factor equal to 2 is arguable. Marjanishvili and Agnew (2006) compared results from all four procedures for progressive collapse analysis in GSA provisions (2003) for a nine-story steel moment frame building. They noted that the limit for DRR in static analysis should be 2 instead of 3 as originally specified in GSA (2003) provisions for non-slender steel components, and the dynamic amplification factor 2 was conservative enough. Ruth et al. (2006) analyzed 11 different models to check the dynamic amplification method as 2 in GSA (2003) provisions, and compared the results from static analysis with ones from dynamic analysis. They commented that using 1.5 might be more appropriate than 2 as the dynamic amplification factor. McKay et al. (2012) looked into a variety of reinforced concrete and steel

moment-frame buildings to investigate the magnitude for dynamic amplification factors, and noted the factor for RC buildings ranged from 1.05 to 1.75, and for steel buildings ranged from 1.2 to 1.8. Although the dynamic amplification factor is variable in GSA (2013) provisions, the value is still nearly 2 as shown in Table 1.1.

NSP is a desirable procedure in progressive collapse analysis. Compared to dynamic analysis, nonlinear static analysis reduces calculation time tremendously while also including material and geometric nonlinearity, unlike linear static analysis. Nevertheless, NSP results cannot provide a detailed representation of structural behaviors. The static methods are quick checks for progressive collapse without details of structures' dynamic behaviors. Compared with static methods, dynamic analysis inherently contains dynamic influence, damping and inertia, which make simulations more accurate (Marjanishvili and Agnew 2006).

The accuracy of nonlinear dynamic analysis makes it ideal for research regarding progressive collapse, and NDP is capable of providing detailed behaviors of structures (Bao et al. 2008; Marjanishvili and Agnew 2006; Kwasniewski and Leslaw 2010; Fu and Feng 2009). In practice, however, the analytical complexities and computational time required for nonlinear dynamics are not always feasible in engineering design. Simplification of nonlinear analysis that retains the increased accuracy but reduces complexity is desirable.

1.3 Simplifications of plastic behavior

Material plasticity, though complex, is one of the necessary factors to be considered to obtain accurate and realistic results from collapse simulation. The finite element method is a widely used method, and its accuracy in progressive collapse analysis has been verified with laboratory

experiments (Kwasniewski, 2010; Fu, 2009). While detailed finite element analysis may provide good results, these detailed analyses are time-consuming, with computational times on the order of days for complicated structures. Kwasniewski (2010) simulated an 8-story steel frame building with the finite element method. He analyzed progressive collapse with the nonlinear dynamic method following GSA provisions, and validated this GSA method with experimental data and detailed FE simulation analysis. For his analysis, the total calculation time using 60 processors was 19 days. Hence, models need to be simplified to make the progressive collapse analysis more practical, though simplification necessarily involves some sacrifice of accuracy. Simplifying dynamic analysis to static ones will improve efficiency, though static analysis neglects the dynamic nature of progressive collapse. Alternatively, the simulation can be simplified by approximating material properties (Marjanishvili, 2004; Marjanishvili and Agnew, 2006; Ruth et al., 2006).

The lumped plasticity method and fiber element method are two popular methods for simplifying the nonlinear and hysteretic behavior of structural elements. In fiber models, structural members are divided into longitudinal fibers. The sectional response is not specified explicitly, but is derived by integration of the response of the fibers, which follows a uniaxial stress-strain relation of the particular material, over the cross section (Taucer et al., 1991). Outcomes of fiber models depend on the number of integration points along the length of a fiber, as well as the meshing and number of fibers through each cross section.

To avoid mesh dependency, the lumped plasticity method is explored in this research instead of the fiber element method. For the lumped plasticity method, nonlinear plastic behavior is lumped into particular zones while all other parts of the model are assumed to remain elastic. Le and Xue

(2014) presented a two-scale numerical model. Coarse-scale cohesive elements represented the potential damage zones in beams, columns and joint panels. The responses of the cohesive elements were available from fine-scale FE simulations. During loading and unloading stages, the modulus was represented as a linear function of a damage factor, for which 1 meant total failure and 0 meant no damage. Meyer and Roufaiel (1983) modified the bi-linear Takeda (1970) model to simulate a fixed-fixed beam. The left and right ends of the beam were considered with changing stiffness while the middle part was elastic. The plastic hinge length was assumed to be equal to the distance from the end of the element to the first section that had an internal moment equal to the yield moment when the maximum loading for each loading loop was applied to the beam. During each loading loop, plastic hinge lengths were recalculated according to the largest load exerted on the beam during that loop. A bilinear moment-curvature relation represented stiffness at each loading stage. Hence, stiffness of elements was represented as functions of the plastic hinge length and stiffness at both ends by direct integration method. Keshavarzian and Schnobrich (1985) expanded Meyer and Roufaiel's (1983) model to reinforced coupled shear walls, adding consideration of axial force to the behaviors of sections. Another method, called the triaxial spring model and shown schematically in Figure 1.1, was developed by Lai and Will (1986). Lai and Will modeled reinforced concrete columns subjected to biaxial bending with constant and varying axial loads using a lumped plasticity method. The lumped plastic zones used zero length plastic hinges to account for plasticity, and each zone (connection) comprised a certain number of effective concrete and steel springs. The uniaxial constitutive law was generalized to a tri-linear moment-curvature relation over the cross section, which accounted for behavior prior to cracking, from cracking to yielding, and from yielding to ultimate. Tributary areas of springs affected the properties of springs. The model was able to represent biaxial bending with stiffness degradation

and variable axial force, and was validated by comparing simulation results of a 10-story RC building with experimental results.

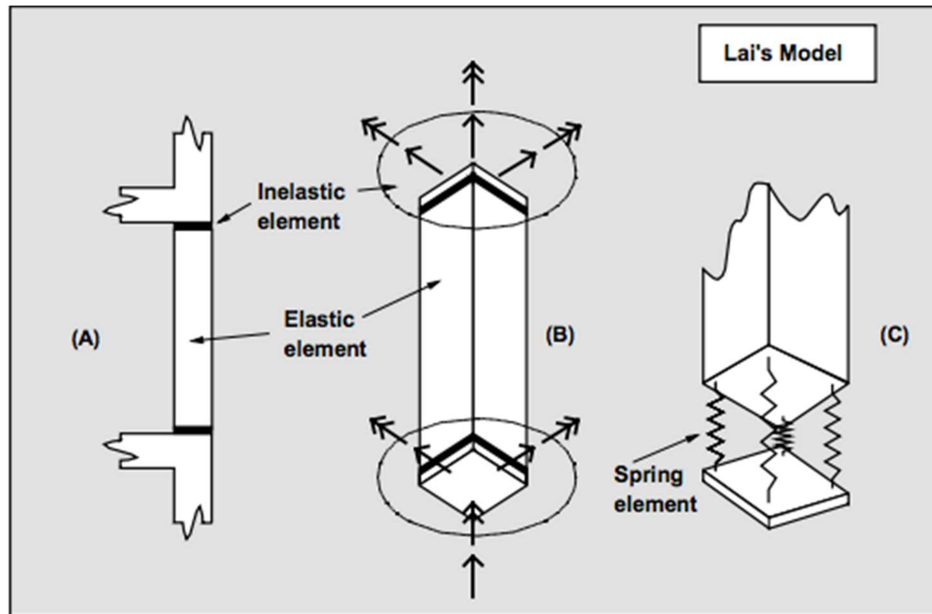


Figure 1.1 Degrading inelastic element for reinforced concrete beam-columns under biaxial bending and axial load (Lai and Will, 1986)

Another lumped plasticity method, including a combination of continuum damage and fracture mechanics, is proposed for this research. This method was developed by Cipollina et al. (1995), and expanded in detail by Flórez-López (1995), Perdomo et al. (1999), and Marante et al. (2002; 2003). This lumped damaged-plasticity method lumps both plasticity and damage at both ends of elements, and considers the remainder of the element as elastic. This lumped damaged-plasticity method can quickly simulate flexural capabilities, though it should be capable of incorporating other theories and actions necessary for progressive collapse analysis. In considering collapse simulation, the current form of this lumped damaged-plasticity model ignores catenary action, effects of falling elements, and removal of ultimately damaged elements in progressive collapse analysis. This method is discussed in more detail in Chapter 2.

1.4 Catenary action

In addition to the material nonlinearity, geometric nonlinearity should also be considered in progressive collapse analysis. The reason is that geometric nonlinearity introduces catenary action into analysis, which is an important collapse resistance mechanism after flexural failure. After flexural failure, horizontal tension in members can provide extra resistance to vertical loadings.

Stinger and Orton (2013) conducted disproportionate collapse experiments on a frame with discontinuous reinforcement, a frame with continuous reinforcement, and a frame with partial-height infill walls. The frame with continuous reinforcement had greater flexural action resistance compared to the frame with discontinuous reinforcement. However, the final collapse resistances of the two frames were similar. This was because the catenary action resistance, which is the controlling resistance mechanism for high collapse loads after flexural failure, of the two frames were similar. Bao et al. (2011) experimentally tested and simulated two reinforced concrete beam-column assemblies, each comprising three columns and two beams. Two full-scale assemblies were tested through monotonically increasing vertical displacement of the center column to simulate removal of column. Both the simulation and test confirmed the main resistance with the column removal scenario was from catenary action. Yi et al. (2008) applied a static test on a four-bay and three-story reinforced concrete frame after removing the center column on the first floor. The experiment suggested that three phases existed for reinforced concrete frames in progressive collapse analysis: elastic, plastic and catenary phases, and the final resistance mechanism was dominated by a beam catenary mechanism.

Efforts have been made by past researchers to calculate the catenary action resistance. Yi et al. (2008) proposed a simplified method to calculate the load-carrying capacity of catenary action, which is stated in Equation (1.4) and described schematically in Figure 1.2.

$$P_c = 2\psi N \sin\alpha \tag{1.4}$$

The force P_c in the equation is the vertical loading resistance from catenary action. The constant ψ is the strain adjustment coefficient, and α is the beam rotational angle. Yi et al. (2008) assumed a value of 0.85 for ψ.

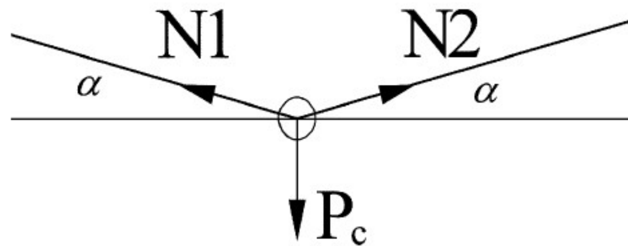


Figure 1.2 Simplified model by Yi et al. (2008) to calculate catenary action

Izzuddin (2005) proposed a simplified model for axially restrained beams subject to extreme loading. The beam he analyzed was a simply supported steel beam with elastic axial restraint at the supports. A midspan point load (MPL) and a uniformly distributed load (UDL) were discussed. The beam's configuration is shown in Figure 1.3. Izzuddin assumed the plastic response of the beam was governed by a plastic hinge at midspan, and the mechanism is displayed in Figure 1.4.

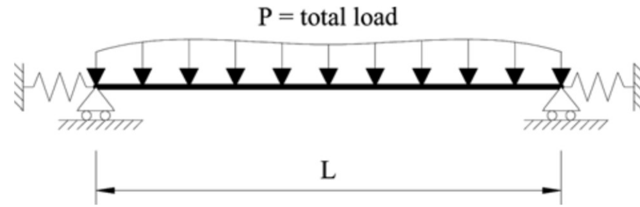


Figure 1.3 Simply supported beam with linear elastic axial restraint (Izzuddin, 2005)

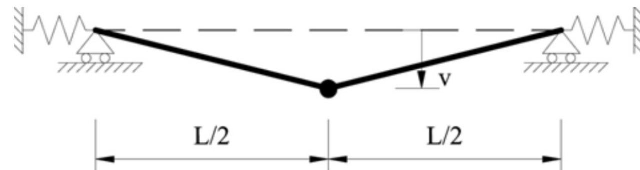


Figure 1.4 Plastic mechanism with a plastic hinge at midspan (Izzuddin, 2005)

Izzuddin (2005) assumed the axial force and bending moment capacities at the midspan of the steel beam had linear interaction as shown in Figure 1.5. The elasto-plastic response of an axially restrained steel beam under transverse loading was assumed to consist of 4 stages as shown in Figure 1.6: elastic, plastic bending, transient catenary, and final catenary stages.

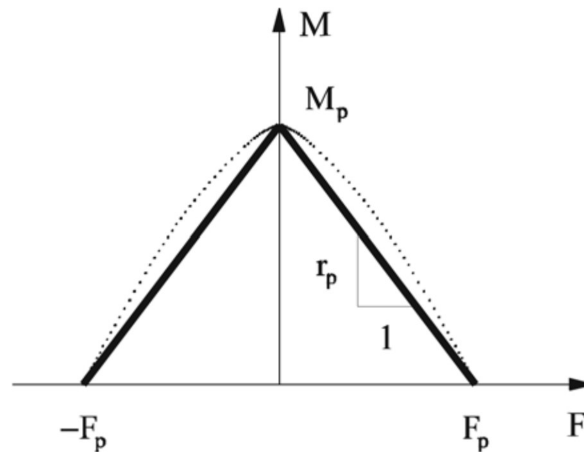


Figure 1.5 Plastic interaction of axial force and bending moment (Izzuddin, 2005)

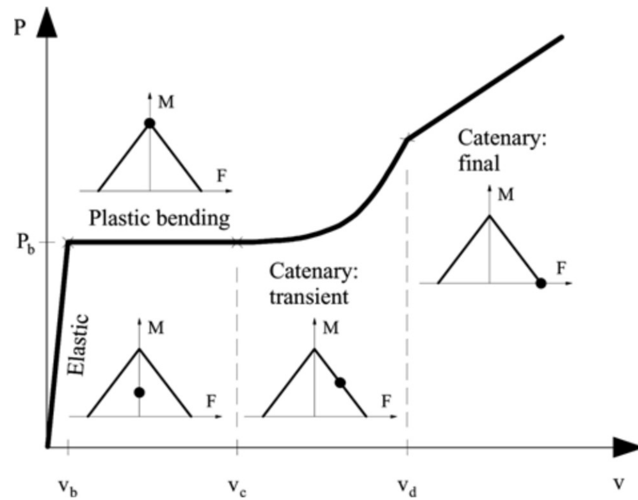


Figure 1.6 loading stages of elasto-plastic response (Izzuddin, 2005)

Moment M_p is the bending moment at yield, F is the axial force of the steel beam, and F_p represents the ultimate axial force. Force P is the total vertical force applied to the beam, and v is the vertical displacement at midspan. In the plastic interaction of axial force and bending moment, Izzuddin (2005) assumed that no axial force appeared during the elastic and plastic bending stages. This was analogous to the applied load moving from the origin of Figure 1.5 up along the vertical axis to the yield moment M_p. In the transient catenary stage, axial support extension was assumed as a second-order function in terms of the midspan vertical displacement. The point in Figure 1.5 started to move toward point (F_p,0) along the failure surface. The axial force in the beam reached and held the ultimate axial force F_p in the final catenary stage until axial fracture of the steel beam.

To analyze catenary action in collapse analysis of reinforced concrete frames, the method proposed by Yi et al. (2008) is reasonable, though the strain adjustment ψ is difficult to determine without experimental support. The method of Izzuddin (2005) is simple to apply, but is mainly applicable to steel beams. Reinforced concrete beams consist of both concrete and steel, and their bonding reactions make RC beams different from steel beams during catenary action analysis. Furthermore,

RC beams exhibit arching action, which is the development of net compression in an axially restrained section as the beam cracks in flexure but prior to the large deflections under which catenary actions begin to control. In this paper, a simplified method to analyze catenary action is proposed and discussed in Section 2.4.

1.5 Outline

The objective of this research was to develop the lumped damaged-plasticity method as an efficient method in progressive collapse analysis that accurately captures resistance of structures subject to disproportionate collapse. Flexural capacities and catenary action were considered in the analysis. The research methodology began with the lumped damaged-plasticity method developed by Cipollina et al. (1995), and expanded it to analyze progressive collapse of 2-D reinforced concrete frame structures. The original lumped damaged-plasticity could only simulate flexural behaviors of members, so catenary actions and flexural failures were introduced into the models. The revised lumped damaged-plasticity method was done via OpenSees (the Open System for Earthquake Engineering Simulation) code; the code is attached in the Appendix. The accuracy and efficiency of this method were checked with experimental data and results from other simulation methods. The thesis is organized as follows: in Chapter 2 theories of the lumped damaged-plasticity method and catenary actions are discussed; in Chapter 3 the lumped damaged-plasticity model is verified and validated using a RC cantilever beam; in Chapter 4 static and dynamic progressive collapse analyses of a particular RC frame are presented; and in Chapter 5 conclusions and directions for future research are given.

Chapter 2. Lumped damaged-plasticity method

The lumped damaged-plasticity method is based on a simplification of continuum damage mechanics, which introduces a damage variable. This damage variable abstracts damage at the microscale, such as the density of microcracks and microvoids, and its influence on the behavior of the material at the continuum level (Cipollina et al., 1993). Unlike theories of continuum damage mechanics, in the following model all the damage and plasticity in a beam or column element are concentrated at both ends of the element. A yield surface and a damage surface are developed to describe the evolution of plastic rotation and damage. The lumped damaged-plasticity model is described thoroughly in this section.

2.1 Basic idea of damage variables

The damage variables used in lumped damaged-plasticity originate from the principle of stress equivalence. The stress associated with a damage state under the applied strain is equivalent with the stress associated with an undamaged state under an effective strain (Mariano, 2000). This principle of stress equivalence is shown in Equation (2.1):

$$\bar{\sigma} = \sigma / (1 - d) \tag{2.1}$$

The strain ε is represented as

$$\varepsilon = \bar{\sigma} / E = \sigma / \left[\left(1 - d\right) E \right] \tag{2.2}$$

Stress $\bar{\sigma}$ is the equivalent stress, and σ is the Cauchy stress. The d is the damage variable, where $d = 0$ is equivalent to no damage, and $d = 1$ means completely destructed.

Reinforced concrete sections crack after the moments reach the cracking moment M_{cr}. After cracking, the moment of inertia of the section decreases, which leads to deterioration of stiffness. The principle of stress equivalence, widely used in continuum damage modeling, can be extended to describe the degradation of the sectional stiffness from intact to cracking, and finally up to failure. The lumped damaged-plasticity method concentrates all damage along the length of the element to points at both ends. Equivalence equations and sectional response equations for a lumped damaged-plasticity method are discussed in Section 2.2.

2.2 Equivalence and constitutive equations

This research focuses on 2D reinforced concrete elements during progressive collapse analysis. In the analysis, equivalence equations and sectional response equations (analogous to constitutive relations in continuum mechanics) are necessary to compute flexural resistances of elements. The equivalence equation results from the principle of stress equivalence in damage mechanics, and equations for the sectional response are developed starting with the yield surface for plasticity and the Griffith criterion for damage.

2.2.1 Equivalence equations

Damage for lumped damage-plasticity finite element formulations is introduced in the element flexibility matrix. In reinforced concrete analysis, damage variables and plastic rotations for negative and positive loads are assumed to be independent. For a 2-D beam element, assuming the two ends of the element are defined as end i and end j separately, the flexibility matrix $\mathbf{F}$ is

$$\mathbf{F}=\begin{bmatrix}\dfrac{F_{11}^0}{1-d_i} & F_{12}^0\\ F_{21}^0 & \dfrac{F_{22}^0}{1-d_j}\end{bmatrix} \tag{2.3}$$

F^0_{11}, F^0_{12}, F^0_{21}, F^0_{22} are the entries of the flexibility matrix for linear elastic FE beam elements. The d_i and d_j are damage variables at each end of the beam element. Damage can be defined separately for both the positive and negative directions, such that d_i^+ and d_j^+ are used in positive loading, and d_i^- and d_j^- are used in negative loading. The sign convention for loads and moments is the same as for traditional FE methods. Considering plastic rotations, the equivalence equation for a beam element with lumped damaged-plasticity model then becomes:

$$\mathbf{F}\cdot\mathbf{M}-\boldsymbol{\theta}^{\mathbf{p}}=\begin{bmatrix}\dfrac{F_{11}^0}{1-d_i} & F_{12}^0\\ F_{21}^0 & \dfrac{F_{22}^0}{1-d_j}\end{bmatrix}\begin{Bmatrix}M_i\\ M_j\end{Bmatrix}-\boldsymbol{\theta}^{\mathbf{p}}=\begin{Bmatrix}\theta_i-\theta_i^p\\ \theta_j-\theta_j^p\end{Bmatrix} \tag{2.4}$$

Moments M_i and M_j are the moments at end i and end j, respectively. Rotations θ_i and θ_j are the total rotations at each end, and θ_i^p and θ_j^p are the plastic rotations at each end. This equivalence equation is derived from the principle of stress equivalence in Section 2.1. Similar with damage variables, θ_i^{p+} and θ_j^{p+} may be used to represent plastic rotations with positive moments, and θ_i^{p-} and θ_j^{p-} may be substituted when negative rotations are applied. The total rotations of the lumped damaged-plasticity modelled element can be separated into three parts: the elastic part $\boldsymbol{\theta}_{\mathbf{e}}$, the damage part $\boldsymbol{\theta}_{\mathbf{d}}$, and the plastic part $\boldsymbol{\theta}_{\mathbf{p}}$.

$$\boldsymbol{\theta}_{\mathbf{total}}=\boldsymbol{\theta}_{\mathbf{e}}+\boldsymbol{\theta}_{\mathbf{d}}+\boldsymbol{\theta}_{\mathbf{p}} \tag{2.5a}$$

$$\boldsymbol{\theta}_{\mathbf{e}} = \begin{bmatrix} F_{11}^0 & F_{12}^0 \\ F_{21}^0 & F_{22}^0 \end{bmatrix} \begin{Bmatrix} M_i \\ M_j \end{Bmatrix} \tag{2.5b}$$

$$\boldsymbol{\theta}_{\mathbf{d}} = \mathbf{F}_{\mathbf{d}} \cdot \mathbf{M} = \begin{bmatrix} \dfrac{d_i}{1-d_i} F_{11}^0 & F_{12}^0 \\ F_{21}^0 & \dfrac{d_j}{1-d_j} F_{22}^0 \end{bmatrix} \begin{Bmatrix} M_i \\ M_j \end{Bmatrix} \tag{2.5c}$$

Matrix $\mathbf{F_d}$ is the flexibility matrix for rotations from damage. No matter how plasticity and damage evolve in the analysis, this equivalence equation must always be satisfied.

2.2.2 Plasticity evolution laws

In mechanics of materials, yield conditions are described in form of a yield surface $f(\sigma)$. The set of stresses that satisfy the yield condition forms the yield surface in the stress space. Plastic flow initiates when the stress state reaches the yield surface $f(\sigma) = 0$, and the stress state remains on the yield surface as the material is yielding. Prior to yielding, $f(\sigma) < 0$. Hence, $f(\sigma) \leq 0$ must always be satisfied. To build the evolution law of plasticity, a flow rule defines the direction of plastic flow (Jirásek et al., 2002). The associated plastic flow rule describing plastic rotation in the lumped damaged-plasticity model is

$$\dot{\theta}_i^p = \dot{\lambda}_i^p \frac{\partial f_i}{\partial M_i} \tag{2.6a}$$

$$\dot{\theta}_j^p = \dot{\lambda}_j^p \frac{\partial f_j}{\partial M_j} \tag{2.6b}$$

Symbols $\dot{\theta}_i^p$ and $\dot{\theta}_j^p$ are plastic rotation rates. The f_i and f_j are, respectively, the yield surface function for each end of the element. Yield surface functions are smaller than zero when the element is in the elastic stage, and $f = 0$ with incremental plasticity. Factors $\dot{\lambda}_i^p$ and $\dot{\lambda}_j^p$ are plastic multipliers, which are defined by Cipollina et al. (1995):

$$\dot{\lambda}^p \begin{cases} = 0 & \text{if } f < 0 \text{ or } \dot{f} < 0 \text{ (no plasticity)} \\ \neq 0 & \text{if } f = 0 \text{ and } \dot{f} = 0 \text{ (plastic increment)} \end{cases} \tag{2.7}$$

Perdomo et al. (1999) proposed a form of the yield surface function for lumped damage-plasticity using continuum damage mechanics and experimental data. This function, which was used to describe plasticity in the current investigation, is given by

$$f_i = \left| \frac{M_i}{1 - d_i} - c_i \theta_i^p \right| - K_{0i} \leq 0 \tag{2.8a}$$

Or alternatively,

$$f_i = \left| M_i - (1 - d_i) c_i \theta_i^p \right| - (1 - d_i) K_{0i} \leq 0 \tag{2.8b}$$

The c_i and K_{0i} in the yield function are constants obtained from section properties of the analyzed element, and are discussed in Sections 2.2.4 and 2.3. If damage d_i and constant c_i are taken as 0, and K_{0i} is set as the yield moment M_{yi}, this yield function constitutes an elastic perfectly-plastic model. In the case where constant c_i is not 0 while damage d_i is taken as 0, this yield function becomes a bilinear model with strain hardening or softening described by the value of c_i. For the other end of the element (point j), the yield function is the same except the subscripts should be changed from i to j.

2.2.3 Damage evolution laws

Damage variables in the lumped damaged-plasticity model evolve in a similar way as plasticity. The Griffith criterion for damage is one of the first theories in fracture mechanics, and uses thermodynamic forces and energy to explain fractures. Crack propagation occurs when the released elastic strain energy is at least equal to the energy required to generate a new crack surface. Hence, a thermodynamic force conjugate to the damage parameter can be used to develop a "damage surface" similar to a yield surface. The potential energy U_d for damage is given by the following equation (Perdomo et al., 1999):

$$U_d = \frac{1}{2}\mathbf{M}^{\mathbf{T}} \cdot \mathbf{F_d} \cdot \mathbf{M} \tag{2.9a}$$

$$\mathbf{F_d} = \begin{bmatrix} \dfrac{d_i}{1-d_i} F_{11}^0 & F_{12}^0 \\ F_{21}^0 & \dfrac{d_j}{1-d_j} F_{22}^0 \end{bmatrix} \tag{2.9b}$$

Flexibility matrix $\mathbf{F_d}$ is derived from additional "damage" rotations. The thermodynamic forces, G_i and G_j, conjugate with damage are described as

$$G_i = -\frac{\partial U_d}{\partial d_i} = \frac{1}{2}\left(\frac{M_i}{1-d_i}\right)^2 F_{11}^0 \tag{2.10a}$$

$$G_j = -\frac{\partial U_d}{\partial d_j} = \frac{1}{2}\left(\frac{M_j}{1-d_j}\right)^2 F_{22}^0 \tag{2.10b}$$

Flórez-López (1995) proposed a damage function for lumped damage-plasticity using the Griffith criterion and experimental data. This function, which was used to describe damage in this current investigation, is given by

$$g_i = G_i - \left[G_{cri} + q_i \frac{\ln\left(1 - d_i\right)}{1 - d_i} \right] \leq 0 \tag{2.11a}$$

$$g_j = G_j - \left[G_{crj} + q_j \frac{\ln\left(1 - d_j\right)}{1 - d_j} \right] \leq 0 \tag{2.11b}$$

$$G_{cri} = \frac{1}{2} M_{cri}^2 F_{11}^0 \tag{2.11c}$$

$$G_{crj} = \frac{1}{2} M_{crj}^2 F_{22}^0 \tag{2.11d}$$

Thermodynamic forces G_{cri} and G_{crj} are the thermodynamic forces at cracking for each end of the element. Cracking is defined when the moment M is the cracking moment M_{cr} and damage $d = 0$. The constants q_i and q_j characterize the element, and may be considered as hardening parameters for damage. The damage function g becomes the traditional Griffith criterion if q is 0. Damage occurs and increases while the damage function $g = 0$, while no additional damage is accumulated if $g < 0$.

Resembling the plasticity flow rule, a damage flow rule is developed to describe the direction and magnitude of the damage increment. Damage increments for both end are described as $\dot{d}_i$ and $\dot{d}_j$. Factors $\dot{\lambda}_i^d$ and $\dot{\lambda}_j^d$ are the damage multipliers for each end, similar to the plastic multiplier (Cipollina et al., 1995).

$$\dot{d}_i = \dot{\lambda}_i^d \frac{\partial g_i}{\partial G_i} \tag{2.12a}$$

$$\dot{d}_j = \dot{\lambda}_j^d \frac{\partial g_j}{\partial G_j} \tag{2.12b}$$

$$\dot{\lambda}^d \begin{cases} = 0 & \text{if } g < 0 \quad \text{or} \quad \dot{g} < 0 \quad \text{(no damage)} \\ > 0 & \text{if } g = 0 \quad \text{and} \quad \dot{g} = 0 \quad \text{(damage increment)} \end{cases} \tag{2.12c}$$

2.2.4 Assumptions of the lumped damaged-plasticity method

Plasticity and damage evolution laws form the sectional response equations (i.e., the generalized constitutive relations), while the equivalence equation is the supplement in solving for damage, plasticity, and reactions of analyzed elements. The following assumptions have been made throughout this report in applying this methodology to RC frame analysis:

1. Damage and plasticity have independent energy dissipation mechanisms. Damage increments may occur with or without plasticity.

2. For the reinforced concrete member using the lumped damaged-plasticity method, cross sections of elements in analysis are assumed to be prismatic over the length of the elements. Because plasticity and damage are lumped at both ends of the finite element in the model, the lumped damaged-plasticity method may not give accurate outcomes for members with changing sections, depending on the finite element discretization.

3. Constants c, K_0, q, and G_{cr} characterize analyzed elements and section properties, such as the yield moment, cracking moment, and ultimate moment. Moments and damage indexes are

independent and separate with different subscripts for both ends i and j in one element. For each end, constants c, K_0, q, and G_{cr} are independent and have different values, but follow the same calculation procedures. The method used to calculate these constants is presented in Section 2.3.

For a beam element, the elastic flexibility matrix is

$$\mathbf{F_e} = \begin{bmatrix} \frac{L}{3E_C I} & -\frac{L}{6E_C I} \\ -\frac{L}{6E_C I} & \frac{L}{3E_C I} \end{bmatrix} \tag{2.13}$$

E_C is the elastic modulus of concrete, I is the transformed moment of inertia for sections, and L is the length of the element. The flexibility matrix **F** becomes the following matrix considering damage at both ends:

$$\mathbf{F} = \begin{bmatrix} \frac{L}{3E_C I(1-d_i)} & -\frac{L}{6E_C I} \\ -\frac{L}{6E_C I} & \frac{L}{3E_C I(1-d_j)} \end{bmatrix} \tag{2.14}$$

Because reinforcement in sections may be different along a beam element, the transformed moment of inertia I can vary. When the ultimate resistance of an element is not of interest, the gross moment of inertia I_g for concrete sections can be used, and the reduction of the moment of inertia after cracking can be modeled by an increase in the damage index. However, under large deformations, unconfined concrete falls off from the reinforced concrete member while the core is confined given enough lateral reinforcement. The damage evolution law proposed in this report and the corresponding damage index d are not sufficient to interpret the damage of falling unconfined concrete. Under these conditions, using the uncracked moment of inertia of only the

confined core I_{core} may be more accurate. Regardless of whether I_g or I_{core} is chosen, the constants computed in Section 2.3 ensure the sectional response retains the same cracking moment, yield moment, ultimate moment, and ultimate curvature. Hence, the choose between I_g or I_{core} subtly changes the shape of the moment-curvature curve, and primarily impacts the elastic behavior. For nonlinear progressive collapse simulation where the ultimate behavior and large deformations are of interest, I_{core} is recommended for simulations.

2.3 Calculation of constants

Plastic and damage evaluation laws of the lumped damaged-plasticity method have 4 constants at each end of the element: c, K_0, q, and G_{cr}. These constants characterize the cross-sectional behavior, and are calculated indirectly. Because the calculation procedure is identical and all factors are independent for both ends i and j in one element, only calculation at node i is shown as an example. Constants in the lumped damaged-plasticity method are derived from the section end moments under monotonic loading. In unloading stages, plastic rotations and damage indexes are fixed for unloading ends, and damage and plastic evolution laws are not involved.

Based on damage functions defined in Section 2.2.3, the damage function g_i at node i is 0 when cracking and damage propagate. Equation (2.11) shows the damage functions. For an intact concrete cross section, damage d_i is 0 when the moment at the section first reaches the cracking moment M_{cri}, and the damage function g_i is 0. Thermodynamic force G_{cri} can be computed through $g_i = 0$. The calculation is shown in Equation (2.15):

$$g_i = G_i - G_{cri} = 0 \tag{2.15a}$$

$$G_{cri} = G_i = \frac{M_{cri}^2 F_{11}^0}{2} \tag{2.15b}$$

At the ultimate loading phase for a section, the differential of the damage function g_i is 0 with no further damage increment. Furthermore, the ultimate phase is at the damage surface with $g_i = 0$, damage factor d_i is the ultimate damage index d_{ui}, and the moment is the ultimate moment M_{ui}, resulting in the following expression for the damage surface:

$$g_i = \frac{M_{ui}^2 F_{11}^0}{2(1-d_{ui})^2} - \left[G_{cri} + q_i \frac{\ln(1-d_{ui})}{1-d_{ui}} \right] = 0 \tag{2.16a}$$

Differentiating both sides of Equation (2.16a) with respect to the plastic rotation θ_i^p gives the following expression:

$$\frac{\mathrm{d}}{\mathrm{d}\theta_i^p}\left[\left(1-d_{ui}\right)^2 g_i \right] = 2M_{ui}F_{11}^0 \frac{\mathrm{d}M_i}{\mathrm{d}\theta_i^p} - \frac{\mathrm{d}}{\mathrm{d}\theta_i^p}\left[\left(1-d_{ui}\right)^2 G_{cri} + q_i\left(1-d_{ui}\right)\ln\left(1-d_{ui}\right) \right] = 0 \tag{2.16b}$$

For the ultimate loading, the section fails with no further plastic rotation or damage increments ($\frac{\mathrm{d}M_i}{\mathrm{d}\theta_i^p} = 0$ and $\frac{\mathrm{d}M_i}{\mathrm{d}d_i} = 0$). Consequently, Equation (2.16b) is simplified as Equation (2.16c):

$$2(1-d_{ui})G_{cri} + q_i[1 + \ln(1-d_{ui})] = 0 \tag{2.16c}$$

The ultimate damage d_{ui} and the constant q_i can be computed simultaneously using a nonlinear numerical solver with Equation (2.16a) and (2.16c).

Constant K_{0i} can be computed at the yield point for a section, where moment reaches the yield moment M_{yi}. In monotonic loading, the damage index d_{yi} is calculated with $g_i = 0$ at the yield point. Moments M_i and damage index d_i are substituted with M_{yi} and d_{yi}:

$$g_i = \frac{M_{yi}^2 F_{11}^0}{2(1-d_{yi})^2} - \left[G_{cri} + q_i \frac{\ln(1-d_{yi})}{1-d_{yi}} \right] = 0 \tag{2.17}$$

Given the yield moment M_{yi}, and having previously computed G_{cr} and q_i from the cracking and ultimate load states, Equation (2.17) can be solved for damage at yield d_{yi}. At the yield point, the plastic function f_i and the plastic rotation θ_i^p are both 0. Therefore,

$$f_i = M_{yi} - (1-d_{yi})K_{0i} = 0 \tag{2.18}$$

Now given d_{yi} from Equation (2.17), the constant K_{0i} can be solved directly from Equation (2.18).

Constant c_i is available after K_{0i} is available. At the ultimate phase of a section, the plastic function f_i is equal to 0, and the moment is the ultimate moment $M_{ui.}$ The plastic rotation at the ultimate stage is computed from the ultimate curvature and plastic hinge length, which are based on the length and supports of the analyzed element. In this report, the plastic hinge length is calculated by the method of Paulay and Priestley (1992), which is shown in Equation (2.19).

$$L_p = 0.08z + 0.022 d_b f_y \tag{2.19}$$

In Paulay and Priestley's method, L_p is the plastic hinge length, z is the distance from critical section to the point of contraflexure for the analyzed beam element, and d_b and f_y are the diameter and yield stress of the longitudinal reinforcement, respectively. Units of L_p, z, and d_b are inches,

and units for yield stress f_y are kips per square inch. The ultimate plastic rotation can be computed by Equation (2.20), with the ultimate curvature and yield curvature at the analyzed section. In Equation (2.20), θ_{ui}^p is the ultimate plastic rotation, and ϕ_i^y and ϕ_i^u are the yield curvature and ultimate curvature of the section at the end i.

$$\phi_{ui}^p = \phi_i^u - \phi_i^y \tag{2.20a}$$

$$\theta_{ui}^p = \phi_{ui}^p L_p \tag{2.20b}$$

Because the plastic function $f_i = 0$ at the ultimate phase, constant c_i is the only unknown in Equation (2.21).

$$f_i = M_{ui} - (1 - d_{ui}) c_i \theta_{ui}^p - (1 - d_{ui}) K_{0i} = 0 \tag{2.21}$$

All constants in the yield function and damage function at point i are derived from the preceding procedures. For the other end (point j) of the beam element, the constants can be calculated with the same procedures. If asymmetric sections exist in node i and node j, constants c, K_0, q, and G_{cr} should be computed for both positive and negative moments with separate monotonic loading reactions.

2.4 Catenary action

Catenary action is the increase of a beam's vertical loading resistance due to the development of axial tension under large deflections. While investigating the effects of carbon fiber reinforced polymers on providing continuity and resisting progressive collapse in reinforced concrete beams, Orton (2007) concluded that catenary action began after the beam had formed a failure mechanism,

meaning the beam was no longer able to sustain additional vertical loads in flexure. This conclusion achieves good agreement with experiments by Yi et al. (2008), Bao et al. (2011), and Stinger and Orton (2013). Additionally, she stated that if the beam did not have continuous reinforcement, the tension line could act through both the negative and positive moment steel if there were sufficient stirrups to transfer the tension force. A sketch showing transfer of catenary forces through stirrups is given in Figure 2.1. In Figure 2.1, detailed behaviors of compressive concrete reactions are not provided and forces from concrete are unclear. For equilibrium, compressive struts would need to be present in the concrete between the ties provided by the stirrups. Where and how these compressive struts develop may depend on the locations of the plastic hinges. Without further understanding of the concrete behavior, the efficiency and accuracy of this catenary force transfer through the stirrups is arguable.

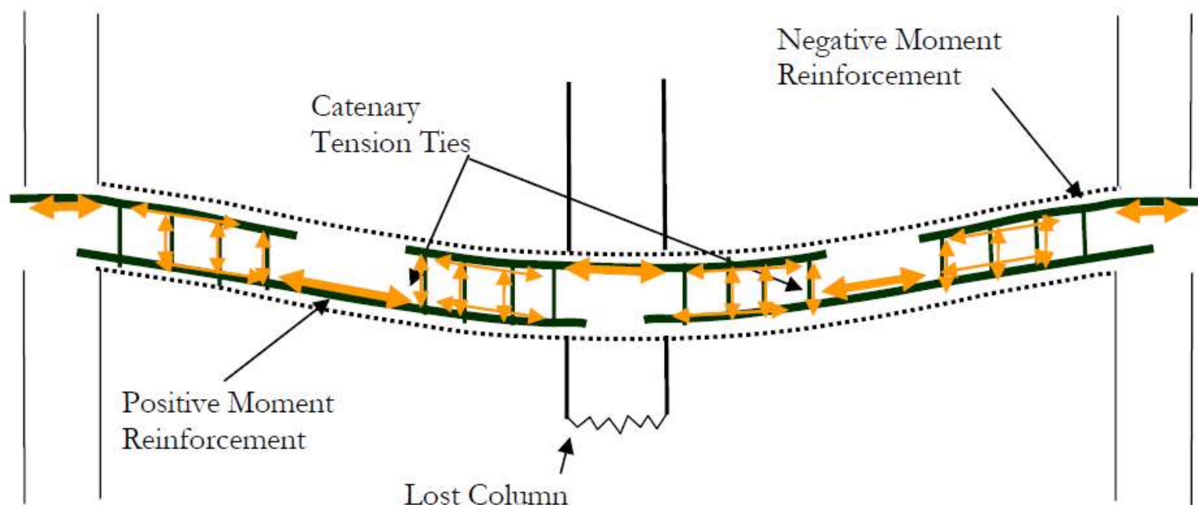


Figure 2.1 Transfer of catenary forces through stirrups (Orton, 2007)

Sasani and Kropelnicki (2008) carried out an experiment to study the behavior of a 3/8 scaled model of a continuous perimeter beam in a reinforced concrete frame structure following the removal of a supporting column. The beam specimen was 13 ft – 8¼ in. long, 12 in. wide, and 20

in. deep. One end of the studied beam was fixed, and rotation and horizontal displacement were constrained for the other end. Monotonically increasing vertical displacement was applied at one end of the beam in a static experiment to check the beam capacity and the development of different resisting mechanisms. After the experiment on the beam specimen, a detailed finite element model was developed. To simulate the whole building, a model with beam-column and shell elements was applied. A sudden removal of column was applied on the model of building, and dynamic nonlinear analysis was used in research. The force-displacement relationship from the static experiment is shown in Figure 2.2.

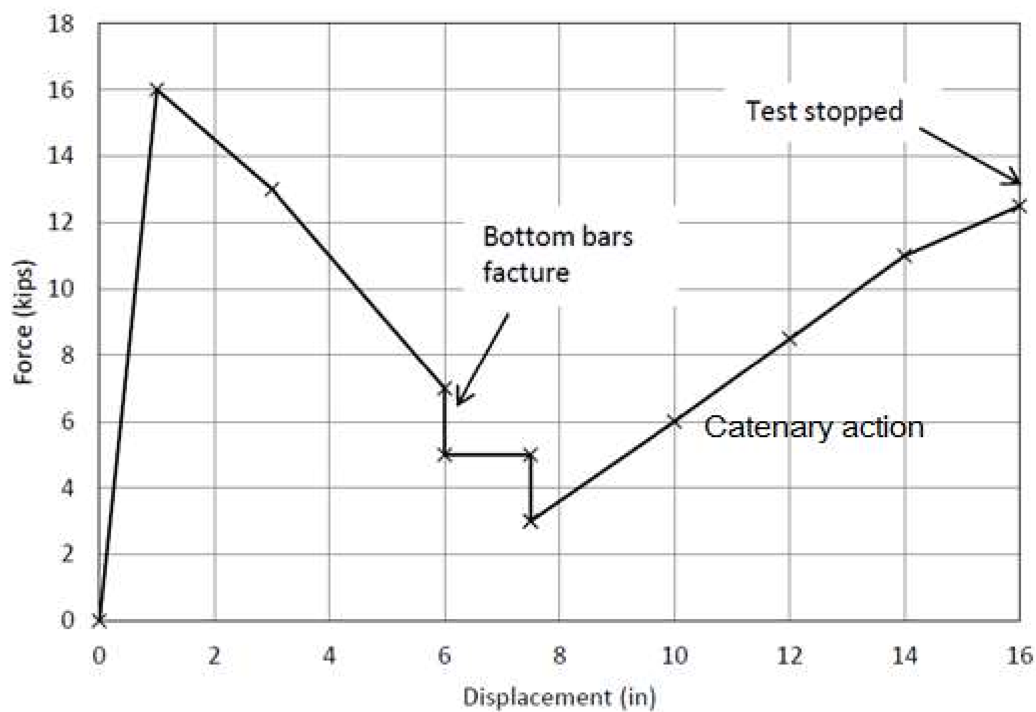


Figure 2.2 Force-displacement relationship (Sasani and Kropelnicki, 2008)

During the static experiment by Sasani and Kropelnicki (2008), flexural bars (bottom bars) fractured at vertical displacements of 6.0 in. and 7.5 in. At a vertical displacement at 8.5 in., Sasani and Kropelnicki observed that the top continuous bars at the center of the beam, which were previously in compression, yielded in tension to provide catenary resistance.

After flexural failure of reinforced concrete beams, the concrete fails and loses bond with the reinforcement in the hinge regions. This behavior leaves reinforcement as the only resistance to axial forces in the hinge regions. For other parts of the beam, the concrete and bars are still bonded, which makes calculation of axial resistance difficult in the catenary action stage because the stress in the reinforcement is not equal along the length of the beam. Additionally, transferring axial resistance through stirrups may not be efficient (Orton, 2007), and the transferred stress is difficult to calculate. To simulate catenary actions in progressive collapse analysis for RC frame structures, some assumptions and simplifications have been made for this investigation:

1. Catenary action is assumed to start after the arching action phase, which is presumed to occur from zero displacement up until vertical displacement equals the depth of the beam. Experiments of Yi et al. (2008), Bao et al. (2011), and Stinger and Orton (2013) display little vertical loading resistance from catenary action before displacements exceed the depth of the beam, and no catenary action resistance exists in the model of Izzuddin (2008) during the plastic phase. When flexural resistance controls and arching action is present, geometric nonlinearity (which is the main cause of catenary action) has little influence on resistance because the displacement is relatively small. This report therefore suggests only calculating catenary action after the deflection is greater than or equal to the depth of the beam. Furthermore, this assumption achieves agreement with Orton's (2007) conclusion

from experiment results. Net compression developed during arching action is neglected in this study.

2. Interaction of moments and axial forces during the catenary action phase is ignored. In Section 1.4, Izzuddin (2008) assumed the existence of a transient catenary phase, and proposed a linear plastic interaction of axial force and bending moment during this phase. In the transient catenary phase, the vertical force applied on Izzuddin's beam increased as the cube of vertical displacement at midspan. His assumptions were for steel beams. However, the linear plastic interaction between axial force and bending moment is not correct for reinforced concrete beams. The loss of concrete decreases moment capacity rapidly after flexural failure, implying two discrete states in practice (i.e., flexural failure followed by catenary action) without a transition state. Ignoring any interaction between moments and axial forces at catenary action is simple to be applied.
3. For a reinforced concrete beam, compression reinforcement in flexural behaviors is assumed to be the only part to provide axial resistance in catenary action. This assumption is from outcomes and conclusions by Orton (2007), Yi et al. (2008), Sasani and Kropelnicki (2008), Bao et al. (2011), and Stinger and Orton (2013). Tension reinforcement provides no resistance because it fractures after flexural failure. If compression bars are not continuous along the length of the beam, stirrups may transfer the axial force as shown in Figure 2.1. Complete transferal of forces through the stirrups is hypothesized.
4. The axial resistance for catenary action is calculated from stress and area of compression reinforcement, and the stress can be calculated with strain and material properties. Concrete falls off in the plastic hinges areas, while the other part of the beam remains nearly intact. Therefore, the axial strain of reinforcement is important to decide the ultimate strength of

members in catenary action. Fadhil (2012) built a simplified model for reinforced concrete members subjected to progressive collapse. The model was built with a rotational spring to represent moment capacities at each end, and an axial spring to introduce axial resistance in the model during catenary action. However, the calculation of axial strain was not given, and the ultimate strength was calculated by a “desired vertical displacement” from experiments. Given limited amount of experiment data, axial deformation of compression reinforcement in this paper is assumed to be concentrated in the unbonded areas near plastic hinges. The unbonded length is assumed to extend beyond the plastic hinge region by a distance equal to development length for tension reinforcement computed in accordance with Chapter 25.4.2 in ACI 318-14. Consequently, the unbonded length L_c for each plastic hinge in catenary action is calculated by summation of L_d and plastic hinge length L_p in Equation (2.22).

$$L_c = L_d + L_p \tag{2.22}$$

Because other parts of the RC member are nearly intact, elasticity is assumed for those parts. The sketch of the assumption for catenary action in a beam element with plastic hinges at both ends is shown in Figure 2.3.

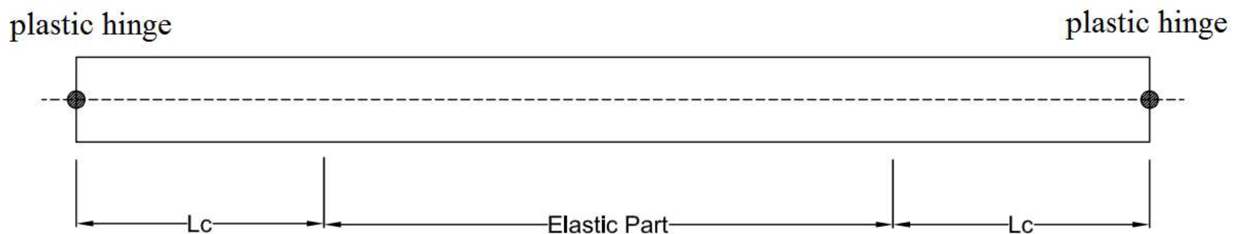


Figure 2.3 Assumption of unbonded length

Equilibrium shall be satisfied for axial force for a beam. Hence, axial forces are the same along the length of the beam. Stiffness is provided by the compressive reinforcement only in the unbonded regions, and by the uncracked reinforced concrete section in the elastic region. Material properties and equilibrium are used to calculate the ultimate axial deformation of the element, equal to the sum of the deformation over any unbonded regions and the elastic region.

If designed according to ACI 318-14, beams are usually tension-controlled. A tension-controlled section is a cross section in which the net tensile strain in the extreme tension steel at nominal strength is greater than or equal to 0.005. Tension-controlled section behaviors prevent steel fracture before failure of concrete, when the ultimate moment M_u at the section has reached. Therefore, fracture of tensile reinforcement before flexural failure is not considered in this investigation.

Chapter 3. Flexural model validation and calibration

Before applying the lumped damaged-plasticity method to simulation of progressive collapse, the methodology was verified on simple structures by comparing lumped damaged-plasticity results with more detailed methods. A cantilever reinforced concrete beam was used as the specimen, with enough stirrups to prevent the shear failure. Results from the lumped damaged-plasticity model were compared to results from fiber element modeling and detailed finite element analysis applied in the commercial finite element software ABAQUS. Because of convergence problems in ABAQUS, only flexural behaviors were compared in this chapter.

3.1 Description of the specimen

The specimen used to compare results from hand calculation, finite element method, the fiber element method and the lumped damage-plasticity method was a 200-in. long cantilever beam. Gravity was not considered, and sufficient shear reinforcement was designed to prevent shear failure. The configuration is shown in Figure 3.1.

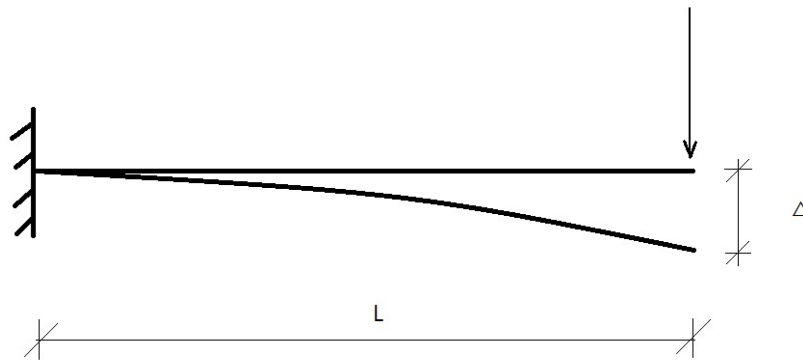


Figure 3.1 Model configuration

A displacement Δ was applied at the end of the beam with 0.01-in. increments, up to a total displacement of 3.5 in. The whole procedure was static. In order to check the unloading behavior, the displacement was decreased from 3.5 in. to –2 in. with 0.01-in. increments. Negative displacements mean upward displacements. Reloading procedure was applied at the free end from –2 in. to 1 in., with 0.01-in. increments. The displacement sequence applied at the free end is displayed in Figure 3.2.

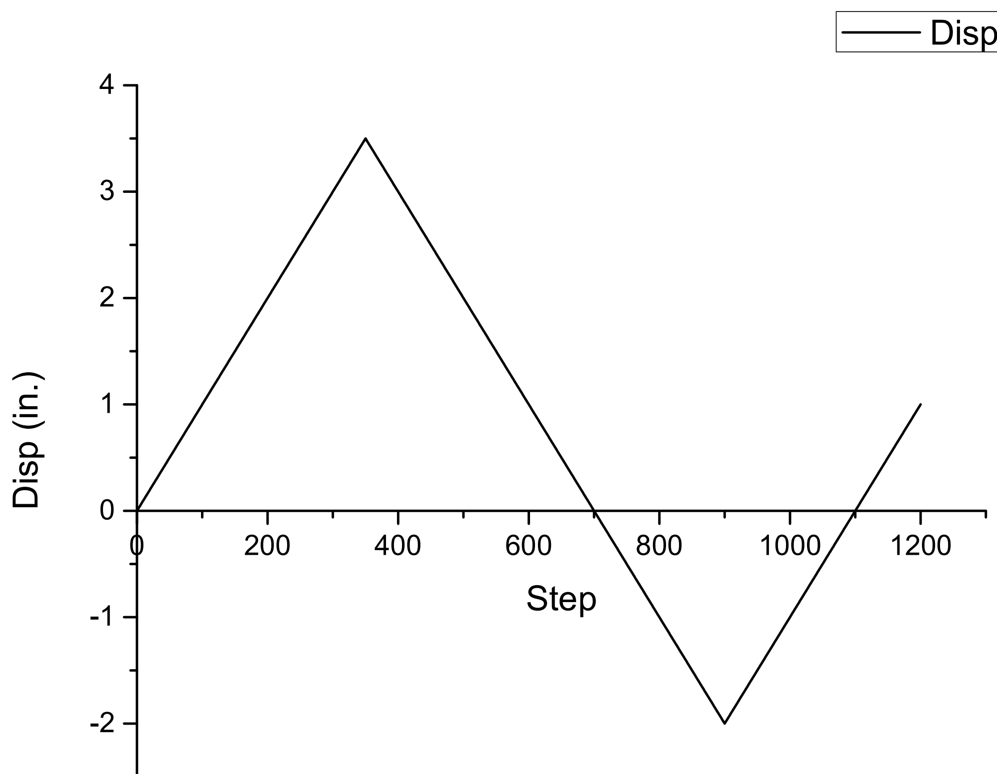


Figure 3.2 Displacement step applied at the free end of specimen

Because the displacement was relatively small compared with the length of the beam (200 in.), second-order effects (i.e., P-Δ effects) and geometric nonlinearity were ignored in simulation. The

specimen had the same cross-section and layout along its length. The section width b was 12 in. and total height h was 20 in. The distance d_b from the centroid of the tension reinforcement to the compression edge of the member was 17.5 in., and the distance d' from the centroid of the compression reinforcement to the compression edge of the member was 2.5 in. Both tension and compression reinforcement consisted of three No. 9 bars, so the area of compression bars A_s' and area of tension bars A_s were both 3 in.^{2}. The layout and dimensions of cross-sections is shown in Figure 3.3.

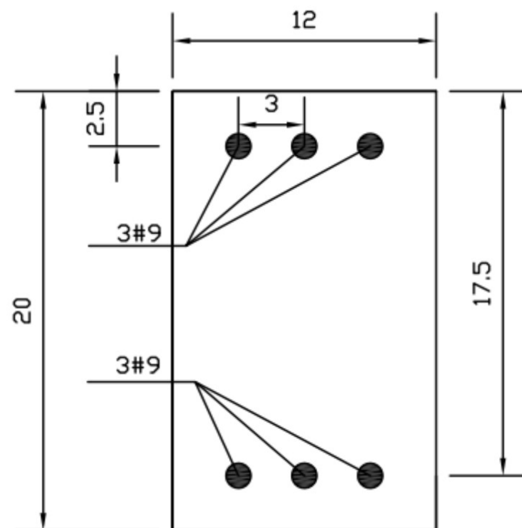


Figure 3.3 Cross-section layout and dimensions (units of inches)

Hognestad's parabola (Hognestad, 1951) was used for concrete compressive properties. Hognestad's parabola is shown in Figure 3.4. Stress f_c is the concrete stress, and ε_c is the concrete strain. Stress f_c' is the compressive strength of 6-in. by 12-in. cylinders or prisms of similar dimensions, f_c'' is the compressive strength of concrete in flexure, and E_c is the modulus of elasticity of concrete. Factors k_1 and k_2 are coefficients related to the magnitude and position of the internal compressive force in the concrete, and are calculated according to Equation (3.1).

Variable C_c is the compression force in the concrete based on value of k_1. Strain ε_0 is the compressive strain in the concrete corresponding to the maximum stress. Strain ε_u is the ultimate concrete strain in flexure, which is 0.0038 for unconfined concrete. Equation (3.2) states the function of the parabolic part in Hognestad's model. Concrete tension was assumed to develop linearly with the elastic modulus E_c until first cracking.

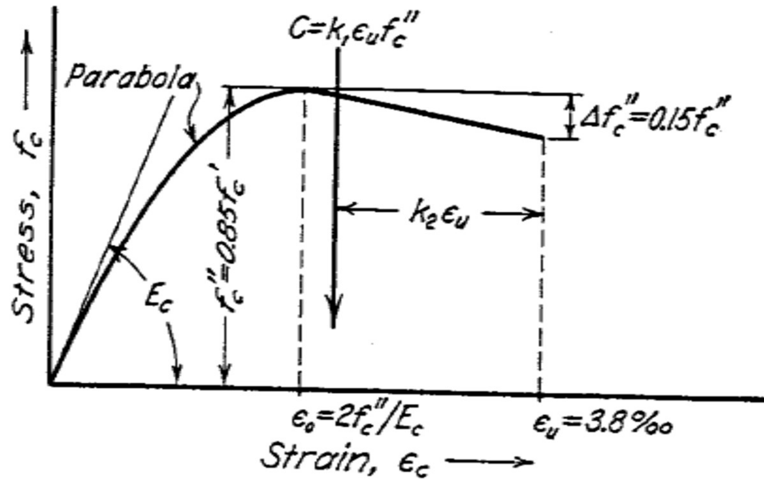


Figure 3.4 Hognestad's stress-strain diagram in flexure (Hognestad, 1951)

$$k_1 = \frac{1}{\varepsilon_c}\int_0^{\varepsilon_c}\left(\frac{f_c}{f''_c}\right)d\varepsilon_c \qquad (3.1a)$$

$$k_2 = 1 - \frac{\int_0^{\varepsilon_c}\left(\frac{f_c}{f''_c}\varepsilon_c\right)d\varepsilon_c}{\varepsilon_c\int_0^{\varepsilon_c}\left(\frac{f_c}{f''_c}\right)d\varepsilon_c} \qquad (3.1b)$$

$$f_c = f_c''\left[2\frac{\varepsilon_c}{\varepsilon_0} - \left(\frac{\varepsilon_c}{\varepsilon_0}\right)^2\right] \tag{3.2}$$

Values for concrete the material properties were assumed throughout Chapter 3, and were the same for the lumped damaged-plasticity model, the fiber element model, and the ABAQUS model. In this investigation, the cylinder strength f_c' and the compressive strength in flexure f_c'' were assumed to be equal, contrary to the indication in Figure 3.4, and both taken as 4 ksi. The modulus of elasticity of concrete E_c was calculated from Equation (3.3) based on ACI 318-14.

$$E_c = 33 w_c^{1.5} \sqrt{f_c'} \tag{3.3}$$

The density of concrete w_c was taken as 150 lbs/ft^3. Tension of concrete contributes little in flexural resistance of RC cross sections. Therefore, the concrete tensile stress was assumed to be 0 after first cracking in traditional flexural analysis. The calculation of the cracking stress was according to Equation (3.4), which is the modulus of rupture f_r from ACI 318-14. Factor λ is the modification factor to reflect the reduced mechanical properties of lightweight concrete relative to normal-weight concrete of the same compressive strength. This analysis used normal-weight concrete, so λ was equal to 1.

$$f_r = 7.5\lambda\sqrt{f_c'} \tag{3.4}$$

Sudden failure in tension caused convergence problems in the finite element software ABAQUS. Therefore, a linear tension softening model with a softening modulus E_{ts} equal to 300 ksi was used only in the ABAQUS results presented in Section 3.2.2. This tension softening model insignificantly enhanced stiffness and moment capacities M_y and M_u but improved convergence in

ABAQUS. Tensile and compressive stress-strain relation of concrete model is shown in Figure 3.5. Tensile stress and strain are positive, and compressive stress and strain are negative. Material properties of concrete used throughout Chapter 3 are summarized in Table 3.1.

Table 3.1 Material property of concrete

Property	E_c (ksi)	f_c'' (ksi)	w_c (lbs/ft^3)	f_r (psi)	ε_0	ε_u
Value	3910	4	150	515	0.0020	0.0038

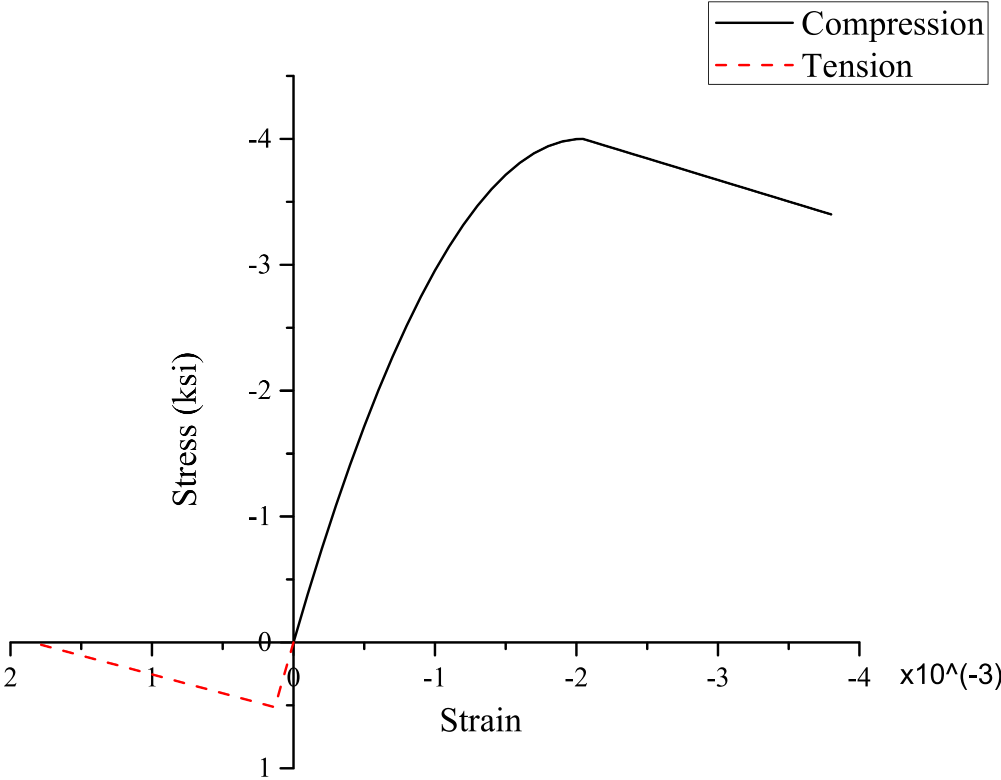


Figure 3.5 Stress-strain relation of concrete

A model with elasticity, yield plateau, and plastic hardening was used for reinforcement properties. The elastic modulus of the steel E_s was 29,000 ksi, yield stress f_y was 60 ksi, and ultimate stress f_u was 105 ksi. Before stress reached f_y, the material was elastic with E_s. During the plastic plateau stage, stress was kept as f_y until strain reached the plastic hardening strain ε_{sh} equal to 0.01. Plastic hardening started after the plastic plateau. The tangent modulus E_{sh} was 1,500 ksi at the point where strain was ε_{sh}. Another plateau was assumed from maximum plastic hardening strain ε_{sm} to ultimate strain ε_{su}. Reinforcement fractured when its strain reached ε_{su} equal to 0.1. Material properties are compiled in Table 3.2. The stress-strain relation during the plastic hardening stage and the maximum plastic hardening strain ε_{sm} were calculated by Equation (3.5), and the total stress-strain relation is shown in Figure 3.6.

$$\varepsilon_{sm} = \varepsilon_{sh} + 2 \cdot \left(\frac{f_{su} - f_y}{E_{sh}} \right) \tag{3.5a}$$

$$f_s = f_y + \left(f_{su} - f_y\right) \cdot \left[2 \frac{\varepsilon_s - \varepsilon_{sh}}{\varepsilon_{sm} - \varepsilon_{sh}} - \left(\frac{\varepsilon_s - \varepsilon_{sh}}{\varepsilon_{sm} - \varepsilon_{sh}} \right)^2 \right] \tag{3.5b}$$

Table 3.2 Material property of reinforcement

Property	E_s (ksi)	f_y (ksi)	f_u (ksi)	E_{sh} (ksi)	ε_{sh}	ε_{sm}	ε_{su}
Value	29000	60	105	1500	0.01	0.07	0.1

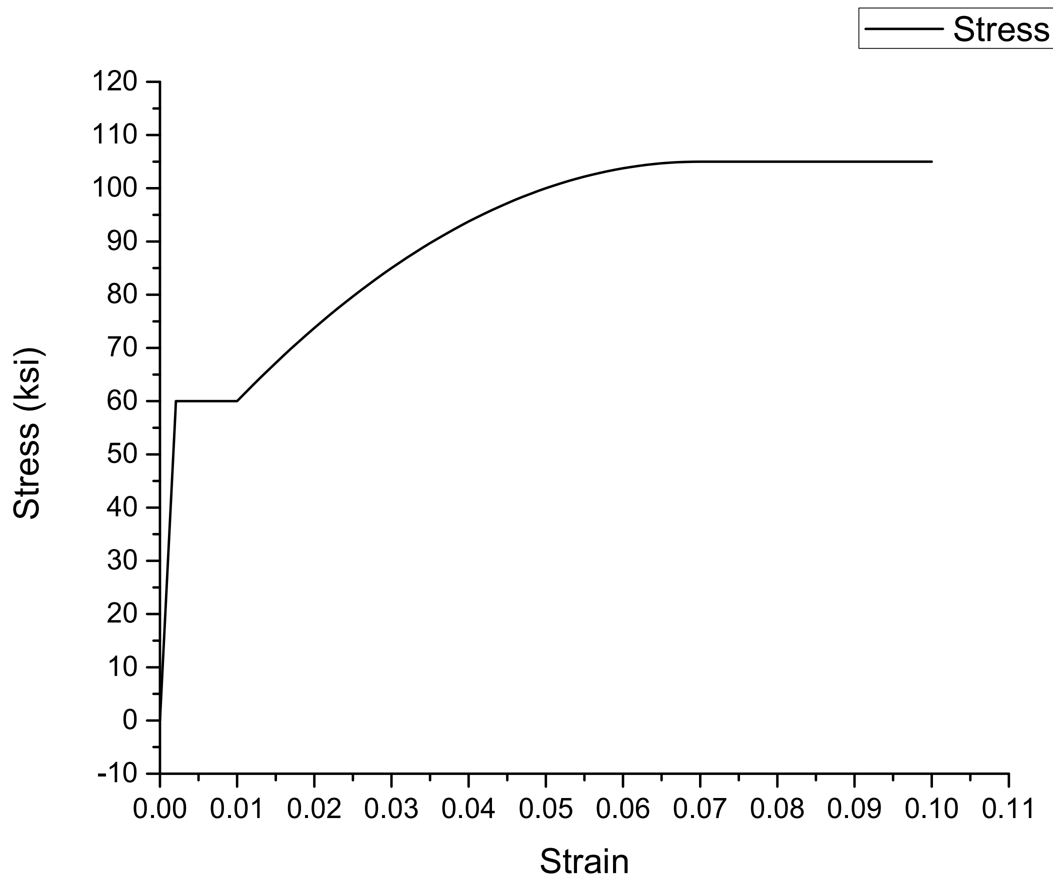


Figure 3.6 Stress-strain relation of longitudinal reinforcement

3.2 Modeling methods

To validate the lumped damaged-plasticity model, a comparison of results from the lumped damaged-plasticity model and data from other methods was conducted using the cantilever beam specimen. Hand calculations, an ABAQUS model, and a fiber element model were applied as points of comparison. The fiber element method only contained flexural behavior of the elements, and convergence problems occurred in ABAQUS when modelling total failure of concrete. Hence, only the flexural resistance of hand calculations, the lumped damaged-plasticity model, the ABAQUS model, and the fiber element model were considered in comparison.

3.2.1 Hand calculations

With material properties of reinforcement and concrete in Section 3.1, the gross moment of inertia I_g, cracking moment M_{cr}, yield moment M_y, ultimate moment M_u, yield curvature ϕ_y, ultimate curvature ϕ_u were hand-calculated. To simplify the hand calculations, concrete tension was assumed to be 0 after cracking and the post-cracking tension softening was ignored in this section. Reactions of the specimen were calculated as displacements were applied at the free end, and the sequence of displacements is given in Section 3.1. Additionally, the amount of transverse reinforcement was assumed to prevent shear failure under the largest applied displacements.

The gross moment of inertia I_g was calculated for the specimen according to the equation for rectangular sections given in Equation (3.6). Lengths b and h are, respectively, the width and height of the analyzed section.

$$I_g = \frac{1}{12} b h^3 \tag{3.6}$$

The cracking moment M_{cr} is the moment when first cracking of concrete occurs in the section. At this point, sections are at the end of the elastic phase. Hence, M_{cr} can be calculated based on the theory of elasticity. Equation (3.7) presents the computation of M_{cr}. After first cracking, concrete tension contributes little to moment capacities of sections. Tension softening is ignored in hand calculation for simplification.

$$M_{cr} = f_r \frac{I_g}{h/2} \tag{3.7}$$

Post-cracking, the theory of elasticity was not applicable because the concrete had cracked and the stress-strain relationships were nonlinear. Instead, equilibrium and compatibility were applied. Assumptions were made according to reinforced concrete design procedures (Wang et al., 2006):

1. Plane sections remain plane after bending up to failure of the beam. This assumption results in a linear strain distribution through the beam depth.
2. Reinforcing bars are bonded perfectly with concrete.

Compatibility was satisfied with the linear strain distribution. Because tension of concrete was ignored in the calculation of flexural strengths after first cracking, only the tension reinforcement provided the tension necessary to satisfy equilibrium with compression from concrete and compression steel.

Yield moment M_y is the moment when longitudinal tension reinforcement starts to yield at yield stress f_y. The assumption regarding the linear strain distribution provided the compatibility equations of strains, and was used to compute the yield curvature ϕ_y in the analyzed section. The linear strain distribution is illustrated by Figure 3.7. Strain of tension reinforcement ε_s is equal to yield strain ε_y. Calculation of ε_y is presented in Equation (3.8), and compatibility equations are given in Equation (3.9). Variable ε_c is the strain of extreme compression concrete fiber, and ε_s' is the strain of compression reinforcement. Length c is the distance from the extreme compression fiber to the neutral axis, and d and d' are the distances from the extreme compression fiber to the centroid of the tensile reinforcement or compression reinforcement, respectively.

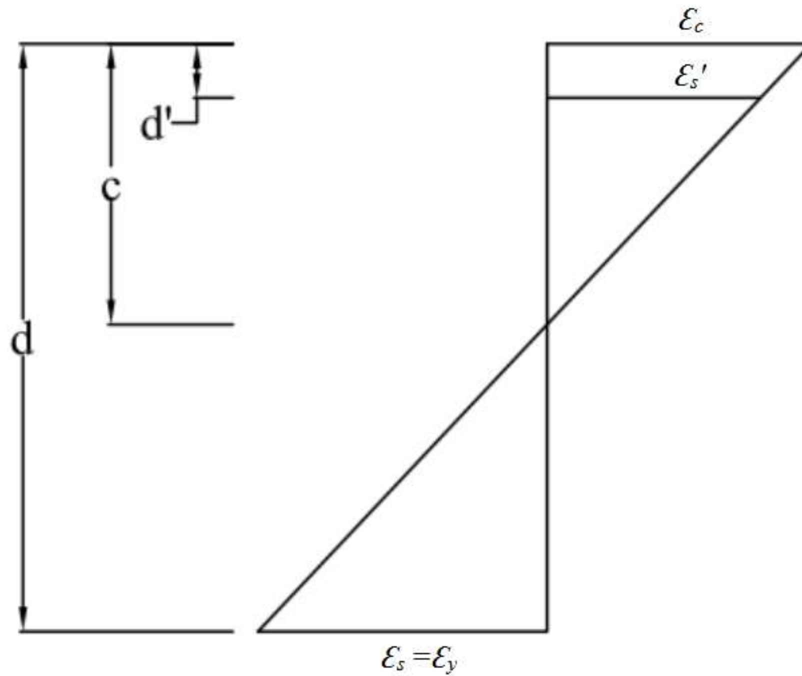


Figure 3.7 Linear strain distribution at first yield point ($\varepsilon_s = \varepsilon_y$)

$$\varepsilon_s = \varepsilon_y = f_y/E_s \tag{3.8}$$

$$\phi = \frac{\varepsilon_s}{d_b - c} = \frac{\varepsilon_s'}{c - d'} = \frac{\varepsilon_c}{c} \tag{3.9}$$

Based on Hognestad's parabola model, compression provided by concrete C_c was calculated from section width b, compression depth c, maximum compression strength of concrete f_c' and factor k_1, computed according to Equation (3.1a). Compression force from compression reinforcement C_s depends on ε_s', material properties, and area A_s' of the compression bars. Equation (3.10) describes the equilibrium and calculation of forces in sections at the yield point. Force T_s is the tension force from tensile reinforcement, and A_s is the area of tension bars.

$$C_c + C_s = T_s \tag{3.10a}$$

$$C_c = k_1 b c f_c'' \tag{3.10b}$$

$$T_s = A_s f_y \tag{3.10c}$$

Strain, stress and yield curvature ϕ_y were derived from compatibility equations, equilibrium equation and material properties. Factor k_2 describes the position of internal compressive force in concrete, and its calculation is in Equation (3.1b). Given k_2 and forces in the concrete and reinforcement, yield moment M_y was computed through Equation (3.11).

$$M = \left[\left(1 - k_2\right) c \right] C_c + \left(c - d'\right) C_s + \left(d_b - c\right) T_s \tag{3.11}$$

Ultimate moment M_u is the ultimate flexural strength of a cross-section. At this loading point, strain of extreme compression fiber ε_c is the ultimate strain ε_u, which is defined as 0.0038 according to Hognestad's parabola. Linear strain distribution is stated as Figure 3.8. Strain and stress of tension reinforcement were calculated from compatibility and equilibrium equations. Equation (3.9) and Equation (3.10), which are compatibility and equilibrium equations at yield point, are still correct at the ultimate point except that the stress of tensile reinforcement may be greater than f_y if the steel is in the strain hardening regime. Ultimate moment M_u was calculated according to Equation (3.11).

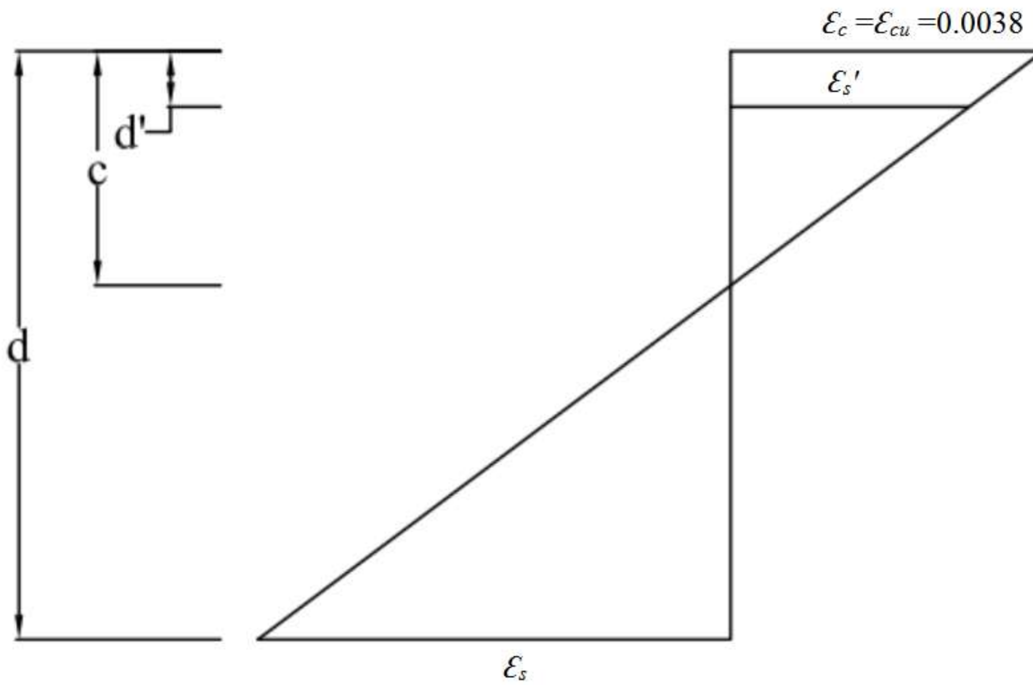


Figure 3.8 Linear strain distribution at ultimate point ($\varepsilon_c = \varepsilon_{cu}$)

Based on material properties and section dimensions in Section 3.1, mechanical properties of sections in specimen were calculated, and values are summarized in Table 3.3.

Table 3.3 Mechanical properties of sections in specimen

Category	Index	Value
General Properties	I_g (in.^{4})	8000
	f_r (psi)	515
	M_{cr} (k-in.)	412
At Yield Point	c (in.)	5.96
	ε_c	1.07×10^{-3}
	ε_s	2.07×10^{-3}
	ε_s'	6.2×10^{-4}
	f_s (ksi)	60
	f_s' (ksi)	17.9
	C_c (kips)	125.9
	T_s (kips)	180
	C_s (kips)	54.0
	ϕ_y (in.$^{-1}$)	1.79×10^{-4}
	M_y (k-in.)	2774
At Ultimate Point	c (in.)	3.24
	ε_c	3.80×10^{-3}
	ε_s	1.67×10^{-2}
	ε_s'	8.68×10^{-4}
	f_s (ksi)	66.5
	f_s' (ksi)	25.2
	C_c (kips)	123.9
	T_s (kips)	199.4
	C_s (kips)	75.5
	ϕ_u (in.$^{-1}$)	1.17×10^{-3}
	M_u (k-in.)	3130

The moment reaction at the fixed end of the cantilever beam is calculated by the displacement sequence applied at the free end in Figure 3.2. After moment reaches M_{cr}, moment of inertia should decrease to account for damage in sections. The method from Bischoff (2007) was used to calculate the effective moment of inertia of the cantilever. The calculation equations are described in Equation (3.12). Variable I_e is the effective moment of inertia, and I_g is the moment of inertia for uncracked sections. I_{cr} is the transformed moment of inertia to account for cracked sections, and was calculated at the yield point from elastic theory shown in Equation (3.13). The value of

I_{cr} was 3963 in.^{4} for the analyzed specimen. Moment M_a is the maximum moment in the analyzed element, which was at the fixed end for the cantilever specimen.

$$I_e = \frac{I_{cr}}{1-\eta\left(M_{cr}/M_a\right)^2} \leq I_g \tag{3.12a}$$

$$\eta = 1 - I_{cr}/I_g \tag{3.12b}$$

$$I_{cr} = \frac{M_y}{E_c\phi_y} \tag{3.13}$$

Moment and shear force at the fixed end were computed with I_e and basic structural analysis. The calculation equation is Equation (3.14). Length L is length of the specimen, and Δ is the displacement applied at the free end.

$$M_a = 3E_cI_e\Delta/L^2 \tag{3.14}$$

In monotonic loading, Equations (3.12) and (3.14) are not correct after the moment M_a is equal to M_y. It was assumed that a yield plateau controls after the yield point, and moment keeps the value of M_y with increasing displacement until failure. In unloading and reloading stages, no damage factor as the one in the lumped damaged-plasticity method was applied. The unloading stiffness was the same with the stiffness at the first yield point, or the stiffness at the unloading point if yield has not yet occurred. Moments of inertia were independent in positive and negative loading. After calculation, the moment-displacement relation of the specimen is shown in Figure 3.9.

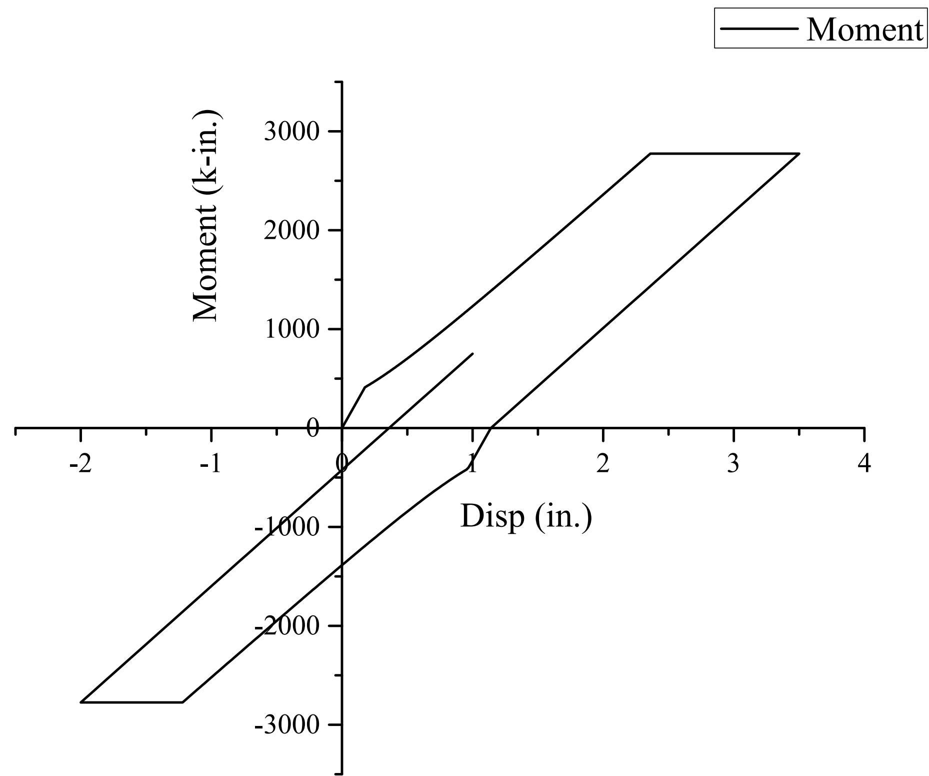


Figure 3.9 Moment-displacement relation from hand calculation

Cracking points and yield points are evident in Figure 3.9. The stiffness decreases greatly after the cracking threshold, and a yield plateau controlled after yielding. At the cracking point, the absolute value of moment was 412 k-in., and the absolute value of the applied force was 2.06 kips. On the plastic plateau, the absolute value of moment was 2774 k-in., and the absolute value of the applied force was 13.9 kips. With small displacement, the applied force and the moment at the fixed end are linearly related.

3.2.2 Detailed finite element model with ABAQUS

ABAQUS was chosen for the detailed finite element analysis because of its wide usage and good accuracy in research. The concrete damage plasticity material model was used. The main two failure mechanisms in this material model are tensile cracking and compressive crushing of the concrete.

Based on ABAQUS Analysis User's Manual version 6.12 (Dassault Systémes, 2012), the concrete damage plasticity material law requires the definition of the dilation angle, eccentricity, ratio of initial equibiaxial compressive yield stress to initial uniaxial compressive yield stress $\frac{\sigma_{b0}}{\sigma_{c0}}$, and invariant stress ratio K_c to properly introduce flow potential, viscosity, and the yield surface in the model. Explanation of these factors can be found in section 23.6.3 of the ABAQUS Analysis User's Manual, version 6.12 (Dassault Systémes, 2012). The Drucker-Prager hyperbolic function was used for non-associated plastic flow. The flow potential function requires the definition of a dilation angle and eccentricity. The dilation angle is measured in the plane defined by the hydrostatic pressure and the Mises equivalent stress at high confining pressure. Eccentricity is a parameter that defines the rate at which the function approaches the asymptote (the flow potential function tends to a straight line as the eccentricity tends to zero). The yield function of Lubliner et. al. (1989) is used in ABAQUS concrete damage plasticity model. Ratio $\frac{\sigma_{b0}}{\sigma_{c0}}$ and K_c are used to define the yield function, where factor K_c defines the shape of the yield surface in the deviatoric plane (Dassault Systémes, 2012). It must satisfy the condition $0.5 < K_c \leq 1$(the default value is 2/3). Figure 3.10 shows the effect of K_c. Stresses S_1, S_2, and S_3 are the principal deviatoric stress components.

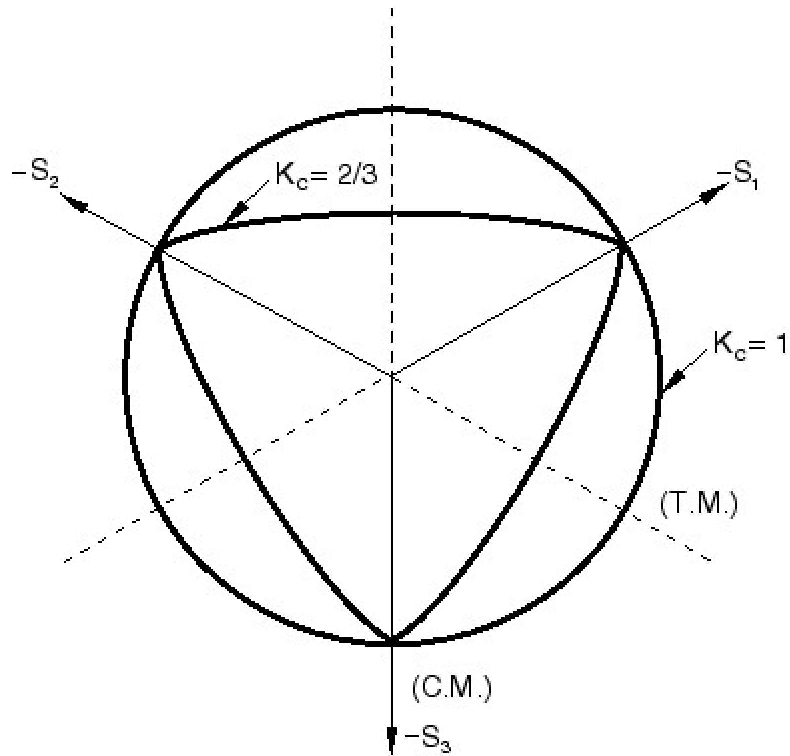


Figure 3.10 Yield surfaces in the deviatoric plane, corresponding to different values of K_c (Dassault Systémes, 2012)

To improve rate of convergence in the softening regime in the ABAQUS model, viscoplastic regularization with a small value as the viscosity parameter can be used. The viscosity parameter represents the relaxation time of the viscoplastic system. The default value of the viscosity parameter in concrete damage plasticity model is 0, which means no viscoplastic regularization is used. More details of explanation can be found in section 23.6.3 in ABAQUS Analysis User's Manual, version 6.12 (Dassault Systémes, 2012).

Values of factors of concrete used in this comparison are summarized in Table 3.4.

Table 3.4 Values of factors defined in concrete damage plasticity model

Factor	Dilation Angle	Eccentricity	$\frac{\sigma_{b0}}{\sigma_{c0}}$	K_c	Viscosity Parameter
Value	30°	0.1	1.16	0.67	5×10^{-4}

Stress-strain relations in compression and tension of concrete were the same as the ones given in Section 3.1. In the unloading stage, a compression damage factor, d_c, was needed to present the effects of concrete cracking on decreasing stiffness. The influence of the compression damage factor on the stress-strain relationship of concrete is shown in Figure 3.11 from ABAQUS Analysis User's Manual 6.12. Factors σ_c and ε_c are compression stress and compression strain, respectively. Index E_0 is the elastic modulus of concrete, $\tilde{\varepsilon}_c^{pl}$ $\tilde{\varepsilon}_c^{pl}$ $\tilde{\varepsilon}_c^{pl}$ is plastic strain. Strain $\tilde{\varepsilon}_c^{in}$ $\tilde{\varepsilon}_c^{in}$ $\tilde{\varepsilon}_c^{in}$ is inelastic strain, which is the plastic strain with zero damage. The relation of d_c, $\tilde{\varepsilon}_c^{in}$ $\tilde{\varepsilon}_c^{in}$ $\tilde{\varepsilon}_c^{in}$ and $\tilde{\varepsilon}_c^{pl}$ $\tilde{\varepsilon}_c^{pl}$ $\tilde{\varepsilon}_c^{pl}$ is shown in Figure 3.11. Plastic strain $\tilde{\varepsilon}_c^{pl}$ $\tilde{\varepsilon}_c^{pl}$ $\tilde{\varepsilon}_c^{pl}$ can be calculated as a function of d_c, $\tilde{\varepsilon}_c^{in}$ $\tilde{\varepsilon}_c^{in}$ $\tilde{\varepsilon}_c^{in}$ and E_0 as presented in Equation (3.15).

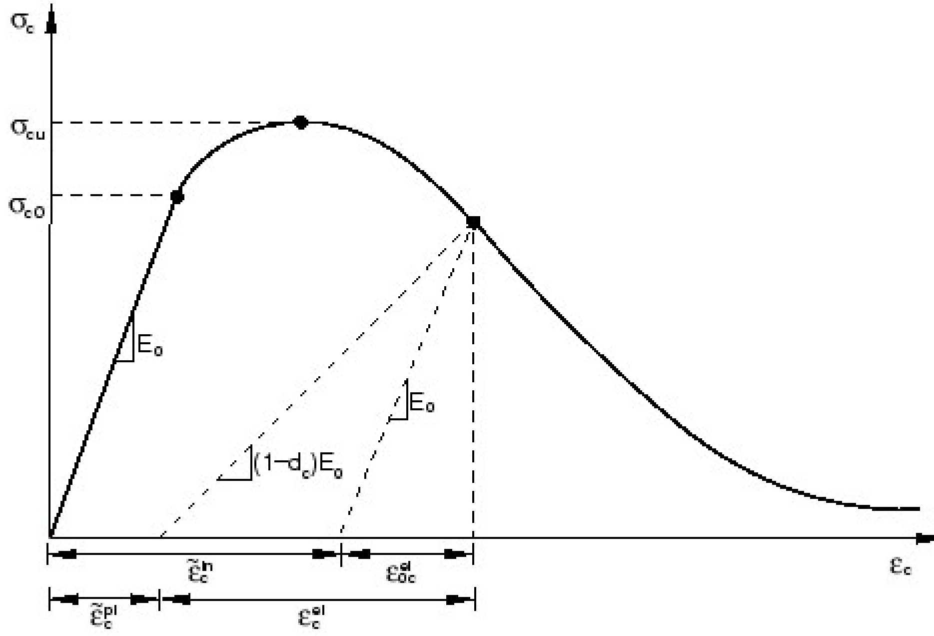


Figure 3.11 Response of concrete to uniaxial loading in compression (Dassault Systémes, 2012)

$$\tilde{\varepsilon}_c^{pl} = \tilde{\varepsilon}_c^{in} - \frac{d_c}{(1-d_c)} \cdot \frac{\sigma_c}{E_0} \tag{3.15}$$

The damage index can be computed by Equation (3.16a) with help of a β_c factor. Factor β_c is the ratio of plastic strain and elastic strain.

$$d_c = \frac{(1-\beta_c)\tilde{\varepsilon}_c^{in} E_0}{\sigma_c + (1-\beta_c)\tilde{\varepsilon}_c^{in} E_0} \tag{3.16a}$$

$$\beta_c = \frac{\tilde{\varepsilon}_c^{pl}}{\tilde{\varepsilon}_c^{in}} \tag{3.16b}$$

Zhang et al. (2008) recommended the β_c ratio should be in the range of 0.35 to 0.7. In this analysis, 0.35 was used as β_c.

The tension recovery factor was set as 0, which means tension strength was 0 after failure in compression. Tension failures (cracks) were assumed not to influence compression strength, so the compression recovery factor was set equal to 1.

An 8-node linear brick with reduced integration and hourglass control element (C3D8R) was set as the element type for the concrete. To get detailed and accurate simulation results, the mesh size should be approximately 3 times the size of the largest aggregate (Bazant and Planas, 1998). Concrete in the ABAQUS model was assumed to have 0.75-in. size for the largest aggregate. Therefore, a 2-in. mesh size was implemented.

Elastic and plastic properties of steel reinforcement were the same as steel properties in Section 3.1. Elastic properties were defined by modulus of elasticity and Poisson's ratio in ABAQUS. Poisson's ratio of steel ranges from 0.27 to 0.30 (Gere el al., 2011). In this ABAQUS model, the modulus of elasticity was 29,000 ksi, and 0.3 was used as Poisson's ratio. Plasticity of reinforcement was applied with a list of plastic stress and plastic strain, which were calculated by Equation (3.4) and Table 3.3.

In the ABAQUS model, lateral reinforcement was needed to provide enough shear resistance to prevent shear failure. Through the loading period of specimen, the largest shear force V_{max} was 13.9 kips. Based on ACI 318-14, shear resistance from concrete was not enough to prevent shear failure. Consequently, to provide enough shear resistance, No. 3 ties at 15-in. spacing were modeled along length of the specimen based on regulations in ACI 318-14. Ties and longitudinal reinforcements were defined as truss elements, and the mesh size was the same as the concrete (2-in.). Using a 0.01-in. displacement increment provided accurate simulation results with no

convergence problems; a sensitivity test using a displacement-increment of 0.001-in. was performed and returned nearly identical results.

Outcomes of moment at the fixed end of the ABAQUS model are shown in Figure 3.12, with relation to displacement applied at the free end. Yield points and unloading points are clear in Figure 3.12.

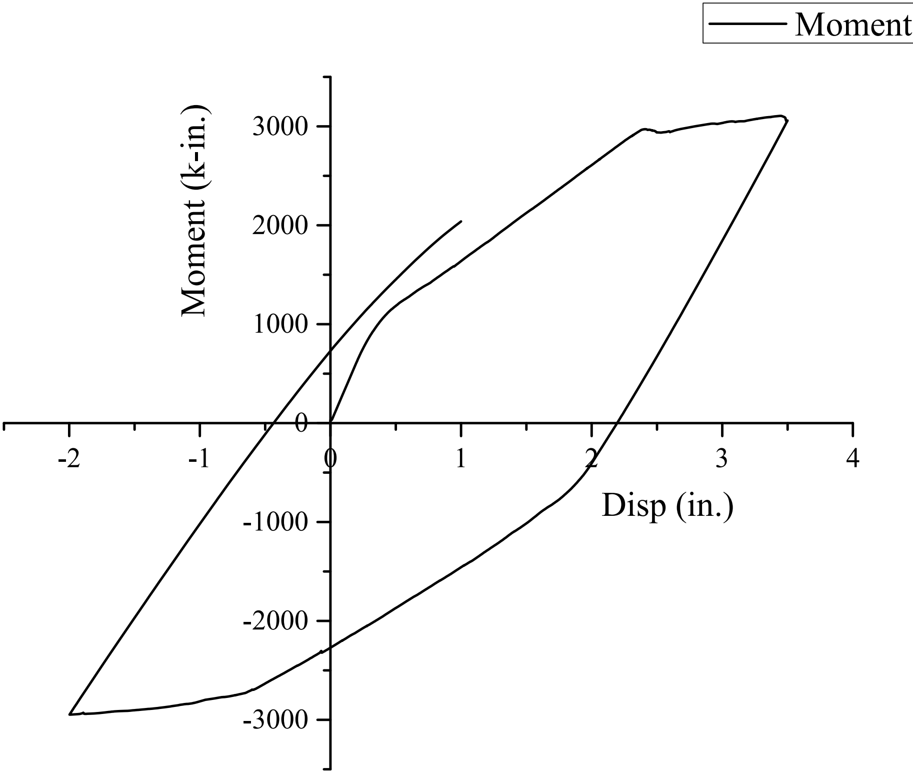


Figure 3.12 Moment-displacement relation from ABAQUS analysis

3.2.3 Model with fiber element method

In the fiber element model, structural members are divided into several elements longitudinally, while cross sections are discretized into a set of longitudinally oriented fibers. The fiber element method assumes the plane section remains plane and normal to the longitudinal axis. The sectional response of a member is derived by integration of the response of the fibers, which follows the uniaxial stress-strain relation of the particular material (Taucer et al., 1998). The fiber element method is based on the flexibility method, in which the internal force distribution is expressed by shape functions. The analysis procedure of the fiber element method is described in Figure 3.13 (Petrangeli et al., 1999).

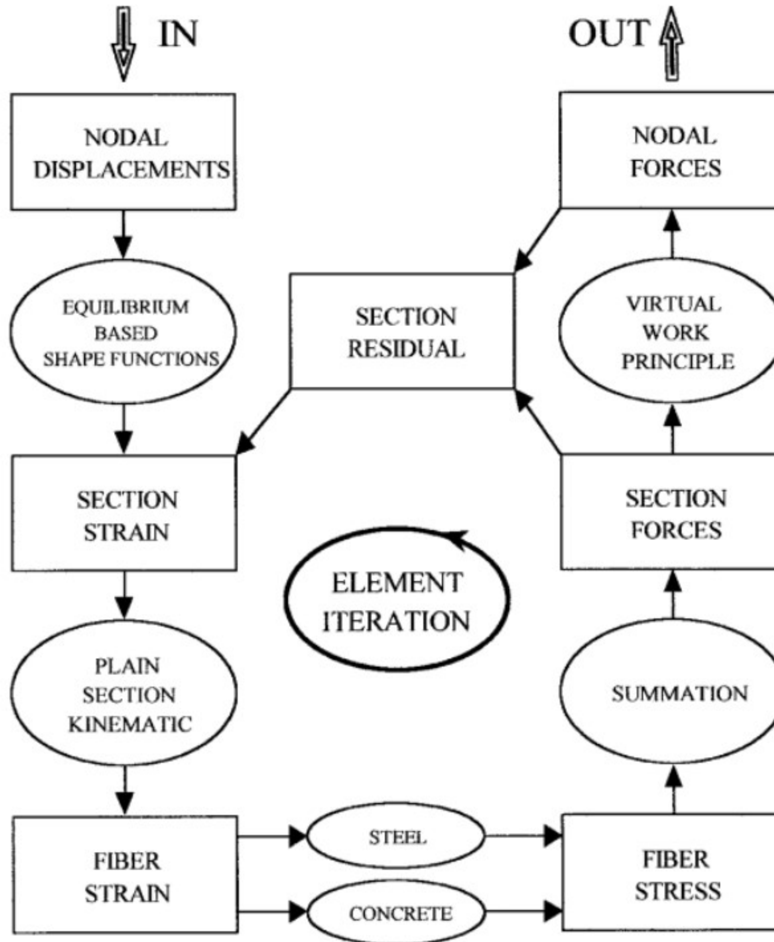


Figure 3.13 Analysis procedure of fiber element method (Petrangeli et al., 1999)

Because the fiber element method strongly depends on the shape function and number of sections integrated, the number of section integrated and fiber numbers are important to the accuracy and efficiency of simulation results. Du et al. (2012) simulated three different elements (a displacement-based element, a force-based element, and a plastic hinge element) and checked the optimal number of section integration points for different elements in Open System for Earthquake Engineering Simulation (OpenSees). OpenSees is a software framework for simulating the seismic response of structural and geotechnical systems, and has been developed as the computational platform for research in performance-based earthquake engineering at the Pacific Earthquake

Engineering Research Center. The displacement-based element was based on the displacement formulation (stiffness formulation), and the force-based element relied on iterative force-based formulation (flexibility formulation). Both considered spread of plasticity along the elements. The plastic hinge element was based on the flexibility formulation, and plasticity was lumped in a plastic hinge area. Du et al. (2012) concluded that for the displacement-based element, six integration points or more should be used to get both consistent structural and sectional level responses; for the force-based element, the use of more integration points made the element more unstable in the case of section response under softening after yield; for the plastic hinge element, the plastic hinge length controlled the structural behavior.

To get accurate results on simulating the cantilever beam specimen with the fiber element method, the number of section integration points and mesh of the beam section needed to be decided carefully. OpenSees was used to build and simulate the fiber element model, and force-based elements were applied. From Du et al.'s (2012) recommendation, more integration points provided better accuracy for the force-based element with strain hardening after yield. Hence, 10 integration points were used for the member in this chapter. To define the number of fibers and the mesh size along the length of the beam section, a 2-in. mesh size (the same with the ABAQUS model) was applied. Material behaviors of concrete and reinforcement were the same as the ones in Section 3.1.

To account for damage of concrete in flexural analysis, OpenSees introduced degraded linear unloading and reloading stiffness according to the work of Karsan and Jirsa (1969). Karsan and Jirsa proposed a linear unloading and reloading path, during which plastic strain ε_p was expressed solely as a function of the maximum compressive strain ε_m ever reached at the considered point of

the analyzed section. The calculation function of plastic strain ε_p is shown in Equation (3.17). Strain ε_0 is the corresponding strain at the largest compressive stress point, which was 0.0021 in this analysis.

$$\varepsilon_p = 0.145 \frac{\varepsilon_m^2}{\varepsilon_0} + 0.13\varepsilon_m \tag{3.17}$$

For instance, Figure 3.14 represents the unloading and reloading procedures using Karsan and Jirsa's model, where the concrete compression properties are the same with the ones in Section 3.1. Stress develops in path O-A-B with increasing monotonic compression strain. If unloading occurs at point C (where strain is 2.6×10^{-3}), remaining plastic strain is 8.2×10^{-4} at point D computed by Equation (3.16). Reloading will follow D-C-B path after unloading happens.

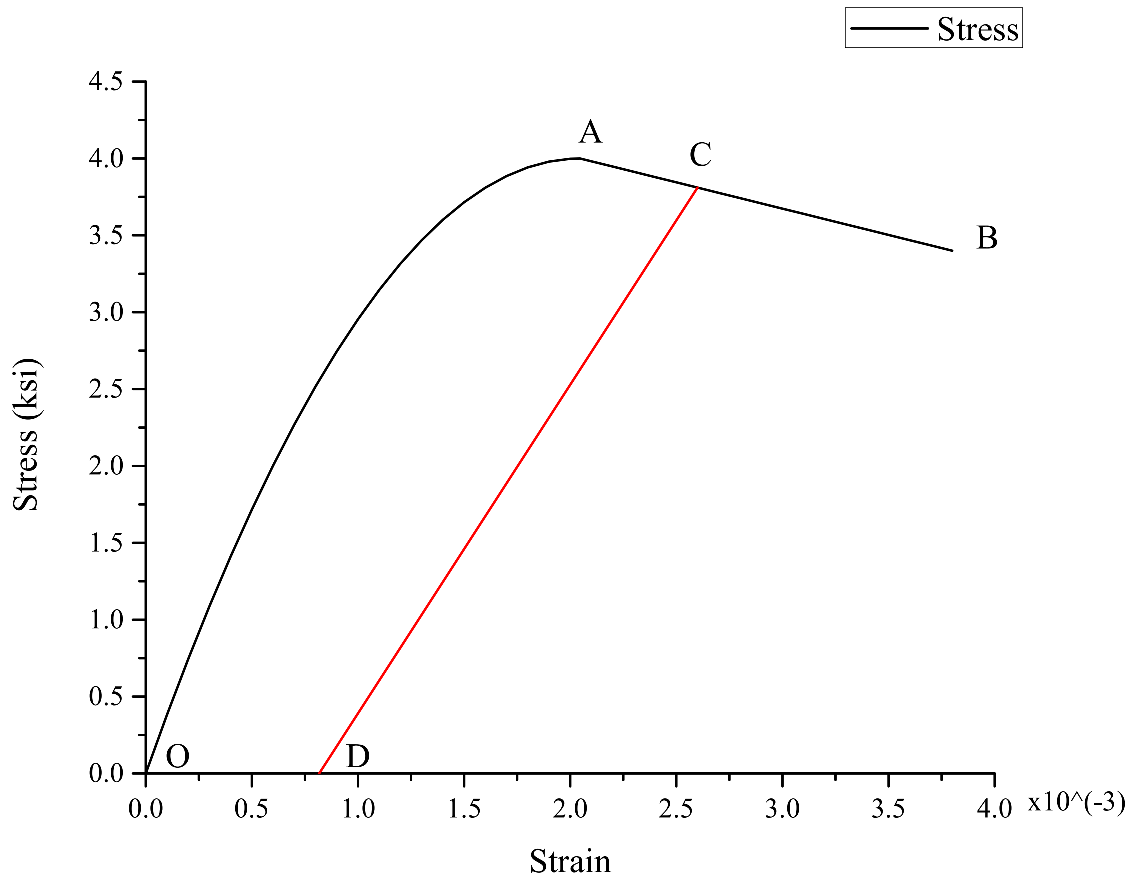


Figure 3.14 Unloading and reloading stages in concrete compression

Moment reactions at the fixed end of specimen computed from the fiber element analysis are presented with respect to displacement at the free end in Figure 3.15.

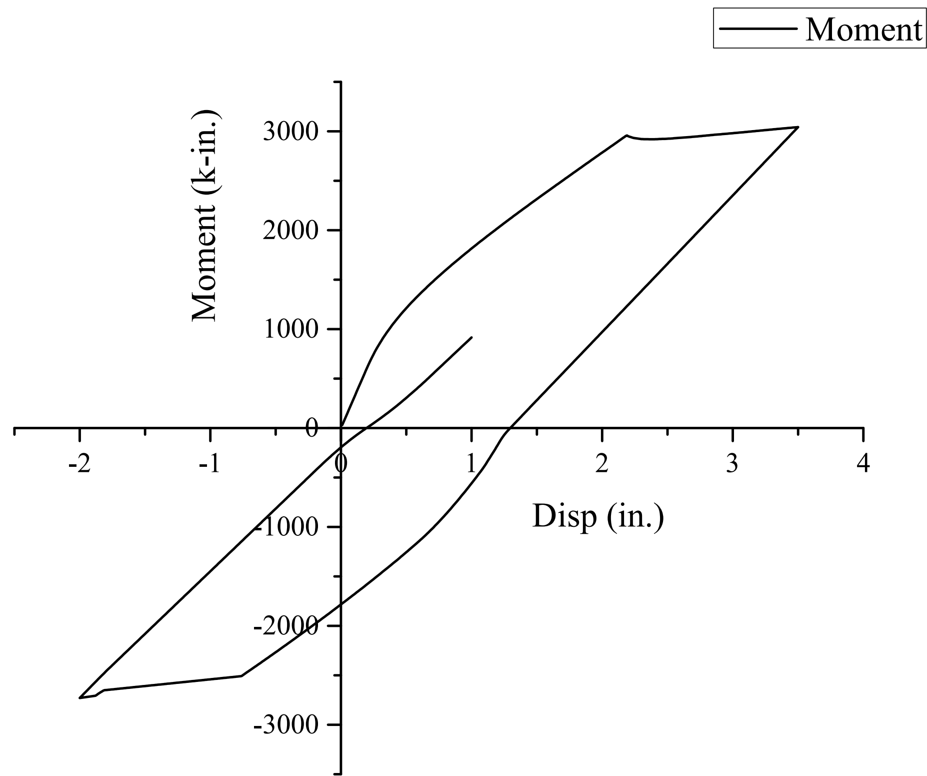


Figure 3.15 Moment-displacement relation from fiber element analysis

3.2.4 Model with the lumped damaged-plasticity method

Theories of the lumped damaged-plasticity method are presented in Chapter 2. OpenSees was applied to achieve the lumped damaged-plasticity model. Equivalence equations, plasticity evolution laws and damage evolution laws of the lumped damaged-plasticity method were introduced into OpenSees by C++. Code for the lumped damaged-plasticity method and its connection code to OpenSees are shown in the Appendix. The procedure for analysis on a beam element is introduced in the following steps:

1. Data of the beam element should be input into OpenSees code. The data include length of the beam, elastic modulus of concrete E_c, gross moment of inertia I_g, area of sections along the beam A, constants q, c, K_0, G_{cr}, d_u calculated according to Section 2.3, and ultimate moment M_u from section analysis. These parameters must be specified for both ends of the beam element (end i and end j).
2. Linearity is assumed first, and moments at both ends of the element M_i and M_j are calculated by equivalence equations (Equation 2.4). Plastic rotations θ_i^p, θ_j^p and damage indexes d_i, d_j are the maximum plastic rotations and damage ever reached at the considered analysis point in the same analysis direction.
3. Values for the yield surface and the damage surface at both ends f_i, g_i, f_j, g_j are computed with data calculated in Step 2. If f_i, g_i, f_j, g_j are all equal to or smaller than 0, the calculations assuming linearity from Step 2 are correct. However, if some of the damage functions and plastic functions are larger than 0, adjustments of damage and plastic rotations at specific points need to be done to make the positive-value-equations equal to 0. Equivalence shall be satisfied simultaneously. Broyden's method (Broyden, 1965) was applied as the nonlinear solver in the OpenSees code.
4. After calculation, moments M_i, M_j, damage indexes d_i, d_j, and plastic rotations θ_i^p, θ_j^p at both ends are recorded.

For the specimen analyzed throughout Chapter 3, factors input into OpenSees are summarized in Table 3.5. Because catenary action was not considered in this comparison, area of compression reinforcement and its ultimate stress were not necessary to input, and I_g was used as the moment of inertia. All factors are calculated following Section 2.3.

Table 3.5 Factors input into OpenSees for the lumped damaged-plasticity model

Factor	Value
E_c (ksi)	3910
I_g (in.^{4})	8000
A (in.^{2})	240
q (k-in.^{3})	-28.3
c (k-in.)	4.38×10^5
K_0 (k-in.)	4.35×10^3
G_{cr} (k-in.^{3})	0.18
M_u (k-in.)	3130
d_u	0.63

The moment-displacement relation after analysis is shown in Figure 3.16, moment-damage relation is shown in Figure 3.17, and moment-plastic rotation relation at the fixed end is shown in Figure 3.18.

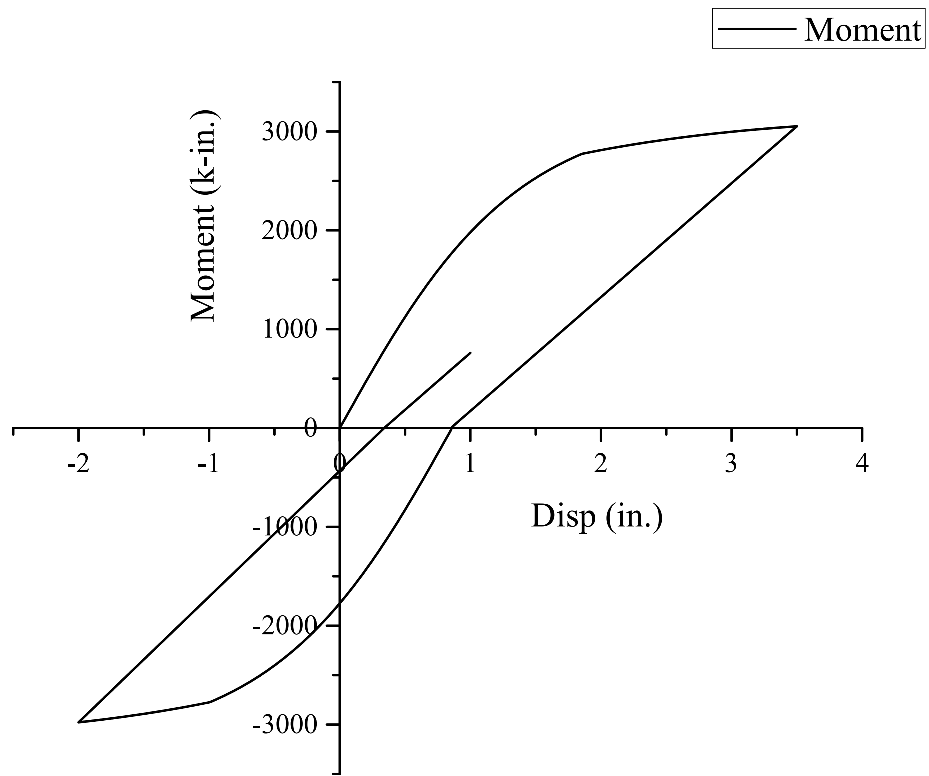


Figure 3.16 Moment-displacement relation for the lumped damaged-plasticity model

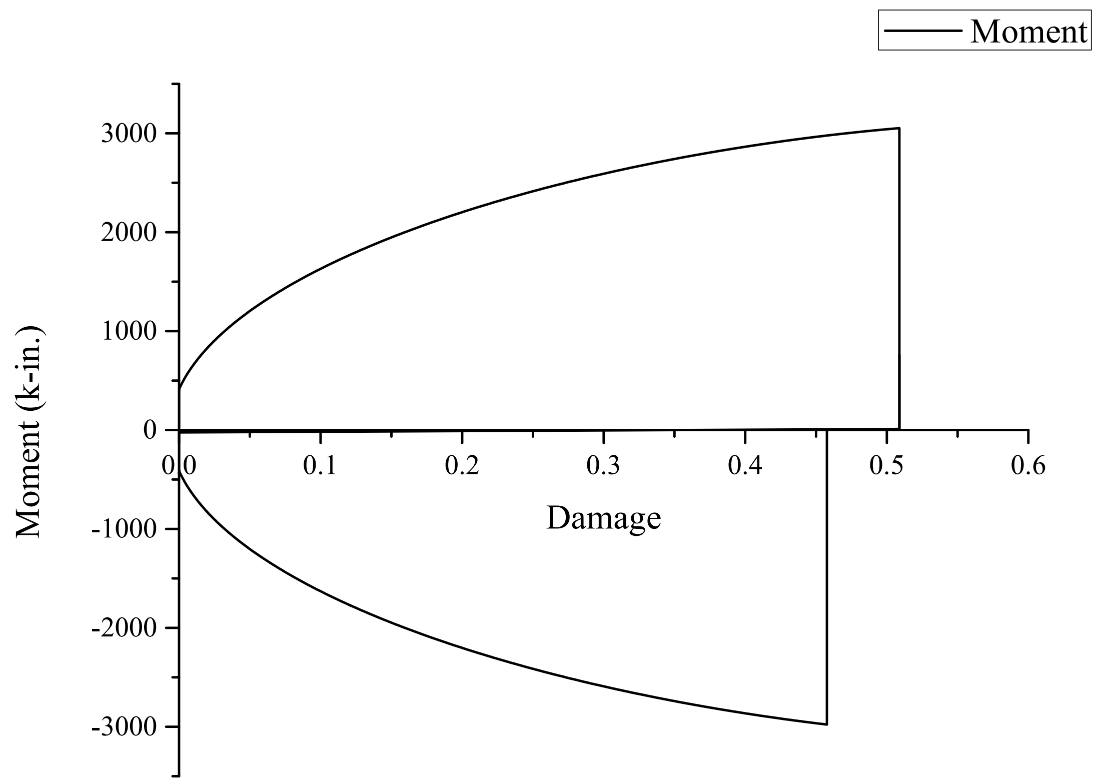


Figure 3.17 Moment-damage relation for the lumped damaged-plasticity model

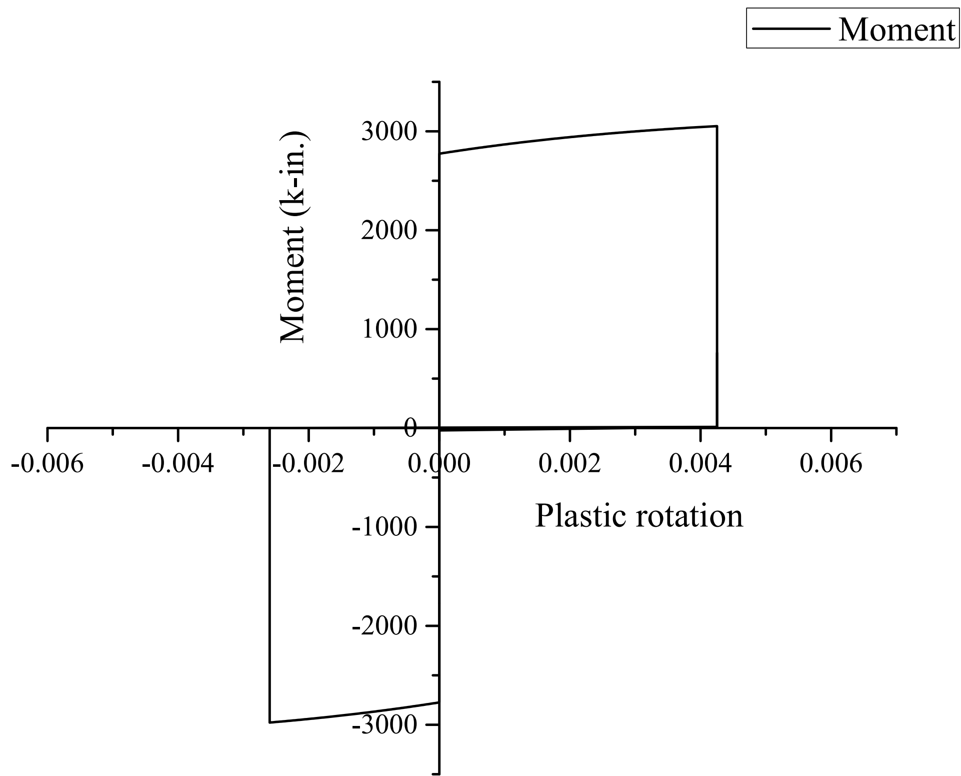


Figure 3.18 Moment-plastic rotation relation for the lumped damaged-plasticity model

From Figure 3.17 and Figure 3.18, damage and plastic rotation upon reloading keep values from the first unloading point. Damage started to accumulate after the cracking moment was reached, and plastic rotation increased after yielding. Damage and plastic rotation were calculated separately for positive and negative loading directions.

3.3 Comparison of models

After analysis, the moment-displacement relations of the examined model from hand calculations, the ABAQUS model, the fiber element model, and the lumped damaged-plasticity model are

shown in Figure 3.19. Yield moment M_y, displacement at first yield point D_y, moment when displacement was equal to 3.5-in. $M_{3.5}$, and hysteretic energy Q for different models are summarized in Table 3.6. Dissipated hysteretic energy Q is computed by the area enclosed by the hysteresis loop.

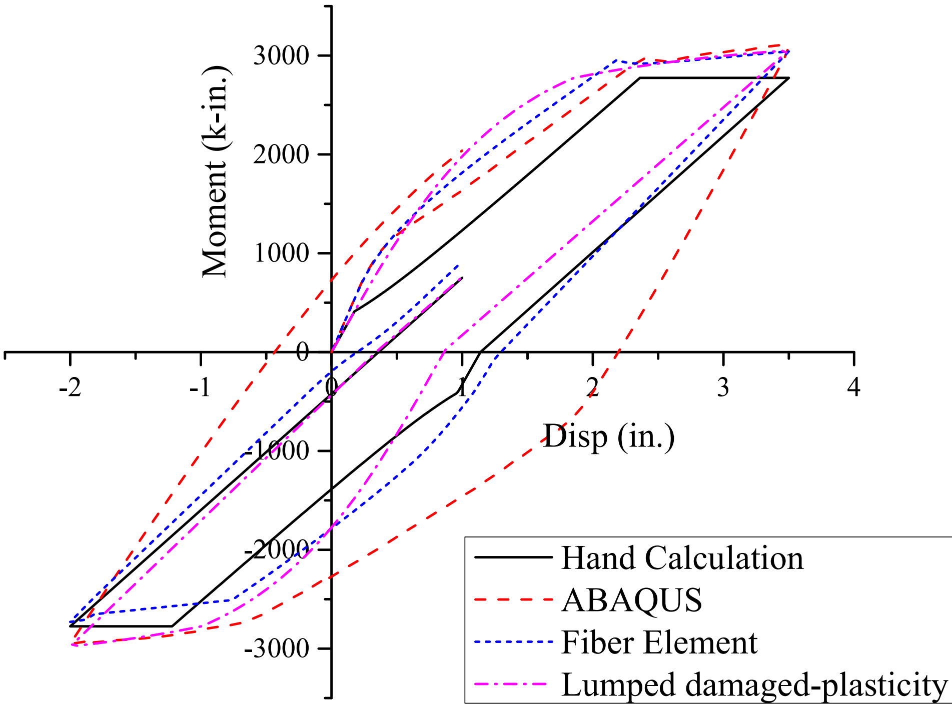


Figure 3.19 Moment-displacement relation for the lumped damaged-plasticity model

Table 3.6 Comparison of different models

Result \ Method	Hand calculation	ABAQUS	Fiber element	Lumped damaged-plasticity method
M_y (k.-in.)	2774	2940	2932	2774
D_y (in.)	2.37	2.35	2.18	2.01
$M_{3.5}$ (k.-in.)	2774	3048	3031	3031
Q (k.-in.)	6196	13400	8165	6840

As shown in Figure 3.19, cracking occurred at nearly the same loading point for hand calculation, ABAQUS, fiber element and the lumped damaged-plasticity models. However, stiffness decreased suddenly for the hand calculations after cracking, while results of other methods showed a gradual decrease in stiffness. The hand calculations was based on the simplified method by Bischoff (2007).

The lumped damage-plasticity model yielded first (that is, with the lowest displacement) in monotonic positive loading, followed by the fiber element model and the ABAQUS model. This difference suggested the lumped damaged-plasticity model accumulated damage faster than the other models before yielding. The fiber element method considered damage before yield from tensile failure of concrete, and its yield displacement was similar with the one of the lumped damaged-plasticity model, with only 7.8% difference. The relative differences of yield moment for the lumped damaged-plasticity model to ABAQUS model and the fiber element model were smaller than 6%.

The ABAQUS model, the fiber element model, and the lumped damaged-plasticity model nearly had the same moment when displacement was equal to 3.5 in., and the relative difference in moment was smaller than 0.6%. After yielding but before the final moment at 3.5 in. displacement, the load paths were similar for the ABAQUS model, the fiber element model and the lumped

damaged-plasticity model. The comparsion in Figure 3.19 suggested the lumped damaged-plasticity model simulated similar resistance with other models.

No damage was considered in the yield plateau of the hand calculation. Upon unloading, hand calculation results had the same stiffness as the stiffness at first yield. Stiffness in the hand calculations rapidly degraded immediately after cracking, which influenced the unloading stiffness and resulted in the smallest hyterestic energy among all the examined methods. Though similar, the fiber element model had greater stiffness and plastic deformation in unloading than the lumped damaged-plasticity model. For the ABAQUS model, unloading stiffness was similar with the elastic stiffness and larger than other models. Thus, the ABAQUS model had much greater plastic displacement compared to the other models. The lumped damaged-plasticity model had the smallest unloading stiffness because of its quick accumulation of damage.

During the loading procedure, dissipated energy Q for each model was calculated and listed in Table 3.6. The hand calculations had the smallest disspated energy because of the sudden degrading stiffness after cracking. Compared with other models, the relative difference of the dissipated energy Q from the lumped damaged-plasticity was 16% and 50% for the fiber element method and the ABAQUS model.

Comparing the lumped damaged-plasticity results and the ones from the detailed finite element ABAQUS model, the cracking moments, yield moments, and final moments (at the 3.5 in. displacement) were similar. However, the unloading paths were largely different, with the ABAQUS model having much larger unloading stiffness and plastic deformation than the lumped damaged-plasticity model, leading to a 50% of difference in Q. The lumped damaged-plasticity model used yield functions and damage functions in Section 2.2.2 and Section 2.2.3 to control

flow of damage and plasticity, while ABAQUS tracked material properties of each point. This difference could be fixed by changing the yield functions and damage functions. Currently, the comparison suggests the lumped damaged-plasticity method provides a good estimate of flexural resistance for the specimen, and the difference in yield moments is smaller than 6% compared with other methods. Although the lumped damaged-plasticity method could provide a smaller unloading resistance, it is still reasonable to be a quick check for flexural resistance considering its theoretical simplicity and short calculation time.

3.4 Deficiencies of the lumped damaged-plasticity model

Although the validity of lumped damaged-plasticity method was demonstrated in this chapter, the method still has deficiencies because of the specific definitions of the yield and damage functions given in Section 2.2.2 and 2.2.3 (Equation 2.8, Equation 2.11). With increasing moments, damage and plastic deformation accumulate when the loading condition reaches yield surface and damage surface. This was consistent with the models examined in this chapter. However, in some cases, the moment capacity may decrease with increasing curvature after yield because of the strain-softening material properties of concrete. If the lumped damaged-plasticity model is applied for elements in which the moment decreases with increasing curvature after yielding, the damage reaches the ultimate damage d_u at the yield point, and further damage fails to accumulate after yielding. Solving this deficiency requires a large number of experiments to propose new yield functions and damage functions, which is outside the scope of this research. To avoid this problem in this research report, if the ultimate moment M_u was smaller than the yield moment M_y, then the ultimate moment was instead assumed to be equal to the yield moment, thus forming a yield plateau.

Additionally, the plasticity and damage evolution laws in Section 2.2.2 and 2.2.3 are moment-based, and damage indexes and plastic rotations are depending heavily on the value of moment. In computation of the parameters (e.g., q, c, K_0) used to define the yield and damage functions, the curvature at first cracking ϕ_{cr} and at first yield ϕ_y are not used. These parameters depend on the flexibility matrix **F**, which is a function of the moment of inertia I. Therefore, the choice of I will impact the response of the lumped damaged-plasticity model. Because the damage indexes do not account for all forms of damage, it is not clear whether I_g, I_{core}, or some other value of moment of inertia should be used. Furthermore, the moment of inertia changes along the length for real structures, and is not simply lumped at the two ends. In theory, the proper moment of inertia could be selected to fit the curvature at first yield or some other feature of the moment-curvature relationship. This topic requires further exploration.

Chapter 4. Comparison to experimental data

The lumped damaged-plasticity model is primarily a flexural model, and its validation to concrete beam elements was demonstrated in Chapter 3. In this chapter, catenary action is included into analysis. A static analysis was applied to a 2-bay-by-2-story two-dimensional structure specimen with the lumped damaged-plasticity model, and the outcomes were compared with experimental data. A nonlinear dynamic analysis was also applied, and the structure survived from progressive collapse based on simulation results. However, the accuracy of results from nonlinear dynamic analysis needs support from further experiment and more information about the structure.

4.1 Experiment description

The structure studied in this research was the continuous reinforcement test frame tested by Stinger (2011) and Stinger and Orton (2013). It was a quarter scaled 2-bay-by-2-story frame from an office building designed following ACI 318-08. The sketch of the office building is shown in Figure 4.1. The original office building was 6 bays by 4 bays on 24-ft center to center spacing, and had 6 stories with 12-ft story height. The reinforced concrete building had 22-in. by 22-in. columns, 36-in. by 18-in. interior girders, 24-in. deep by 12-in. wide exterior girders and 6-in. thick slabs. The columns contained eight #10 longitudinal bars and #4 stirrups at 18-in. spacing. Beams contained three #8 bars for positive moments and four #8 bars for negative moments as tensile reinforcement, and #3 stirrups were placed every 6 in. until 9 ft from the face of columns. Layouts of columns and beams cross-sections from the original office building are shown in Figure 4.2 and Figure 4.3, respectively. Details of shear reinforcement are described in Figure 4.4. Continuous reinforcement in the perimeter frames was provided as required in ACI 318-08.

The bolded portion of the exterior frame of the structure in Figure 4.1 was quarter scaled to be the test frame. For the quarter-scaled frame, columns were 5.5 in. by 5.5 in. with six #3 bars, and beams were 6 in. by 3 in. with three #2 bars as positive moment tension bars and four #2 for negative moment. Transverse reinforcement was scaled down to D2 bars (0.16-in. diameter). Layouts of sections and shear reinforcement for the quarter-scaled test are shown in Figure 4.5, Figure 4.6, and Figure 4.7. Details of reinforcement position in the test frame are in Figure 4.8 and Figure 4.9.

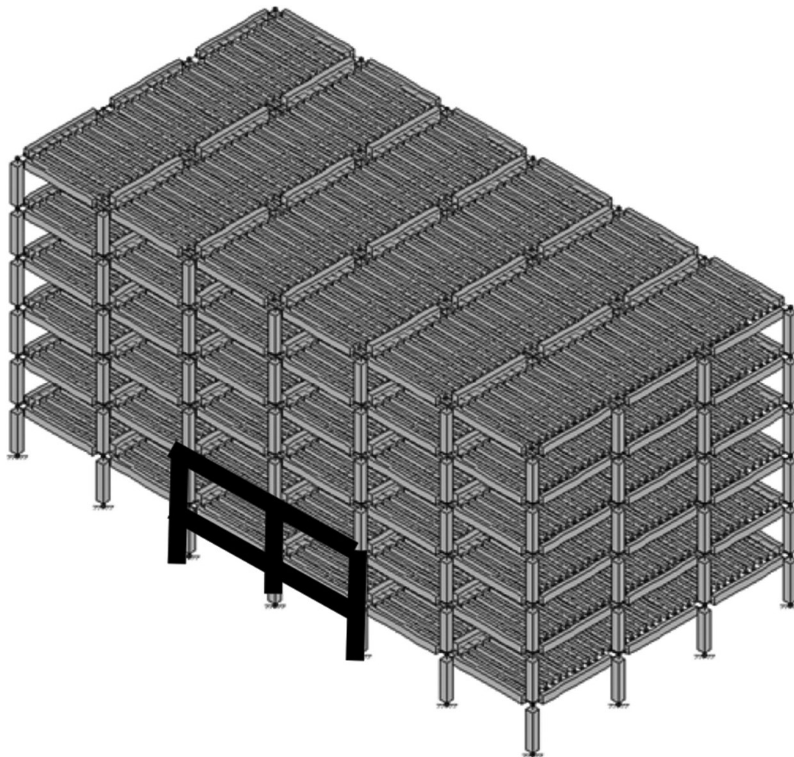


Figure 4.1 Sketch of original office building (Stinger and Orton, 2013)

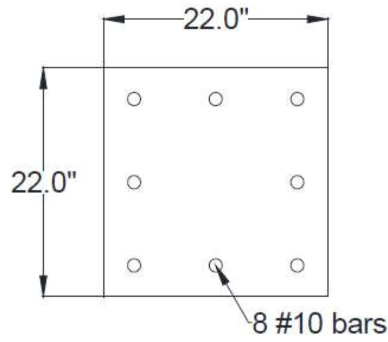
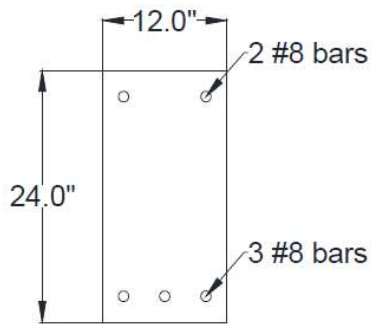


Figure 4.2 Layout of column cross-sections in original building (Stinger, 2011)



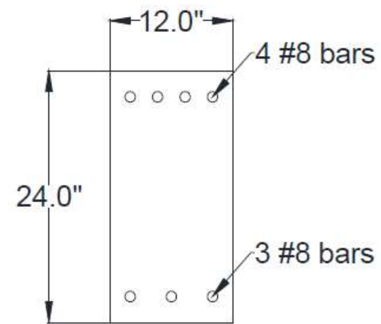


Figure 4.3 Layout of beam cross-sections in original building (Stinger, 2011)

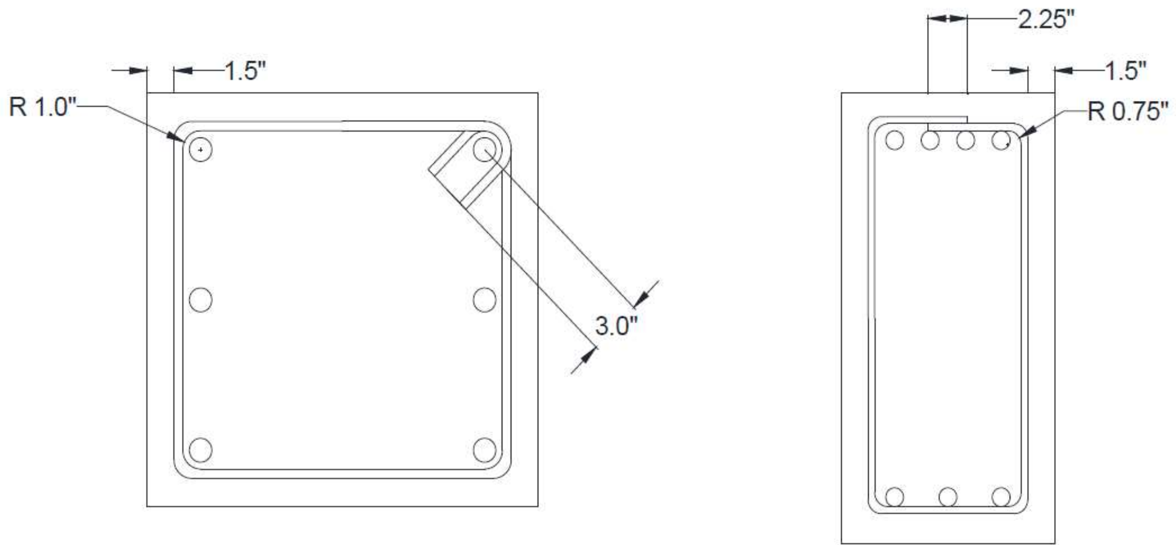


Figure 4.4 Details of shear reinforcement in original building (Stinger, 2011)

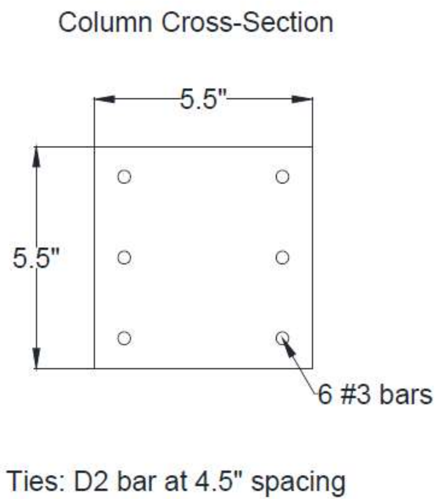
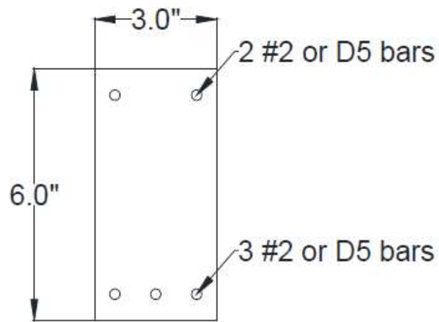


Figure 4.5 Layout of column cross-sections in the test frame (Stinger, 2011)



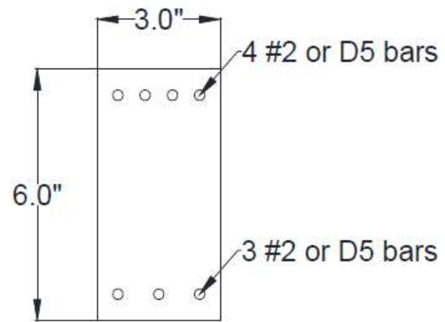


Figure 4.6 Layout of beam cross-sections in the test frame (Stinger, 2011)

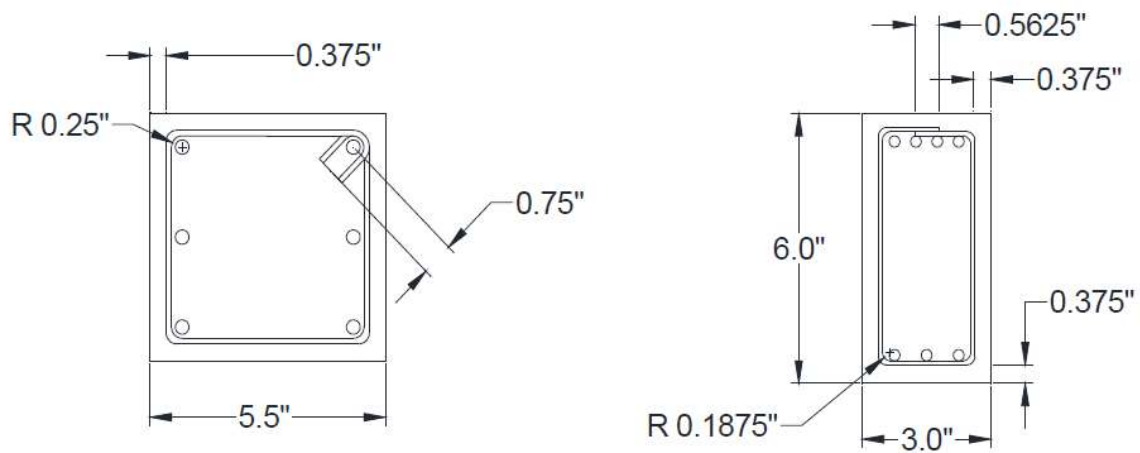


Figure 4.7 Details of shear reinforcement in the test frame (Stinger, 2011)

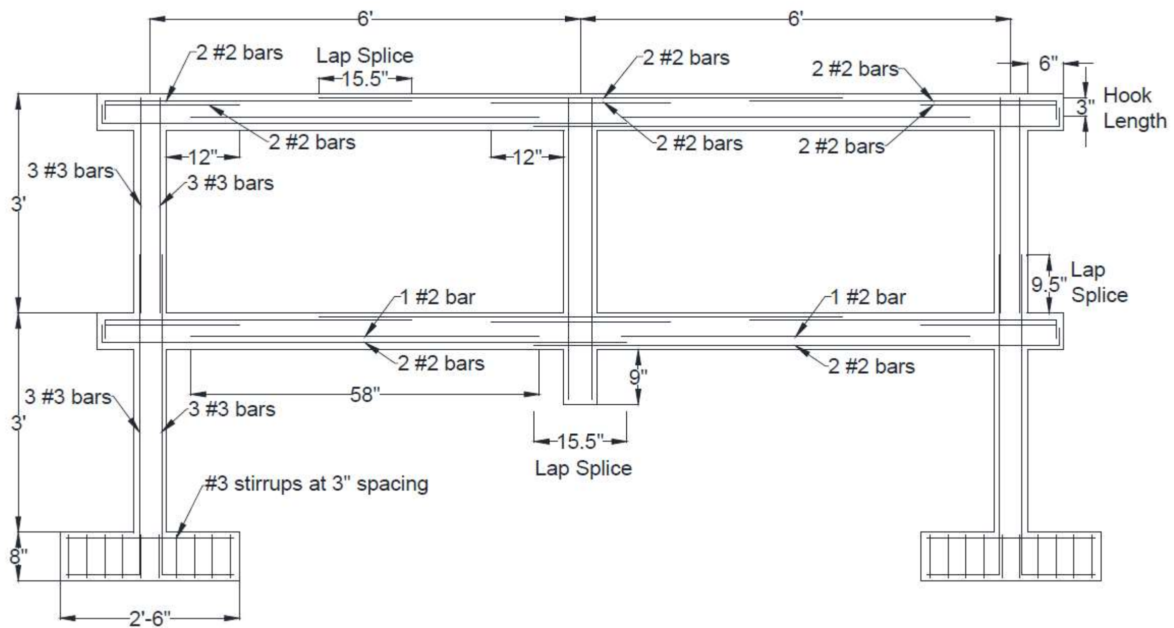


Figure 4.8 Details of longitudinal reinforcement for the continuous test frame (Stinger, 2011)

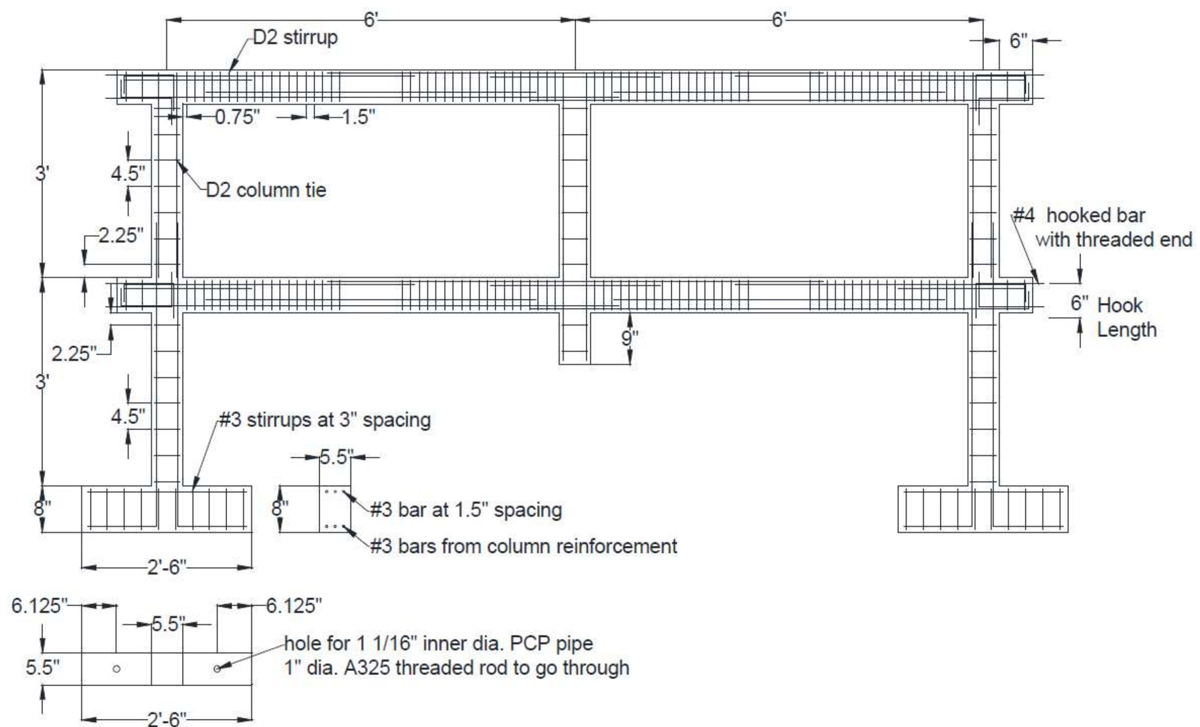


Figure 4.9 Details of transverse reinforcement for the continuous test frame (Stinger, 2011)

To simulate the stiffness provided by the surrounding frame, a reaction frame was applied at both sides of the test frame. After calculation, the lateral stiffness from the original structure was 5000 kips/in., and 1250 kips/in. for the quarter-scaled test frame (Stinger and Orton, 2013). Reaction frames were built in the experiment, and its design is shown in Figure 4.10.

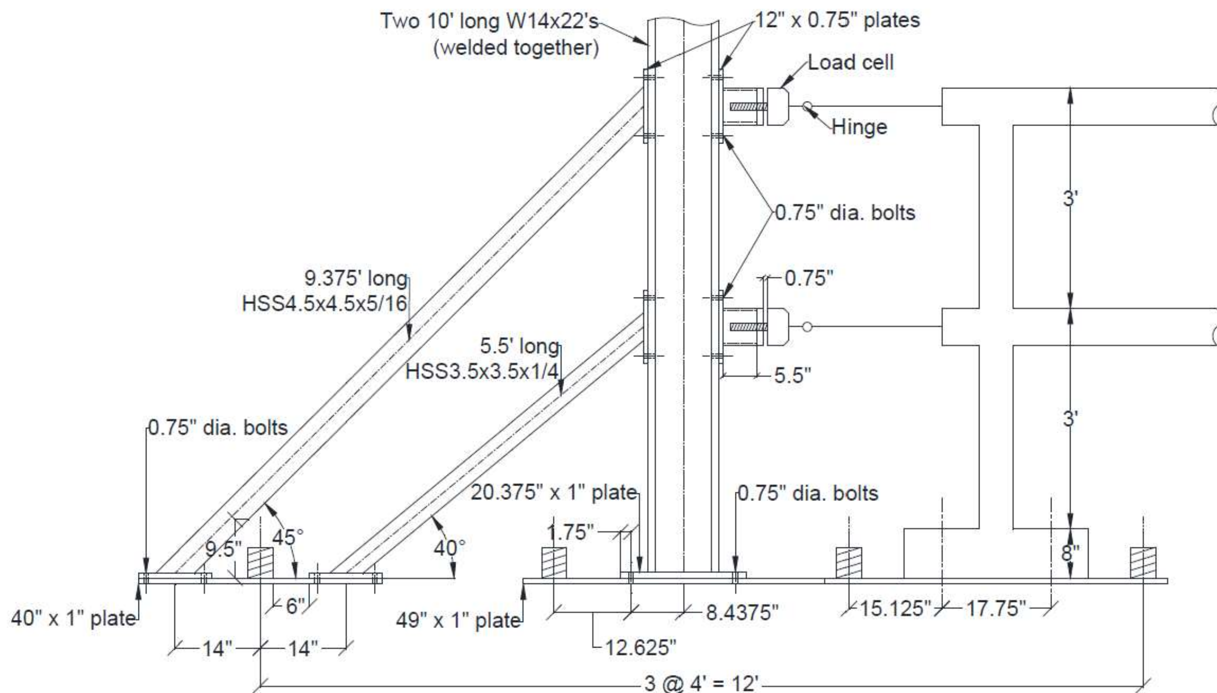


Figure 4.10 Design of reaction frame (Stinger, 2011)

Monotonic vertical displacement was applied at the top of central column in the test frame during the experiment. Rate of exerted displacement was 1/300 in./sec for the first 2 in., and 1/120 in./sec for the remainder of test. The vertical force applied at the top of central column was recorded with corresponding displacement. The force-displacement relationship at the top of the central column is shown in Figure 4.11.

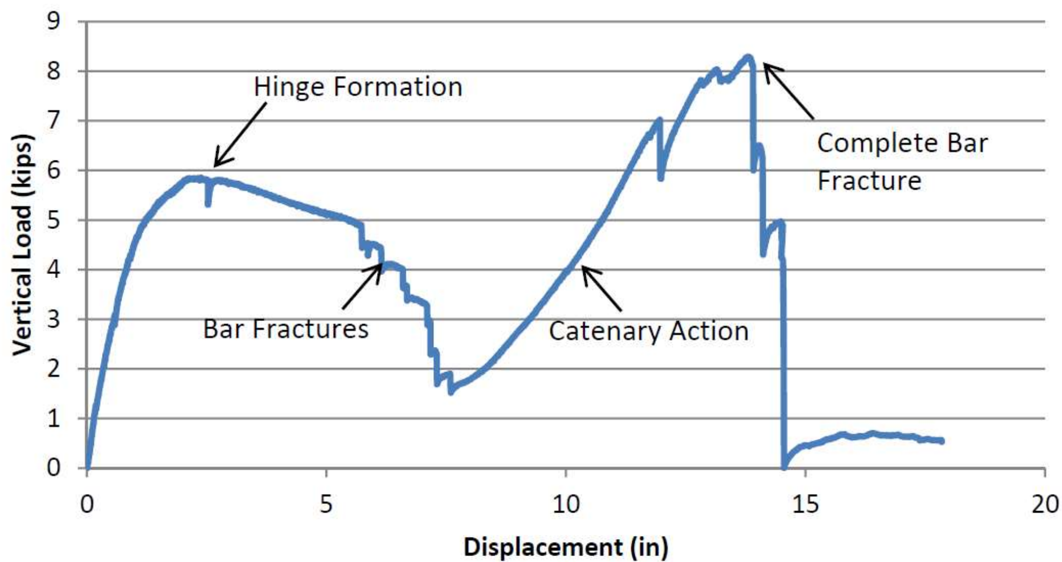


Figure 4.11 Relation of vertical load and displacement at the top of central column (Stinger, 2011)

The frame's collapse resistance consisted of four stages in monotonic loading: the flexural stage, tensile bars fracture stage, catenary action stage and the final failure. Because the test frame had continuous reinforcement and sufficient ductility, flexural behaviors controlled at initial levels of loading until plastic hinges formed. Failure of tensile reinforcement occurred when the strain in the tensile reinforcement reached the ultimate strain, at which point the load dropped rapidly with increasing displacement. Catenary action, developed by the compression bars crossing the plastic hinge regions, provided extra collapse resistance after all tensile bars fractured. With increasing displacement and load, compression bars also fractured after reaching ultimate strain, and the total structure failed. Assumptions of catenary actions presented in Section 2.4 were in agreement with the experiment results.

4.2 Calculation of sectional properties

To use the lumped damaged-plasticity model to analyze the frame with continuous reinforcement, mechanical properties must be calculated. Constants needed in the lumped damaged-plasticity model were calculated following Section 2.3. Material properties used in the experiments by Stinger (2011) and Stinger and Orton (2013) are summarized below, followed by computation of the lumped damaged-plasticity model constants. Mechanical properties of columns and beams were calculated separately.

4.2.1 Material properties

Before the frame experiment, material properties were tested by Stinger (2011). These material properties for concrete and reinforcement are summarized in Table 4.1.

Table 4.1 Experimental material properties for test frame (Stinger 2011)

Material	Property	Value
concrete	Compressive strength of beams f_c' (psi)	3618
	Concrete density w_c (lb/ft^3)	145
#3 reinforcing bars	Elastic modulus E_s (ksi)	13300
	Yield strength f_y (ksi)	58
	Ultimate strength f_u (ksi)	93.9
	ε_{sh}	0.02
	ε_{su}	0.2
#2 reinforcing bars	Elastic modulus E_s (ksi)	17500
	Yield strength f_y (ksi)	84.9
	Ultimate strength f_u (ksi)	97.2
	ε_{sh}	0.027
	ε_{su}	0.1

Steel reinforcement had a yield plateau and strain hardening. Index ε_{sh} is the strain at the start of strain hardening and end of the yield plateau, and ε_{su} is the ultimate strain. Stinger (2011) did not provide E_{sh}, the tangent modulus at the start point of strain hardening, so a value of 1,500 ksi was

assumed for E_{sh} for both #2 bars and #3 bars. The stress-strain relationships of the #2 and #3 bars were assumed to follow Equation (3.5). The value of E_{sh} influenced the behavior of #3 bars in the columns primarily, and had little effect on #2 reinforcing bars in the beams because the yield strength and ultimate strength of #2 bars were similar. During testing, plastic hinges formed in the beams without hinging or failure in the columns. Hence, assuming a value of 1,500 ksi for E_{sh} was not expected to influence simulation results significantly. From material properties in Table 4.1, stress-strain relationships for #2 and #3 reinforcing bars are shown in Figure 4.12.

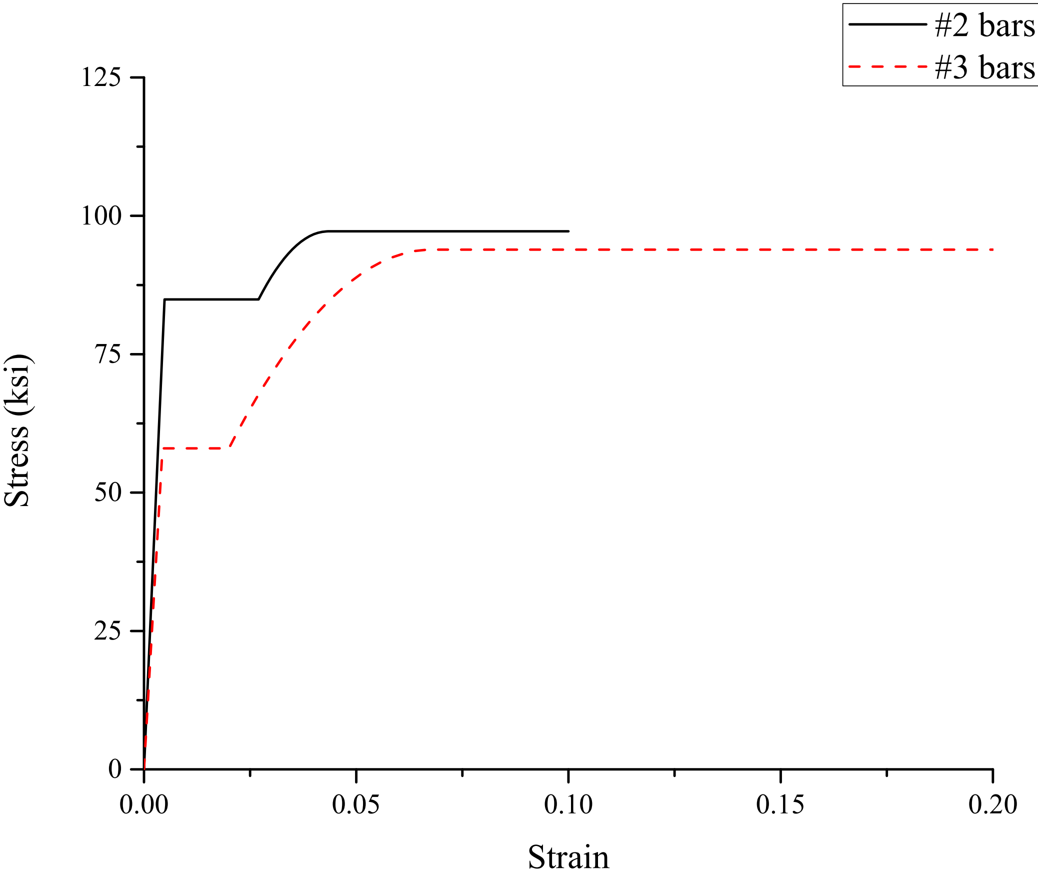


Figure 4.12 Assumed stress-strain relationship for #2 and #3 reinforcing bars

Hognestad's parabola (Hognestad, 1951) was used as the unconfined concrete stress-strain model with 0.0038 as the ultimate strain. Tensile steel in beams (#2 bars) fractured in the experiments on the test frame. Using the hand calculation procedures documented in Section 3.2, the strain of the tensile reinforcement sections resisting positive moments after removal of the central column was 0.018 if concrete failed at the ultimate strain 0.0038. The strain 0.018 is smaller than the ultimate strain 0.1, and thus the ultimate strain for concrete equal to 0.0038 could not allow enough curvature for fracture of tension reinforcement. Therefore, other concrete models were introduced into the analysis to attempt to model the beam up until fracture of the flexural reinforcement. Ultimately, the confined concrete model of Kent and Park (1971) was suggested. The rectangular ties and stirrups used to confine the concrete core were assumed to not be effective enough to increase the maximum compressive strength of concrete significantly, though the confinement was presumed to increase the ductility of the reinforced concrete sections. Before the maximum compressive stress point, the Kent and Park stress-strain relationship of concrete was similar with Hognestad's parabola in Section 3.1 except the strain at maximum stress was defined as 0.002 by Kent and Park. An empirical equation (Equation 4.1) was used to describe post-peak stress development. Concrete failed when the stress reached 20% of maximum compressive stress in the post-peak descending stage.

$$f_c = f_c'' \cdot \left[1 - Z\left(\varepsilon_c - \varepsilon_0\right)\right] \tag{4.1}$$

Variables f_c and ε_c are the stress and strain of concrete separately. Stress f_c'' is the maximum compressive stress and ε_0 is the corresponding strain. Constant Z is the slope describing the linear post-peak stress-strain relation, where Z is calculated by Equation (4.2).

$$Z = \frac{0.5}{\varepsilon_{50u} + \varepsilon_{50h} - \varepsilon_0} \tag{4.2}$$

Index ε_{50u} is strain corresponding to 50% of the maximum compressive stress for unconfined concrete, and ε_{50h} is the difference between ε_{50u} and the strain at 50% maximum stress for confined concrete ε_{50c}. Calculation of ε_{50u} is by Equation (4.3), and the unit of f_c'' is psi.

$$\varepsilon_{50u} = \frac{3 + 0.002 f_c''}{f_c'' - 1000} \tag{4.3}$$

The empirical equation of ε_{50h} is Equation (4.4).

$$\varepsilon_{50h} = \varepsilon_{50c} - \varepsilon_{50u} = \frac{3}{4} p'' \sqrt{\frac{b''}{s}} \tag{4.4a}$$

$$p'' = \frac{2(b'' + d'') A_s''}{b'' d'' s} \tag{4.4b}$$

In Equation (4.4), p'' is the volumetric ratio of transverse reinforcement to confined concrete cores. Dimension d'' and b'' are the depth and width of confined concrete core, s is the center to center spacing of reinforcing hoops, and A_s'' is the area of transverse reinforcement.

Values of variables used in Kent and Park (1971) concrete model for beams of the test frame are calculated and summarized in Table 4.2. For simplification of calculation, concrete tension was ignored after the first cracking point. The stress-strain relationship of concrete based on Kent and Park model is shown in Figure 4.13. Hognestad's model is also included as comparison. It is observed that the Kent and Park (1971) model provides a larger ultimate curvature than Hognestad's parabola (1951), while both models have similar stress-strain development before the

peak. Table 4.3 listed concrete and section properties of columns, which also followed the Kent and Park (1971) model in calculation.

Table 4.2 Concrete and section properties of beams in lumped damaged-plasticity model

Property	Value
f_c'' (psi)	3075
E_c (ksi)	3428
f_r (psi)	451
b'' (in.)	1.93
d'' (in.)	4.93
A_s'' (in.^{2})	0.02
s (in.)	1.50
p'' (in.)	1.92×10^{-2}
ε_0	2×10^{-3}
ε_{50u}	4.4×10^{-3}
ε_{50h}	1.63×10^{-2}
ε_{50c}	2.07×10^{-2}
Z	26.74

Table 4.3 Concrete and section properties of columns in lumped damaged-plasticity model

Property	Value
f_c'' (psi)	3075
E_c (ksi)	3428
f_r (psi)	451
b'' (in.)	4.43
d'' (in.)	4.43
A_s'' (in.^{2})	0.02
s (in.)	4.5
p'' (in.)	0.004
ε_0	2×10^{-3}
ε_{50u}	4.4×10^{-3}
ε_{50h}	2.98×10^{-3}
ε_{50c}	7.4×10^{-3}
Z	92.6

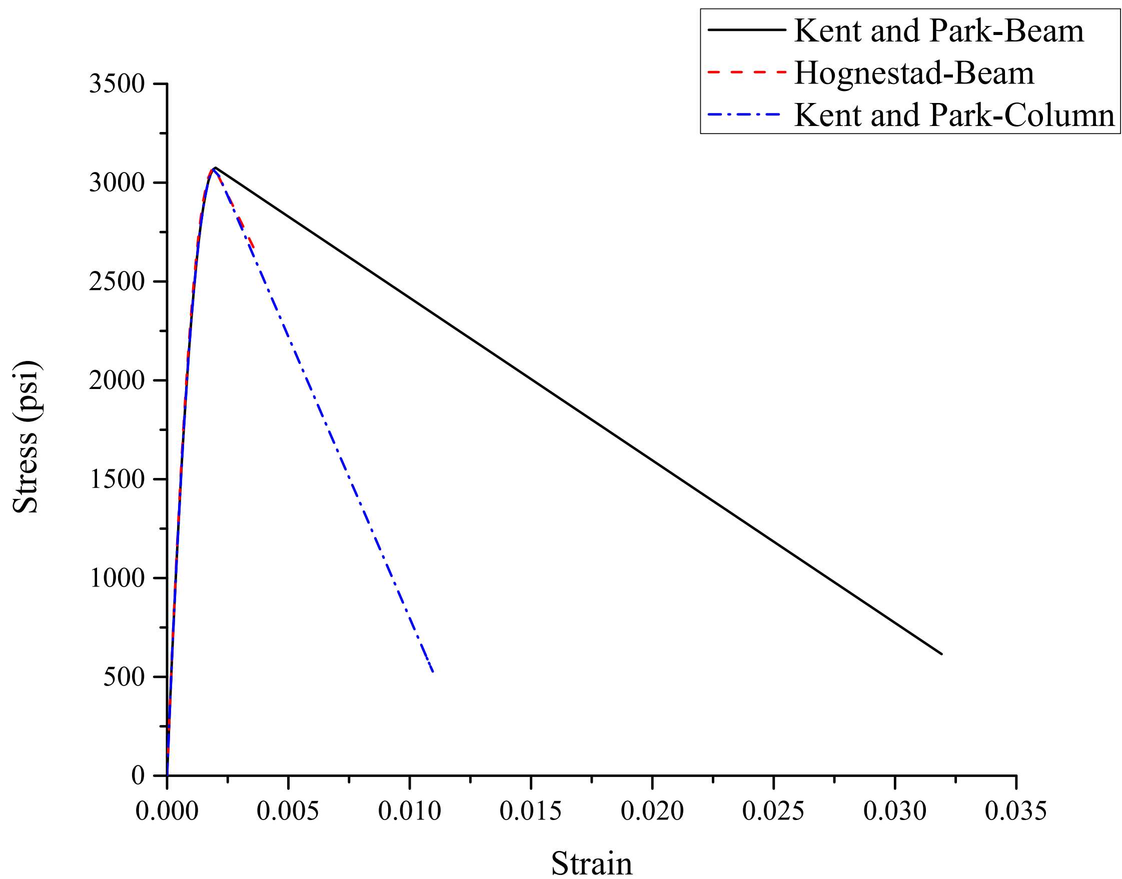


Figure 4.13 Stress-strain relationship of concrete

4.2.2 Mechanical properties of sections

As discussed in Chapter 2, mechanical properties of sections at plastic hinge areas, such as yield moment M_y and ultimate moment M_u, need to be calculated before modeling a beam as a lumped damaged-plasticity element. The lumped damaged-plasticity model can only model plastic hinging at element boundaries, so the first step is to decide upon the possible locations of plastic hinges before computation for the test frame.

Plastic hinge locations are based on details of longitudinal reinforcement and direction of moments applied on beams. Details of the longitudinal reinforcement is shown again in Figure 4.14 with possible locations for plastic hinging. The center column was loaded with increasing downward vertical displacement. Therefore, the beam sections at both sides of the center columns carried positive moment (i.e., bottom bars were in tension), and the beam sections near the left and right column surfaces were loaded with negative moment (i.e., top bars in tension). Places of discontinuous tension reinforcement created critical regions for flexural cracking, as the discontinuous bars will be pulled out given large deformations and concrete cracking. Plastic hinges developed when large flexural cracks pierced depth of sections in large deformation. Hence, positions of plastic hinges area were near positions labeled A and B in Figure 4.14. Under downward vertical loading of the central column, sections at positions A had two #2 bars in tension and four #2 bars in compression, whereas sections at positions B had two #2 bars in tension and three #2 bars in compression. Plastic hinges were expected to begin at 5 in. from the face of center column, and at 12 in. from the surfaces of left and right columns. Assuming cracking in the plastic hinge region at nearly 45 degrees with a beam depth of 6 in., plastic hinge areas were assumed to span from 6 in. to 12 in. of the faces of the left and right columns, and were assumed to span from both faces of the center column out 5.5 in. Experiments conducted by Stinger (2011) showed similar results of plastic hinge areas. From the experimental evidence, flexural cracks formed at 11.5 in. from the faces of left and right columns, and other large flexural cracks formed at 5 in. from the faces of center column. These places were nearly at positions A and B, and were thus in agreement with the assumptions.

Figure 4.14 Plastic hinge positions A and B

Mechanical properties of sections at plastic hinges areas and column sections were calculated following procedures in Section 3.2.1. Mechanical properties of sections at positions A and B are summarized in Table 4.4 and Table 4.5, respectively, and properties of the column sections are in Table 4.6.

Table 4.4 Mechanical properties of sections at A

Category	Index	Value
General Properties	I_g (in.^{4})	54
	f_r (psi)	451
	M_{cr} (k.-in.)	8.12
At Yield Point	c (in.)	1.19
	ε_c	1.39×10^{-3}
	ε_s	4.85×10^{-3}
	ε_s'	6.2×10^{-4}
	f_s (ksi)	84.9
	f_s' (ksi)	10.8
	C_c (kips)	6.24
	T_s (kips)	8.34
	C_s (kips)	2.13
	ϕ_y (in.$^{-1}$)	1.17×10^{-3}
	M_y (k.-in.)	40.6
At Ultimate Point	c (in.)	0.72
	ε_c	1.56×10^{-2}
	ε_s	0.1
	ε_s'	1.30×10^{-3}
	f_s (ksi)	97.2
	f_s' (ksi)	22.73
	C_c (kips)	5.29
	T_s (kips)	9.54
	C_s (kips)	4.46
	ϕ_u (in.$^{-1}$)	2.16×10^{-2}
	M_u (k.-in.)	46.3

Table 4.5 Mechanical properties of sections at B

Category	Index	Value
General Properties	I_g (in.^{4})	54
	f_r (psi)	451
	M_{cr} (k.-in.)	8.12
At Yield Point	c (in.)	1.23
	ε_c	1.45×10^{-3}
	ε_s	4.85×10^{-3}
	ε_s'	6.73×10^{-4}
	f_s (ksi)	84.9
	f_s' (ksi)	11.8
	C_c (kips)	6.45
	T_s (kips)	8.34
	C_s (kips)	1.73
	ϕ_y (in.$^{-1}$)	1.18×10^{-3}
	M_y (k.-in.)	40.5
At Ultimate Point	c (in.)	0.74
	ε_c	1.61×10^{-2}
	ε_s	0.1
	ε_s'	1.74×10^{-3}
	f_s (ksi)	97.2
	f_s' (ksi)	30.4
	C_c (kips)	5.37
	T_s (kips)	9.54
	C_s (kips)	4.48
	ϕ_u (in.$^{-1}$)	2.17×10^{-2}
	M_u (k.-in.)	46.3

Table 4.6 Mechanical properties of sections for columns

Category	Index	Value
General Properties	I_g (in.^{4})	76.3
	f_r (psi)	451
	M_{cr} (k.-in.)	12.5
At Yield Point	c (in.)	1.4
	ε_c	1.81×10^{-3}
	ε_s	4.36×10^{-3}
	ε_s'	8.75×10^{-4}
	f_s (ksi)	58
	f_s' (ksi)	11.6
	C_c (kips)	15.6
	T_s (kips)	19.2
	C_s (kips)	3.85
	ϕ_y (in.$^{-1}$)	1.29×10^{-3}
	M_y (k.-in.)	81.6
At Ultimate Point	c (in.)	1.1
	ε_c	1.06×10^{-2}
	ε_s	3.56×10^{-2}
	ε_s'	3.65×10^{-3}
	f_s (ksi)	79.9
	f_s' (ksi)	48.6
	C_c (kips)	10.8
	T_s (kips)	26.5
	C_s (kips)	16.1
	ϕ_u (in.$^{-1}$)	9.67×10^{-3}
	M_u (k.-in.)	108.8

Mechanical properties of beam sections at other parts of the test frame (i.e., where no plastic hinging was expected to occur) were required to build the rest of the model. Those sections were at areas with layouts designed for original beam sections with negative moments (i.e., near the original supports of each span), with four #2 bars on top and two #2 bars at bottom. After the removal of the center column, those sections were exerted with negative moments at the face of left and right column, and with positive moments at the face of the center column. Mechanical properties for these beam sections are summarized in Table 4.7 and Table 4.8.

Table 4.7 Mechanical properties of beam sections applied with positive moment after column removal

Category	Index	Value
General Properties	I_g (in.^{4})	54
	f_r (psi)	451
	M_{cr} (k.-in.)	8.12
At Yield Point	c (in.)	1.19
	ε_c	1.39×10^{-3}
	ε_s	4.85×10^{-3}
	ε_s'	6.2×10^{-4}
	f_s (ksi)	84.9
	f_s' (ksi)	10.8
	C_c (kips)	6.24
	T_s (kips)	8.34
	C_s (kips)	2.13
	ϕ_y (in.$^{-1}$)	1.17×10^{-3}
	M_y (k.-in.)	40.6
At Ultimate Point	c (in.)	0.72
	ε_c	1.56×10^{-2}
	ε_s	0.1
	ε_s'	1.30×10^{-3}
	f_s (ksi)	97.2
	f_s' (ksi)	22.73
	C_c (kips)	5.29
	T_s (kips)	9.54
	C_s (kips)	4.46
	ϕ_u (in.$^{-1}$)	2.16×10^{-2}
	M_u (k.-in.)	46.3

Table 4.8 Mechanical properties of beam sections applied with negative moment after column removal

Category	Index	Value
General Properties	I_g (in.^{4})	54
	f_r (psi)	451
	M_{cr} (k.-in.)	8.12
At Yield Point	c (in.)	1.95
	ε_c	2.79×10^{-3}
	ε_s	4.85×10^{-3}
	ε_s'	1.85×10^{-3}
	f_s (ksi)	84.9
	f_s' (ksi)	32.3
	C_c (kips)	14.1
	T_s (kips)	16.7
	C_s (kips)	3.2
	ϕ_y (in.$^{-1}$)	1.43×10^{-3}
	M_y (k.-in.)	78.9
At Ultimate Point	c (in.)	1.9
	ε_c	3.19×10^{-2}
	ε_s	8.97×10^{-2}
	ε_s'	2.08×10^{-2}
	f_s (ksi)	97.2
	f_s' (ksi)	84.9
	C_c (kips)	10.4
	T_s (kips)	19.1
	C_s (kips)	8.3
	ϕ_u (in.$^{-1}$)	1.68×10^{-2}
	M_u (k.-in.)	84.1

The reaction frames in the test frame were also included into the numerical model. Details of reaction frames are shown in Figure 4.10, and mechanical properties of the steel elements constituting the reaction frames are listed in Table 4.9 based on the Steel Construction Manual (14th edition, 2011) by American Institute of Steel Construction. Index I_s and Z_s are, respectively, the moment of inertia and section modulus in the strong axis, A is the area, and M_y is the yield moment in the strong axis.

Table 4.9 Material properties and mechanical properties of reaction frame steel elements

Category	Index	Value
Material property	Yield strength f_y (ksi)	60
	Elastic modulus E_s (ksi)	29000
2 welded W14x22's	A (in^2)	12.98
	I_s (in^4)	398
	Z_s (in^3)	66.4
	M_y (k.-in.)	3984
HSS3.5x3.5x1/4	A (in^2)	2.91
	I_s (in^4)	5.04
	Z_s (in^3)	3.5
	M_y (k.-in.)	210
HSS4.5x4.5x5/16	A (in^2)	4.68
	I_s (in^4)	13.5
	Z_s (in^3)	7.27
	M_y (k.-in.)	436.2

4.2.3 Lumped-damaged plasticity elements

Based on mechanical properties of sections and length of elements, three categories of the lumped-damaged plasticity elements were modeled. The first category, Category (a), contained parts of beams which first failed in flexure with formation of plastic hinges on the test frame. Based on analysis in Section 4.2.2, the Category (a) beam parts were from 6 in. at the face of left and right columns to the face of the center column. The parts of beams originally designed for negative moments at the faces of the left and right columns were modeled as the second category, Category (b). Columns were modeled as the last category, Category (c). The beam sections in Category (d) are the beam-column connections with length equal to half the width of the column, and they were modelled as rigid elements. The sketch of element categories and length of each element in the test frame are shown in Figure 4.15. Bold points in Figure 4.15 represent the possible plastic hinge places when a vertical load is applied at the top of center column.

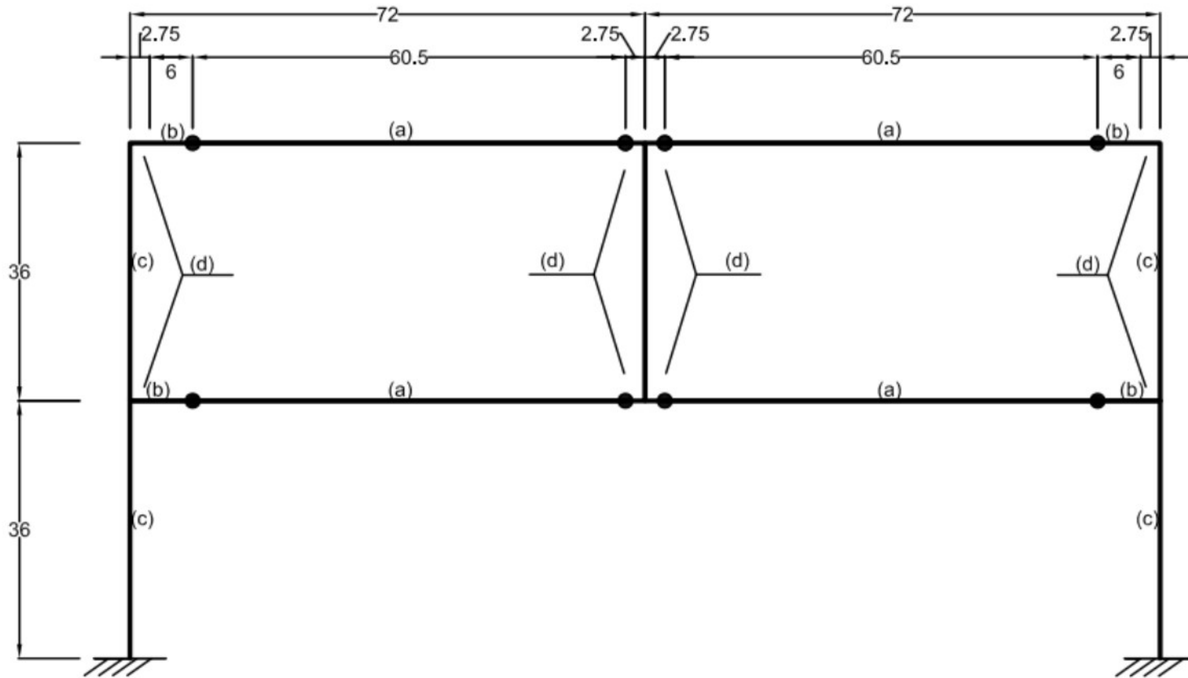


Figure 4.15 Element categories and lengths

To model the lumped-damaged plasticity elements, constants in the yield and damage functions were computed by the methods in Section 2.3. The moment of inertia of the concrete core, instead of the gross moment of inertia for the entire section, was used in simulation because the model was meant to investigate the ultimate strength of members under large deformation. Constants used for the model are listed in Table 4.10.

Table 4.10 Factors for the lumped damaged-plasticity model

Category / Constant	Beam elements with plastic hinges		Beam elements at face of columns exerted with negative moments	Column
	Sections at position A with plastic hinge	Sections at position B with plastic hinge		
L **(in.)**	60.5		6	36
I_{core} **(in.^{4})**	19.27		19.27	32.10
q **(k-in.^{3})**	-0.89	-0.89	-0.29	-1.75
c **(k-in.)**	1094	1098	5700	6051
K_0 **(k-in.)**	62.84	62.51	140.3	106.9
G_{cr} **(k-in.^{3})**	1.9×10^{-3}	1.9×10^{-3}	1.88×10^{-4}	2.3×10^{-3}
M_u **(k-in.)**	46.3	46.27	84.1	108.8
d_u	0.63	0.63	0.63	0.63
A_s' **(in.^{2})**	0.20	0.15	0.10	0.33
L_p **(in.)**	3.0		1.0	3.72
L_d **(in.)**	14.1		14.1	14.5

After calculation of constants for the lumped-damaged plasticity elements, the model of the test frame was formed by OpenSees. The reaction frame was modelled by elastic elements because of its large yield moment and tension capacity compared with the reinforced concrete test frame. The OpenSees code is attached in the Appendix.

4.3 Outcomes of the lumped-damaged plasticity model

Using the factors in Table 4.10 and the methodology presented in Chapter 2, the lumped-damaged plasticity model for the test frame was built and analyzed in OpenSees. The top of the center column was loaded with increasing vertical displacement statically. The load applied at the top of center column versus vertical displacement is shown in Figure 4.16, and the results are compared with experimental data provided by Stinger (2011). Important data about the results of the simulation and experiment are provided in Table 4.11.

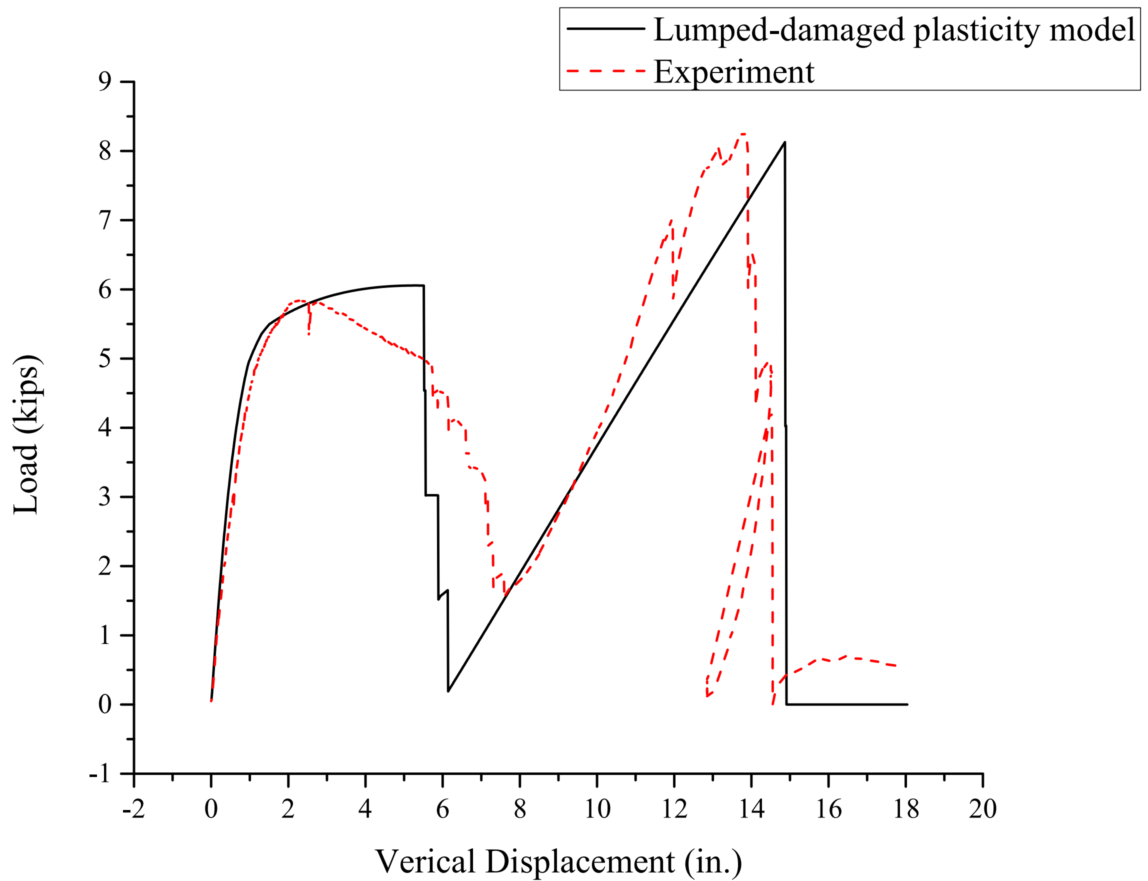


Figure 4.16 Comparison of vertical load and displacement at the top of center column between lumped damaged-plasticity model and Stinger (2011) experimental data

Table 4.11 Data of the experiment and simulation

Data point		Simulation	Experiment
Yield	Displacement (in.)	1.66	2.19
	Load (kips)	5.55	5.81
Flexural failure	Displacement (in.)	5.51	5.68
	Load (kips)	6.02	4.95
Ultimate stage	Displacement (in.)	14.9	13.9
	Load (kips)	8.14	8.24
Dissipated energy (k-in.)		67.7	72.5

When small displacement was applied, the test frame was mainly controlled by flexural behaviors. Before yielding, results from the model were similar with ones of the experiment, and the yield points were nearly the same. However, differences started to accumulate from the yield point to the first bars fracture point. Loading capacity increased with increasing displacement for the model, while the experiment showed a descending trend. Using the material properties from Stringer (2011), the expected ultimate load capacities at flexural failure were expected to be larger than the capacities in yield points, a prediction at odds with the experimental observations. Therefore, the difference of loads was believed to be from uncertainties in the material modeling. If the reinforcing bars in the test frame experiment had no plastic hardening after yielding, the descending trend observed in the experimental data was plausible.

Tension reinforcement started to fracture after exhaustion of flexural capacities, and hinges formed in plastic hinge areas. The simulation and experiment results showed bar fractures at four discrete times with sudden drops of resistance in flexural failure. At the start of flexural failure, the difference of displacement between the simulation and experiment was just 4.5%. In the simulation, the fracture of tension reinforcement and formation of hinges followed the sequence: position A on the first floor, position A on the second floor, position B on the first floor, position B on the second floor, where position labels are shown in Figure 4.14. However, in the experiment, compression reinforcement at positions B fractured instead of the tension reinforcement, in contrast with the behavior assumed for the simulation.

Catenary action, computed according to the assumptions listed in Section 2.4, started to control after flexural failure. In simulation, because tension reinforcement had fractured, compression bars provided axial resistance for reinforced concrete beams. Results of the simulation and experiment

were similar. Orton (2007) assumed that the deflection at which catenary action began was nearly equal with the height of the beam. Catenary action started to control at a 6.15 in. vertical displacement in simulation, which is close to the assumption of Orton (2007). Four #2 compression bars at position A, three #2 compression bars at position B, and the elastic concrete part in the middle provided axial resistance. But in the experiment, two #2 tension reinforcement at position B held the catenary action, while the simulation assumed tension reinforcement fractured in flexural failure. The picture of position B in the experiment is shown in Figure 4.17. The failure of compression reinforcement might have been caused by buckling, and needs further research. Nevertheless, the simulation results are similar with the experiment ones in catenary action.

Figure 0.17 Beams at position B in catenary action (Stinger, 2011)

At the ultimate stage (where the total failure of the test frame happened), the differences of the simulation and experiment are 7.2% in displacement and 1.2% in resistance load. A 6.6% difference of dissipated energy is shown for the simulation and experiment during the total loading

stage. Small differences of the yield points, flexural failure points, ultimate points and dissipated energies suggested that the lumped damaged-plasticity model provided good accuracy for the test frame.

In the lumped damaged-plasticity model, the reaction frame was modeled with elastic elements. The largest moment exerted on the members of the reaction frame was listed and compared with the corresponding yield moment in Table 4.12.

Table 4.12 Modeled moments of the reaction frame members

Member type	Yield moment (k-in.)	Maximum moment in simulation (k-in.)
2 welded W14x22's	3984	73.5
HSS3.5x3.5x1/4	210	1.13
HSS4.5x4.5x5/16	436.2	2.28

The moments exerted on members of the reaction frame were far less than the corresponding yield moments. Consequently, modeling the reaction frame with elastic elements did not compromise the simulation at all.

The total calculation time for running the lumped damaged-plasticity model in OpenSees was less than one minute. Hence, the preceding comparison suggested that the lumped damaged-plasticity model provided high efficiency and little sacrifice of accuracy in static progressive collapse analysis on the test frame.

4.4 Nonlinear dynamic analysis

As a demonstration of the capabilities of the lumped-damaged plasticity model, nonlinear dynamic analysis was applied to the test frame after the efficiency and accuracy were shown for the developed method under nonlinear static conditions in Section 4.3. Material behaviors, section

dimensions, layouts, and reaction frames were the same as the static model described in Section 4.2. The configuration of the nonlinear dynamic model was similar with the corresponding static one. To achieve a nonlinear dynamic analysis procedure in progressive collapse, the middle column on the first floor was first modeled with the same properties as the other columns. After application of the static loads, the middle column on the first floor was removed suddenly, and the dynamic behavior of the frame with time was computed. More information was needed to get an accurate progressive collapse analysis for the test frame, such as values and distributions of live load and superimposed dead load, damping, and environmental loads on the corresponding full-scale structure from which the frame was designed. To make use of the static analysis results in Section 4.3, 2 vertical loads were exerted on the top of the center column, and their magnitudes were chosen from the flexure failure point and the yield point. Damping was stiffness-proportional Rayleigh damping based on the 1st mode, and the damping ratio in the first mode was set to 5%. Because of a lack of detailed information and experimental data, this nonlinear dynamic simulation for progressive collapse analysis was just a trial on the test frame. The accuracy of nonlinear dynamic simulation results requires further research and experimental verification. 0.01 second was chosen as time increment in dynamic analysis. Time period of the structure is 0.4 second. Consequently, 0.01 second as time increment will not cause problems such as resonance.

A concentrated downward load of 5.5 kips was applied at the top of the center column on the second floor. This load simulated the effects of loads on the structure before failure of one main structure element (in this case, the middle column on the first floor). First yield occurred in the beams at a load of 6.0 kips during static analysis, as shown in Figure 4.16. After applying the load statically, the center column on the first floor was removed suddenly, and a transient analysis was applied in OpenSees. In addition to the nonlinear dynamic analysis, a linear dynamic analysis was

applied with the same procedure, except that all members were elastic. Outcomes of the linear dynamic analysis and nonlinear dynamic analysis are shown in Figure 4.18 and Figure 4.19. Figure 4.18 presents the time series for the vertical displacement at the top of the center column on the second floor, and Figure 4.19 shows the time series for moments in the beams on the second floor at positions A as labeled in Figure 4.14 (i.e., plastic hinge location near the center column). Table 4.13 lists peak values and final values for vertical displacements and moments.

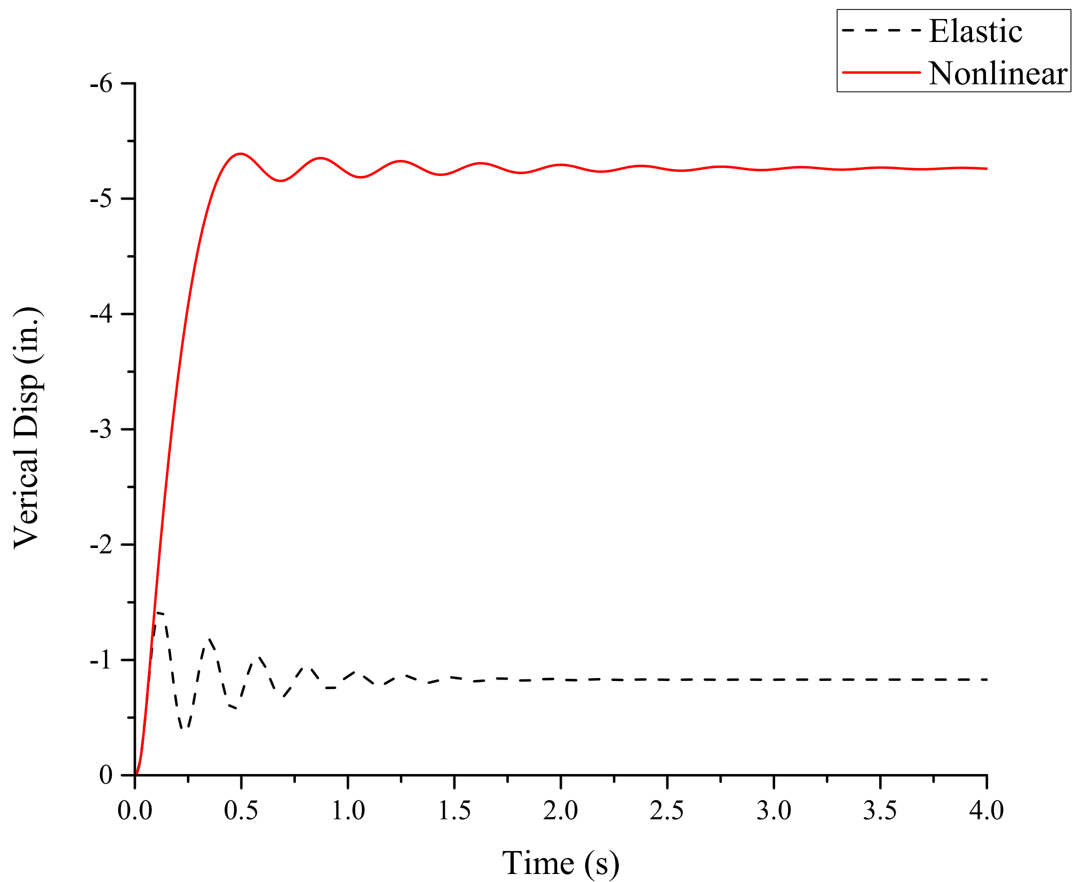


Figure 4.18 Time series for the vertical displacement at the top of center column on second floor (with 5.5-kip load)

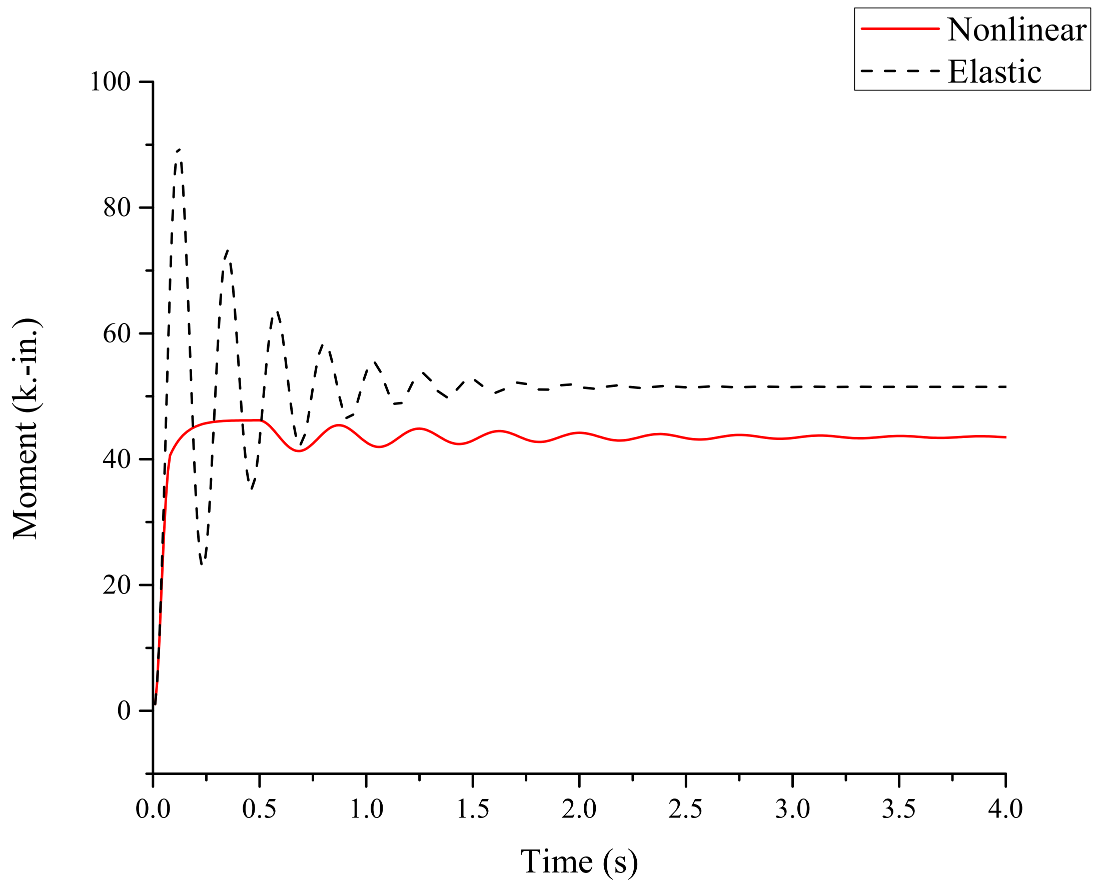


Figure 4.19 Time series for moments of beams on the second floor at position A (with 5.5-kip load)

Table 4.13 Peak values and final values for 5.5-kip load

Category	Nonlinear analysis		Linear analysis	
	Static	Dynamic	Static	Dynamic
Peak value for displacement (in.)	1.18	2.36	0.73	1.26
Final value for displacement (in.)	1.18	1.93	0.73	0.73
Peak value for moment (k-in.)	42.3	45.7	46.4	80.1
Final value for moment (k-in.)	42.3	41.6	46.4	46.4

From Table 4.13, linear dynamic analysis always returned the same results as linear static analysis after vibrations had damped to zero. Linear dynamic analysis results amplified the maximum displacement and moment in linear static analysis by 1.73, and thus the dynamic amplification factor for linear analysis of the analyzed test frame was effectively 1.75. In nonlinear analysis, the dynamic procedure presented much higher peak and final displacements than the static procedure, while the ultimate moments and peak moments were similar between to the two methods because of nonlinearity and yielding in the materials. Consequently, the dynamic procedure in important to get accurate displacement actions in nonlinear analysis.

At the top of the center column on the second floor, the vertical displacement in nonlinear analysis was larger than elastic analysis in both peak value and final value. In Figure 4.19, moments in nonlinear dynamic analysis were smaller than ones in elastic analysis because of material nonlinearity. Both Figure 4.18 and Figure 4.19 suggest with a vertical loading of 5.5 kips at the top of the center column on the second floor, the test frame survived from progressive collapse.

A concentrated downward load of 6 kips was then applied at the top of the center column on the second floor. This magnitude of 6 kips is similar with the flexural failure load in nonlinear static analysis in Table 4.11. This load was originally proposed to examine the efficacy of catenary action in arresting the failure of the frame after flexural failure. Outcomes of the linear analysis and nonlinear analysis are shown in Figure 4.20 and Figure 4.21.

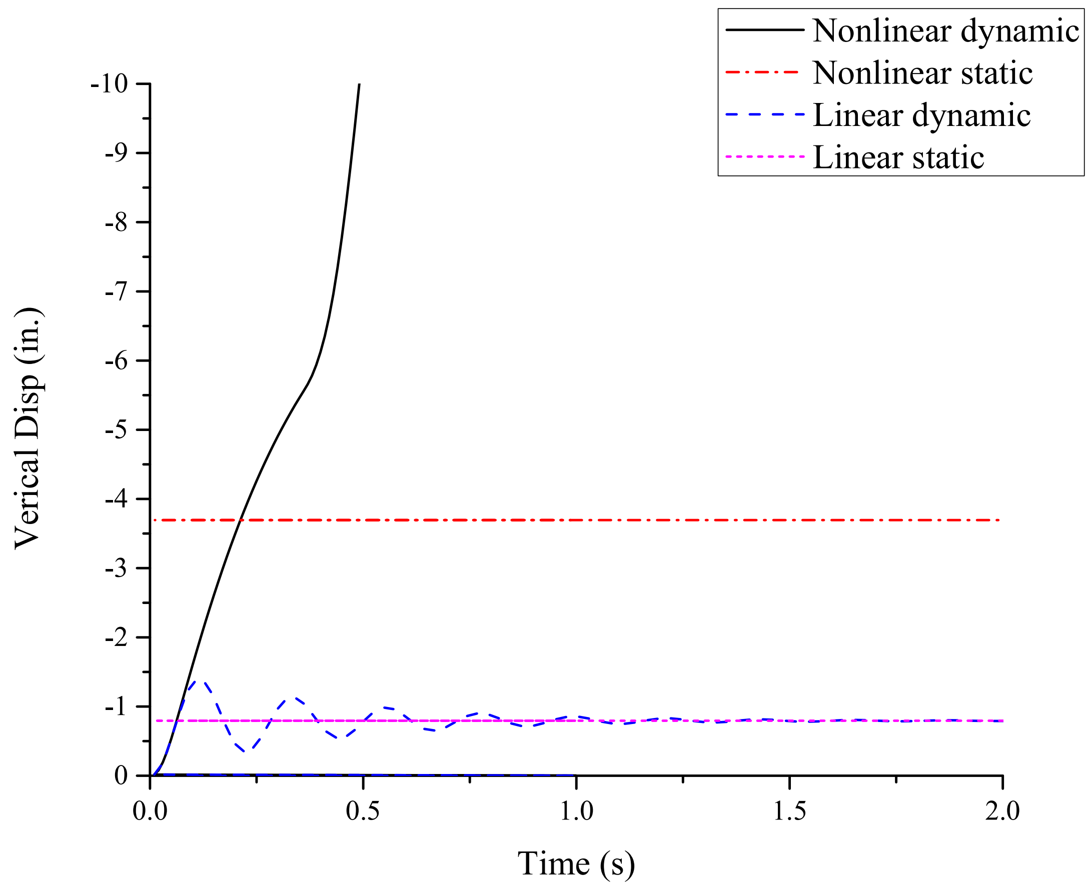


Figure 4.20 Time series for the vertical displacement at the top of center column on second floor (with 6-kip load)

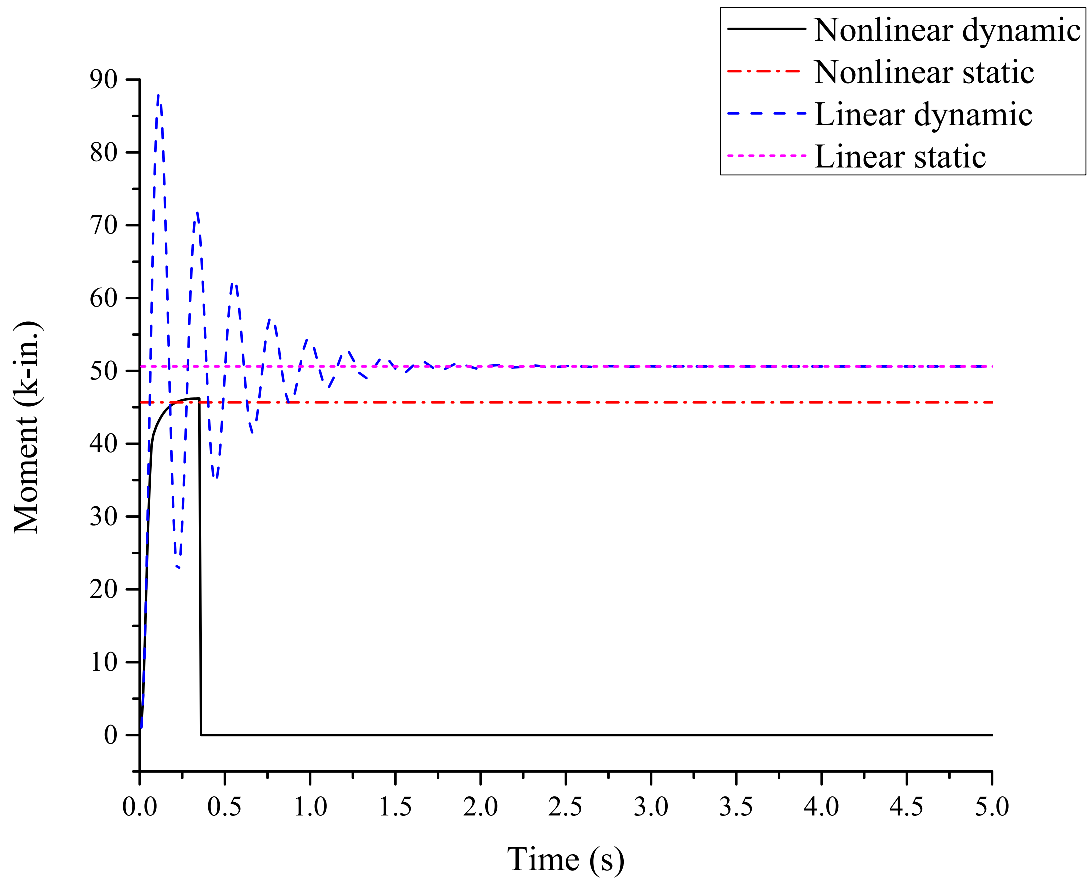


Figure 4.21 Time series for moments of beams on the second floor at positions A (with 6-kip load)

With a 6-kip vertical load at the top of center column, hinges formed at positions labeled as A and B in progressive collapse analysis. Figure 4.21 suggests a flexural failure for the beams on the second floor at positions A at nearly 0.36 s. Although flexural failures had happened, catenary action provided resistance by developing axial forces in the beams. Figure 4.22 shows the time series for the axial force of beams on the second floor.

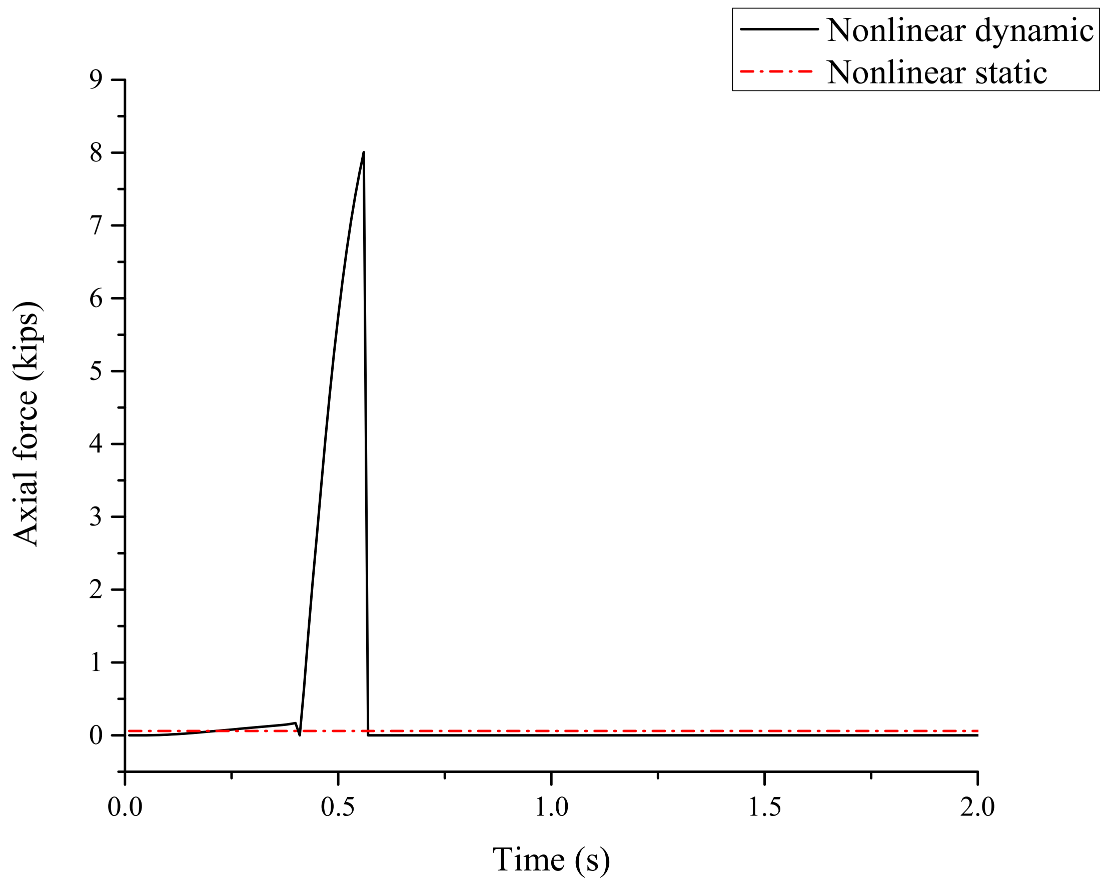


Figure 4.22 Time series for axial forces of beams on the second floor (with 6-kip load)

In nonlinear static analysis, the axial force was negligible because the structure was still controlled by the flexural capacity with a 6-kip load at the top of center column. All data of nonlinear analysis and linear analysis for a 6-kip load are summarized in Table 4.14.

Table 4.14 Peak values and final values for 6 kips load

Category	Nonlinear analysis		Linear analysis	
	Static	Dynamic	Static	Dynamic
Peak value for displacement (in.)	3.69		0.79	1.38
Final value for displacement (in.)	3.69		0.79	0.79
Peak value for moment (k-in.)	45.7	46.2	50.6	80.1
Final value for moment (k-in.)	45.7	0	50.6	50.6
Peak value for axial force (kips)	0.06	8.14		
Final value for axial force (kips)	0.06	0		

Comparing Table 4.13 and Table 4.14, with 0.5-kip increase of load, results of linear analysis results are similar for both loads. However, for the nonlinear analysis the results differ significantly. With a 6-kip load, the structure developed hinges and catenary action during the nonlinear dynamic procedure, and the structure collapsed.

Conclusively, the test frame survived a 5.5-kip load but not a 6-kip load at the top of the center column in progressive collapse. The reactions were completely different in nonlinear dynamic analysis for these two loads. Catenary action provided extra resistance in progressive collapse in the 6-kip load situation, but was ultimately unable to arrest collapse. The nonlinear dynamic analysis was presumed to provide a more accurate representation of the deflections and reaction of the structure compared with static analysis and nonlinear dynamic analysis. However, the dynamic results from the lumped damaged-plasticity method in this section need further experimental validation and more detailed information about the structure.

Chapter 5. Conclusions and future research

The lumped damaged-plasticity method was validated in static analysis in Chapter 3 and Chapter 4. Conclusions and deficiencies of the lumped damaged-plasticity method are summarized in this chapter, and possible future research directions are documented.

5.1 Conclusions and deficiencies

In this paper, the lumped damaged-plasticity method is introduced and developed as an efficient and accurate way to capture reactions of structures subjected to progressive collapse. The lumped damaged-plasticity method was developed from the flexural model by Flórez-López (1995). In order to apply the model into progressive collapse analysis, resistance from catenary action was introduced. Theories of the lumped damaged-plasticity model are discussed in Chapter 2. In Chapter 3, capabilities of the method in flexural analysis were validated by comparing results with ones from an ABAQUS model and the fiber element method. A reinforced concrete cantilever beam was used as the analysis specimen. In Chapter 4, the lumped damaged-plasticity method was applied to static progressive collapse of a quarter scaled 2-bay-by-2-story frame from an office building designed following ACI 318-08. The method showed accuracy and efficiency compared with results from experiments on the same test frame by Stinger (2011). Nonlinear dynamic progressive collapse analysis was applied on the test frame with the lumped damaged-plasticity method as a demonstration of the capabilities of the developed methodology.

The conclusions drawn from the results of this research are listed as follows:

1. In progressive collapse analysis, flexural capacity and catenary action provided the main resistance mechanisms for the analyzed frame structure. As increasing load was applied to

the structure, flexural resistance controlled first. Catenary action was the main resistance mechanism after flexural failure.

2. The resistance provided by catenary action is from geometric nonlinearity and tensile axial force in the reinforced concrete beams. This can only develop if the structure is designed with enough ductility in the flexural members. After flexural failure causes fracture of tensile reinforcement, axial resistance of former compressive reinforcement provides catenary resistance under large deformation. In dynamic progressive collapse analysis, results from Section 4.4 suggest that the catenary action provides resistance to structures, but may not always be sufficient to arrest collapse.
3. The nonlinear dynamic procedure in progressive collapse analysis is important. Compared with static procedures, the nonlinear dynamic procedure provides more detailed reactions and displacements of structures.
4. The lumped damaged-plasticity method is shown to be an efficient and accurate method to compute static and dynamic reactions of the frame structure researched in this paper. The calculation time for the test frame was less than one minute for both static and dynamic analyses. Results of the static analysis fit experimental results well. Computational code for the lumped damaged-plasticity is attached in Appendix, and further analysis and revision with the lumped damaged-plasticity method are straightforward.

At the same time, the research has deficiencies and unfinished work because of restrictions of the lumped damaged-plasticity method, limited time, and limited quantities of experimental data.

Firstly, the availability of experimental data for structures subjected to progressive collapse analysis is inadequate. Large-scale progressive collapse experiments can be exorbitantly

expensive, and require proper experimental facilities and instrumentation. The lack of experimental data limits comparison of the results of the lumped damaged-plasticity method to only a few structures. This problem is especially true for dynamic progressive collapse analysis.

Secondly, the mechanism of catenary action requires more research in order to accurately predict an ultimate catenary action capacity. Orton (2007) stated that tension force in reinforced concrete beams was transferred by stirrups if discontinuous reinforcement provided axial resistance, and this transferal was not completely efficient. However, the efficiency of the transferal of catenary forces was not quantified. In calculation of tensile strains in the reinforcement during catenary action, the development length plus the plastic hinge length were assumed to concentrate a beam's axial deformation at each end of an element. While the assumptions for catenary action in the lumped damaged-plasticity method provided good accuracy for the analyzed test frame in Chapter 4, more comparisons are required for proper validation of these assumptions.

Lastly, the flexural analysis of the lumped damaged-plasticity method may need to be revised. The lumped damaged-plasticity method may not be able to provide accurate flexural analysis for members with degrading moment capacities after yielding. This is because of the method's definition in yield functions and damage functions in Sections 2.2.2 and 2.2.3 (Equation 2.8, Equation 2.11). With increasing moments, damage and plastic deformations accumulate when the loading condition reaches the yield surface $f = 0$ and damage surface $g = 0$. Though the method has been verified with other modeling techniques in Section 3.4, in some cases the moment capacity may decrease with increasing curvature after yielding. If the lumped damaged-plasticity model is applied for elements with this feature, the damage reaches ultimate damage d_u at the yield point and fails to accumulate after yielding. Additionally, because of ignoring certain details of the

moment-curvature relationship (for example, the curvature at yield) in defining yield surface functions and damage functions, analysis results are not correct with a wrong moment of inertia I. Proper selection of the moment of inertia that incorporates damage along the length of the element while returning the correct moment-curvature behavior requires further research.

Although the deficiencies of the lumped damaged-plasticity method require further research to overcome, the lumped damaged-plasticity method can be a quick and efficient check for progressive collapse resistance of structures.

5.2 Future research

The simulation results and conclusions suggest the need for further research in progressive collapse. Suggestions of further research directions are listed as follows:

1. Yield functions and damage functions in the lumped damaged-plasticity method can be improved to simulate general reinforced concrete sections. As stated in Section 5.1, currently the yield functions and damage functions are not accurate for RC sections with decreasing moment capacities after yield.
2. Additional tests on reinforced concrete frames in progressive collapse, especially in dynamic progressive collapse analysis, could better support the lumped damaged-plasticity method in this paper.
3. The lumped damaged-plasticity method in this paper is only applied to 2-D reinforced concrete frame structures. The method can be expanded to more types of structures and to more dimensions.
4. Slabs in reinforced concrete structure may be simulated with revised lumped damaged-

plasticity method. As important members, slabs hold people and property, and may provide considerable robustness with respect to progressive collapse. Development of collapse analysis techniques for slabs is necessary. The developed lumped damaged-plasticity methodology could be extended to slabs by subdividing the slab into sets of continuous beams in a grillage analogy.

References

1. ACI Committee, American Concrete Institute, & International Organization for Standardization. (2014). Building code requirements for structural concrete (ACI 318-14) and commentary. American Concrete Institute.
2. AISC committee (2011). Steel Construction Manual, 14th Edition, First Print. American Institute of Steel Construction.
3. ASCE/SEI Seismic Rehabilitation Standards Committee. (2007). Seismic Rehabilitation of Existing Buildings (ASCE/SEI 41-06). *American Society of Civil Engineers, Reston, VA*.
4. Bao, Y., Kunnath, S. K., El-Tawil, S., & Lew, H. S. (2008). Macromodel-based simulation of progressive collapse: RC frame structures. *Journal of Structural Engineering*, *134*(7), 1079-1091.
5. Bao, Y., Sadek, F., Main, J. A., Pujol, S., & Sozen, M. A. (2011). *An experimental and computational study of reinforced concrete assemblies under a column removal scenario*. US Department of Commerce, National Institute of Standards and Technology.
6. Bazant, Z. P., & Planas, J. (1997). *Fracture and size effect in concrete and other quasibrittle materials* (Vol. 16). CRC press.
7. Bischoff, P. H. (2007). Rational model for calculating deflection of reinforced concrete beams and slabs. *Canadian Journal of Civil Engineering*, *34*(8), 992-1002.
8. Broyden, C. G. (1965). "A Class of Methods for Solving Nonlinear Simultaneous Equations". Mathematics of Computation. American Mathematical Society. 19 (92): 577–593.
9. Byfield, M. P. (2006). Behavior and design of commercial multistory buildings subjected to blast. *Journal of performance of constructed facilities*, *20*(4), 324-329.
10. Cipollina, A., López-Inojosa, A., & Flórez-López, J. (1995). A simplified damage mechanics approach to nonlinear analysis of frames. *Computers & Structures*, *54*(6), 1113-1126.
11. Dassault Systémes Simulia CorpHibbett, Karlsson, & Sorensen, Inc. (2012). ABAQUS/standard: User's Manual(Vol. 1). Hibbitt, Karlsson & Sorensen, Inc.
12. Du, K., Sun, J., & Xu, W. (2012). Evaluation of Section and Fiber Integration Points in Fiber Model. 15th World Conference on Earthquake Engineering.
13. Dusenberry, D. O., & Hamburger, R. O. (2006). Practical means for energy-based analyses of disproportionate collapse potential. *Journal of Performance of Constructed Facilities*, *20*(4), 336-348.
14. Fadhil, A. T. (2012). Simplified analysis to predict the behavior of RC beams under collapse (Doctoral dissertation, University of Missouri--Columbia).
15. Flórez-López, J. (1995). Simplified model of unilateral damage for RC frames. *Journal of Structural Engineering*, *121*(12), 1765-1772.
16. Fu, F. (2009). Progressive collapse analysis of high-rise building with 3-D finite element modeling method. *Journal of Constructional Steel Research*,*65*(6), 1269-1278.
17. General Services Administration (GSA) (2003). Progressive collapse analysis and design guidelines for new federal office buildings and major modernizations projects. GSA Guidelines.
18. General Services Administration (GSA) (2013). Alternate Path Analysis and Design Guidelines for Progressive Collapse Resistance. GSA Guidelines.
19. Gere, J. M., & Timoshenko, S. P. (2001). Mechanics of materials Brooks.Cole, Pacific Grove, CA, 815-39.
20. Hognestad, E. (1951). A study on combined bending and axial load in reinforced concrete

members. Univ. of Illinois Engineering Experiment Station, Univ. of Illinois at Urbana-Champaign, IL, 43-46.

21. Izzuddin, B. A. (2005). A simplified model for axially restrained beams subject to extreme loading. *International Journal of Steel Structures*, *5*(5), 421-429.
22. Jirásek, M., & Bazant, Z. P. (2002). *Inelastic analysis of structures*. John Wiley & Sons.
23. Karsan, I. D., & Jirsa, J. O. (1969). Behavior of concrete under compressive loadings. Journal of the Structural Division, 95(12), 2543-2564.
24. Kent, D. C., & Park, R. (1971). Flexural members with confined concrete. Journal of the Structural Division.
25. Keshavarzian, M., & Schnobrich, W. C. (1985). Inelastic analysis of R/C coupled shear walls. *Earthquake Engineering & Structural Dynamics*, *13*(4), 427-448.
26. Kokot, S., Anthoine, A., Negro, P., & Solomos, G. (2012). Static and dynamic analysis of a reinforced concrete flat slab frame building for progressive collapse. *Engineering Structures*, *40*, 205-217.
27. Kwasniewski, L. (2010). Nonlinear dynamic simulations of progressive collapse for a multistory building. *Engineering Structures*, *32*(5), 1223-1235.
28. Lai, S. S., & Will, G. T. (1986). R/C space frames with column axial force and biaxial bending moment interactions. *Journal of Structural Engineering*,*112*(7), 1553-1572.
29. Le, J. L., & Xue, B. (2014). Probabilistic analysis of reinforced concrete frame structures against progressive collapse. *Engineering Structures*, *76*, 313-323.
30. Leyendecker, E. V., & Ellingwood, B. (1977). *Design methods for reducing the risk of progressive collapse in buildings* (No. 98). US Dept. of Commerce, National Bureau of Standards.
31. Lubliner, J., Oliver, J., Oller, S., & Onate, E. (1989). A plastic-damage model for concrete. International Journal of solids and structures, 25(3), 299-326.
32. Marante, M. E., & Flórez-López, J. (2002). Model of damage for RC elements subjected to biaxial bending. *Engineering Structures*, *24*(9), 1141-1152.
33. Marante, M. E., & Flórez-López, J. (2003). Three-dimensional analysis of reinforced concrete frames based on lumped damage mechanics.*International Journal of Solids and Structures*, *40*(19), 5109-5123.
34. Mariano, P. M. (2000). Premises to a Multifield Approach to Stochastic Damage Evolution. *Damage and fracture of disordered materials*, Chapter 6. New York: Springer.
35. Marjanishvili, S. M. (2004). Progressive analysis procedure for progressive collapse. *Journal of Performance of Constructed Facilities*, *18*(2), 79-85.
36. Marjanishvili, S., & Agnew, E. (2006). Comparison of various procedures for progressive collapse analysis. *Journal of Performance of Constructed Facilities*, *20*(4), 365-374.
37. McKay, A., Marchand, K., & Diaz, M. (2012). Alternate path method in progressive collapse analysis: Variation of dynamic and nonlinear load increase factors. *Practice Periodical on Structural Design and Construction*,*17*(4), 152-160.
38. Murtha-Smith, E. (1988). Alternate path analysis of space trusses for progressive collapse. *Journal of Structural Engineering*, *114*(9), 1978-1999.
39. OpenSees-About. Retrieved March 16, 2016, from http://opensees.berkeley.edu/OpenSees/home/about.php
40. Orton, S. L. (2007). Development of a CFRP system to provide continuity in existing reinforced concrete buildings vulnerable to progressive collapse. ProQuest.
41. Perdomo, M. E., Ramírez, A., & Flórez-López, J. (1999). Simulation of damage in RC frames

with variable axial forces. *Earthquake engineering & structural dynamics*, *28*(3), 311-328.
42. Petrangeli, M., Pinto, P. E., & Ciampi, V. (1999). Fiber element for cyclic bending and shear of RC structures. I: Theory. *Journal of Engineering Mechanics*, *125*(9), 994-1001.
43. Ruth, P., Marchand, K. A., & Williamson, E. B. (2006). Static equivalency in progressive collapse alternate path analysis: reducing conservatism while retaining structural integrity. *Journal of Performance of Constructed Facilities*,*20*(4), 349-364.
44. Sasani, M., & Kropelnicki, J. (2008). Progressive collapse analysis of an RC structure. *The Structural Design of Tall and Special Buildings*, *17*(4), 757-771.
45. Stinger, S. M. (2011). Evaluation of alternative resistance mechanisms for progressive collapse (Doctoral dissertation, University of Missouri--Columbia).
46. Stinger, S. M., & Orton, S. L. (2013). Experimental evaluation of disproportionate collapse resistance in reinforced concrete frames. *ACI Structural journal*, *110*(3), 521.
47. Taucer, F., Spacone, E., & Filippou, F. C. (1991). *A fiber beam-column element for seismic response analysis of reinforced concrete structures* (Vol. 91, No. 17). Berkekey, California: Earthquake Engineering Research Center, College of Engineering, University of California.
48. Wang, C. K., Salmon, C. G, J Pincheira. (2006). *Reinforced concrete design*.
49. Yi, W. J., He, Q. F., Xiao, Y., & Kunnath, S. K. (2008). Experimental study on progressive collapse-resistant behavior of reinforced concrete frame structures. ACI Structural Journal, 105(4), 433.
50. Zhang, J., Wang, Q., Hu, S., & Wang, C. (2008). Parameters of the concrete damaged-plasticity model in ABAQUS. *Buliding structures,* 38(8), 127-130. (In Chinese: 张劲, 王庆扬, 胡守营, & 王传甲. (2008). ABAQUS 混凝土损伤塑性模型参数验证. *建筑结构, 38*(8), 127-130.)

Appendix

Three files are needed to build the lumped damaged-plasticity method into OpenSees: .h file (header file), main .cpp file (main code file) and another .cpp file to connect OpenSees with TclEditor. These three files are included in the Appendix.

Appendix 1. Header file

```
#ifndef LumpedDamageEle2D2_h
#define LumpedDamageEle2D2_h

#include <Element.h>
#include <Node.h>
#include <Matrix.h>
#include <Vector.h>

class Channel;
class Information;
class CrdTransf;
class Response;
class Renderer;
class FEM_ObjectBroker;

class LumpedDamageEle2D2 : public Element
{
public:
	LumpedDamageEle2D2();
	LumpedDamageEle2D2(int tag, double E, double I, double A,
		double qq, double c, double k0, double Gcr, double Mu, double du,
		double qq2, double c2, double k02, double Gcr2, double Mu2, double du2,
		double As1, double As2, double fu, double strain_u, double Lp,
		int Nd1, int Nd2, CrdTransf &theTransf,
		double rho = 0.0, int cMass = 0);
	~LumpedDamageEle2D2();

	const char *getClassType(void) const { return "LumpedDamageEle2D2"; };

	int getNumExternalNodes(void) const;
	const ID &getExternalNodes(void);
	Node **getNodePtrs(void);

	int getNumDOF(void);
	void setDomain(Domain *theDomain);

	int commitState(void);
	int revertToLastCommit(void);
	int revertToStart(void);

	int update(void);
```

```
	const Matrix &getTangentStiff(void);
	const Matrix &getInitialStiff(void);
	const Matrix &getMass(void);

	void zeroLoad(void);
	int addLoad(ElementalLoad *theLoad, double loadFactor);
	int addInertiaLoadToUnbalance(const Vector &accel);

	const Vector &getResistingForce(void);
	const Vector &getResistingForceIncInertia(void);

	int sendSelf(int commitTag, Channel &theChannel);
	int recvSelf(int commitTag, Channel &theChannel, FEM_ObjectBroker &theBroker);

	void Print(OPS_Stream &s, int flag = 0);
	int displaySelf(Renderer &theViewer, int displayMode, float fact, const char
**modes = 0, int numModes = 0);

	Response *setResponse(const char **argv, int argc, OPS_Stream &s);
	int getResponse(int responseID, Information &info);

	int setParameter(const char **argv, int argc, Parameter &param);
	int updateParameter(int parameterID, Information &info);

private:

	double E, I, A;     // area, elastic modulus, moment of inertia
	double qq, c, k0, Gcr;  // coeff. of lumped damage model
	double du, du2, Mu,Mu2;
	double qq2, c2, k02, Gcr2;
	double As1, As2,  fu, strain_u, Lp;
	double axial,plastic_strain;
	double rho;       // mass per unit length
	int cMass;        // consistent mass flag


	double compense_1;
	double compense_2;
	double Mi;
	double Mj;

	static Matrix K;
	static Vector P;
	Vector Q;

	static Matrix kb;
	Vector q;

	double q0[3];  // Fixed end forces in basic system
	double p0[3];  // Reactions in basic system
	double dv[3];
	double cv[3];
	double d[3];
	double rot_p[3];

	// damage variables
	double di_po, di_neg, dj_po, dj_neg;
	double rot_p_i_po, rot_p_i_neg, rot_p_j_po, rot_p_j_neg;
```

```
        // converged history damage variables
        double Cdi_po, Cdi_neg, Cdj_po, Cdj_neg;
        double Crot_p_i_po, Crot_p_i_neg, Crot_p_j_po, Crot_p_j_neg;

        Node *theNodes[2];

        ID  connectedExternalNodes;

        CrdTransf *theCoordTransf;


};

#endif
```

Appendix 2. Main .cpp file

```
#include "LumpedDamageEle2D2.h"
#include <ElementalLoad.h>
#include <Element.h>

#include <Vector.h>
#include <Matrix.h>
#include <MatrixUtil.h>

#include <Domain.h>
#include <Channel.h>
#include <FEM_ObjectBroker.h>

#include <CrdTransf.h>
#include <Information.h>
#include <Parameter.h>
#include <ElementResponse.h>
#include <Renderer.h>

#include <math.h>
#include <stdlib.h>
#include <elementAPI.h>
#include <string>
#include <stdio.h>
#include <DummyNode.h>

Matrix LumpedDamageEle2D2::K(6, 6);
Vector LumpedDamageEle2D2::P(6);
Matrix LumpedDamageEle2D2::kb(3, 3);

// set all to zero
LumpedDamageEle2D2::LumpedDamageEle2D2()
	:Element(0, ELE_TAG_LumpedDamageEle2D),
	E(0.0), I(0.0), A(0.0), qq(0.0), c(0.0), k0(0.0), Gcr(0.0), Mu(0.0), du(0.0),
	qq2(0.0), c2(0.0), k02(0.0), Gcr2(0.0), Mu2(0.0), du2(0.0),
	As1(0.0),As2(0.0),fu(0.0),strain_u(0.0), Lp(0.0),
	rho(0.0), cMass(0),
	Q(6), q(3), connectedExternalNodes(2), theCoordTransf(0),
	di_po(0.0), di_neg(0.0), dj_po(0.0) , dj_neg(0.0),
	rot_p_i_po(0.0), rot_p_i_neg(0.0), rot_p_j_po(0.0), rot_p_j_neg(0.0),
	Cdi_po(0.0), Cdi_neg(0.0), Cdj_po(0.0), Cdj_neg(0.0),
	Crot_p_i_po(0.0), Crot_p_i_neg(0.0), Crot_p_j_po(0.0), Crot_p_j_neg(0.0),
compense_1(0.0), compense_2(0.0), Mi(0.0), Mj(0.0), axial(0.0),plastic_strain(0.0)
{
	// does nothing
	q0[0] = 0.0;
	q0[1] = 0.0;
	q0[2] = 0.0;

	p0[0] = 0.0;
	p0[1] = 0.0;
	p0[2] = 0.0;

	dv[0] = 0.0;
	dv[1] = 0.0;
```

```
        dv[2] = 0.0;

        cv[0] = 0.0;
        cv[1] = 0.0;
        cv[2] = 0.0;

        d[0] = 0.0;
        d[1] = 0.0;
        d[2] = 0.0;

        rot_p[0] = 0.0;
        rot_p[1] = 0.0;
        rot_p[2] = 0.0;
        // set node pointers to NULL
        for (int i = 0; i<2; i++)
               theNodes[i] = 0;
}
// apply inputs
LumpedDamageEle2D2::LumpedDamageEle2D2(int tag, double e, double i, double a, double QQ,
double C, double K0, double gcr, double mu, double DU,
        double QQ2, double C2, double K02, double gcr2, double mu2, double DU2,
        double as1, double as2, double FU, double STRAIN_U, double lp,
        int Nd1, int Nd2, CrdTransf &coordTransf,
    double r, int cm)
        :Element(tag, ELE_TAG_LumpedDamageEle2D),
        E(e), I(i), A(a), qq(QQ), c(C), k0(K0), Gcr(gcr), Mu(mu), du(DU),
        qq2(QQ2), c2(C2), k02(K02), Gcr2(gcr2), Mu2(mu2), du2(DU2),
        As1(as1), As2(as2), fu(FU),strain_u(STRAIN_U), Lp(lp),
        rho(r), cMass(cm),
        connectedExternalNodes(2), theCoordTransf(0),
        di_po(0.0), di_neg(0.0), dj_po(1.0), dj_neg(1.0),
        rot_p_i_po(0.0), rot_p_i_neg(0.0), rot_p_j_po(0.0), rot_p_j_neg(0.0),
        Cdi_po(0.0), Cdi_neg(0.0), Cdj_po(0.0), Cdj_neg(0.0),
        Crot_p_i_po(0.0), Crot_p_i_neg(0.0), Crot_p_j_po(0.0), Crot_p_j_neg(0.0),
        Q(6), q(3), compense_1(0.0), compense_2(0.0), Mi(0.0), Mj(0.0), axial(0.0),
plastic_strain(0.0)
{
        connectedExternalNodes(0) = Nd1;
        connectedExternalNodes(1) = Nd2;

         if (E <= 0.0)  {
    opserr << "BeamWithHinges2d::BeamWithHinges2d -- input parameter E is <= 0.0\n";
      exit(-1);
    }
    if (I <= 0.0)  {
      opserr << "BeamWithHinges2d::BeamWithHinges2d -- input parameter I is <= 0.0\n";
      exit(-1);
    }
   if (A <= 0.0)  {
     opserr << "BeamWithHinges2d::BeamWithHinges2d -- input parameter A is <= 0.0\n";
     exit(-1);
    }

        theCoordTransf = coordTransf.getCopy2d();

        if (!theCoordTransf) {
               opserr << "LumpedDamageEle2D2::LumpedDamageEle2D2 -- failed to get copy of
coordinate transformation\n";
```

```
		exit(01);
	}

	q0[0] = 0.0;
	q0[1] = 0.0;
	q0[2] = 0.0;

	p0[0] = 0.0;
	p0[1] = 0.0;
	p0[2] = 0.0;

	dv[0] = 0.0;
	dv[1] = 0.0;
	dv[2] = 0.0;

	cv[0] = 0.0;
	cv[1] = 0.0;
	cv[2] = 0.0;

	d[0] = 0.0;
	d[1] = 0.0;
	d[2] = 0.0;

	rot_p[0] = 0.0;
	rot_p[1] = 0.0;
	rot_p[2] = 0.0;
	// set node pointers to NULL
	theNodes[0] = 0;
	theNodes[1] = 0;

}

LumpedDamageEle2D2::~LumpedDamageEle2D2()
{
	if (theCoordTransf)
		delete theCoordTransf;
}

int
LumpedDamageEle2D2::getNumExternalNodes(void) const
{
	return 2;
}

const ID &
LumpedDamageEle2D2::getExternalNodes(void)
{
	return connectedExternalNodes;
}

Node **
LumpedDamageEle2D2::getNodePtrs(void)
{
	return theNodes;
}

int
LumpedDamageEle2D2::getNumDOF(void)
```

```
{
	return 6;
}

void
LumpedDamageEle2D2::setDomain(Domain *theDomain)
{
	if (theDomain == 0) {
		opserr << "LumpedDamageEle2D2::setDomain -- Domain is null\n";
		exit(-1);
	}
	// check nodes
	theNodes[0] = theDomain->getNode(connectedExternalNodes(0));
	theNodes[1] = theDomain->getNode(connectedExternalNodes(1));

	if (theNodes[0] == 0) {
		opserr << "LumpedDamageEle2D2::setDomain -- Node 1: " <<
connectedExternalNodes(0) << " does not exist\n";
		exit(-1);
	}

	if (theNodes[1] == 0) {
		opserr << "LumpedDamageEle2D2::setDomain -- Node 2: " <<
connectedExternalNodes(1) << " does not exist\n";
		exit(-1);
	}

	int dofNd1 = theNodes[0]->getNumberDOF();
	int dofNd2 = theNodes[1]->getNumberDOF();

	if (dofNd1 != 3) {
		opserr << "LumpedDamageEle2D2::setDomain -- Node 1: " <<
connectedExternalNodes(0)
			<< " has incorrect number of DOF\n";
		exit(-1);
	}

	if (dofNd2 != 3) {
		opserr << "LumpedDamageEle2D2::setDomain -- Node 2: " <<
connectedExternalNodes(1)
			<< " has incorrect number of DOF\n";
		exit(-1);
	}

	this->DomainComponent::setDomain(theDomain);

	if (theCoordTransf->initialize(theNodes[0], theNodes[1]) != 0) {
		opserr << "LumpedDamageEle2D2::setDomain -- Error initializing coordinate
transformation\n";
		exit(-1);
	}

	double L = theCoordTransf->getInitialLength();

	if (L == 0.0) {
		opserr << "LumpedDamageEle2D2::setDomain -- Element has zero length\n";
		exit(-1);
	}
```

```
}

int
LumpedDamageEle2D2::commitState()
{

        int retVal = 0;
        // call element commitState to do any base class stuff
        if ((retVal = this->Element::commitState()) != 0) {
                opserr << "LumpedDamageEle2D2::commitState () - failed in base class";
        }
        retVal += theCoordTransf->commitState();
        if (Mi>0){
                if (d[1]>Cdi_po)
                {Cdi_po=d[1];}
                if (rot_p[1]>Crot_p_i_po){
                Crot_p_i_po=rot_p[1];
                }
        }
        else if (Mi<0){
                if (d[1]>Cdi_neg)
                {Cdi_neg=d[1];}
                if (rot_p[1]<Crot_p_i_neg){
                Crot_p_i_neg=rot_p[1];
                }
        }
        if (Mj>0){
                if (d[2]>Cdj_po)
                {Cdj_po=d[2];}
                if (rot_p[2]>Crot_p_j_po){
                Crot_p_j_po=rot_p[2];
                }
        }
        else if (Mj<0){
                if (d[2]>Cdj_neg)
                {Cdj_neg=d[2];}
                if (rot_p[2]<Crot_p_j_neg){
                Crot_p_j_neg=rot_p[2];
                }
        }
        di_po=Mi;
        di_neg=Mj;
        double L=theCoordTransf->getInitialLength();
        static Vector v(3);
        v=theCoordTransf->getBasicTrialDisp();
        double rot_p_u_i=dj_po*(Mu-(1-du)*k0)/((1-du)*c);
        double rot_p_u_j=dj_neg*(Mu2-(1-du2)*k02)/((1-du2)*c2);
        double h=6;
        double Ltotal=60.5;
        const Vector& disp1 = theNodes[0]->getDisp();
     const Vector& disp2 = theNodes[1]->getDisp();

     Vector diff = disp2-disp1;
        double angle=abs(diff(1))/L;

        if (angle > h/Ltotal && cv[0] == 0)
                cv[0]=v(0);
```

```
	int eleTag = this->getTag();
	if (dj_neg != 0 && (abs(rot_p[2]) >= rot_p_u_j || d[2] > du2)){
		d[2]=du2;
		dj_neg=0;


		opserr << "element: " << eleTag << "node: " << connectedExternalNodes(1) <<
"has failed" << endln;}
	if (dj_po != 0 && ((rot_p[1]) >= rot_p_u_i || d[1] > du )){
		d[1]=du;
		dj_po=0;

		opserr << "element: " << eleTag << "node: " << connectedExternalNodes(0) <<
"has failed" << endln;}

	cv[1]=v(1);
	cv[2]=v(2);
	axial=v(0);
	return retVal;
}

int
LumpedDamageEle2D2::revertToLastCommit()
{
	return theCoordTransf->revertToLastCommit();
}

int
LumpedDamageEle2D2::revertToStart()
{
	return theCoordTransf->revertToStart();
}

int
LumpedDamageEle2D2::update(void)
{
	return theCoordTransf->update();
}

const Matrix &
LumpedDamageEle2D2::getTangentStiff(void)
{
	theCoordTransf->update();
	static Vector v(3);
	v=theCoordTransf->getBasicTrialDisp();
	double L=theCoordTransf->getInitialLength();

	int signal;
	double EoverL = E / L;
	double EAoverL = A*EoverL;			// EA/L
	double EI = E*I;
	double Lover3EI=L/(3*EI);
	double accu=pow((double)10,-10);

	double cd1=d[1];
	double cd2=d[2];
```

```
        //decide loading, unloading or reloading, need to get M first
        double denomina = L*(d[1] + d[2] - d[1]* d[2] + 3);
        if (abs(di_po)< pow((double)1.0,-2)){
               di_po=0;}
        if (abs(di_neg)< pow((double)1.0,-2)){
               di_neg=0;}

Mi = dj_po*(12 * EI*(1 - d[1])*((v(1)-compense_1) - rot_p[1]) / denomina + 6 * EI*(1 -
d[1])*(1 - d[2])*((v(2)-compense_2) - rot_p[2]) / denomina);
Mj = dj_neg*(6 * EI*(1 - d[1])*(1 - d[2])*((v(1)-compense_1) - rot_p[1]) / denomina + 12
* EI*(1 - d[2])*((v(2)-compense_2) - rot_p[2]) / denomina);
if (Mi*di_po<0){

        if (Mi > 0){
                       compense_1=cv[1]-Crot_p_i_po;
                              d[1] = Cdi_po;
                              rot_p[1] = Crot_p_i_po;
                      }
                      else if (Mi < 0){
                              compense_1=cv[1]-Crot_p_i_neg;
                              d[1] = Cdi_neg;
                              rot_p[1] = Crot_p_i_neg;
                      }
}
if (Mj*di_neg<0){

        if (Mj > 0){
                       compense_2=cv[2]-Crot_p_j_po;
                              d[2] = Cdj_po;
                              rot_p[2] = Crot_p_j_po;
                      }
                      else if (Mj < 0){
                              compense_2=cv[2]-Crot_p_j_neg;
                              d[2] = Cdj_neg;
                              rot_p[2] = Crot_p_j_neg;
                      }
}

        //assign changed values
        denomina = L*(d[1] + d[2] - d[1]* d[2] + 3);
        Mi = dj_po*(12 * EI*(1 - d[1])*((v(1)-compense_1) - rot_p[1]) / denomina + 6 *
EI*(1 - d[1])*(1 - d[2])*((v(2)-compense_2) - rot_p[2]) / denomina);
        Mj = dj_neg*(6 * EI*(1 - d[1])*(1 - d[2])*((v(1)-compense_1) - rot_p[1]) /
denomina + 12 * EI*(1 - d[2])*((v(2)-compense_2) - rot_p[2]) / denomina);

        double f1 = Mi - dj_po*(12 * EI*(1 - d[1])*((v(1)-compense_1) - rot_p[1]) /
denomina + 6 * EI*(1 - d[1])*(1 - d[2])*((v(2)-compense_2) - rot_p[2]) / denomina);
        double f2 = Mj - dj_neg*(6 * EI*(1 - d[1])*(1 - d[2])*((v(1)-compense_1) -
rot_p[1]) / denomina + 12 * EI*(1 - d[2])*((v(2)-compense_2) - rot_p[2]) / denomina);

        double fi = abs(Mi - dj_po*(1 - d[1])*c*rot_p[1]) - dj_po*(1 - d[1])*k0;
        double gi = pow(Mi,2) * (L / (3 * EI)) / (2 * (pow(1-d[1],2))) - dj_po*(Gcr +
qq*log(1 - d[1]) / (1 - d[1]));
        double fj = abs(Mj - dj_neg*(1 - d[2])*c2*rot_p[2]) - dj_neg*(1 - d[2])*k02;
        double gj = pow(Mj,2) * (L / (3 * EI)) / (2 * (pow(1-d[2],2))) - dj_neg*(Gcr2 +
qq2*log(1 - d[2]) / (1 - d[2]));

        if (fi <= 0 && gi > 0 && fj <= 0 && gj <= 0){
```

```
            static Matrix B(3, 3);
            B(0,0)=1;
            B(0,1)=0;
            B(0,2)=-dj_po*((12*EI*(compense_1 - v(1) + rot_p[1]))/denomina -
(6*EI*(d[2] - 1)*(compense_2 - v(2) + rot_p[2]))/denomina + (12*EI*(d[1] - 1)*(d[2] -
1)*(compense_1 - v(1) + rot_p[1]))/(pow(denomina,2)/L) - (6*EI*(d[1] - 1)*pow((d[2] -
1),2)*(compense_2 - v(2) + rot_p[2]))/(pow(denomina,2)/L));
            B(1,0)=0;
            B(1,1)=1;
            B(1,2)=dj_neg*((6*EI*(d[2] - 1)*(compense_1 - v(1) + rot_p[1]))/denomina -
(12*EI*pow((d[2] - 1),2)*(compense_2 - v(2) + rot_p[2]))/(pow(denomina,2)/L) +
(6*EI*(d[1] - 1)*pow((d[2] - 1),2)*(compense_1 - v(1) + rot_p[1]))/(pow(denomina,2)/L));
        B(2,0)=(L*Mi)/(3*EI*pow((d[1] - 1),2));
            B(2,1)=0;
            B(2,2)=dj_po*(qq/pow((d[1] - 1),2) - (qq*log(1 - d[1]))/pow((d[1] - 1),2))
- (L*pow(Mi,2))/(3*EI*pow((d[1] - 1),3));

            static Matrix B0(3, 3);
            invert3by3Matrix(B, B0);

            static Matrix s(3, 1);
            static Matrix s_trans(1, 3);
            static Matrix y(3, 1);
            static Matrix x_plus(3, 1);
            static Matrix x(3, 1);
            static Matrix F(3, 1);
            static Matrix F_plus(3, 1);
            double sby;

            for (int i = 0; i < 10000; i++){
                    signal=0;
                    x(0, 0) = Mi;
                    x(1, 0) = Mj;
                    x(2, 0) = d[1];
                    //define F
                    denomina = L*(d[1] + d[2] - d[1]* d[2] + 3);
                    F(0, 0) = f1 = Mi - dj_po*(12 * EI*(1 - d[1])*((v(1)-compense_1) -
rot_p[1]) / denomina + 6 * EI*(1 - d[1])*(1 - d[2])*((v(2)-compense_2) - rot_p[2]) /
denomina);
                    F(1, 0) = f2 = Mj - dj_neg*(6 * EI*(1 - d[1])*(1 - d[2])*((v(1)-
compense_1) - rot_p[1]) / denomina + 12 * EI*(1 - d[2])*((v(2)-compense_2) - rot_p[2]) /
denomina);
                    F(2, 0) = gi = pow(Mi, 2) * (L / (3 * EI)) / (2 * (pow(1 - d[1],
2))) - dj_po*(Gcr + qq*log(1 - d[1]) / (1 - d[1]));
                    static Matrix mid1(3, 1);
                    mid1.addMatrixProduct(0.0, B0, F, 1.0);
                    for (int j = 0; j <= 2; j++){
                            x_plus(j, 0) = x(j, 0) - mid1(j, 0);
                            s(j, 0) = x_plus(j, 0) - x(j, 0);
                            s_trans(0, j) = s(j, 0);
                    }

                    Mi = x_plus(0, 0);
                    Mj = x_plus(1, 0);
                    d[1] = x_plus(2, 0);

                    if (abs(s(0, 0)) <= accu && abs(s(1, 0)) <=accu && abs(s(2, 0)) <=
accu){
```

```
                    if (abs(Mi)>Mu && Mi != 0){
                          Mi=dj_po*Mu*abs(Mi)/Mi;
                          signal=1;
                    }
                    if (abs(Mj)>Mu2 && Mj != 0){
                          Mj=dj_neg*Mu2*abs(Mj)/Mj;
                          signal=1;
                    }
                    if (d[1]>du){
                          d[1]=du;
                          signal=1;
                    }
                    if (signal == 1) continue;
                    else break;
                    }

                    F_plus(0, 0) = f1 = Mi - dj_po*(12 * EI*(1 - d[1])*((v(1)-
compense_1) - rot_p[1]) / denomina + 6 * EI*(1 - d[1])*(1 - d[2])*((v(2)-compense_2) -
rot_p[2]) / denomina);
                    F_plus(1, 0) = f2 = Mj - dj_neg*(6 * EI*(1 - d[1])*(1 -
d[2])*((v(1)-compense_1) - rot_p[1]) / denomina + 12 * EI*(1 - d[2])*((v(2)-compense_2) -
rot_p[2]) / denomina);
                    F_plus(2, 0) = gi = pow(Mi, 2) * (L / (3 * EI)) / (2 * (pow(1 -
d[1], 2))) - dj_po*(Gcr + qq*log(1 - d[1]) / (1 - d[1]));
                    for (int k = 0; k <= 2; k++){
                          y(k, 0) = F_plus(k, 0) - F(k, 0);
                    }
                    static Matrix mid2(3, 1);
                    static Matrix mid3(3, 1);
                    static Matrix mid4(3, 3);
                    static Matrix mid5(3, 3);
                    mid2.addMatrixProduct(0.0, B0, y, 1.0);
                    for (int m = 0; m <= 2; m++){
                          mid3(m, 0) = s(m, 0) - mid2(m, 0);
                    }
                    mid4.addMatrixProduct(0.0, mid3, s_trans, 1.0);
                    mid5.addMatrixProduct(0.0, mid4, B0, 1.0);
                    sby = s(0, 0)*mid2(0, 0) + s(1, 0)*mid2(1, 0) + s(2, 0)*mid2(2, 0);
                    for (int n = 0; n <= 2; n++){
                          for (int o = 0; o <= 2; o++){
                                B0(n, o) += mid5(n, o) / sby;
                          }
                    }
              }
       }
       else if (fi > 0 && gi > 0 && fj <= 0 && gj <= 0){
              static Matrix B(4, 4);
              //f=[f1 f2 fi gi];
              //df = [diff(f, 'Mi'); diff(f, 'Mj'); diff(f, 'd[1]'); diff(f,
'rot_p[1]')]'
              B(0,0)=1;
              B(0,1)=0;
              B(0,2)=-dj_po*((12*EI*(compense_1 - v(1) + rot_p[1]))/denomina -
(6*EI*(d[2] - 1)*(compense_2 - v(2) + rot_p[2]))/denomina + (12*EI*(d[1] - 1)*(d[2] -
1)*(compense_1 - v(1) + rot_p[1]))/(pow(denomina,2)/L) - (6*EI*(d[1] - 1)*pow((d[2] -
1),2)*(compense_2 - v(2) + rot_p[2]))/(pow(denomina,2)/L));
              B(0,3)=-(12*EI*dj_po*(d[1] - 1))/denomina;
        B(1,0)=0;
```

```
            B(1,1)=1;
            B(1,2)=dj_neg*((6*EI*(d[2] - 1)*(compense_1 - v(1) + rot_p[1]))/denomina -
(12*EI*pow((d[2] - 1),2)*(compense_2 - v(2) + rot_p[2]))/(pow(denomina,2)/L) +
(6*EI*(d[1] - 1)*pow((d[2] - 1),2)*(compense_1 - v(1) + rot_p[1]))/(pow(denomina,2)/L));
            B(1,3)=(6*EI*dj_neg*(d[1] - 1)*(d[2] - 1))/denomina;
        B(2,0)=abs(Mi + c*dj_po*rot_p[1]*(d[1] - 1))/(Mi + c*dj_po*rot_p[1]*(d[1] - 1));
            B(2,1)=0;
            B(2,2)=dj_po*k0 + c*dj_po*rot_p[1]*abs(Mi + c*dj_po*rot_p[1]*(d[1] -
1))/(Mi + c*dj_po*rot_p[1]*(d[1] - 1));
            B(2,3)=c*dj_po*abs(Mi + c*dj_po*rot_p[1]*(d[1] - 1))/(Mi +
c*dj_po*rot_p[1]*(d[1] - 1))*(d[1] - 1);
        B(3,0)=(L*Mi)/(3*EI*pow((d[1] - 1),2));
        B(3,1)=0;
            B(3,2)=dj_po*(qq/pow((d[1] - 1),2) - (qq*log(1 - d[1]))/pow((d[1] - 1),2))
- (L*pow(Mi,2))/(3*EI*pow((d[1] - 1),3));
            B(3,3)=0;


            static Matrix B0(4, 4);
            invertMatrix(4, B, B0);

            static Matrix s(4, 1);
            static Matrix s_trans(1, 4);
            static Matrix y(4, 1);
            static Matrix x_plus(4, 1);
            static Matrix x(4, 1);
            static Matrix F(4, 1);
            static Matrix F_plus(4, 1);
            double sby;

            for (int i = 0; i < 10000; i++){
                  signal=0;
                  x(0, 0) = Mi;
                  x(1, 0) = Mj;
                  x(2, 0) = d[1];
                  x(3, 0) = rot_p[1];
                  //define F
                  denomina = L*(d[1] + d[2] - d[1]* d[2] + 3);
                  F(0, 0) = f1 = Mi - dj_po*(12 * EI*(1 - d[1])*((v(1)-compense_1) -
rot_p[1]) / denomina + 6 * EI*(1 - d[1])*(1 - d[2])*((v(2)-compense_2) - rot_p[2]) /
denomina);
                  F(1, 0) = f2 = Mj - dj_neg*(6 * EI*(1 - d[1])*(1 - d[2])*((v(1)-
compense_1) - rot_p[1]) / denomina + 12 * EI*(1 - d[2])*((v(2)-compense_2) - rot_p[2]) /
denomina);
                  F(2, 0) = fi = abs(Mi - dj_po*(1 - d[1])*c*rot_p[1]) - dj_po* (1 -
d[1])*k0;
                  F(3, 0) = gi = pow(Mi, 2) * (L / (3 * EI)) / (2 * (pow((1 - d[1]),
2))) - dj_po*(Gcr + qq*log(1 - d[1]) / (1 - d[1]));
                  static Matrix mid1(4, 1);
                  mid1.addMatrixProduct(0.0, B0, F, 1.0);
                  for (int j = 0; j <= 3; j++){
                        x_plus(j, 0) = x(j, 0) - mid1(j, 0);
                        s(j, 0) = x_plus(j, 0) - x(j, 0);
                        s_trans(0, j) = s(j, 0);
                  }

                  Mi = x_plus(0, 0);
                  Mj = x_plus(1, 0);
```

```
                    d[1] = x_plus(2, 0);
                    rot_p[1] = x_plus(3, 0);

                    //don't forget to change
                    if (abs(s(0, 0)) <=accu && abs(s(1, 0)) <=accu && abs(s(2, 0))
<=accu && abs(s(3, 0)) <=accu){
                    if (abs(Mi)>Mu && Mi != 0){
                           Mi=dj_po*Mu*abs(Mi)/Mi;
                           signal=1;
                    }
                    if (abs(Mj)>Mu2 && Mj != 0){
                           Mj=dj_neg*Mu2*abs(Mj)/Mj;
                           signal=1;
                    }
                    if (d[1]>du){
                           d[1]=du;
                           signal=1;
                    }
                    if (signal == 1) continue;
                    else break;
                    }

                    F_plus(0, 0) = f1 = Mi - dj_po*(12 * EI*(1 - d[1])*((v(1)-
compense_1) - rot_p[1]) / denomina + 6 * EI*(1 - d[1])*(1 - d[2])*((v(2)-compense_2) -
rot_p[2]) / denomina);
                    F_plus(1, 0) = f2 = Mj - dj_neg*(6 * EI*(1 - d[1])*(1 -
d[2])*((v(1)-compense_1) - rot_p[1]) / denomina + 12 * EI*(1 - d[2])*((v(2)-compense_2) -
rot_p[2]) / denomina);
                    F_plus(2, 0) = fi = abs(Mi - dj_po*(1 - d[1])*c*rot_p[1]) - dj_po*
(1 - d[1])*k0;
                    F_plus(3, 0) = gi = pow(Mi, 2) * (L / (3 * EI)) / (2 * (pow((1 -
d[1]), 2))) - dj_po*(Gcr + qq*log(1 - d[1]) / (1 - d[1]));
                    for (int k = 0; k <= 3; k++){
                           y(k, 0) = F_plus(k, 0) - F(k, 0);
                    }
                    static Matrix mid2(4, 1);
                    static Matrix mid3(4, 1);
                    static Matrix mid4(4, 4);
                    static Matrix mid5(4, 4);
                    mid2.addMatrixProduct(0.0, B0, y, 1.0);
                    for (int m = 0; m <= 3; m++){
                           mid3(m, 0) = s(m, 0) - mid2(m, 0);
                    }
                    mid4.addMatrixProduct(0.0, mid3, s_trans, 1.0);
                    mid5.addMatrixProduct(0.0, mid4, B0, 1.0);
                    // don't forget to change sby
                    sby = s(0, 0)*mid2(0, 0) + s(1, 0)*mid2(1, 0) + s(2, 0)*mid2(2, 0) +
s(3, 0)*mid2(3, 0);
                    for (int n = 0; n <= 3; n++){
                           for (int o = 0; o <= 3; o++){
                                  B0(n, o) += mid5(n, o) / sby;
                           }
                    }

             }
      }

      else if (fi <= 0 && gi <= 0 && fj <= 0 && gj > 0){
```

```
            //f=[f1 f2 gj];
            //df = [diff(f, 'Mi'); diff(f, 'Mj'); diff(f, 'd[2]')]'
            static Matrix B(3, 3);
            B(0,0)=1;
            B(0,1)=0;
            B(0,2)=dj_po*((6*EI*(d[1] - 1)*(compense_2 - v(2) + rot_p[2]))/denomina -
(12*EI*pow((d[1] - 1),2)*(compense_1 - v(1) + rot_p[1]))/(pow(denomina,2)/L) +
(6*EI*pow((d[1] - 1),2)*(d[2] - 1)*(compense_2 - v(2) + rot_p[2]))/(pow(denomina,2)/L));
        B(1,0)=0;
            B(1,1)=1;
            B(1,2)=-dj_neg*((12*EI*(compense_2 - v(2) + rot_p[2]))/denomina -
(6*EI*(d[1] - 1)*(compense_1 - v(1) + rot_p[1]))/denomina + (12*EI*(d[1] - 1)*(d[2] -
1)*(compense_2 - v(2) + rot_p[2]))/(pow(denomina,2)/L) - (6*EI*pow((d[1] - 1),2)*(d[2] -
1)*(compense_1 - v(1) + rot_p[1]))/(pow(denomina,2)/L));
        B(2,0)=0;
            B(2,1)=(L*Mj)/(3*EI*pow((d[2] - 1),2));
            B(2,2)=dj_neg*(qq2/pow((d[2] - 1),2) - (qq2*log(1 - d[2]))/pow((d[2] -
1),2)) - (L*pow(Mj,2))/(3*EI*pow((d[2] - 1),3));

        static Matrix B0(3, 3);
            invert3by3Matrix(B, B0);

            static Matrix s(3, 1);
            static Matrix s_trans(1, 3);
            static Matrix y(3, 1);
            static Matrix x_plus(3, 1);
            static Matrix x(3, 1);
            static Matrix F(3, 1);
            static Matrix F_plus(3, 1);
            double sby;

            for (int i = 0; i < 10000; i++){
                    signal=0;
                    x(0, 0) = Mi;
                    x(1, 0) = Mj;
                    x(2, 0) = d[2];
                    //define F
                    denomina = L*(d[1] + d[2] - d[1]* d[2] + 3);
                    F(0, 0) = f1 = Mi - dj_po*(12 * EI*(1 - d[1])*((v(1)-compense_1) -
rot_p[1]) / denomina + 6 * EI*(1 - d[1])*(1 - d[2])*((v(2)-compense_2) - rot_p[2]) /
denomina);
                    F(1, 0) = f2 = Mj - dj_neg*(6 * EI*(1 - d[1])*(1 - d[2])*((v(1)-
compense_1) - rot_p[1]) / denomina + 12 * EI*(1 - d[2])*((v(2)-compense_2) - rot_p[2]) /
denomina);
                    F(2, 0) = gj = pow(Mj, 2) * (L / (3 * EI)) / (2 * (pow(1 - d[2],
2))) - dj_neg*(Gcr2 + qq2*log(1 - d[2]) / (1 - d[2]));
                    static Matrix mid1(3, 1);
                    mid1.addMatrixProduct(0.0, B0, F, 1.0);
                    for (int j = 0; j <= 2; j++){
                            x_plus(j, 0) = x(j, 0) - mid1(j, 0);
                            s(j, 0) = x_plus(j, 0) - x(j, 0);
                            s_trans(0, j) = s(j, 0);
                    }

                    Mi = x_plus(0, 0);
                    Mj = x_plus(1, 0);
                    d[2] = x_plus(2, 0);
```

```
                    if (abs(s(0, 0)) <=accu && abs(s(1, 0)) <=accu && abs(s(2, 0)) 
<=accu){
            if (abs(Mi)>Mu && Mi != 0){
                        Mi=dj_po*Mu*abs(Mi)/Mi;
                        signal=1;
                    }
                    if (abs(Mj)>Mu2 && Mj != 0){
                        Mj=dj_neg*Mu2*abs(Mj)/Mj;
                        signal=1;
                    }
                    if (d[2]>du2){
                        d[2]=du2;
                        signal=1;
                    }
                    if (signal == 1) continue;
                    else break;
                    }

                    F_plus(0, 0) = f1 = Mi - dj_po*(12 * EI*(1 - d[1])*((v(1)-
compense_1) - rot_p[1]) / denomina + 6 * EI*(1 - d[1])*(1 - d[2])*((v(2)-compense_2) - 
rot_p[2]) / denomina);
                    F_plus(1, 0) = f2 = Mj - dj_neg*(6 * EI*(1 - d[1])*(1 - 
d[2])*((v(1)-compense_1) - rot_p[1]) / denomina + 12 * EI*(1 - d[2])*((v(2)-compense_2) - 
rot_p[2]) / denomina);
                    F_plus(2, 0) = gj = pow(Mj, 2) * (L / (3 * EI)) / (2 * (pow(1 - 
d[2], 2))) - dj_neg*(Gcr2 + qq2*log(1 - d[2]) / (1 - d[2]));
                    for (int k = 0; k <= 2; k++){
                        y(k, 0) = F_plus(k, 0) - F(k, 0);
                    }
                    static Matrix mid2(3, 1);
                    static Matrix mid3(3, 1);
                    static Matrix mid4(3, 3);
                    static Matrix mid5(3, 3);
                    mid2.addMatrixProduct(0.0, B0, y, 1.0);
                    for (int m = 0; m <= 2; m++){
                        mid3(m, 0) = s(m, 0) - mid2(m, 0);
                    }
                    mid4.addMatrixProduct(0.0, mid3, s_trans, 1.0);
                    mid5.addMatrixProduct(0.0, mid4, B0, 1.0);
                    sby = s(0, 0)*mid2(0, 0) + s(1, 0)*mid2(1, 0) + s(2, 0)*mid2(2, 0);
                    for (int n = 0; n <= 2; n++){
                        for (int o = 0; o <= 2; o++){
                            B0(n, o) += mid5(n, o) / sby;
                        }
                    }
                }
        }


        else if (fi <= 0 && gi <= 0 && fj > 0 && gj > 0){
                static Matrix B(4, 4);
                //f=[f1 f2 fj gj];
                //df = [diff(f, 'Mi'); diff(f, 'Mj'); diff(f, 'd[2]'); diff(f, 
'rot_p[2]')]'
                B(0,0)=1;
                B(0,1)=0;
```

```
            B(0,2)=dj_po*((6*EI*(d[1] - 1)*(compense_2 - v(2) + rot_p[2]))/denomina -
(12*EI*pow((d[1] - 1),2)*(compense_1 - v(1) + rot_p[1]))/(pow(denomina,2)/L) +
(6*EI*pow((d[1] - 1),2)*(d[2] - 1)*(compense_2 - v(2) + rot_p[2]))/(pow(denomina,2)/L));
            B(0,3)=(6*EI*dj_po*(d[1] - 1)*(d[2] - 1))/denomina;
        B(1,0)=0;
            B(1,1)=1;
            B(1,2)=-dj_neg*((12*EI*(compense_2 - v(2) + rot_p[2]))/denomina -
(6*EI*(d[1] - 1)*(compense_1 - v(1) + rot_p[1]))/denomina + (12*EI*(d[1] - 1)*(d[2] -
1)*(compense_2 - v(2) + rot_p[2]))/(pow(denomina,2)/L) - (6*EI*pow((d[1] - 1),2)*(d[2] -
1)*(compense_1 - v(1) + rot_p[1]))/(pow(denomina,2)/L));
            B(1,3)=-(12*EI*dj_neg*(d[2] - 1))/denomina;
        B(2,0)=0;
            B(2,1)=abs(Mj + c2*dj_neg*rot_p[2]*(d[2] - 1))/(Mj +
c2*dj_neg*rot_p[2]*(d[2] - 1));
            B(2,2)=dj_neg*k02 + c2*dj_neg*rot_p[2]*abs(Mj + c2*dj_neg*rot_p[2]*(d[2] -
1))/(Mj + c2*dj_neg*rot_p[2]*(d[2] - 1));
            B(2,3)=c2*dj_neg*abs(Mj + c2*dj_neg*rot_p[2]*(d[2] - 1))/(Mj +
c2*dj_neg*rot_p[2]*(d[2] - 1))*(d[2] - 1);
        B(3,0)=0;
            B(3,1)=(L*Mj)/(3*EI*pow((d[2] - 1),2));
            B(3,2)=dj_neg*(qq2/pow((d[2] - 1),2) - (qq2*log(1 - d[2]))/pow((d[2] -
1),2)) - (L*pow(Mj,2))/(3*EI*pow((d[2] - 1),3));
            B(3,3)=0;


            static Matrix B0(4, 4);
            invertMatrix(4, B, B0);

            static Matrix s(4, 1);
            static Matrix s_trans(1, 4);
            static Matrix y(4, 1);
            static Matrix x_plus(4, 1);
            static Matrix x(4, 1);
            static Matrix F(4, 1);
            static Matrix F_plus(4, 1);
            double sby;

            for (int i = 0; i < 10000; i++){
                signal=0;
                x(0, 0) = Mi;
                x(1, 0) = Mj;
                x(2, 0) = d[2];
                x(3, 0) = rot_p[2];
                //define F
                denomina = L*(d[1] + d[2] - d[1]* d[2] + 3);
                F(0, 0) = f1 = Mi - dj_po*(12 * EI*(1 - d[1])*((v(1)-compense_1) -
rot_p[1]) / denomina + 6 * EI*(1 - d[1])*(1 - d[2])*((v(2)-compense_2) - rot_p[2]) /
denomina);
                F(1, 0) = f2 = Mj - dj_neg*(6 * EI*(1 - d[1])*(1 - d[2])*((v(1)-
compense_1) - rot_p[1]) / denomina + 12 * EI*(1 - d[2])*((v(2)-compense_2) - rot_p[2]) /
denomina);
                F(2, 0) = fj = abs(Mj - dj_neg*(1 - d[2])*c2*rot_p[2]) - dj_neg*(1 -
d[2])*k02;
                F(3, 0) = gj = pow(Mj, 2) * (L / (3 * EI)) / (2 * (pow(1 - d[2],
2))) - dj_neg*(Gcr2+ qq2*log(1 - d[2]) / (1 - d[2]));
                static Matrix mid1(4, 1);
                mid1.addMatrixProduct(0.0, B0, F, 1.0);
                for (int j = 0; j <= 3; j++){
```

```
                x_plus(j, 0) = x(j, 0) - mid1(j, 0);
                s(j, 0) = x_plus(j, 0) - x(j, 0);
                s_trans(0, j) = s(j, 0);
            }

            Mi = x_plus(0, 0);
            Mj = x_plus(1, 0);
            d[2] = x_plus(2, 0);
            rot_p[2] = x_plus(3, 0);

            //don't forget to change
            if (abs(s(0, 0)) <=accu && abs(s(1, 0)) <=accu && abs(s(2, 0))
<=accu && abs(s(3, 0)) <=accu){
            if (abs(Mi)>Mu && Mi != 0){
                Mi=dj_po*Mu*abs(Mi)/Mi;
                signal=1;
            }
            if (abs(Mj)>Mu2 && Mj != 0){
                Mj=dj_neg*Mu2*abs(Mj)/Mj;
                signal=1;
            }
            if (d[2]>du2){
                d[2]=du2;
                signal=1;
            }
            if (signal == 1) continue;
            else break;
            }

            F_plus(0, 0) = f1 = Mi - dj_po*(12 * EI*(1 - d[1])*((v(1)-
compense_1) - rot_p[1]) / denomina + 6 * EI*(1 - d[1])*(1 - d[2])*((v(2)-compense_2) -
rot_p[2]) / denomina);
            F_plus(1, 0) = f2 = Mj - dj_neg*(6 * EI*(1 - d[1])*(1 -
d[2])*((v(1)-compense_1) - rot_p[1]) / denomina + 12 * EI*(1 - d[2])*((v(2)-compense_2) -
rot_p[2]) / denomina);
            F_plus(2, 0) = fj = abs(Mj - dj_neg*(1 - d[2])*c2*rot_p[2]) -
dj_neg*(1 - d[2])*k02;
            F_plus(3, 0) = gj = pow(Mj, 2) * (L / (3 * EI)) / (2 * (pow(1 -
d[2], 2))) - dj_neg*(Gcr2+ qq2*log(1 - d[2]) / (1 - d[2]));
            for (int k = 0; k <= 3; k++){
                y(k, 0) = F_plus(k, 0) - F(k, 0);
            }
            static Matrix mid2(4, 1);
            static Matrix mid3(4, 1);
            static Matrix mid4(4, 4);
            static Matrix mid5(4, 4);
            mid2.addMatrixProduct(0.0, B0, y, 1.0);
            for (int m = 0; m <= 3; m++){
                mid3(m, 0) = s(m, 0) - mid2(m, 0);
            }
            mid4.addMatrixProduct(0.0, mid3, s_trans, 1.0);
            mid5.addMatrixProduct(0.0, mid4, B0, 1.0);
            // don't forget to change sby
            sby = s(0, 0)*mid2(0, 0) + s(1, 0)*mid2(1, 0) + s(2, 0)*mid2(2, 0) +
s(3, 0)*mid2(3, 0);
            for (int n = 0; n <= 3; n++){
                for (int o = 0; o <= 3; o++){
                    B0(n, o) += mid5(n, o) / sby;
```

```
                        }
                }
            }
        }
        else if (fi <= 0 && gi > 0 && fj <= 0 && gj > 0){
            static Matrix B(4, 4);
            //f = [f1 f2 gi gj];
            //df = [diff(f, 'Mi'); diff(f, 'Mj'); diff(f, 'd[1]'); diff(f, 'd[2]')]'
            B(0,0)=1;
            B(0,1)=0;
            B(0,2)=-dj_po*((12*EI*(compense_1 - v(1) + rot_p[1]))/denomina -
(6*EI*(d[2] - 1)*(compense_2 - v(2) + rot_p[2]))/denomina + (12*EI*(d[1] - 1)*(d[2] -
1)*(compense_1 - v(1) + rot_p[1]))/(pow(denomina,2)/L) - (6*EI*(d[1] - 1)*pow((d[2] -
1),2)*(compense_2 - v(2) + rot_p[2]))/(pow(denomina,2)/L));
            B(0,3)=dj_po*((6*EI*(d[1] - 1)*(compense_2 - v(2) + rot_p[2]))/denomina -
(12*EI*pow((d[1] - 1),2)*(compense_1 - v(1) + rot_p[1]))/(pow(denomina,2)/L) +
(6*EI*pow((d[1] - 1),2)*(d[2] - 1)*(compense_2 - v(2) + rot_p[2]))/(pow(denomina,2)/L));
        B(1,0)=0;
            B(1,1)=1;
            B(1,2)=dj_neg*((6*EI*(d[2] - 1)*(compense_1 - v(1) + rot_p[1]))/denomina -
(12*EI*pow((d[2] - 1),2)*(compense_2 - v(2) + rot_p[2]))/(pow(denomina,2)/L) +
(6*EI*(d[1] - 1)*pow((d[2] - 1),2)*(compense_1 - v(1) + rot_p[1]))/(pow(denomina,2)/L));
            B(1,3)=-dj_neg*((12*EI*(compense_2 - v(2) + rot_p[2]))/denomina -
(6*EI*(d[1] - 1)*(compense_1 - v(1) + rot_p[1]))/denomina + (12*EI*(d[1] - 1)*(d[2] -
1)*(compense_2 - v(2) + rot_p[2]))/(pow(denomina,2)/L) - (6*EI*pow((d[1] - 1),2)*(d[2] -
1)*(compense_1 - v(1) + rot_p[1]))/(pow(denomina,2)/L));
        B(2,0)=(L*Mi)/(3*EI*pow((d[1] - 1),2));
            B(2,1)=0;
            B(2,2)=dj_po*(qq/pow((d[1] - 1),2) - (qq*log(1 - d[1]))/pow((d[1] - 1),2))
- (L*pow(Mi,2))/(3*EI*pow((d[1] - 1),3));
            B(2,3)=0;
        B(3,0)=0;
            B(3,1)=(L*Mj)/(3*EI*pow((d[2] - 1),2));
            B(3,2)=0;
            B(3,3)=dj_neg*(qq2/pow((d[2] - 1),2) - (qq2*log(1 - d[2]))/pow((d[2] -
1),2)) - (L*pow(Mj,2))/(3*EI*pow((d[2] - 1),3));

            static Matrix B0(4, 4);
            invertMatrix(4, B, B0);

            static Matrix s(4, 1);
            static Matrix s_trans(1, 4);
            static Matrix y(4, 1);
            static Matrix x_plus(4, 1);
            static Matrix x(4, 1);
            static Matrix F(4, 1);
            static Matrix F_plus(4, 1);
            double sby;

            for (int i = 0; i < 10000; i++){
                signal=0;
                x(0, 0) = Mi;
                x(1, 0) = Mj;
                x(2, 0) = d[1];
                x(3, 0) = d[2];
                //define F
                denomina = L*(d[1] + d[2] - d[1]* d[2] + 3);
```

```
			F(0, 0) = f1 = Mi - dj_po*(12 * EI*(1 - d[1])*((v(1)-compense_1) - 
rot_p[1]) / denomina + 6 * EI*(1 - d[1])*(1 - d[2])*((v(2)-compense_2) - rot_p[2]) / 
denomina);
			F(1, 0) = f2 = Mj - dj_neg*(6 * EI*(1 - d[1])*(1 - d[2])*((v(1)-
compense_1) - rot_p[1]) / denomina + 12 * EI*(1 - d[2])*((v(2)-compense_2) - rot_p[2]) / 
denomina);
			F(2, 0) = gi = pow(Mi, 2) * (L / (3 * EI)) / (2 * (pow(1 - d[1], 
2))) - dj_po*(Gcr + qq*log(1 - d[1]) / (1 - d[1]));
			F(3, 0) = gj = pow(Mj, 2) * (L / (3 * EI)) / (2 * (pow(1 - d[2], 
2))) - dj_neg*(Gcr2+ qq2*log(1 - d[2]) / (1 - d[2]));
			static Matrix mid1(4, 1);
			mid1.addMatrixProduct(0.0, B0, F, 1.0);
			for (int j = 0; j <= 3; j++){
				x_plus(j, 0) = x(j, 0) - mid1(j, 0);
				s(j, 0) = x_plus(j, 0) - x(j, 0);
				s_trans(0, j) = s(j, 0);
			}

			Mi = x_plus(0, 0);
			Mj = x_plus(1, 0);
			d[1] = x_plus(2, 0);
			d[2] = x_plus(3, 0);

			//don't forget to change
			if (abs(s(0, 0)) <=accu && abs(s(1, 0)) <=accu && abs(s(2, 0)) 
<=accu && abs(s(3, 0)) <=accu){
			if (abs(Mi)>Mu && Mi != 0){
				Mi=dj_po*Mu*abs(Mi)/Mi;
				signal=1;
			}
			if (abs(Mj)>Mu2 && Mj != 0){
				Mj=dj_neg*Mu2*abs(Mj)/Mj;
				signal=1;
			}
			if (d[1]>du){
				d[1]=du;
				signal=1;
			}
			if (d[2]>du2){
				d[2]=du2;
				signal=1;
			}
			if (signal == 1) continue;
			else break;
			}

			F_plus(0, 0) = f1 = Mi - dj_po*(12 * EI*(1 - d[1])*((v(1)-
compense_1) - rot_p[1]) / denomina + 6 * EI*(1 - d[1])*(1 - d[2])*((v(2)-compense_2) - 
rot_p[2]) / denomina);
			F_plus(1, 0) = f2 = Mj - dj_neg*(6 * EI*(1 - d[1])*(1 - 
d[2])*((v(1)-compense_1) - rot_p[1]) / denomina + 12 * EI*(1 - d[2])*((v(2)-compense_2) - 
rot_p[2]) / denomina);
			F_plus(2, 0) = gi = pow(Mi, 2) * (L / (3 * EI)) / (2 * (pow(1 - 
d[1], 2))) - dj_po*(Gcr + qq*log(1 - d[1]) / (1 - d[1]));
			F_plus(3, 0) = gj = pow(Mj, 2) * (L / (3 * EI)) / (2 * (pow(1 - 
d[2], 2))) - dj_neg*(Gcr2+ qq2*log(1 - d[2]) / (1 - d[2]));
			for (int k = 0; k <= 3; k++){
				y(k, 0) = F_plus(k, 0) - F(k, 0);
```

```
                    }
                    static Matrix mid2(4, 1);
                    static Matrix mid3(4, 1);
                    static Matrix mid4(4, 4);
                    static Matrix mid5(4, 4);
                    mid2.addMatrixProduct(0.0, B0, y, 1.0);
                    for (int m = 0; m <= 3; m++){
                           mid3(m, 0) = s(m, 0) - mid2(m, 0);
                    }
                    mid4.addMatrixProduct(0.0, mid3, s_trans, 1.0);
                    mid5.addMatrixProduct(0.0, mid4, B0, 1.0);
                    // don't forget to change sby
                    sby = s(0, 0)*mid2(0, 0) + s(1, 0)*mid2(1, 0) + s(2, 0)*mid2(2, 0) +
s(3, 0)*mid2(3, 0);
                    for (int n = 0; n <= 3; n++){
                           for (int o = 0; o <= 3; o++){
                                  B0(n, o) += mid5(n, o) / sby;
                           }
                    }
             }
      }
      else if (fi > 0 && gi > 0 && fj <= 0 && gj > 0){
             static Matrix B(5, 5);
             //f = [f1 f2 gi gj fi];
             //df = [diff(f, 'Mi'); diff(f, 'Mj'); diff(f, 'd[1]'); diff(f, 'd[2]');
diff(f, 'rot_p[1]')]'

        B(0,0)=1;
             B(0,1)=0;
             B(0,2)=-dj_po*((12*EI*(compense_1 - v(1) + rot_p[1]))/denomina -
(6*EI*(d[2] - 1)*(compense_2 - v(2) + rot_p[2]))/denomina + (12*EI*(d[1] - 1)*(d[2] -
1)*(compense_1 - v(1) + rot_p[1]))/(pow(denomina,2)/L) - (6*EI*(d[1] - 1)*pow((d[2] -
1),2)*(compense_2 - v(2) + rot_p[2]))/(pow(denomina,2)/L));
             B(0,3)=dj_po*((6*EI*(d[1] - 1)*(compense_2 - v(2) + rot_p[2]))/denomina -
(12*EI*pow((d[1] - 1),2)*(compense_1 - v(1) + rot_p[1]))/(pow(denomina,2)/L) +
(6*EI*pow((d[1] - 1),2)*(d[2] - 1)*(compense_2 - v(2) + rot_p[2]))/(pow(denomina,2)/L));
             B(0,4)=-(12*EI*dj_po*(d[1] - 1))/denomina;
        B(1,0)=0;
             B(1,1)=1;
             B(1,2)=dj_neg*((6*EI*(d[2] - 1)*(compense_1 - v(1) + rot_p[1]))/denomina -
(12*EI*pow((d[2] - 1),2)*(compense_2 - v(2) + rot_p[2]))/(pow(denomina,2)/L) +
(6*EI*(d[1] - 1)*pow((d[2] - 1),2)*(compense_1 - v(1) + rot_p[1]))/(pow(denomina,2)/L));
             B(1,3)=-dj_neg*((12*EI*(compense_2 - v(2) + rot_p[2]))/denomina -
(6*EI*(d[1] - 1)*(compense_1 - v(1) + rot_p[1]))/denomina + (12*EI*(d[1] - 1)*(d[2] -
1)*(compense_2 - v(2) + rot_p[2]))/(pow(denomina,2)/L) - (6*EI*pow((d[1] - 1),2)*(d[2] -
1)*(compense_1 - v(1) + rot_p[1]))/(pow(denomina,2)/L));
             B(1,4)=(6*EI*dj_neg*(d[1] - 1)*(d[2] - 1))/denomina;
        B(2,0)=(L*Mi)/(3*EI*pow((d[1] - 1),2));
             B(2,1)=0;
             B(2,2)=dj_po*(qq/pow((d[1] - 1),2) - (qq*log(1 - d[1]))/pow((d[1] - 1),2))
- (L*pow(Mi,2))/(3*EI*pow((d[1] - 1),3));
             B(2,3)=0;
             B(2,4)=0;
        B(3,0)=0;
             B(3,1)=(L*Mj)/(3*EI*pow((d[2] - 1),2));
             B(3,2)=0;
             B(3,3)=dj_neg*(qq2/pow((d[2] - 1),2) - (qq2*log(1 - d[2]))/pow((d[2] -
1),2)) - (L*pow(Mj,2))/(3*EI*pow((d[2] - 1),3));
```

```
              B(3,4)=0;
        B(4,0)=abs(Mi + c*dj_po*rot_p[1]*(d[1] - 1))/(Mi + c*dj_po*rot_p[1]*(d[1] - 1));;
              B(4,1)=0;
              B(4,2)=dj_po*k0 + c*dj_po*rot_p[1]*abs(Mi + c*dj_po*rot_p[1]*(d[1] -
1))/(Mi + c*dj_po*rot_p[1]*(d[1] - 1));
              B(4,3)=0;
              B(4,4)=c*dj_po*abs(Mi + c*dj_po*rot_p[1]*(d[1] - 1))/(Mi +
c*dj_po*rot_p[1]*(d[1] - 1))*(d[1] - 1);


              static Matrix B0(5, 5);
              invertMatrix(5, B, B0);

              static Matrix s(5, 1);
              static Matrix s_trans(1, 5);
              static Matrix y(5, 1);
              static Matrix x_plus(5, 1);
              static Matrix x(5, 1);
              static Matrix F(5, 1);
              static Matrix F_plus(5, 1);
              double sby;

              for (int i = 0; i < 10000; i++){
                     signal=0;
                     x(0, 0) = Mi;
                     x(1, 0) = Mj;
                     x(2, 0) = d[1];
                     x(3, 0) = d[2];
                     x(4, 0) = rot_p[1];
                     //define F
                     denomina = L*(d[1] + d[2] - d[1]* d[2] + 3);
                     F(0, 0) = f1 = Mi - dj_po*(12 * EI*(1 - d[1])*((v(1)-compense_1) -
rot_p[1]) / denomina + 6 * EI*(1 - d[1])*(1 - d[2])*((v(2)-compense_2) - rot_p[2]) /
denomina);
                     F(1, 0) = f2 = Mj - dj_neg*(6 * EI*(1 - d[1])*(1 - d[2])*((v(1)-
compense_1) - rot_p[1]) / denomina + 12 * EI*(1 - d[2])*((v(2)-compense_2) - rot_p[2]) /
denomina);
                     F(2, 0) = gi = pow(Mi, 2) * (L / (3 * EI)) / (2 * (pow(1 - d[1],
2))) - dj_po*(Gcr + qq*log(1 - d[1]) / (1 - d[1]));
                     F(3, 0) = gj = pow(Mj, 2) * (L / (3 * EI)) / (2 * (pow(1 - d[2],
2))) - dj_neg*(Gcr2+ qq2*log(1 - d[2]) / (1 - d[2]));
                     F(4, 0) = fi = abs(Mi - dj_po*(1 - d[1])*c*rot_p[1]) - dj_po* (1 -
d[1])*k0;
                     static Matrix mid1(5, 1);
                     mid1.addMatrixProduct(0.0, B0, F, 1.0);
                     for (int j = 0; j <= 4; j++){
                            x_plus(j, 0) = x(j, 0) - mid1(j, 0);
                            s(j, 0) = x_plus(j, 0) - x(j, 0);
                            s_trans(0, j) = s(j, 0);
                     }

                     Mi = x_plus(0, 0);
                     Mj = x_plus(1, 0);
                     d[1] = x_plus(2, 0);
                     d[2] = x_plus(3, 0);
                     rot_p[1] = x_plus(4, 0);

                     //don't forget to change
```

```
                    if (abs(s(0, 0)) <=accu && abs(s(1, 0)) <=accu && abs(s(2, 0))
<=accu && abs(s(3, 0)) <=accu && abs(s(4, 0)) <=accu){
            if (abs(Mi)>Mu && Mi != 0){
                           Mi=dj_po*Mu*abs(Mi)/Mi;
                           signal=1;
                    }
                    if (abs(Mj)>Mu2 && Mj != 0){
                           Mj=dj_neg*Mu2*abs(Mj)/Mj;
                           signal=1;
                    }
                    if (d[1]>du){
                           d[1]=du;
                           signal=1;
                    }
                    if (d[2]>du2){
                           d[2]=du2;
                           signal=1;
                    }
                    if (signal == 1) continue;
                    else break;
                    }

                    F_plus(0, 0) = f1 = Mi - dj_po*(12 * EI*(1 - d[1])*((v(1)-
compense_1) - rot_p[1]) / denomina + 6 * EI*(1 - d[1])*(1 - d[2])*((v(2)-compense_2) -
rot_p[2]) / denomina);
                    F_plus(1, 0) = f2 = Mj - dj_neg*(6 * EI*(1 - d[1])*(1 -
d[2])*((v(1)-compense_1) - rot_p[1]) / denomina + 12 * EI*(1 - d[2])*((v(2)-compense_2) -
rot_p[2]) / denomina);
                    F_plus(2, 0) = gi = pow(Mi, 2) * (L / (3 * EI)) / (2 * (pow(1 -
d[1], 2))) - dj_po*(Gcr + qq*log(1 - d[1]) / (1 - d[1]));
                    F_plus(3, 0) = gj = pow(Mj, 2) * (L / (3 * EI)) / (2 * (pow(1 -
d[2], 2))) - dj_neg*(Gcr2+ qq2*log(1 - d[2]) / (1 - d[2]));
                    F_plus(4, 0) = fi = abs(Mi - dj_po*(1 - d[1])*c*rot_p[1]) - dj_po*
(1 - d[1])*k0;
                    for (int k = 0; k <= 4; k++){
                           y(k, 0) = F_plus(k, 0) - F(k, 0);
                    }
                    static Matrix mid2(5, 1);
                    static Matrix mid3(5, 1);
                    static Matrix mid4(5, 5);
                    static Matrix mid5(5, 5);
                    mid2.addMatrixProduct(0.0, B0, y, 1.0);
                    for (int m = 0; m <= 4; m++){
                           mid3(m, 0) = s(m, 0) - mid2(m, 0);
                    }
                    mid4.addMatrixProduct(0.0, mid3, s_trans, 1.0);
                    mid5.addMatrixProduct(0.0, mid4, B0, 1.0);
                    // don't forget to change sby
                    sby = s(0, 0)*mid2(0, 0) + s(1, 0)*mid2(1, 0) + s(2, 0)*mid2(2, 0) +
s(3, 0)*mid2(3, 0) + s(4, 0)*mid2(4, 0);
                    for (int n = 0; n <= 4; n++){
                           for (int o = 0; o <= 4; o++){
                                  B0(n, o) += mid5(n, o) / sby;
                           }
                    }
             }
      }
      else if (fi <= 0 && gi > 0 && fj > 0 && gj > 0){
```

```
            static Matrix B(5, 5);
            //f = [f1 f2 gi gj fj];
            //df = [diff(f, 'Mi'); diff(f, 'Mj'); diff(f, 'd[1]'); diff(f, 'd[2]');
diff(f, 'rot_p[2]')]'
            B(0,0)=1;
            B(0,1)=0;
            B(0,2)=-dj_po*((12*EI*(compense_1 - v(1) + rot_p[1]))/denomina -
(6*EI*(d[2] - 1)*(compense_2 - v(2) + rot_p[2]))/denomina + (12*EI*(d[1] - 1)*(d[2] -
1)*(compense_1 - v(1) + rot_p[1]))/(pow(denomina,2)/L) - (6*EI*(d[1] - 1)*pow((d[2] -
1),2)*(compense_2 - v(2) + rot_p[2]))/(pow(denomina,2)/L));
            B(0,3)=dj_po*((6*EI*(d[1] - 1)*(compense_2 - v(2) + rot_p[2]))/denomina -
(12*EI*pow((d[1] - 1),2)*(compense_1 - v(1) + rot_p[1]))/(pow(denomina,2)/L) +
(6*EI*pow((d[1] - 1),2)*(d[2] - 1)*(compense_2 - v(2) + rot_p[2]))/(pow(denomina,2)/L));
            B(0,4)=(6*EI*dj_po*(d[1] - 1)*(d[2] - 1))/denomina;
        B(1,0)=0;
            B(1,1)=1;
            B(1,2)=dj_neg*((6*EI*(d[2] - 1)*(compense_1 - v(1) + rot_p[1]))/denomina -
(12*EI*pow((d[2] - 1),2)*(compense_2 - v(2) + rot_p[2]))/(pow(denomina,2)/L) +
(6*EI*(d[1] - 1)*pow((d[2] - 1),2)*(compense_1 - v(1) + rot_p[1]))/(pow(denomina,2)/L));
            B(1,3)=-dj_neg*((12*EI*(compense_2 - v(2) + rot_p[2]))/denomina -
(6*EI*(d[1] - 1)*(compense_1 - v(1) + rot_p[1]))/denomina + (12*EI*(d[1] - 1)*(d[2] -
1)*(compense_2 - v(2) + rot_p[2]))/(pow(denomina,2)/L) - (6*EI*pow((d[1] - 1),2)*(d[2] -
1)*(compense_1 - v(1) + rot_p[1]))/(pow(denomina,2)/L));
            B(1,4)=-(12*EI*dj_neg*(d[2] - 1))/denomina;
        B(2,0)=(L*Mi)/(3*EI*pow((d[1] - 1),2));
            B(2,1)=0;
            B(2,2)=dj_po*(qq/pow((d[1] - 1),2) - (qq*log(1 - d[1]))/pow((d[1] - 1),2))
- (L*pow(Mi,2))/(3*EI*pow((d[1] - 1),3));
            B(2,3)=0;
            B(2,4)=0;
        B(3,0)=0;
            B(3,1)=(L*Mj)/(3*EI*pow((d[2] - 1),2));
            B(3,2)=0;
            B(3,3)=dj_neg*(qq2/pow((d[2] - 1),2) - (qq2*log(1 - d[2]))/pow((d[2] -
1),2)) - (L*pow(Mj,2))/(3*EI*pow((d[2] - 1),3));
            B(3,4)=0;
        B(4,0)=0;
            B(4,1)=abs(Mj + c2*dj_neg*rot_p[2]*(d[2] - 1))/(Mj +
c2*dj_neg*rot_p[2]*(d[2] - 1));
            B(4,2)=0;
            B(4,3)=dj_neg*k02 + c2*dj_neg*rot_p[2]*abs(Mj + c2*dj_neg*rot_p[2]*(d[2] -
1))/(Mj + c2*dj_neg*rot_p[2]*(d[2] - 1));
            B(4,4)=c2*dj_neg*abs(Mj + c2*dj_neg*rot_p[2]*(d[2] - 1))/(Mj +
c2*dj_neg*rot_p[2]*(d[2] - 1))*(d[2] - 1);

            static Matrix B0(5, 5);
            invertMatrix(5, B, B0);

            static Matrix s(5, 1);
            static Matrix s_trans(1, 5);
            static Matrix y(5, 1);
            static Matrix x_plus(5, 1);
            static Matrix x(5, 1);
            static Matrix F(5, 1);
            static Matrix F_plus(5, 1);
            double sby;

            for (int i = 0; i < 10000; i++){
```

```
                    signal=0;
                    x(0, 0) = Mi;
                    x(1, 0) = Mj;
                    x(2, 0) = d[1];
                    x(3, 0) = d[2];
                    x(4, 0) = rot_p[2];

                    //define F
                    denomina = L*(d[1] + d[2] - d[1]* d[2] + 3);
                    F(0, 0) = f1 = Mi - dj_po*(12 * EI*(1 - d[1])*((v(1)-compense_1) - 
rot_p[1]) / denomina + 6 * EI*(1 - d[1])*(1 - d[2])*((v(2)-compense_2) - rot_p[2]) / 
denomina);
                    F(1, 0) = f2 = Mj - dj_neg*(6 * EI*(1 - d[1])*(1 - d[2])*((v(1)-
compense_1) - rot_p[1]) / denomina + 12 * EI*(1 - d[2])*((v(2)-compense_2) - rot_p[2]) / 
denomina);
                    F(2, 0) = gi = pow(Mi, 2) * (L / (3 * EI)) / (2 * (pow(1 - d[1], 
2))) - dj_po*(Gcr + qq*log(1 - d[1]) / (1 - d[1]));
                    F(3, 0) = gj = pow(Mj, 2) * (L / (3 * EI)) / (2 * (pow(1 - d[2], 
2))) - dj_neg*(Gcr2+ qq2*log(1 - d[2]) / (1 - d[2]));
                    F(4, 0) = fj = abs(Mj - dj_neg*(1 - d[2])*c2*rot_p[2]) - dj_neg*(1 - 
d[2])*k02;
                    static Matrix mid1(5, 1);
                    mid1.addMatrixProduct(0.0, B0, F, 1.0);
                    for (int j = 0; j <= 4; j++){
                           x_plus(j, 0) = x(j, 0) - mid1(j, 0);
                           s(j, 0) = x_plus(j, 0) - x(j, 0);
                           s_trans(0, j) = s(j, 0);
                    }

                    Mi = x_plus(0, 0);
                    Mj = x_plus(1, 0);
                    d[1] = x_plus(2, 0);
                    d[2] = x_plus(3, 0);
                    rot_p[2] = x_plus(4, 0);

                    //don't forget to change
                    if (abs(s(0, 0)) <=accu && abs(s(1, 0)) <=accu && abs(s(2, 0)) 
<=accu && abs(s(3, 0)) <=accu && abs(s(4, 0)) <=accu){
                    if (abs(Mi)>Mu && Mi != 0){
                           Mi=dj_po*Mu*abs(Mi)/Mi;
                           signal=1;
                    }
                    if (abs(Mj)>Mu2 && Mj != 0){
                           Mj=dj_neg*Mu2*abs(Mj)/Mj;
                           signal=1;
                    }
                    if (d[1]>du){
                           d[1]=du;
                           signal=1;
                    }
                    if (d[2]>du2){
                           d[2]=du2;
                           signal=1;
                    }
                    if (signal == 1) continue;
                    else break;
                    }
```

```
                    F_plus(0, 0) = f1 = Mi - dj_po*(12 * EI*(1 - d[1])*((v(1)-
compense_1) - rot_p[1]) / denomina + 6 * EI*(1 - d[1])*(1 - d[2])*((v(2)-compense_2) -
rot_p[2]) / denomina);
                    F_plus(1, 0) = f2 = Mj - dj_neg*(6 * EI*(1 - d[1])*(1 -
d[2])*((v(1)-compense_1) - rot_p[1]) / denomina + 12 * EI*(1 - d[2])*((v(2)-compense_2) -
rot_p[2]) / denomina);
                    F_plus(2, 0) = gi = pow(Mi, 2) * (L / (3 * EI)) / (2 * (pow(1 -
d[1], 2))) - dj_po*(Gcr + qq*log(1 - d[1]) / (1 - d[1]));
                    F_plus(3, 0) = gj = pow(Mj, 2) * (L / (3 * EI)) / (2 * (pow(1 -
d[2], 2))) - dj_neg*(Gcr2+ qq2*log(1 - d[2]) / (1 - d[2]));
                    F_plus(4, 0) = fj = abs(Mj - dj_neg*(1 - d[2])*c2*rot_p[2]) -
dj_neg*(1 - d[2])*k02;
                    for (int k = 0; k <= 4; k++){
                           y(k, 0) = F_plus(k, 0) - F(k, 0);
                    }
                    static Matrix mid2(5, 1);
                    static Matrix mid3(5, 1);
                    static Matrix mid4(5, 5);
                    static Matrix mid5(5, 5);
                    mid2.addMatrixProduct(0.0, B0, y, 1.0);
                    for (int m = 0; m <= 4; m++){
                           mid3(m, 0) = s(m, 0) - mid2(m, 0);
                    }
                    mid4.addMatrixProduct(0.0, mid3, s_trans, 1.0);
                    mid5.addMatrixProduct(0.0, mid4, B0, 1.0);
                    // don't forget to change sby
                    sby = s(0, 0)*mid2(0, 0) + s(1, 0)*mid2(1, 0) + s(2, 0)*mid2(2, 0) +
s(3, 0)*mid2(3, 0) + s(4, 0)*mid2(4, 0);
                    for (int n = 0; n <= 4; n++){
                           for (int o = 0; o <= 4; o++){
                                  B0(n, o) += mid5(n, o) / sby;
                           }
                    }
             }
      }
      else if (fi > 0 && gi > 0 && fj > 0 && gj > 0){
             static Matrix B(6, 6);
             //f = [f1 f2 gi gj fi fj];
             //df = [diff(f, 'Mi'); diff(f, 'Mj'); diff(f, 'd[1]'); diff(f, 'd[2]');
diff(f, 'rot_p[1]'); diff(f, 'rot_p[2]')]'
             B(0,0)=1;
             B(0,1)=0;
             B(0,2)=-dj_po*((12*EI*(compense_1 - v(1) + rot_p[1]))/denomina -
(6*EI*(d[2] - 1)*(compense_2 - v(2) + rot_p[2]))/denomina + (12*EI*(d[1] - 1)*(d[2] -
1)*(compense_1 - v(1) + rot_p[1]))/(pow(denomina,2)/L) - (6*EI*(d[1] - 1)*pow((d[2] -
1),2)*(compense_2 - v(2) + rot_p[2]))/(pow(denomina,2)/L));
             B(0,3)=dj_po*((6*EI*(d[1] - 1)*(compense_2 - v(2) + rot_p[2]))/denomina -
(12*EI*pow((d[1] - 1),2)*(compense_1 - v(1) + rot_p[1]))/(pow(denomina,2)/L) +
(6*EI*pow((d[1] - 1),2)*(d[2] - 1)*(compense_2 - v(2) + rot_p[2]))/(pow(denomina,2)/L));
             B(0,4)=-(12*EI*dj_po*(d[1] - 1))/denomina;
             B(0,5)=(6*EI*dj_po*(d[1] - 1)*(d[2] - 1))/denomina;
        B(1,0)=0;
             B(1,1)=1;
             B(1,2)=dj_neg*((6*EI*(d[2] - 1)*(compense_1 - v(1) + rot_p[1]))/denomina -
(12*EI*pow((d[2] - 1),2)*(compense_2 - v(2) + rot_p[2]))/(pow(denomina,2)/L) +
(6*EI*(d[1] - 1)*pow((d[2] - 1),2)*(compense_1 - v(1) + rot_p[1]))/(pow(denomina,2)/L));
             B(1,3)=-dj_neg*((12*EI*(compense_2 - v(2) + rot_p[2]))/denomina -
(6*EI*(d[1] - 1)*(compense_1 - v(1) + rot_p[1]))/denomina + (12*EI*(d[1] - 1)*(d[2] -
```

```
1)*(compense_2 - v(2) + rot_p[2]))/(pow(denomina,2)/L) - (6*EI*pow((d[1] - 1),2)*(d[2] -
1)*(compense_1 - v(1) + rot_p[1]))/(pow(denomina,2)/L));
                B(1,4)=(6*EI*dj_neg*(d[1] - 1)*(d[2] - 1))/denomina;
                B(1,5)=-(12*EI*dj_neg*(d[2] - 1))/denomina;
        B(2,0)=(L*Mi)/(3*EI*pow((d[1] - 1),2));
                B(2,1)=0;
                B(2,2)=dj_po*(qq/pow((d[1] - 1),2) - (qq*log(1 - d[1]))/pow((d[1] - 1),2))
- (L*pow(Mi,2))/(3*EI*pow((d[1] - 1),3));
                B(2,3)=0;
                B(2,4)=0;
                B(2,5)=0;
        B(3,0)=0;
                B(3,1)=(L*Mj)/(3*EI*pow((d[2] - 1),2));
                B(3,2)=0;
                B(3,3)=dj_neg*(qq2/pow((d[2] - 1),2) - (qq2*log(1 - d[2]))/pow((d[2] -
1),2)) - (L*pow(Mj,2))/(3*EI*pow((d[2] - 1),3));
                B(3,4)=0;
                B(3,5)=0;
        B(4,0)=abs(Mi + c*dj_po*rot_p[1]*(d[1] - 1))/(Mi + c*dj_po*rot_p[1]*(d[1] - 1));
                B(4,1)=0;
                B(4,2)=dj_po*k0 + c*dj_po*rot_p[1]*abs(Mi + c*dj_po*rot_p[1]*(d[1] -
1))/(Mi + c*dj_po*rot_p[1]*(d[1] - 1));
                B(4,3)=0;
                B(4,4)=c*dj_po*abs(Mi + c*dj_po*rot_p[1]*(d[1] - 1))/(Mi +
c*dj_po*rot_p[1]*(d[1] - 1))*(d[1] - 1);
                B(4,5)=0;
        B(5,0)=0;
                B(5,1)=abs(Mj + c2*dj_neg*rot_p[2]*(d[2] - 1))/(Mj +
c2*dj_neg*rot_p[2]*(d[2] - 1));
                B(5,2)=0;
                B(5,3)=dj_neg*k02 + c2*dj_neg*rot_p[2]*abs(Mj + c2*dj_neg*rot_p[2]*(d[2] -
1))/(Mj + c2*dj_neg*rot_p[2]*(d[2] - 1));
                B(5,4)=0;
                B(5,5)=c2*dj_neg*abs(Mj + c2*dj_neg*rot_p[2]*(d[2] - 1))/(Mj +
c2*dj_neg*rot_p[2]*(d[2] - 1))*(d[2] - 1);

                static Matrix B0(6, 6);
                invertMatrix(6, B, B0);

                static Matrix s(6, 1);
                static Matrix s_trans(1, 6);
                static Matrix y(6, 1);
                static Matrix x_plus(6, 1);
                static Matrix x(6, 1);
                static Matrix F(6, 1);
                static Matrix F_plus(6, 1);
                double sby;

                for (int i = 0; i < 10000; i++){
                        signal=0;
                        x(0, 0) = Mi;
                        x(1, 0) = Mj;
                        x(2, 0) = d[1];
                        x(3, 0) = d[2];
                        x(4, 0) = rot_p[1];
                        x(5, 0) = rot_p[2];
                        //define F
                        denomina = L*(d[1] + d[2] - d[1]* d[2] + 3);
```

```
                    F(0, 0) = f1 = Mi - dj_po*(12 * EI*(1 - d[1])*((v(1)-compense_1) -
rot_p[1]) / denomina + 6 * EI*(1 - d[1])*(1 - d[2])*((v(2)-compense_2) - rot_p[2]) /
denomina);
                    F(1, 0) = f2 = Mj - dj_neg*(6 * EI*(1 - d[1])*(1 - d[2])*((v(1)-
compense_1) - rot_p[1]) / denomina + 12 * EI*(1 - d[2])*((v(2)-compense_2) - rot_p[2]) /
denomina);
                    F(2, 0) = gi = pow(Mi, 2) * (L / (3 * EI)) / (2 * (pow(1 - d[1],
2))) - dj_po*(Gcr + qq*log(1 - d[1]) / (1 - d[1]));
                    F(3, 0) = gj = pow(Mj, 2) * (L / (3 * EI)) / (2 * (pow(1 - d[2],
2))) - dj_neg*(Gcr2+ qq2*log(1 - d[2]) / (1 - d[2]));
                    F(4, 0) = fi = abs(Mi - dj_po*(1 - d[1])*c*rot_p[1]) - dj_po* (1 -
d[1])*k0;
                    F(5, 0) = fj = abs(Mj - dj_neg*(1 - d[2])*c2*rot_p[2]) - dj_neg*(1 -
d[2])*k02;
                    static Matrix mid1(6, 1);
                    mid1.addMatrixProduct(0.0, B0, F, 1.0);
                    for (int j = 0; j <= 5; j++){
                           x_plus(j, 0) = x(j, 0) - mid1(j, 0);
                           s(j, 0) = x_plus(j, 0) - x(j, 0);
                           s_trans(0, j) = s(j, 0);
                    }

                    Mi = x_plus(0, 0);
                    Mj = x_plus(1, 0);
                    d[1] = x_plus(2, 0);
                    d[2] = x_plus(3, 0);
                    rot_p[1] = x_plus(4, 0);
                    rot_p[2] = x_plus(5, 0);

                    //don't forget to change
                    if (abs(s(0, 0)) <=accu && abs(s(1, 0)) <=accu && abs(s(2, 0))
<=accu && abs(s(3, 0)) <=accu && abs(s(4, 0)) <=accu && abs(s(5, 0)) <=accu){
                 if (abs(Mi)>Mu && Mi != 0){
                           Mi=dj_po*Mu*abs(Mi)/Mi;
                           signal=1;
                    }
                    if (abs(Mj)>Mu2 && Mj != 0){
                           Mj=dj_neg*Mu2*abs(Mj)/Mj;
                           signal=1;
                    }
                    if (d[1]>du){
                           d[1]=du;
                           signal=1;
                    }
                    if (d[2]>du2){
                           d[2]=du2;
                           signal=1;
                    }
                    if (signal == 1) continue;
                    else break;
                    }

                    F_plus(0, 0) = f1 = Mi - dj_po*(12 * EI*(1 - d[1])*((v(1)-
compense_1) - rot_p[1]) / denomina + 6 * EI*(1 - d[1])*(1 - d[2])*((v(2)-compense_2) -
rot_p[2]) / denomina);
                    F_plus(1, 0) = f2 = Mj - dj_neg*(6 * EI*(1 - d[1])*(1 -
d[2])*((v(1)-compense_1) - rot_p[1]) / denomina + 12 * EI*(1 - d[2])*((v(2)-compense_2) -
rot_p[2]) / denomina);
```

```
                    F_plus(2, 0) = gi = pow(Mi, 2) * (L / (3 * EI)) / (2 * (pow(1 - 
d[1], 2))) - dj_po*(Gcr + qq*log(1 - d[1]) / (1 - d[1]));
                    F_plus(3, 0) = gj = pow(Mj, 2) * (L / (3 * EI)) / (2 * (pow(1 - 
d[2], 2))) - dj_neg*(Gcr2+ qq2*log(1 - d[2]) / (1 - d[2]));
                    F_plus(4, 0) = fi = abs(Mi - dj_po*(1 - d[1])*c*rot_p[1]) - dj_po*
(1 - d[1])*k0;
                    F_plus(5, 0) = fj = abs(Mj - dj_neg*(1 - d[2])*c2*rot_p[2]) - 
dj_neg*(1 - d[2])*k02;
                    for (int k = 0; k <= 5; k++){
                           y(k, 0) = F_plus(k, 0) - F(k, 0);
                    }
                    static Matrix mid2(6, 1);
                    static Matrix mid3(6, 1);
                    static Matrix mid4(6, 6);
                    static Matrix mid5(6, 6);
                    mid2.addMatrixProduct(0.0, B0, y, 1.0);
                    for (int m = 0; m <= 5; m++){
                           mid3(m, 0) = s(m, 0) - mid2(m, 0);
                    }
                    mid4.addMatrixProduct(0.0, mid3, s_trans, 1.0);
                    mid5.addMatrixProduct(0.0, mid4, B0, 1.0);
                    // don't forget to change sby
                    sby = s(0, 0)*mid2(0, 0) + s(1, 0)*mid2(1, 0) + s(2, 0)*mid2(2, 0) + 
s(3, 0)*mid2(3, 0) + s(4, 0)*mid2(4, 0) + s(5, 0)*mid2(5, 0);
                    for (int n = 0; n <= 5; n++){
                           for (int o = 0; o <= 5; o++){
                                  B0(n, o) += mid5(n, o) / sby;
                           }
                    }
             }
      }


      kb(0,0)=E*A/L;
      double strain_s=0;
      double LL=theCoordTransf->getInitialLength();
      double Lu = Lp;
      double Le = LL-2*Lu;
      double Es = 17500;
      double A2=18;
      double h = 6;
   // As1<As2
      if (cv[0] != 0)
             strain_s=v(0)/(Lu/(As1*Es)+Le/(A2*E)+Lu/(As2*Es))/(Es*As1);
      strain_s=strain_s+plastic_strain;
      double strain_y=fu/17500+plastic_strain;
      if (cv[0] == 0 || strain_s == 0){
             q(0)=EAoverL*v(0);

      }
      else{
      if (strain_s<strain_y && strain_s > 0){
      q(0) = Es*strain_s*As1*(1-sqrt(cv[0]/v(0)));
      kb(0,0)=(1-0.5*sqrt(cv[0]/v(0)))/(Lu/(As1*Es)+Le/(A2*E)+Lu/(As2*Es));
      }
      else if (strain_s>strain_y && strain_s<strain_u){
             strain_s=(v(0)-As1*fu*Le/(E*A2)-As1*fu*Lu/(Es*As2))/Lu;
   q(0) = fu*As1*(1-sqrt(cv[0]/v(0)));
```

```
	kb(0,0)=fu*As1*(sqrt(cv[0])*pow(v(0),-3/2)/2);
	   if (strain_s>strain_u)
			q(0)=0;
	}
	}
    //unloading
	double ss=axial/(Lu/(As1*Es)+Le/(A2*E)+Lu/(As2*Es))/(Es*As1);

	if (ss >= strain_y && ss < strain_u && v(0)<axial && v(0)>0 && v(0) > cv[0]){
		ss=(axial-As1*fu*Le/(E*A2)-As1*fu*Lu/(Es*As2))/Lu;
		plastic_strain=ss-strain_y;
		strain_s=(v(0)-
(Lu*plastic_strain))/(Lu/(As1*Es)+Le/(A2*E)+Lu/(As2*Es))/(Es*As1);
		q(0) = Es*strain_s*As1*(1-sqrt(cv[0]/v(0)));
		kb(0,0)=(1-0.5*sqrt(cv[0]/v(0)))/(Lu/(As1*Es)+Le/(A2*E)+Lu/(As2*Es));
	}



	q(1) = Mi;
	q(2) = Mj;

	q(0) += q0[0];
	q(1) += q0[1];
	q(2) += q0[2];


	kb(1, 1) = dj_po*(12 * EI*(1 - d[1]) / denomina);
	kb(1, 2) = dj_po*(6 * EI*(1 - d[1])*(1 - d[2]) / denomina);
	kb(2, 2) = dj_neg*(12 * EI*(1 - d[2]) / denomina);
	kb(2, 1) = dj_neg*(6 * EI*(1 - d[1])*(1 - d[2]) / denomina);

	/*cv[0] = v(0);
	cv[1] = v(1);
	cv[2] = v(2);*/

	return theCoordTransf->getGlobalStiffMatrix(kb, q);
}
const Matrix &
LumpedDamageEle2D2::getInitialStiff(void)
{
	double L = theCoordTransf->getInitialLength();

	double EoverL = E / L;
	double EAoverL = A*EoverL;			// EA/L
	double EIoverL2 = 2.0*I*EoverL;			// 2EI/L
	double EIoverL4 = 2.0*EIoverL2;			// 4EI/L

	kb(0, 0) = EAoverL;
	kb(1, 1) = kb(2, 2) = EIoverL4;
	kb(2, 1) = kb(1, 2) = EIoverL2;

	return theCoordTransf->getInitialGlobalStiffMatrix(kb);
}

const Matrix &
LumpedDamageEle2D2::getMass(void)
{
```

```
	K.Zero();

	if (rho > 0.0)  {
		// get initial element length
		double L = theCoordTransf->getInitialLength();
		if (cMass == 0)  {
			// lumped mass matrix
			double m = 0.5*rho*L;
			K(0, 0) = K(1, 1) = K(3, 3) = K(4, 4) = m;
		}
		else  {
			// consistent mass matrix
			static Matrix ml(6, 6);
			double m = rho*L / 420.0;
			ml(0, 0) = ml(3, 3) = m*140.0;
			ml(0, 3) = ml(3, 0) = m*70.0;

			ml(1, 1) = ml(4, 4) = m*156.0;
			ml(1, 4) = ml(4, 1) = m*54.0;
			ml(2, 2) = ml(5, 5) = m*4.0*L*L;
			ml(2, 5) = ml(5, 2) = -m*3.0*L*L;
			ml(1, 2) = ml(2, 1) = m*22.0*L;
			ml(4, 5) = ml(5, 4) = -ml(1, 2);
			ml(1, 5) = ml(5, 1) = -m*13.0*L;
			ml(2, 4) = ml(4, 2) = -ml(1, 5);

			// transform local mass matrix to global system
			K = theCoordTransf->getGlobalMatrixFromLocal(ml);
		}
	}

	return K;
}

void
LumpedDamageEle2D2::zeroLoad(void)
{
	Q.Zero();

	q0[0] = 0.0;
	q0[1] = 0.0;
	q0[2] = 0.0;

	p0[0] = 0.0;
	p0[1] = 0.0;
	p0[2] = 0.0;

	return;
}

int
LumpedDamageEle2D2::addLoad(ElementalLoad *theLoad, double loadFactor)
{
	int type;
	const Vector &data = theLoad->getData(type, loadFactor);
	double L = theCoordTransf->getInitialLength();

	if (type == LOAD_TAG_Beam2dUniformLoad) {
```

```
            double wt = data(0)*loadFactor;  // Transverse (+ve upward)
            double wa = data(1)*loadFactor;  // Axial (+ve from node I to J)

            double V = 0.5*wt*L;
            double M = V*L / 6.0; // wt*L*L/12
            double P = wa*L;

            // Reactions in basic system
            p0[0] -= P;
            p0[1] -= V;
            p0[2] -= V;

            // Fixed end forces in basic system
            q0[0] -= 0.5*P;
            q0[1] -= M;
            q0[2] += M;
        }

        else if (type == LOAD_TAG_Beam2dPointLoad) {
            double P = data(0)*loadFactor;
            double N = data(1)*loadFactor;
            double aOverL = data(2);

            if (aOverL < 0.0 || aOverL > 1.0)
                return 0;

            double a = aOverL*L;
            double b = L - a;

            // Reactions in basic system
            p0[0] -= N;
            double V1 = P*(1.0 - aOverL);
            double V2 = P*aOverL;
            p0[1] -= V1;
            p0[2] -= V2;

            double L2 = 1.0 / (L*L);
            double a2 = a*a;
            double b2 = b*b;

            // Fixed end forces in basic system
            q0[0] -= N*aOverL;
            double M1 = -a * b2 * P * L2;
            double M2 = a2 * b * P * L2;
            q0[1] += M1;
            q0[2] += M2;
        }
        else {
            opserr << "LumpedDamageEle2D2::addLoad()  -- load type unknown for element 
with tag: " << this->getTag() << endln;
            return -1;
        }

        return 0;
}

int
LumpedDamageEle2D2::addInertiaLoadToUnbalance(const Vector &accel)
```

```
{
	if (rho == 0.0)
		return 0;

	// get R * accel from the nodes
	const Vector &Raccel1 = theNodes[0]->getRV(accel);
	const Vector &Raccel2 = theNodes[1]->getRV(accel);

	if (3 != Raccel1.Size() || 3 != Raccel2.Size()) {
		opserr << "LumpedDamageEle2D2::addInertiaLoadToUnbalance matrix and vector
sizes are incompatable\n";
		return -1;
	}

	// want to add ( - fact * M R * accel ) to unbalance
	if (cMass == 0)  {
		// take advantage of lumped mass matrix
		double L = theCoordTransf->getInitialLength();
		double m = 0.5*rho*L;

		Q(0) -= m * Raccel1(0);
		Q(1) -= m * Raccel1(1);

		Q(3) -= m * Raccel2(0);
		Q(4) -= m * Raccel2(1);
	}
	else  {
		// use matrix vector multip. for consistent mass matrix
		static Vector Raccel(6);
		for (int i = 0; i<3; i++)  {
			Raccel(i) = Raccel1(i);
			Raccel(i + 3) = Raccel2(i);
		}
		Q.addMatrixVector(1.0, this->getMass(), Raccel, -1.0);
	}

	return 0;
}

const Vector &
LumpedDamageEle2D2::getResistingForceIncInertia()
{
	P = this->getResistingForce();

	// subtract external load P = P - Q
	P.addVector(1.0, Q, -1.0);

	// add the damping forces if rayleigh damping
	if (alphaM != 0.0 || betaK != 0.0 || betaK0 != 0.0 || betaKc != 0.0)
		P.addVector(1.0, this->getRayleighDampingForces(), 1.0);

	if (rho == 0.0)
		return P;

	// add inertia forces from element mass
	const Vector &accel1 = theNodes[0]->getTrialAccel();
	const Vector &accel2 = theNodes[1]->getTrialAccel();
```

```
	if (cMass == 0)  {
		// take advantage of lumped mass matrix
		double L = theCoordTransf->getInitialLength();
		double m = 0.5*rho*L;

		P(0) += m * accel1(0);
		P(1) += m * accel1(1);

		P(3) += m * accel2(0);
		P(4) += m * accel2(1);
	}
	else  {
		// use matrix vector multip. for consistent mass matrix
		static Vector accel(6);
		for (int i = 0; i<3; i++)  {
			accel(i) = accel1(i);
			accel(i + 3) = accel2(i);
		}
		P.addMatrixVector(1.0, this->getMass(), accel, 1.0);
	}

	return P;
}


const Vector &
LumpedDamageEle2D2::getResistingForce()
{

	const Vector &v = theCoordTransf->getBasicTrialDisp();

	double L=theCoordTransf->getInitialLength();

	double EoverL = E / L;
	double EAoverL = A*EoverL;			// EA/L
	double EI=E*I;
	double Lover3EI=L/(3*EI);
	double accu=pow((double)10,-10);
	int signal;
	//double rot_p_u=(Mu-(1-du)*k0)/((1-du)*c);

	double cd1=d[1];
	double cd2=d[2];
	//decide loading, unloading or reloading, need to get M first
	double denomina = L*(d[1] + d[2] - d[1]* d[2] + 3);
	if (abs(di_po)< pow((double)1.0,-2)){
		di_po=0;}
	if (abs(di_neg)< pow((double)1.0,-2)){
		di_neg=0;}

    Mi = dj_po*(12 * EI*(1 - d[1])*((v(1)-compense_1) - rot_p[1]) / denomina + 6 * EI*(1
- d[1])*(1 - d[2])*((v(2)-compense_2) - rot_p[2]) / denomina);
    Mj = dj_neg*(6 * EI*(1 - d[1])*(1 - d[2])*((v(1)-compense_1) - rot_p[1]) / denomina +
12 * EI*(1 - d[2])*((v(2)-compense_2) - rot_p[2]) / denomina);
if (Mi*di_po<0){

	if (Mi > 0){
			compense_1=cv[1]-Crot_p_i_po;
```

```
                            d[1] = Cdi_po;
                            rot_p[1] = Crot_p_i_po;
                    }
                    else if (Mi < 0){
                            compense_1=cv[1]-Crot_p_i_neg;
                            d[1] = Cdi_neg;
                            rot_p[1] = Crot_p_i_neg;
                    }
}
if (Mj*di_neg<0){

       if (Mj > 0){
                     compense_2=cv[2]-Crot_p_j_po;
                            d[2] = Cdj_po;
                            rot_p[2] = Crot_p_j_po;
                    }
                    else if (Mj < 0){
                            compense_2=cv[2]-Crot_p_j_neg;
                            d[2] = Cdj_neg;
                            rot_p[2] = Crot_p_j_neg;
                    }
}
       //assign changed values
       denomina = L*(d[1] + d[2] - d[1]* d[2] + 3);
       Mi = dj_po*(12 * EI*(1 - d[1])*((v(1)-compense_1) - rot_p[1]) / denomina + 6 *
EI*(1 - d[1])*(1 - d[2])*((v(2)-compense_2) - rot_p[2]) / denomina);
       Mj = dj_neg*(6 * EI*(1 - d[1])*(1 - d[2])*((v(1)-compense_1) - rot_p[1]) /
denomina + 12 * EI*(1 - d[2])*((v(2)-compense_2) - rot_p[2]) / denomina);


       double f1 = Mi - dj_po*(12 * EI*(1 - d[1])*((v(1)-compense_1) - rot_p[1]) /
denomina + 6 * EI*(1 - d[1])*(1 - d[2])*((v(2)-compense_2) - rot_p[2]) / denomina);
       double f2 = Mj - dj_neg*(6 * EI*(1 - d[1])*(1 - d[2])*((v(1)-compense_1) -
rot_p[1]) / denomina + 12 * EI*(1 - d[2])*((v(2)-compense_2) - rot_p[2]) / denomina);

       double fi = abs(Mi - dj_po*(1 - d[1])*c*rot_p[1]) - dj_po*(1 - d[1])*k0;
       double gi = pow(Mi,2) * (L / (3 * EI)) / (2 * (pow(1-d[1],2))) - dj_po*(Gcr +
qq*log(1 - d[1]) / (1 - d[1]));
       double fj = abs(Mj - dj_neg*(1 - d[2])*c2*rot_p[2]) - dj_neg*(1 - d[2])*k02;
       double gj = pow(Mj,2) * (L / (3 * EI)) / (2 * (pow(1-d[2],2))) - dj_neg*(Gcr2 +
qq2*log(1 - d[2]) / (1 - d[2]));

       if (fi <= 0 && gi > 0 && fj <= 0 && gj <= 0){
              static Matrix B(3, 3);
              B(0,0)=1;
              B(0,1)=0;
              B(0,2)=-dj_po*((12*EI*(compense_1 - v(1) + rot_p[1]))/denomina -
(6*EI*(d[2] - 1)*(compense_2 - v(2) + rot_p[2]))/denomina + (12*EI*(d[1] - 1)*(d[2] -
1)*(compense_1 - v(1) + rot_p[1]))/(pow(denomina,2)/L) - (6*EI*(d[1] - 1)*pow((d[2] -
1),2)*(compense_2 - v(2) + rot_p[2]))/(pow(denomina,2)/L));
              B(1,0)=0;
              B(1,1)=1;
              B(1,2)=dj_neg*((6*EI*(d[2] - 1)*(compense_1 - v(1) + rot_p[1]))/denomina -
(12*EI*pow((d[2] - 1),2)*(compense_2 - v(2) + rot_p[2]))/(pow(denomina,2)/L) +
(6*EI*(d[1] - 1)*pow((d[2] - 1),2)*(compense_1 - v(1) + rot_p[1]))/(pow(denomina,2)/L));
        B(2,0)=(L*Mi)/(3*EI*pow((d[1] - 1),2));
              B(2,1)=0;
```

```
            B(2,2)=dj_po*(qq/pow((d[1] - 1),2) - (qq*log(1 - d[1]))/pow((d[1] - 1),2))
- (L*pow(Mi,2))/(3*EI*pow((d[1] - 1),3));

            static Matrix B0(3, 3);
            invert3by3Matrix(B, B0);

            static Matrix s(3, 1);
            static Matrix s_trans(1, 3);
            static Matrix y(3, 1);
            static Matrix x_plus(3, 1);
            static Matrix x(3, 1);
            static Matrix F(3, 1);
            static Matrix F_plus(3, 1);
            double sby;

            for (int i = 0; i < 10000; i++){
                    signal=0;
                    x(0, 0) = Mi;
                    x(1, 0) = Mj;
                    x(2, 0) = d[1];
                    //define F
                    denomina = L*(d[1] + d[2] - d[1]* d[2] + 3);
                    F(0, 0) = f1 = Mi - dj_po*(12 * EI*(1 - d[1])*((v(1)-compense_1) -
rot_p[1]) / denomina + 6 * EI*(1 - d[1])*(1 - d[2])*((v(2)-compense_2) - rot_p[2]) /
denomina);
                    F(1, 0) = f2 = Mj - dj_neg*(6 * EI*(1 - d[1])*(1 - d[2])*((v(1)-
compense_1) - rot_p[1]) / denomina + 12 * EI*(1 - d[2])*((v(2)-compense_2) - rot_p[2]) /
denomina);
                    F(2, 0) = gi = pow(Mi, 2) * (L / (3 * EI)) / (2 * (pow(1 - d[1],
2))) - dj_po*(Gcr + qq*log(1 - d[1]) / (1 - d[1]));
                    static Matrix mid1(3, 1);
                    mid1.addMatrixProduct(0.0, B0, F, 1.0);
                    for (int j = 0; j <= 2; j++){
                            x_plus(j, 0) = x(j, 0) - mid1(j, 0);
                            s(j, 0) = x_plus(j, 0) - x(j, 0);
                            s_trans(0, j) = s(j, 0);
                    }

                    Mi = x_plus(0, 0);
                    Mj = x_plus(1, 0);
                    d[1] = x_plus(2, 0);

                    if (abs(s(0, 0)) <= accu && abs(s(1, 0)) <=accu && abs(s(2, 0)) <=
accu){
                    if (abs(Mi)>Mu && Mi != 0){
                            Mi=dj_po*Mu*abs(Mi)/Mi;
                            signal=1;
                    }
                    if (abs(Mj)>Mu2 && Mj != 0){
                            Mj=dj_neg*Mu2*abs(Mj)/Mj;
                            signal=1;
                    }
                    if (d[1]>du){
                            d[1]=du;
                            signal=1;
                    }
                    if (signal == 1) continue;
                    else break;
```

```
                    }

                    F_plus(0, 0) = f1 = Mi - dj_po*(12 * EI*(1 - d[1])*((v(1)-
compense_1) - rot_p[1]) / denomina + 6 * EI*(1 - d[1])*(1 - d[2])*((v(2)-compense_2) -
rot_p[2]) / denomina);
                    F_plus(1, 0) = f2 = Mj - dj_neg*(6 * EI*(1 - d[1])*(1 -
d[2])*((v(1)-compense_1) - rot_p[1]) / denomina + 12 * EI*(1 - d[2])*((v(2)-compense_2) -
rot_p[2]) / denomina);
                    F_plus(2, 0) = gi = pow(Mi, 2) * (L / (3 * EI)) / (2 * (pow(1 -
d[1], 2))) - dj_po*(Gcr + qq*log(1 - d[1]) / (1 - d[1]));
                    for (int k = 0; k <= 2; k++){
                           y(k, 0) = F_plus(k, 0) - F(k, 0);
                    }
                    static Matrix mid2(3, 1);
                    static Matrix mid3(3, 1);
                    static Matrix mid4(3, 3);
                    static Matrix mid5(3, 3);
                    mid2.addMatrixProduct(0.0, B0, y, 1.0);
                    for (int m = 0; m <= 2; m++){
                           mid3(m, 0) = s(m, 0) - mid2(m, 0);
                    }
                    mid4.addMatrixProduct(0.0, mid3, s_trans, 1.0);
                    mid5.addMatrixProduct(0.0, mid4, B0, 1.0);
                    sby = s(0, 0)*mid2(0, 0) + s(1, 0)*mid2(1, 0) + s(2, 0)*mid2(2, 0);
                    for (int n = 0; n <= 2; n++){
                           for (int o = 0; o <= 2; o++){
                                  B0(n, o) += mid5(n, o) / sby;
                           }
                    }
             }
      }
      else if (fi > 0 && gi > 0 && fj <= 0 && gj <= 0){
             static Matrix B(4, 4);
             //f=[f1 f2 fi gi];
             //df = [diff(f, 'Mi'); diff(f, 'Mj'); diff(f, 'd[1]'); diff(f,
'rot_p[1]')]'
             B(0,0)=1;
             B(0,1)=0;
             B(0,2)=-dj_po*((12*EI*(compense_1 - v(1) + rot_p[1]))/denomina -
(6*EI*(d[2] - 1)*(compense_2 - v(2) + rot_p[2]))/denomina + (12*EI*(d[1] - 1)*(d[2] -
1)*(compense_1 - v(1) + rot_p[1]))/(pow(denomina,2)/L) - (6*EI*(d[1] - 1)*pow((d[2] -
1),2)*(compense_2 - v(2) + rot_p[2]))/(pow(denomina,2)/L));
             B(0,3)=-(12*EI*dj_po*(d[1] - 1))/denomina;
        B(1,0)=0;
             B(1,1)=1;
             B(1,2)=dj_neg*((6*EI*(d[2] - 1)*(compense_1 - v(1) + rot_p[1]))/denomina -
(12*EI*pow((d[2] - 1),2)*(compense_2 - v(2) + rot_p[2]))/(pow(denomina,2)/L) +
(6*EI*(d[1] - 1)*pow((d[2] - 1),2)*(compense_1 - v(1) + rot_p[1]))/(pow(denomina,2)/L));
             B(1,3)=(6*EI*dj_neg*(d[1] - 1)*(d[2] - 1))/denomina;
        B(2,0)=abs(Mi + c*dj_po*rot_p[1]*(d[1] - 1))/(Mi + c*dj_po*rot_p[1]*(d[1] - 1));
             B(2,1)=0;
             B(2,2)=dj_po*k0 + c*dj_po*rot_p[1]*abs(Mi + c*dj_po*rot_p[1]*(d[1] -
1))/(Mi + c*dj_po*rot_p[1]*(d[1] - 1));
             B(2,3)=c*dj_po*abs(Mi + c*dj_po*rot_p[1]*(d[1] - 1))/(Mi +
c*dj_po*rot_p[1]*(d[1] - 1))*(d[1] - 1);
        B(3,0)=(L*Mi)/(3*EI*pow((d[1] - 1),2));
        B(3,1)=0;
```

```
            B(3,2)=dj_po*(qq/pow((d[1] - 1),2) - (qq*log(1 - d[1]))/pow((d[1] - 1),2))
- (L*pow(Mi,2))/(3*EI*pow((d[1] - 1),3));
            B(3,3)=0;


            static Matrix B0(4, 4);
            invertMatrix(4, B, B0);

            static Matrix s(4, 1);
            static Matrix s_trans(1, 4);
            static Matrix y(4, 1);
            static Matrix x_plus(4, 1);
            static Matrix x(4, 1);
            static Matrix F(4, 1);
            static Matrix F_plus(4, 1);
            double sby;

            for (int i = 0; i < 10000; i++){
                   signal=0;
                   x(0, 0) = Mi;
                   x(1, 0) = Mj;
                   x(2, 0) = d[1];
                   x(3, 0) = rot_p[1];
                   //define F
                   denomina = L*(d[1] + d[2] - d[1]* d[2] + 3);
                   F(0, 0) = f1 = Mi - dj_po*(12 * EI*(1 - d[1])*((v(1)-compense_1) -
rot_p[1]) / denomina + 6 * EI*(1 - d[1])*(1 - d[2])*((v(2)-compense_2) - rot_p[2]) /
denomina);
                   F(1, 0) = f2 = Mj - dj_neg*(6 * EI*(1 - d[1])*(1 - d[2])*((v(1)-
compense_1) - rot_p[1]) / denomina + 12 * EI*(1 - d[2])*((v(2)-compense_2) - rot_p[2]) /
denomina);
                   F(2, 0) = fi = abs(Mi - dj_po*(1 - d[1])*c*rot_p[1]) - dj_po* (1 -
d[1])*k0;
                   F(3, 0) = gi = pow(Mi, 2) * (L / (3 * EI)) / (2 * (pow((1 - d[1]),
2))) - dj_po*(Gcr + qq*log(1 - d[1]) / (1 - d[1]));
                   static Matrix mid1(4, 1);
                   mid1.addMatrixProduct(0.0, B0, F, 1.0);
                   for (int j = 0; j <= 3; j++){
                          x_plus(j, 0) = x(j, 0) - mid1(j, 0);
                          s(j, 0) = x_plus(j, 0) - x(j, 0);
                          s_trans(0, j) = s(j, 0);
                   }

                   Mi = x_plus(0, 0);
                   Mj = x_plus(1, 0);
                   d[1] = x_plus(2, 0);
                   rot_p[1] = x_plus(3, 0);

                   //don't forget to change
                   if (abs(s(0, 0)) <=accu && abs(s(1, 0)) <=accu && abs(s(2, 0))
<=accu && abs(s(3, 0)) <=accu){
                   if (abs(Mi)>Mu && Mi != 0){
                          Mi=dj_po*Mu*abs(Mi)/Mi;
                          signal=1;
                   }
                   if (abs(Mj)>Mu2 && Mj != 0){
                          Mj=dj_neg*Mu2*abs(Mj)/Mj;
                          signal=1;
```

```
			}
			if (d[1]>du){
				d[1]=du;
				signal=1;
			}
			if (signal == 1) continue;
			else break;
			}

			F_plus(0, 0) = f1 = Mi - dj_po*(12 * EI*(1 - d[1])*((v(1)-
compense_1) - rot_p[1]) / denomina + 6 * EI*(1 - d[1])*(1 - d[2])*((v(2)-compense_2) -
rot_p[2]) / denomina);
			F_plus(1, 0) = f2 = Mj - dj_neg*(6 * EI*(1 - d[1])*(1 -
d[2])*((v(1)-compense_1) - rot_p[1]) / denomina + 12 * EI*(1 - d[2])*((v(2)-compense_2) -
rot_p[2]) / denomina);
			F_plus(2, 0) = fi = abs(Mi - dj_po*(1 - d[1])*c*rot_p[1]) - dj_po*
(1 - d[1])*k0;
			F_plus(3, 0) = gi = pow(Mi, 2) * (L / (3 * EI)) / (2 * (pow((1 -
d[1]), 2))) - dj_po*(Gcr + qq*log(1 - d[1]) / (1 - d[1]));
			for (int k = 0; k <= 3; k++){
				y(k, 0) = F_plus(k, 0) - F(k, 0);
			}
			static Matrix mid2(4, 1);
			static Matrix mid3(4, 1);
			static Matrix mid4(4, 4);
			static Matrix mid5(4, 4);
			mid2.addMatrixProduct(0.0, B0, y, 1.0);
			for (int m = 0; m <= 3; m++){
				mid3(m, 0) = s(m, 0) - mid2(m, 0);
			}
			mid4.addMatrixProduct(0.0, mid3, s_trans, 1.0);
			mid5.addMatrixProduct(0.0, mid4, B0, 1.0);
			// don't forget to change sby
			sby = s(0, 0)*mid2(0, 0) + s(1, 0)*mid2(1, 0) + s(2, 0)*mid2(2, 0) +
s(3, 0)*mid2(3, 0);
			for (int n = 0; n <= 3; n++){
				for (int o = 0; o <= 3; o++){
					B0(n, o) += mid5(n, o) / sby;
				}
			}

		}
	}

	else if (fi <= 0 && gi <= 0 && fj <= 0 && gj > 0){
		//f=[f1 f2 gj];
		//df = [diff(f, 'Mi'); diff(f, 'Mj'); diff(f, 'd[2]')]'
		static Matrix B(3, 3);
		B(0,0)=1;
		B(0,1)=0;
		B(0,2)=dj_po*((6*EI*(d[1] - 1)*(compense_2 - v(2) + rot_p[2]))/denomina -
(12*EI*pow((d[1] - 1),2)*(compense_1 - v(1) + rot_p[1]))/(pow(denomina,2)/L) +
(6*EI*pow((d[1] - 1),2)*(d[2] - 1)*(compense_2 - v(2) + rot_p[2]))/(pow(denomina,2)/L));
	 B(1,0)=0;
		B(1,1)=1;
		B(1,2)=-dj_neg*((12*EI*(compense_2 - v(2) + rot_p[2]))/denomina -
(6*EI*(d[1] - 1)*(compense_1 - v(1) + rot_p[1]))/denomina + (12*EI*(d[1] - 1)*(d[2] -
```

```
1)*(compense_2 - v(2) + rot_p[2]))/(pow(denomina,2)/L) - (6*EI*pow((d[1] - 1),2)*(d[2] -
1)*(compense_1 - v(1) + rot_p[1]))/(pow(denomina,2)/L));
        B(2,0)=0;
              B(2,1)=(L*Mj)/(3*EI*pow((d[2] - 1),2));
              B(2,2)=dj_neg*(qq2/pow((d[2] - 1),2) - (qq2*log(1 - d[2]))/pow((d[2] -
1),2)) - (L*pow(Mj,2))/(3*EI*pow((d[2] - 1),3));

        static Matrix B0(3, 3);
              invert3by3Matrix(B, B0);

              static Matrix s(3, 1);
              static Matrix s_trans(1, 3);
              static Matrix y(3, 1);
              static Matrix x_plus(3, 1);
              static Matrix x(3, 1);
              static Matrix F(3, 1);
              static Matrix F_plus(3, 1);
              double sby;

              for (int i = 0; i < 10000; i++){
                     signal=0;
                     x(0, 0) = Mi;
                     x(1, 0) = Mj;
                     x(2, 0) = d[2];
                     //define F
                     denomina = L*(d[1] + d[2] - d[1]* d[2] + 3);
                     F(0, 0) = f1 = Mi - dj_po*(12 * EI*(1 - d[1])*((v(1)-compense_1) -
rot_p[1]) / denomina + 6 * EI*(1 - d[1])*(1 - d[2])*((v(2)-compense_2) - rot_p[2]) /
denomina);
                     F(1, 0) = f2 = Mj - dj_neg*(6 * EI*(1 - d[1])*(1 - d[2])*((v(1)-
compense_1) - rot_p[1]) / denomina + 12 * EI*(1 - d[2])*((v(2)-compense_2) - rot_p[2]) /
denomina);
                     F(2, 0) = gj = pow(Mj, 2) * (L / (3 * EI)) / (2 * (pow(1 - d[2],
2))) - dj_neg*(Gcr2 + qq2*log(1 - d[2]) / (1 - d[2]));
                     static Matrix mid1(3, 1);
                     mid1.addMatrixProduct(0.0, B0, F, 1.0);
                     for (int j = 0; j <= 2; j++){
                            x_plus(j, 0) = x(j, 0) - mid1(j, 0);
                            s(j, 0) = x_plus(j, 0) - x(j, 0);
                            s_trans(0, j) = s(j, 0);
                     }

                     Mi = x_plus(0, 0);
                     Mj = x_plus(1, 0);
                     d[2] = x_plus(2, 0);

                     if (abs(s(0, 0)) <=accu && abs(s(1, 0)) <=accu && abs(s(2, 0))
<=accu){
            if (abs(Mi)>Mu && Mi != 0){
                            Mi=dj_po*Mu*abs(Mi)/Mi;
                            signal=1;
                     }
                     if (abs(Mj)>Mu2 && Mj != 0){
                            Mj=dj_neg*Mu2*abs(Mj)/Mj;
                            signal=1;
                     }
                     if (d[2]>du2){
                            d[2]=du2;
```

```
                        signal=1;
                    }
                    if (signal == 1) continue;
                    else break;
                    }

                    F_plus(0, 0) = f1 = Mi - dj_po*(12 * EI*(1 - d[1])*((v(1)-
compense_1) - rot_p[1]) / denomina + 6 * EI*(1 - d[1])*(1 - d[2])*((v(2)-compense_2) -
rot_p[2]) / denomina);
                    F_plus(1, 0) = f2 = Mj - dj_neg*(6 * EI*(1 - d[1])*(1 -
d[2])*((v(1)-compense_1) - rot_p[1]) / denomina + 12 * EI*(1 - d[2])*((v(2)-compense_2) -
rot_p[2]) / denomina);
                    F_plus(2, 0) = gj = pow(Mj, 2) * (L / (3 * EI)) / (2 * (pow(1 -
d[2], 2))) - dj_neg*(Gcr2 + qq2*log(1 - d[2]) / (1 - d[2]));
                    for (int k = 0; k <= 2; k++){
                        y(k, 0) = F_plus(k, 0) - F(k, 0);
                    }
                    static Matrix mid2(3, 1);
                    static Matrix mid3(3, 1);
                    static Matrix mid4(3, 3);
                    static Matrix mid5(3, 3);
                    mid2.addMatrixProduct(0.0, B0, y, 1.0);
                    for (int m = 0; m <= 2; m++){
                        mid3(m, 0) = s(m, 0) - mid2(m, 0);
                    }
                    mid4.addMatrixProduct(0.0, mid3, s_trans, 1.0);
                    mid5.addMatrixProduct(0.0, mid4, B0, 1.0);
                    sby = s(0, 0)*mid2(0, 0) + s(1, 0)*mid2(1, 0) + s(2, 0)*mid2(2, 0);
                    for (int n = 0; n <= 2; n++){
                        for (int o = 0; o <= 2; o++){
                            B0(n, o) += mid5(n, o) / sby;
                        }
                    }
                }
        }


        else if (fi <= 0 && gi <= 0 && fj > 0 && gj > 0){
                static Matrix B(4, 4);
                //f=[f1 f2 fj gj];
                //df = [diff(f, 'Mi'); diff(f, 'Mj'); diff(f, 'd[2]'); diff(f,
'rot_p[2]')]'
                B(0,0)=1;
                B(0,1)=0;
                B(0,2)=dj_po*((6*EI*(d[1] - 1)*(compense_2 - v(2) + rot_p[2]))/denomina -
(12*EI*pow((d[1] - 1),2)*(compense_1 - v(1) + rot_p[1]))/(pow(denomina,2)/L) +
(6*EI*pow((d[1] - 1),2)*(d[2] - 1)*(compense_2 - v(2) + rot_p[2]))/(pow(denomina,2)/L));
                B(0,3)=(6*EI*dj_po*(d[1] - 1)*(d[2] - 1))/denomina;
            B(1,0)=0;
                B(1,1)=1;
                B(1,2)=-dj_neg*((12*EI*(compense_2 - v(2) + rot_p[2]))/denomina -
(6*EI*(d[1] - 1)*(compense_1 - v(1) + rot_p[1]))/denomina + (12*EI*(d[1] - 1)*(d[2] -
1)*(compense_2 - v(2) + rot_p[2]))/(pow(denomina,2)/L) - (6*EI*pow((d[1] - 1),2)*(d[2] -
1)*(compense_1 - v(1) + rot_p[1]))/(pow(denomina,2)/L));
                B(1,3)=-(12*EI*dj_neg*(d[2] - 1))/denomina;
            B(2,0)=0;
                B(2,1)=abs(Mj + c2*dj_neg*rot_p[2]*(d[2] - 1))/(Mj +
c2*dj_neg*rot_p[2]*(d[2] - 1));
```

```
		B(2,2)=dj_neg*k02 + c2*dj_neg*rot_p[2]*abs(Mj + c2*dj_neg*rot_p[2]*(d[2] - 
1))/(Mj + c2*dj_neg*rot_p[2]*(d[2] - 1));
		B(2,3)=c2*dj_neg*abs(Mj + c2*dj_neg*rot_p[2]*(d[2] - 1))/(Mj + 
c2*dj_neg*rot_p[2]*(d[2] - 1))*(d[2] - 1);
	B(3,0)=0;
		B(3,1)=(L*Mj)/(3*EI*pow((d[2] - 1),2));
		B(3,2)=dj_neg*(qq2/pow((d[2] - 1),2) - (qq2*log(1 - d[2]))/pow((d[2] - 
1),2)) - (L*pow(Mj,2))/(3*EI*pow((d[2] - 1),3));
		B(3,3)=0;


		static Matrix B0(4, 4);
		invertMatrix(4, B, B0);

		static Matrix s(4, 1);
		static Matrix s_trans(1, 4);
		static Matrix y(4, 1);
		static Matrix x_plus(4, 1);
		static Matrix x(4, 1);
		static Matrix F(4, 1);
		static Matrix F_plus(4, 1);
		double sby;

		for (int i = 0; i < 10000; i++){
			signal=0;
			x(0, 0) = Mi;
			x(1, 0) = Mj;
			x(2, 0) = d[2];
			x(3, 0) = rot_p[2];
			//define F
			denomina = L*(d[1] + d[2] - d[1]* d[2] + 3);
			F(0, 0) = f1 = Mi - dj_po*(12 * EI*(1 - d[1])*((v(1)-compense_1) - 
rot_p[1]) / denomina + 6 * EI*(1 - d[1])*(1 - d[2])*((v(2)-compense_2) - rot_p[2]) / 
denomina);
			F(1, 0) = f2 = Mj - dj_neg*(6 * EI*(1 - d[1])*(1 - d[2])*((v(1)-
compense_1) - rot_p[1]) / denomina + 12 * EI*(1 - d[2])*((v(2)-compense_2) - rot_p[2]) / 
denomina);
			F(2, 0) = fj = abs(Mj - dj_neg*(1 - d[2])*c2*rot_p[2]) - dj_neg*(1 - 
d[2])*k02;
			F(3, 0) = gj = pow(Mj, 2) * (L / (3 * EI)) / (2 * (pow(1 - d[2], 
2))) - dj_neg*(Gcr2+ qq2*log(1 - d[2]) / (1 - d[2]));
			static Matrix mid1(4, 1);
			mid1.addMatrixProduct(0.0, B0, F, 1.0);
			for (int j = 0; j <= 3; j++){
				x_plus(j, 0) = x(j, 0) - mid1(j, 0);
				s(j, 0) = x_plus(j, 0) - x(j, 0);
				s_trans(0, j) = s(j, 0);
			}

			Mi = x_plus(0, 0);
			Mj = x_plus(1, 0);
			d[2] = x_plus(2, 0);
			rot_p[2] = x_plus(3, 0);

			//don't forget to change
			if (abs(s(0, 0)) <=accu && abs(s(1, 0)) <=accu && abs(s(2, 0)) 
<=accu && abs(s(3, 0)) <=accu){
			if (abs(Mi)>Mu && Mi != 0){
```

```
                    Mi=dj_po*Mu*abs(Mi)/Mi;
                    signal=1;
                }
                if (abs(Mj)>Mu2 && Mj != 0){
                    Mj=dj_neg*Mu2*abs(Mj)/Mj;
                    signal=1;
                }
                if (d[2]>du2){
                    d[2]=du2;
                    signal=1;
                }
                if (signal == 1) continue;
                else break;
                }

                F_plus(0, 0) = f1 = Mi - dj_po*(12 * EI*(1 - d[1])*((v(1)-
compense_1) - rot_p[1]) / denomina + 6 * EI*(1 - d[1])*(1 - d[2])*((v(2)-compense_2) -
rot_p[2]) / denomina);
                F_plus(1, 0) = f2 = Mj - dj_neg*(6 * EI*(1 - d[1])*(1 -
d[2])*((v(1)-compense_1) - rot_p[1]) / denomina + 12 * EI*(1 - d[2])*((v(2)-compense_2) -
rot_p[2]) / denomina);
                F_plus(2, 0) = fj = abs(Mj - dj_neg*(1 - d[2])*c2*rot_p[2]) -
dj_neg*(1 - d[2])*k02;
                F_plus(3, 0) = gj = pow(Mj, 2) * (L / (3 * EI)) / (2 * (pow(1 -
d[2], 2))) - dj_neg*(Gcr2+ qq2*log(1 - d[2]) / (1 - d[2]));
                for (int k = 0; k <= 3; k++){
                    y(k, 0) = F_plus(k, 0) - F(k, 0);
                }
                static Matrix mid2(4, 1);
                static Matrix mid3(4, 1);
                static Matrix mid4(4, 4);
                static Matrix mid5(4, 4);
                mid2.addMatrixProduct(0.0, B0, y, 1.0);
                for (int m = 0; m <= 3; m++){
                    mid3(m, 0) = s(m, 0) - mid2(m, 0);
                }
                mid4.addMatrixProduct(0.0, mid3, s_trans, 1.0);
                mid5.addMatrixProduct(0.0, mid4, B0, 1.0);
                // don't forget to change sby
                sby = s(0, 0)*mid2(0, 0) + s(1, 0)*mid2(1, 0) + s(2, 0)*mid2(2, 0) +
s(3, 0)*mid2(3, 0);
                for (int n = 0; n <= 3; n++){
                    for (int o = 0; o <= 3; o++){
                        B0(n, o) += mid5(n, o) / sby;
                    }
                }
            }
        }
        else if (fi <= 0 && gi > 0 && fj <= 0 && gj > 0){
            static Matrix B(4, 4);
            //f = [f1 f2 gi gj];
            //df = [diff(f, 'Mi'); diff(f, 'Mj'); diff(f, 'd[1]'); diff(f, 'd[2]')]'
            B(0,0)=1;
            B(0,1)=0;
            B(0,2)=-dj_po*((12*EI*(compense_1 - v(1) + rot_p[1]))/denomina -
(6*EI*(d[2] - 1)*(compense_2 - v(2) + rot_p[2]))/denomina + (12*EI*(d[1] - 1)*(d[2] -
1)*(compense_1 - v(1) + rot_p[1]))/(pow(denomina,2)/L) - (6*EI*(d[1] - 1)*pow((d[2] -
1),2)*(compense_2 - v(2) + rot_p[2]))/(pow(denomina,2)/L));
```

```
            B(0,3)=dj_po*((6*EI*(d[1] - 1)*(compense_2 - v(2) + rot_p[2]))/denomina -
(12*EI*pow((d[1] - 1),2)*(compense_1 - v(1) + rot_p[1]))/(pow(denomina,2)/L) +
(6*EI*pow((d[1] - 1),2)*(d[2] - 1)*(compense_2 - v(2) + rot_p[2]))/(pow(denomina,2)/L));
        B(1,0)=0;
            B(1,1)=1;
            B(1,2)=dj_neg*((6*EI*(d[2] - 1)*(compense_1 - v(1) + rot_p[1]))/denomina -
(12*EI*pow((d[2] - 1),2)*(compense_2 - v(2) + rot_p[2]))/(pow(denomina,2)/L) +
(6*EI*(d[1] - 1)*pow((d[2] - 1),2)*(compense_1 - v(1) + rot_p[1]))/(pow(denomina,2)/L));
            B(1,3)=-dj_neg*((12*EI*(compense_2 - v(2) + rot_p[2]))/denomina -
(6*EI*(d[1] - 1)*(compense_1 - v(1) + rot_p[1]))/denomina + (12*EI*(d[1] - 1)*(d[2] -
1)*(compense_2 - v(2) + rot_p[2]))/(pow(denomina,2)/L) - (6*EI*pow((d[1] - 1),2)*(d[2] -
1)*(compense_1 - v(1) + rot_p[1]))/(pow(denomina,2)/L));
        B(2,0)=(L*Mi)/(3*EI*pow((d[1] - 1),2));
            B(2,1)=0;
            B(2,2)=dj_po*(qq/pow((d[1] - 1),2) - (qq*log(1 - d[1]))/pow((d[1] - 1),2))
- (L*pow(Mi,2))/(3*EI*pow((d[1] - 1),3));
            B(2,3)=0;
        B(3,0)=0;
            B(3,1)=(L*Mj)/(3*EI*pow((d[2] - 1),2));
            B(3,2)=0;
            B(3,3)=dj_neg*(qq2/pow((d[2] - 1),2) - (qq2*log(1 - d[2]))/pow((d[2] -
1),2)) - (L*pow(Mj,2))/(3*EI*pow((d[2] - 1),3));

            static Matrix B0(4, 4);
            invertMatrix(4, B, B0);

            static Matrix s(4, 1);
            static Matrix s_trans(1, 4);
            static Matrix y(4, 1);
            static Matrix x_plus(4, 1);
            static Matrix x(4, 1);
            static Matrix F(4, 1);
            static Matrix F_plus(4, 1);
            double sby;

            for (int i = 0; i < 10000; i++){
                signal=0;
                x(0, 0) = Mi;
                x(1, 0) = Mj;
                x(2, 0) = d[1];
                x(3, 0) = d[2];
                //define F
                denomina = L*(d[1] + d[2] - d[1]* d[2] + 3);
                F(0, 0) = f1 = Mi - dj_po*(12 * EI*(1 - d[1])*((v(1)-compense_1) -
rot_p[1]) / denomina + 6 * EI*(1 - d[1])*(1 - d[2])*((v(2)-compense_2) - rot_p[2]) /
denomina);
                F(1, 0) = f2 = Mj - dj_neg*(6 * EI*(1 - d[1])*(1 - d[2])*((v(1)-
compense_1) - rot_p[1]) / denomina + 12 * EI*(1 - d[2])*((v(2)-compense_2) - rot_p[2]) /
denomina);
                F(2, 0) = gi = pow(Mi, 2) * (L / (3 * EI)) / (2 * (pow(1 - d[1],
2))) - dj_po*(Gcr + qq*log(1 - d[1]) / (1 - d[1]));
                F(3, 0) = gj = pow(Mj, 2) * (L / (3 * EI)) / (2 * (pow(1 - d[2],
2))) - dj_neg*(Gcr2+ qq2*log(1 - d[2]) / (1 - d[2]));
                static Matrix mid1(4, 1);
                mid1.addMatrixProduct(0.0, B0, F, 1.0);
                for (int j = 0; j <= 3; j++){
                    x_plus(j, 0) = x(j, 0) - mid1(j, 0);
                    s(j, 0) = x_plus(j, 0) - x(j, 0);
```

```
                        s_trans(0, j) = s(j, 0);
                }

                Mi = x_plus(0, 0);
                Mj = x_plus(1, 0);
                d[1] = x_plus(2, 0);
                d[2] = x_plus(3, 0);

                //don't forget to change
                if (abs(s(0, 0)) <=accu && abs(s(1, 0)) <=accu && abs(s(2, 0))
<=accu && abs(s(3, 0)) <=accu){
                if (abs(Mi)>Mu && Mi != 0){
                        Mi=dj_po*Mu*abs(Mi)/Mi;
                        signal=1;
                }
                if (abs(Mj)>Mu2 && Mj != 0){
                        Mj=dj_neg*Mu2*abs(Mj)/Mj;
                        signal=1;
                }
                if (d[1]>du){
                        d[1]=du;
                        signal=1;
                }
                if (d[2]>du2){
                        d[2]=du2;
                        signal=1;
                }
                if (signal == 1) continue;
                else break;
                }

                F_plus(0, 0) = f1 = Mi - dj_po*(12 * EI*(1 - d[1])*((v(1)-
compense_1) - rot_p[1]) / denomina + 6 * EI*(1 - d[1])*(1 - d[2])*((v(2)-compense_2) -
rot_p[2]) / denomina);
                F_plus(1, 0) = f2 = Mj - dj_neg*(6 * EI*(1 - d[1])*(1 -
d[2])*((v(1)-compense_1) - rot_p[1]) / denomina + 12 * EI*(1 - d[2])*((v(2)-compense_2) -
rot_p[2]) / denomina);
                F_plus(2, 0) = gi = pow(Mi, 2) * (L / (3 * EI)) / (2 * (pow(1 -
d[1], 2))) - dj_po*(Gcr + qq*log(1 - d[1]) / (1 - d[1]));
                F_plus(3, 0) = gj = pow(Mj, 2) * (L / (3 * EI)) / (2 * (pow(1 -
d[2], 2))) - dj_neg*(Gcr2+ qq2*log(1 - d[2]) / (1 - d[2]));
                for (int k = 0; k <= 3; k++){
                        y(k, 0) = F_plus(k, 0) - F(k, 0);
                }
                static Matrix mid2(4, 1);
                static Matrix mid3(4, 1);
                static Matrix mid4(4, 4);
                static Matrix mid5(4, 4);
                mid2.addMatrixProduct(0.0, B0, y, 1.0);
                for (int m = 0; m <= 3; m++){
                        mid3(m, 0) = s(m, 0) - mid2(m, 0);
                }
                mid4.addMatrixProduct(0.0, mid3, s_trans, 1.0);
                mid5.addMatrixProduct(0.0, mid4, B0, 1.0);
                // don't forget to change sby
                sby = s(0, 0)*mid2(0, 0) + s(1, 0)*mid2(1, 0) + s(2, 0)*mid2(2, 0) +
s(3, 0)*mid2(3, 0);
                for (int n = 0; n <= 3; n++){
```

```
                    for (int o = 0; o <= 3; o++){
                        B0(n, o) += mid5(n, o) / sby;
                    }
                }
            }
        }
        else if (fi > 0 && gi > 0 && fj <= 0 && gj > 0){
            static Matrix B(5, 5);
            //f = [f1 f2 gi gj fi];
            //df = [diff(f, 'Mi'); diff(f, 'Mj'); diff(f, 'd[1]'); diff(f, 'd[2]');
diff(f, 'rot_p[1]')]'

        B(0,0)=1;
            B(0,1)=0;
            B(0,2)=-dj_po*((12*EI*(compense_1 - v(1) + rot_p[1]))/denomina -
(6*EI*(d[2] - 1)*(compense_2 - v(2) + rot_p[2]))/denomina + (12*EI*(d[1] - 1)*(d[2] -
1)*(compense_1 - v(1) + rot_p[1]))/(pow(denomina,2)/L) - (6*EI*(d[1] - 1)*pow((d[2] -
1),2)*(compense_2 - v(2) + rot_p[2]))/(pow(denomina,2)/L));
            B(0,3)=dj_po*((6*EI*(d[1] - 1)*(compense_2 - v(2) + rot_p[2]))/denomina -
(12*EI*pow((d[1] - 1),2)*(compense_1 - v(1) + rot_p[1]))/(pow(denomina,2)/L) +
(6*EI*pow((d[1] - 1),2)*(d[2] - 1)*(compense_2 - v(2) + rot_p[2]))/(pow(denomina,2)/L));
            B(0,4)=-(12*EI*dj_po*(d[1] - 1))/denomina;
        B(1,0)=0;
            B(1,1)=1;
            B(1,2)=dj_neg*((6*EI*(d[2] - 1)*(compense_1 - v(1) + rot_p[1]))/denomina -
(12*EI*pow((d[2] - 1),2)*(compense_2 - v(2) + rot_p[2]))/(pow(denomina,2)/L) +
(6*EI*(d[1] - 1)*pow((d[2] - 1),2)*(compense_1 - v(1) + rot_p[1]))/(pow(denomina,2)/L));
            B(1,3)=-dj_neg*((12*EI*(compense_2 - v(2) + rot_p[2]))/denomina -
(6*EI*(d[1] - 1)*(compense_1 - v(1) + rot_p[1]))/denomina + (12*EI*(d[1] - 1)*(d[2] -
1)*(compense_2 - v(2) + rot_p[2]))/(pow(denomina,2)/L) - (6*EI*pow((d[1] - 1),2)*(d[2] -
1)*(compense_1 - v(1) + rot_p[1]))/(pow(denomina,2)/L));
            B(1,4)=(6*EI*dj_neg*(d[1] - 1)*(d[2] - 1))/denomina;
        B(2,0)=(L*Mi)/(3*EI*pow((d[1] - 1),2));
            B(2,1)=0;
            B(2,2)=dj_po*(qq/pow((d[1] - 1),2) - (qq*log(1 - d[1]))/pow((d[1] - 1),2))
- (L*pow(Mi,2))/(3*EI*pow((d[1] - 1),3));
            B(2,3)=0;
            B(2,4)=0;
        B(3,0)=0;
            B(3,1)=(L*Mj)/(3*EI*pow((d[2] - 1),2));
            B(3,2)=0;
            B(3,3)=dj_neg*(qq2/pow((d[2] - 1),2) - (qq2*log(1 - d[2]))/pow((d[2] -
1),2)) - (L*pow(Mj,2))/(3*EI*pow((d[2] - 1),3));
            B(3,4)=0;
        B(4,0)=abs(Mi + c*dj_po*rot_p[1]*(d[1] - 1))/(Mi + c*dj_po*rot_p[1]*(d[1] - 1));;
            B(4,1)=0;
            B(4,2)=dj_po*k0 + c*dj_po*rot_p[1]*abs(Mi + c*dj_po*rot_p[1]*(d[1] -
1))/(Mi + c*dj_po*rot_p[1]*(d[1] - 1));
            B(4,3)=0;
            B(4,4)=c*dj_po*abs(Mi + c*dj_po*rot_p[1]*(d[1] - 1))/(Mi +
c*dj_po*rot_p[1]*(d[1] - 1))*(d[1] - 1);


            static Matrix B0(5, 5);
            invertMatrix(5, B, B0);

            static Matrix s(5, 1);
            static Matrix s_trans(1, 5);
```

```
            static Matrix y(5, 1);
            static Matrix x_plus(5, 1);
            static Matrix x(5, 1);
            static Matrix F(5, 1);
            static Matrix F_plus(5, 1);
            double sby;

            for (int i = 0; i < 10000; i++){
                    signal=0;
                    x(0, 0) = Mi;
                    x(1, 0) = Mj;
                    x(2, 0) = d[1];
                    x(3, 0) = d[2];
                    x(4, 0) = rot_p[1];
                    //define F
                    denomina = L*(d[1] + d[2] - d[1]* d[2] + 3);
                    F(0, 0) = f1 = Mi - dj_po*(12 * EI*(1 - d[1])*((v(1)-compense_1) -
rot_p[1]) / denomina + 6 * EI*(1 - d[1])*(1 - d[2])*((v(2)-compense_2) - rot_p[2]) /
denomina);
                    F(1, 0) = f2 = Mj - dj_neg*(6 * EI*(1 - d[1])*(1 - d[2])*((v(1)-
compense_1) - rot_p[1]) / denomina + 12 * EI*(1 - d[2])*((v(2)-compense_2) - rot_p[2]) /
denomina);
                    F(2, 0) = gi = pow(Mi, 2) * (L / (3 * EI)) / (2 * (pow(1 - d[1],
2))) - dj_po*(Gcr + qq*log(1 - d[1]) / (1 - d[1]));
                    F(3, 0) = gj = pow(Mj, 2) * (L / (3 * EI)) / (2 * (pow(1 - d[2],
2))) - dj_neg*(Gcr2+ qq2*log(1 - d[2]) / (1 - d[2]));
                    F(4, 0) = fi = abs(Mi - dj_po*(1 - d[1])*c*rot_p[1]) - dj_po* (1 -
d[1])*k0;
                    static Matrix mid1(5, 1);
                    mid1.addMatrixProduct(0.0, B0, F, 1.0);
                    for (int j = 0; j <= 4; j++){
                            x_plus(j, 0) = x(j, 0) - mid1(j, 0);
                            s(j, 0) = x_plus(j, 0) - x(j, 0);
                            s_trans(0, j) = s(j, 0);
                    }

                    Mi = x_plus(0, 0);
                    Mj = x_plus(1, 0);
                    d[1] = x_plus(2, 0);
                    d[2] = x_plus(3, 0);
                    rot_p[1] = x_plus(4, 0);

                    //don't forget to change
                    if (abs(s(0, 0)) <=accu && abs(s(1, 0)) <=accu && abs(s(2, 0))
<=accu && abs(s(3, 0)) <=accu && abs(s(4, 0)) <=accu){
            if (abs(Mi)>Mu && Mi != 0){
                            Mi=dj_po*Mu*abs(Mi)/Mi;
                            signal=1;
                    }
                    if (abs(Mj)>Mu2 && Mj != 0){
                            Mj=dj_neg*Mu2*abs(Mj)/Mj;
                            signal=1;
                    }
                    if (d[1]>du){
                            d[1]=du;
                            signal=1;
                    }
                    if (d[2]>du2){
```

```
                        d[2]=du2;
                        signal=1;
                    }
                    if (signal == 1) continue;
                    else break;
                    }

                    F_plus(0, 0) = f1 = Mi - dj_po*(12 * EI*(1 - d[1])*((v(1)-
compense_1) - rot_p[1]) / denomina + 6 * EI*(1 - d[1])*(1 - d[2])*((v(2)-compense_2) -
rot_p[2]) / denomina);
                    F_plus(1, 0) = f2 = Mj - dj_neg*(6 * EI*(1 - d[1])*(1 -
d[2])*((v(1)-compense_1) - rot_p[1]) / denomina + 12 * EI*(1 - d[2])*((v(2)-compense_2) -
rot_p[2]) / denomina);
                    F_plus(2, 0) = gi = pow(Mi, 2) * (L / (3 * EI)) / (2 * (pow(1 -
d[1], 2))) - dj_po*(Gcr + qq*log(1 - d[1]) / (1 - d[1]));
                    F_plus(3, 0) = gj = pow(Mj, 2) * (L / (3 * EI)) / (2 * (pow(1 -
d[2], 2))) - dj_neg*(Gcr2+ qq2*log(1 - d[2]) / (1 - d[2]));
                    F_plus(4, 0) = fi = abs(Mi - dj_po*(1 - d[1])*c*rot_p[1]) - dj_po*
(1 - d[1])*k0;
                    for (int k = 0; k <= 4; k++){
                        y(k, 0) = F_plus(k, 0) - F(k, 0);
                    }
                    static Matrix mid2(5, 1);
                    static Matrix mid3(5, 1);
                    static Matrix mid4(5, 5);
                    static Matrix mid5(5, 5);
                    mid2.addMatrixProduct(0.0, B0, y, 1.0);
                    for (int m = 0; m <= 4; m++){
                        mid3(m, 0) = s(m, 0) - mid2(m, 0);
                    }
                    mid4.addMatrixProduct(0.0, mid3, s_trans, 1.0);
                    mid5.addMatrixProduct(0.0, mid4, B0, 1.0);
                    // don't forget to change sby
                    sby = s(0, 0)*mid2(0, 0) + s(1, 0)*mid2(1, 0) + s(2, 0)*mid2(2, 0) +
s(3, 0)*mid2(3, 0) + s(4, 0)*mid2(4, 0);
                    for (int n = 0; n <= 4; n++){
                        for (int o = 0; o <= 4; o++){
                            B0(n, o) += mid5(n, o) / sby;
                        }
                    }
                }
        }
        else if (fi <= 0 && gi > 0 && fj > 0 && gj > 0){
                static Matrix B(5, 5);
                //f = [f1 f2 gi gj fj];
                //df = [diff(f, 'Mi'); diff(f, 'Mj'); diff(f, 'd[1]'); diff(f, 'd[2]');
diff(f, 'rot_p[2]')]'
                B(0,0)=1;
                B(0,1)=0;
                B(0,2)=-dj_po*((12*EI*(compense_1 - v(1) + rot_p[1]))/denomina -
(6*EI*(d[2] - 1)*(compense_2 - v(2) + rot_p[2]))/denomina + (12*EI*(d[1] - 1)*(d[2] -
1)*(compense_1 - v(1) + rot_p[1]))/(pow(denomina,2)/L) - (6*EI*(d[1] - 1)*pow((d[2] -
1),2)*(compense_2 - v(2) + rot_p[2]))/(pow(denomina,2)/L));
                B(0,3)=dj_po*((6*EI*(d[1] - 1)*(compense_2 - v(2) + rot_p[2]))/denomina -
(12*EI*pow((d[1] - 1),2)*(compense_1 - v(1) + rot_p[1]))/(pow(denomina,2)/L) +
(6*EI*pow((d[1] - 1),2)*(d[2] - 1)*(compense_2 - v(2) + rot_p[2]))/(pow(denomina,2)/L));
                B(0,4)=(6*EI*dj_po*(d[1] - 1)*(d[2] - 1))/denomina;
         B(1,0)=0;
```

```
            B(1,1)=1;
            B(1,2)=dj_neg*((6*EI*(d[2] - 1)*(compense_1 - v(1) + rot_p[1]))/denomina -
(12*EI*pow((d[2] - 1),2)*(compense_2 - v(2) + rot_p[2]))/(pow(denomina,2)/L) +
(6*EI*(d[1] - 1)*pow((d[2] - 1),2)*(compense_1 - v(1) + rot_p[1]))/(pow(denomina,2)/L));
            B(1,3)=-dj_neg*((12*EI*(compense_2 - v(2) + rot_p[2]))/denomina -
(6*EI*(d[1] - 1)*(compense_1 - v(1) + rot_p[1]))/denomina + (12*EI*(d[1] - 1)*(d[2] -
1)*(compense_2 - v(2) + rot_p[2]))/(pow(denomina,2)/L) - (6*EI*pow((d[1] - 1),2)*(d[2] -
1)*(compense_1 - v(1) + rot_p[1]))/(pow(denomina,2)/L));
            B(1,4)=-(12*EI*dj_neg*(d[2] - 1))/denomina;
      B(2,0)=(L*Mi)/(3*EI*pow((d[1] - 1),2));
            B(2,1)=0;
            B(2,2)=dj_po*(qq/pow((d[1] - 1),2) - (qq*log(1 - d[1]))/pow((d[1] - 1),2))
- (L*pow(Mi,2))/(3*EI*pow((d[1] - 1),3));
            B(2,3)=0;
            B(2,4)=0;
      B(3,0)=0;
            B(3,1)=(L*Mj)/(3*EI*pow((d[2] - 1),2));
            B(3,2)=0;
            B(3,3)=dj_neg*(qq2/pow((d[2] - 1),2) - (qq2*log(1 - d[2]))/pow((d[2] -
1),2)) - (L*pow(Mj,2))/(3*EI*pow((d[2] - 1),3));
            B(3,4)=0;
      B(4,0)=0;
            B(4,1)=abs(Mj + c2*dj_neg*rot_p[2]*(d[2] - 1))/(Mj +
c2*dj_neg*rot_p[2]*(d[2] - 1));
            B(4,2)=0;
            B(4,3)=dj_neg*k02 + c2*dj_neg*rot_p[2]*abs(Mj + c2*dj_neg*rot_p[2]*(d[2] -
1))/(Mj + c2*dj_neg*rot_p[2]*(d[2] - 1));
            B(4,4)=c2*dj_neg*abs(Mj + c2*dj_neg*rot_p[2]*(d[2] - 1))/(Mj +
c2*dj_neg*rot_p[2]*(d[2] - 1))*(d[2] - 1);

            static Matrix B0(5, 5);
            invertMatrix(5, B, B0);

            static Matrix s(5, 1);
            static Matrix s_trans(1, 5);
            static Matrix y(5, 1);
            static Matrix x_plus(5, 1);
            static Matrix x(5, 1);
            static Matrix F(5, 1);
            static Matrix F_plus(5, 1);
            double sby;

            for (int i = 0; i < 10000; i++){
                  signal=0;
                  x(0, 0) = Mi;
                  x(1, 0) = Mj;
                  x(2, 0) = d[1];
                  x(3, 0) = d[2];
                  x(4, 0) = rot_p[2];

                  //define F
                  denomina = L*(d[1] + d[2] - d[1]* d[2] + 3);
                  F(0, 0) = f1 = Mi - dj_po*(12 * EI*(1 - d[1])*((v(1)-compense_1) -
rot_p[1]) / denomina + 6 * EI*(1 - d[1])*(1 - d[2])*((v(2)-compense_2) - rot_p[2]) /
denomina);
                  F(1, 0) = f2 = Mj - dj_neg*(6 * EI*(1 - d[1])*(1 - d[2])*((v(1)-
compense_1) - rot_p[1]) / denomina + 12 * EI*(1 - d[2])*((v(2)-compense_2) - rot_p[2]) /
denomina);
```

```
                    F(2, 0) = gi = pow(Mi, 2) * (L / (3 * EI)) / (2 * (pow(1 - d[1],
2))) - dj_po*(Gcr + qq*log(1 - d[1]) / (1 - d[1]));
                    F(3, 0) = gj = pow(Mj, 2) * (L / (3 * EI)) / (2 * (pow(1 - d[2],
2))) - dj_neg*(Gcr2+ qq2*log(1 - d[2]) / (1 - d[2]));
                    F(4, 0) = fj = abs(Mj - dj_neg*(1 - d[2])*c2*rot_p[2]) - dj_neg*(1 -
d[2])*k02;
                    static Matrix mid1(5, 1);
                    mid1.addMatrixProduct(0.0, B0, F, 1.0);
                    for (int j = 0; j <= 4; j++){
                          x_plus(j, 0) = x(j, 0) - mid1(j, 0);
                          s(j, 0) = x_plus(j, 0) - x(j, 0);
                          s_trans(0, j) = s(j, 0);
                    }

                    Mi = x_plus(0, 0);
                    Mj = x_plus(1, 0);
                    d[1] = x_plus(2, 0);
                    d[2] = x_plus(3, 0);
                    rot_p[2] = x_plus(4, 0);

                    //don't forget to change
                    if (abs(s(0, 0)) <=accu && abs(s(1, 0)) <=accu && abs(s(2, 0))
<=accu && abs(s(3, 0)) <=accu && abs(s(4, 0)) <=accu){
                    if (abs(Mi)>Mu && Mi != 0){
                          Mi=dj_po*Mu*abs(Mi)/Mi;
                          signal=1;
                    }
                    if (abs(Mj)>Mu2 && Mj != 0){
                          Mj=dj_neg*Mu2*abs(Mj)/Mj;
                          signal=1;
                    }
                    if (d[1]>du){
                          d[1]=du;
                          signal=1;
                    }
                    if (d[2]>du2){
                          d[2]=du2;
                          signal=1;
                    }
                    if (signal == 1) continue;
                    else break;
                    }

                    F_plus(0, 0) = f1 = Mi - dj_po*(12 * EI*(1 - d[1])*((v(1)-
compense_1) - rot_p[1]) / denomina + 6 * EI*(1 - d[1])*(1 - d[2])*((v(2)-compense_2) -
rot_p[2]) / denomina);
                    F_plus(1, 0) = f2 = Mj - dj_neg*(6 * EI*(1 - d[1])*(1 -
d[2])*((v(1)-compense_1) - rot_p[1]) / denomina + 12 * EI*(1 - d[2])*((v(2)-compense_2) -
rot_p[2]) / denomina);
                    F_plus(2, 0) = gi = pow(Mi, 2) * (L / (3 * EI)) / (2 * (pow(1 -
d[1], 2))) - dj_po*(Gcr + qq*log(1 - d[1]) / (1 - d[1]));
                    F_plus(3, 0) = gj = pow(Mj, 2) * (L / (3 * EI)) / (2 * (pow(1 -
d[2], 2))) - dj_neg*(Gcr2+ qq2*log(1 - d[2]) / (1 - d[2]));
                    F_plus(4, 0) = fj = abs(Mj - dj_neg*(1 - d[2])*c2*rot_p[2]) -
dj_neg*(1 - d[2])*k02;
                    for (int k = 0; k <= 4; k++){
                          y(k, 0) = F_plus(k, 0) - F(k, 0);
                    }
```

```
                        static Matrix mid2(5, 1);
                        static Matrix mid3(5, 1);
                        static Matrix mid4(5, 5);
                        static Matrix mid5(5, 5);
                        mid2.addMatrixProduct(0.0, B0, y, 1.0);
                        for (int m = 0; m <= 4; m++){
                               mid3(m, 0) = s(m, 0) - mid2(m, 0);
                        }
                        mid4.addMatrixProduct(0.0, mid3, s_trans, 1.0);
                        mid5.addMatrixProduct(0.0, mid4, B0, 1.0);
                        // don't forget to change sby
                        sby = s(0, 0)*mid2(0, 0) + s(1, 0)*mid2(1, 0) + s(2, 0)*mid2(2, 0) +
s(3, 0)*mid2(3, 0) + s(4, 0)*mid2(4, 0);
                        for (int n = 0; n <= 4; n++){
                               for (int o = 0; o <= 4; o++){
                                      B0(n, o) += mid5(n, o) / sby;
                               }
                        }
                 }
          }
          else if (fi > 0 && gi > 0 && fj > 0 && gj > 0){
                 static Matrix B(6, 6);
                 //f = [f1 f2 gi gj fi fj];
                 //df = [diff(f, 'Mi'); diff(f, 'Mj'); diff(f, 'd[1]'); diff(f, 'd[2]');
diff(f, 'rot_p[1]'); diff(f, 'rot_p[2]')]'
                 B(0,0)=1;
                 B(0,1)=0;
                 B(0,2)=-dj_po*((12*EI*(compense_1 - v(1) + rot_p[1]))/denomina -
(6*EI*(d[2] - 1)*(compense_2 - v(2) + rot_p[2]))/denomina + (12*EI*(d[1] - 1)*(d[2] -
1)*(compense_1 - v(1) + rot_p[1]))/(pow(denomina,2)/L) - (6*EI*(d[1] - 1)*pow((d[2] -
1),2)*(compense_2 - v(2) + rot_p[2]))/(pow(denomina,2)/L));
                 B(0,3)=dj_po*((6*EI*(d[1] - 1)*(compense_2 - v(2) + rot_p[2]))/denomina -
(12*EI*pow((d[1] - 1),2)*(compense_1 - v(1) + rot_p[1]))/(pow(denomina,2)/L) +
(6*EI*pow((d[1] - 1),2)*(d[2] - 1)*(compense_2 - v(2) + rot_p[2]))/(pow(denomina,2)/L));
                 B(0,4)=-(12*EI*dj_po*(d[1] - 1))/denomina;
                 B(0,5)=(6*EI*dj_po*(d[1] - 1)*(d[2] - 1))/denomina;
            B(1,0)=0;
                 B(1,1)=1;
                 B(1,2)=dj_neg*((6*EI*(d[2] - 1)*(compense_1 - v(1) + rot_p[1]))/denomina -
(12*EI*pow((d[2] - 1),2)*(compense_2 - v(2) + rot_p[2]))/(pow(denomina,2)/L) +
(6*EI*(d[1] - 1)*pow((d[2] - 1),2)*(compense_1 - v(1) + rot_p[1]))/(pow(denomina,2)/L));
                 B(1,3)=-dj_neg*((12*EI*(compense_2 - v(2) + rot_p[2]))/denomina -
(6*EI*(d[1] - 1)*(compense_1 - v(1) + rot_p[1]))/denomina + (12*EI*(d[1] - 1)*(d[2] -
1)*(compense_2 - v(2) + rot_p[2]))/(pow(denomina,2)/L) - (6*EI*pow((d[1] - 1),2)*(d[2] -
1)*(compense_1 - v(1) + rot_p[1]))/(pow(denomina,2)/L));
                 B(1,4)=(6*EI*dj_neg*(d[1] - 1)*(d[2] - 1))/denomina;
                 B(1,5)=-(12*EI*dj_neg*(d[2] - 1))/denomina;
            B(2,0)=(L*Mi)/(3*EI*pow((d[1] - 1),2));
                 B(2,1)=0;
                 B(2,2)=dj_po*(qq/pow((d[1] - 1),2) - (qq*log(1 - d[1]))/pow((d[1] - 1),2))
- (L*pow(Mi,2))/(3*EI*pow((d[1] - 1),3));
                 B(2,3)=0;
                 B(2,4)=0;
                 B(2,5)=0;
            B(3,0)=0;
                 B(3,1)=(L*Mj)/(3*EI*pow((d[2] - 1),2));
                 B(3,2)=0;
```

```
              B(3,3)=dj_neg*(qq2/pow((d[2] - 1),2) - (qq2*log(1 - d[2]))/pow((d[2] -
1),2)) - (L*pow(Mj,2))/(3*EI*pow((d[2] - 1),3));
              B(3,4)=0;
              B(3,5)=0;
        B(4,0)=abs(Mi + c*dj_po*rot_p[1]*(d[1] - 1))/(Mi + c*dj_po*rot_p[1]*(d[1] - 1));
              B(4,1)=0;
              B(4,2)=dj_po*k0 + c*dj_po*rot_p[1]*abs(Mi + c*dj_po*rot_p[1]*(d[1] -
1))/(Mi + c*dj_po*rot_p[1]*(d[1] - 1));
              B(4,3)=0;
              B(4,4)=c*dj_po*abs(Mi + c*dj_po*rot_p[1]*(d[1] - 1))/(Mi +
c*dj_po*rot_p[1]*(d[1] - 1))*(d[1] - 1);
              B(4,5)=0;
        B(5,0)=0;
              B(5,1)=abs(Mj + c2*dj_neg*rot_p[2]*(d[2] - 1))/(Mj +
c2*dj_neg*rot_p[2]*(d[2] - 1));
              B(5,2)=0;
              B(5,3)=dj_neg*k02 + c2*dj_neg*rot_p[2]*abs(Mj + c2*dj_neg*rot_p[2]*(d[2] -
1))/(Mj + c2*dj_neg*rot_p[2]*(d[2] - 1));
              B(5,4)=0;
              B(5,5)=c2*dj_neg*abs(Mj + c2*dj_neg*rot_p[2]*(d[2] - 1))/(Mj +
c2*dj_neg*rot_p[2]*(d[2] - 1))*(d[2] - 1);

              static Matrix B0(6, 6);
              invertMatrix(6, B, B0);

              static Matrix s(6, 1);
              static Matrix s_trans(1, 6);
              static Matrix y(6, 1);
              static Matrix x_plus(6, 1);
              static Matrix x(6, 1);
              static Matrix F(6, 1);
              static Matrix F_plus(6, 1);
              double sby;

              for (int i = 0; i < 10000; i++){
                     signal=0;
                     x(0, 0) = Mi;
                     x(1, 0) = Mj;
                     x(2, 0) = d[1];
                     x(3, 0) = d[2];
                     x(4, 0) = rot_p[1];
                     x(5, 0) = rot_p[2];
                     //define F
                     denomina = L*(d[1] + d[2] - d[1]* d[2] + 3);
                     F(0, 0) = f1 = Mi - dj_po*(12 * EI*(1 - d[1])*((v(1)-compense_1) -
rot_p[1]) / denomina + 6 * EI*(1 - d[1])*(1 - d[2])*((v(2)-compense_2) - rot_p[2]) /
denomina);
                     F(1, 0) = f2 = Mj - dj_neg*(6 * EI*(1 - d[1])*(1 - d[2])*((v(1)-
compense_1) - rot_p[1]) / denomina + 12 * EI*(1 - d[2])*((v(2)-compense_2) - rot_p[2]) /
denomina);
                     F(2, 0) = gi = pow(Mi, 2) * (L / (3 * EI)) / (2 * (pow(1 - d[1],
2))) - dj_po*(Gcr + qq*log(1 - d[1]) / (1 - d[1]));
                     F(3, 0) = gj = pow(Mj, 2) * (L / (3 * EI)) / (2 * (pow(1 - d[2],
2))) - dj_neg*(Gcr2+ qq2*log(1 - d[2]) / (1 - d[2]));
                     F(4, 0) = fi = abs(Mi - dj_po*(1 - d[1])*c*rot_p[1]) - dj_po* (1 -
d[1])*k0;
                     F(5, 0) = fj = abs(Mj - dj_neg*(1 - d[2])*c2*rot_p[2]) - dj_neg*(1 -
d[2])*k02;
```

```
                    static Matrix mid1(6, 1);
                    mid1.addMatrixProduct(0.0, B0, F, 1.0);
                    for (int j = 0; j <= 5; j++){
                           x_plus(j, 0) = x(j, 0) - mid1(j, 0);
                           s(j, 0) = x_plus(j, 0) - x(j, 0);
                           s_trans(0, j) = s(j, 0);
                    }

                    Mi = x_plus(0, 0);
                    Mj = x_plus(1, 0);
                    d[1] = x_plus(2, 0);
                    d[2] = x_plus(3, 0);
                    rot_p[1] = x_plus(4, 0);
                    rot_p[2] = x_plus(5, 0);

                    //don't forget to change
                    if (abs(s(0, 0)) <=accu && abs(s(1, 0)) <=accu && abs(s(2, 0))
<=accu && abs(s(3, 0)) <=accu && abs(s(4, 0)) <=accu && abs(s(5, 0)) <=accu){
                 if (abs(Mi)>Mu && Mi != 0){
                           Mi=dj_po*Mu*abs(Mi)/Mi;
                           signal=1;
                    }
                    if (abs(Mj)>Mu2 && Mj != 0){
                           Mj=dj_neg*Mu2*abs(Mj)/Mj;
                           signal=1;
                    }
                    if (d[1]>du){
                           d[1]=du;
                           signal=1;
                    }
                    if (d[2]>du2){
                           d[2]=du2;
                           signal=1;
                    }
                    if (signal == 1) continue;
                    else break;
                    }

                    F_plus(0, 0) = f1 = Mi - dj_po*(12 * EI*(1 - d[1])*((v(1)-
compense_1) - rot_p[1]) / denomina + 6 * EI*(1 - d[1])*(1 - d[2])*((v(2)-compense_2) -
rot_p[2]) / denomina);
                    F_plus(1, 0) = f2 = Mj - dj_neg*(6 * EI*(1 - d[1])*(1 -
d[2])*((v(1)-compense_1) - rot_p[1]) / denomina + 12 * EI*(1 - d[2])*((v(2)-compense_2) -
rot_p[2]) / denomina);
                    F_plus(2, 0) = gi = pow(Mi, 2) * (L / (3 * EI)) / (2 * (pow(1 -
d[1], 2))) - dj_po*(Gcr + qq*log(1 - d[1]) / (1 - d[1]));
                    F_plus(3, 0) = gj = pow(Mj, 2) * (L / (3 * EI)) / (2 * (pow(1 -
d[2], 2))) - dj_neg*(Gcr2+ qq2*log(1 - d[2]) / (1 - d[2]));
                    F_plus(4, 0) = fi = abs(Mi - dj_po*(1 - d[1])*c*rot_p[1]) - dj_po*
(1 - d[1])*k0;
                    F_plus(5, 0) = fj = abs(Mj - dj_neg*(1 - d[2])*c2*rot_p[2]) -
dj_neg*(1 - d[2])*k02;
                    for (int k = 0; k <= 5; k++){
                           y(k, 0) = F_plus(k, 0) - F(k, 0);
                    }
                    static Matrix mid2(6, 1);
                    static Matrix mid3(6, 1);
                    static Matrix mid4(6, 6);
```

```
                    static Matrix mid5(6, 6);
                    mid2.addMatrixProduct(0.0, B0, y, 1.0);
                    for (int m = 0; m <= 5; m++){
                           mid3(m, 0) = s(m, 0) - mid2(m, 0);
                    }
                    mid4.addMatrixProduct(0.0, mid3, s_trans, 1.0);
                    mid5.addMatrixProduct(0.0, mid4, B0, 1.0);
                    // don't forget to change sby
                    sby = s(0, 0)*mid2(0, 0) + s(1, 0)*mid2(1, 0) + s(2, 0)*mid2(2, 0) +
s(3, 0)*mid2(3, 0) + s(4, 0)*mid2(4, 0) + s(5, 0)*mid2(5, 0);
                    for (int n = 0; n <= 5; n++){
                           for (int o = 0; o <= 5; o++){
                                  B0(n, o) += mid5(n, o) / sby;
                           }
                    }
             }
      }
      // failure process


      double strain_s=0;
      double LL=theCoordTransf->getInitialLength();
      double Lu = Lp;
      double Le = LL-2*Lu;
      double Es = 17500;
      double A2=18;
      double h = 6;
   // As1<As2
      if (cv[0] != 0)
      strain_s=v(0)/(Lu/(As1*Es)+Le/(A2*E)+Lu/(As2*Es))/(Es*As1);
      strain_s=strain_s+plastic_strain;
      double strain_y=fu/Es+plastic_strain;
      if (cv[0]==0 || strain_s == 0){
             q(0)=EAoverL*v(0);

      }
      else{
      if (strain_s<strain_y && strain_s > 0){
      q(0) = Es*strain_s*As1*(1-sqrt(cv[0]/v(0)));
      }
      else if (strain_s>strain_y && strain_s<strain_u){
             strain_s=(v(0)-As1*fu*Le/(E*A2)-As1*fu*Lu/(Es*As2))/Lu;
   q(0) = fu*As1*(1-sqrt(cv[0]/v(0)));

      if (strain_s>strain_u)
                q(0)=0;
      }
      }
  //unloading
      double ss=axial/(Lu/(As1*Es)+Le/(A2*E)+Lu/(As2*Es))/(Es*As1);

      if (ss >= strain_y && ss < strain_u && v(0)<axial && v(0)>0 && v(0) > cv[0]){
             ss=(axial-As1*fu*Le/(E*A2)-As1*fu*Lu/(Es*As2))/Lu;
             plastic_strain=ss-strain_y;
             strain_s=(v(0)-
(Lu*plastic_strain))/(Lu/(As1*Es)+Le/(A2*E)+Lu/(As2*Es))/(Es*As1);
             q(0) = Es*strain_s*As1*(1-sqrt(cv[0]/v(0)));
      }
```

```
	q(1) = Mi;
	q(2) = Mj;

	q(0) += q0[0];
	q(1) += q0[1];
	q(2) += q0[2];

	// Vector for reactions in basic system
	Vector p0Vec(p0, 3);

	P = theCoordTransf->getGlobalResistingForce(q, p0Vec);


	return P;

}

int
LumpedDamageEle2D2::sendSelf(int cTag, Channel &theChannel)
{
	int res = 0;

	static Vector data(31);

	data(0) = E;
	data(1) = I;
	data(2) = A;
	data(3) = qq;
	data(4) = c;
	data(5) = k0;
	data(6) = Gcr;
	data(7) = Mu;
	data(8) = du;
	data(9) = qq2;
	data(10) = c2;
	data(11) = k02;
	data(12) = Gcr2;
	data(13) = Mu2;
	data(14) = du2;
	data(15) = As1;
	data(16) = As2;
	data(17) = fu;
	data(18) = strain_u;
	data(19) = Lp;
	data(20) = rho;
	data(21) = cMass;
	data(22) = this->getTag();
	data(23) = connectedExternalNodes(0);
	data(24) = connectedExternalNodes(1);
	data(25) = theCoordTransf->getClassTag();

	int dbTag = theCoordTransf->getDbTag();

	if (dbTag == 0) {
		dbTag = theChannel.getDbTag();
		if (dbTag != 0)
```

```
			theCoordTransf->setDbTag(dbTag);
	}

	data(26) = dbTag;

	data(27) = alphaM;
	data(28) = betaK;
	data(29) = betaK0;
	data(30) = betaKc;

	// Send the data vector
	res += theChannel.sendVector(this->getDbTag(), cTag, data);
	if (res < 0) {
		opserr << "LumpedDamageEle2D2::sendSelf -- could not send data Vector\n";
		return res;
	}

	// Ask the CoordTransf to send itself
	res += theCoordTransf->sendSelf(cTag, theChannel);
	if (res < 0) {
		opserr << "LumpedDamageEle2D2::sendSelf -- could not send CoordTransf\n";
		return res;
	}

	return res;
}

int
LumpedDamageEle2D2::recvSelf(int cTag, Channel &theChannel, FEM_ObjectBroker &theBroker)
{
	int res = 0;

	static Vector data(31);

	res += theChannel.recvVector(this->getDbTag(), cTag, data);
	if (res < 0) {
		opserr << "LumpedDamageEle2D2::recvSelf -- could not receive data
Vector\n";
		return res;
	}

	E = data(0);
	I = data(1);
	A = data(2);


	alphaM = data(27);
	betaK = data(28);
	betaK0 = data(29);
	betaKc = data(30);

	rho = data(20);
	cMass = (int)data(21);
	this->setTag((int)data(22));
	connectedExternalNodes(0) = (int)data(23);
	connectedExternalNodes(1) = (int)data(24);

	// Check if the CoordTransf is null; if so, get a new one
```

```
	int crdTag = (int)data(25);
	if (theCoordTransf == 0) {
		theCoordTransf = theBroker.getNewCrdTransf(crdTag);
		if (theCoordTransf == 0) {
			opserr << "LumpedDamageEle2D2::recvSelf -- could not get a 
CrdTransf2d\n";
			exit(-1);
		}
	}

	// Check that the CoordTransf is of the right type; if not, delete
	// the current one and get a new one of the right type
	if (theCoordTransf->getClassTag() != crdTag) {
		delete theCoordTransf;
		theCoordTransf = theBroker.getNewCrdTransf(crdTag);
		if (theCoordTransf == 0) {
			opserr << "LumpedDamageEle2D2::recvSelf -- could not get a 
CrdTransf2d\n";
			exit(-1);
		}
	}

	// Now, receive the CoordTransf
	theCoordTransf->setDbTag((int)data(26));
	res += theCoordTransf->recvSelf(cTag, theChannel, theBroker);
	if (res < 0) {
		opserr << "LumpedDamageEle2D2::recvSelf -- could not receive 
CoordTransf\n";
		return res;
	}

	return res;
}

void
LumpedDamageEle2D2::Print(OPS_Stream &s, int flag)
{
	// to update forces!
	this->getResistingForce();

	if (flag == -1) {
		int eleTag = this->getTag();
		s << "DM_BEAM\t" << eleTag << "\t";
		s << 0 << "\t" << 0 << "\t" << connectedExternalNodes(0) << "\t" << 
connectedExternalNodes(1);
		s << "0\t0.0000000\n";
	}
	else {
		this->getResistingForce();
		s << "\nLumpedDamageEle2D: " << this->getTag() << endln;
		s << "\tConnected Nodes: " << connectedExternalNodes;
		s << "\tCoordTransf: " << theCoordTransf->getTag() << endln;
		s << "\tmass density:  " << rho << ", cMass: " << cMass << endln;
		double P = q(0);
		double M1 = q(1);
		double M2 = q(2);
		const Vector& disp1 = theNodes[0]->getDisp();
	 const Vector& disp2 = theNodes[1]->getDisp();
```

```
            const Vector& crds1 = theNodes[0]->getCrds();
            const Vector& crds2 = theNodes[0]->getCrds();

         Vector diff = disp2-disp1;

            double L=sqrt(pow(crds1(0)-crds2(0)+diff(0),2)+pow(crds1(1)-
crds2(1)+diff(1),2));
               double V = (M1 + M2) / L;
               s << "\tEnd 1 Forces (P V M): " << -P + p0[0]
                      << " " << V + p0[1] << " " << M1 << endln;
               s << "\tEnd 2 Forces (P V M): " << P
                      << " " << -V + p0[2] << " " << M2 << endln;
       }
}

int
LumpedDamageEle2D2::displaySelf(Renderer &theViewer, int displayMode, float fact, const
char **modes, int numMode)
{
       static Vector v1(3);
       static Vector v2(3);
       static Vector vp(3);

       theNodes[0]->getDisplayCrds(v1, fact);
       theNodes[1]->getDisplayCrds(v2, fact);

       float d1 = 0.0;
       float d2 = 0.0;
       float d3 = 0.0;

       int res = 0;

       if (displayMode > 0 && numMode == 0) {

              res += theViewer.drawLine(v1, v2, d1, d1, this->getTag(), 0);

       }
       else if (displayMode < 0) {

              theNodes[0]->getDisplayCrds(v1, 0.);
              theNodes[1]->getDisplayCrds(v2, 0.);

              // add eigenvector values
              int mode = displayMode  *  -1;

              const Matrix &eigen1 = theNodes[0]->getEigenvectors();
              const Matrix &eigen2 = theNodes[1]->getEigenvectors();
              if (eigen1.noCols() >= mode) {
                     for (int i = 0; i < 2; i++) {
                            v1(i) += eigen1(i, mode - 1)*fact;
                            v2(i) += eigen2(i, mode - 1)*fact;
                     }
              }

              res = theViewer.drawLine(v1, v2, 0.0, 0.0, this->getTag(), 0);
       }

       if (numMode > 0) {
```

```
        // calculate q for potential need below
        this->getResistingForce();
        vp = theCoordTransf->getBasicTrialDisp();
}

for (int i = 0; i<numMode; i++) {

        const char *theMode = modes[i];
        if (strcmp(theMode, "axialForce") == 0) {
                d1 = q(0);
                res += theViewer.drawLine(v1, v2, d1, d1, this->getTag(), i);

        }
        else if (strcmp(theMode, "endMoments") == 0) {

                d1 = q(1);
                d2 = q(2);
                static Vector delta(3); delta = v2 - v1; delta /= 20.;
                res += theViewer.drawPoint(v1 + delta, d1, this->getTag(), i);
                res += theViewer.drawPoint(v2 - delta, d2, this->getTag(), i);

        }
        else if (strcmp(theMode, "localForces") == 0) {
                d1 = q(0);
                d2 = q(1);
                d3 = q(2);
                static Vector delta(3); delta = v2 - v1; delta /= 20;
                res += theViewer.drawPoint(v1 + delta, d2, this->getTag(), i);
                res += theViewer.drawPoint(v2 - delta, d3, this->getTag(), i);
                res += theViewer.drawLine(v1, v2, d1, d1, this->getTag(), i);

        }
        else if (strcmp(theMode, "axialDeformation") == 0) {
                d1 = vp(0);
                res += theViewer.drawLine(v1, v2, d1, d1, this->getTag(), i);

        }
        else if (strcmp(theMode, "endRotations") == 0) {

                d1 = vp(1);
                d2 = vp(2);
                static Vector delta(3); delta = v2 - v1; delta /= 20.;
                res += theViewer.drawPoint(v1 + delta, d1, this->getTag(), i);
                res += theViewer.drawPoint(v2 - delta, d2, this->getTag(), i);

        }
        else if (strcmp(theMode, "localDeformations") == 0) {
                d1 = vp(0);
                d2 = vp(1);
                d3 = vp(2);
                static Vector delta(3); delta = v2 - v1; delta /= 20;
                res += theViewer.drawPoint(v1 + delta, d2, this->getTag(), i);
                res += theViewer.drawPoint(v2 - delta, d3, this->getTag(), i);
                res += theViewer.drawLine(v1, v2, d1, d1, this->getTag(), i);

        }
        else if (strcmp(theMode, "plasticDeformations") == 0) {
                d1 = 0.;
```

```
                    d2 = 0.;
                    d3 = 0.;
                    static Vector delta(3); delta = v2 - v1; delta /= 20;
                    res += theViewer.drawPoint(v1 + delta, d2, this->getTag(), i);
                    res += theViewer.drawPoint(v2 - delta, d3, this->getTag(), i);
                    res += theViewer.drawLine(v1, v2, d1, d1, this->getTag(), i);
                }

        }

        return res;
}

Response*
LumpedDamageEle2D2::setResponse(const char **argv, int argc, OPS_Stream &output)
{

        Response *theResponse = 0;
        output.tag("ElementOutput");
        output.attr("eleType", "LumpedDamageEle2D2");
        output.attr("eleTag", this->getTag());
        output.attr("node1", connectedExternalNodes[0]);
        output.attr("node2", connectedExternalNodes[1]);

        static Vector dd(3);
         dd(1)=d[0];
         dd(2)=d[1];
         dd(3)=d[2];
        static Vector rot_pp(3);
        rot_pp(1)=rot_p[0];
        rot_pp(2)=rot_p[1];
        rot_pp(3)=rot_p[2];
        // damage
     if (strcmp(argv[0], "damage") == 0) {
                output.tag("ResponseType", "d[0]");
                output.tag("ResponseType", "d[1]");
                output.tag("ResponseType", "d[2]");
     theResponse = new ElementResponse(this, 2, dd);
        }
        else if (strcmp(argv[0], "plasticrotation") == 0) {
            output.tag("ResponseType", "rot_p_0");
                output.tag("ResponseType", "rot_p_1");
                output.tag("ResponseType", "rot_p_2");
        theResponse = new ElementResponse(this, 3, rot_pp);
        }
        // global forces
        else if (strcmp(argv[0], "force") == 0 || strcmp(argv[0], "forces") == 0 ||
                strcmp(argv[0], "globalForce") == 0 || strcmp(argv[0], "globalForces") ==
0) {

                output.tag("ResponseType", "Px_1");
                output.tag("ResponseType", "Py_1");
                output.tag("ResponseType", "Mz_1");
                output.tag("ResponseType", "Px_2");
                output.tag("ResponseType", "Py_2");
                output.tag("ResponseType", "Mz_2");

                theResponse = new ElementResponse(this, 4, P);
```

```
            // local forces
        }
        else if (strcmp(argv[0], "localForce") == 0 || strcmp(argv[0], "localForces") ==
0) {

            output.tag("ResponseType", "N_1");
            output.tag("ResponseType", "V_1");
            output.tag("ResponseType", "M_1");
            output.tag("ResponseType", "N_2");
            output.tag("ResponseType", "V_2");
            output.tag("ResponseType", "M_2");

            theResponse = new ElementResponse(this, 5, P);

            // basic forces
        }
        else if (strcmp(argv[0], "basicForce") == 0 || strcmp(argv[0], "basicForces") ==
0) {

            output.tag("ResponseType", "N");
            output.tag("ResponseType", "M_1");
            output.tag("ResponseType", "M_2");

            theResponse = new ElementResponse(this, 6, Vector(3));

            // deformations
        }
        else if (strcmp(argv[0], "deformatons") == 0 ||
            strcmp(argv[0], "basicDeformations") == 0) {

            output.tag("ResponseType", "eps");
            output.tag("ResponseType", "theta1");
            output.tag("ResponseType", "theta2");
            theResponse = new ElementResponse(this, 7, Vector(3));

            // chord rotation -
        }
        else if (strcmp(argv[0], "chordRotation") == 0 || strcmp(argv[0],
"chordDeformation") == 0
            || strcmp(argv[0], "basicDeformation") == 0) {

            output.tag("ResponseType", "eps");
            output.tag("ResponseType", "theta1");
            output.tag("ResponseType", "theta2");

            theResponse = new ElementResponse(this, 7, Vector(3));
        }
        output.endTag(); // ElementOutput

        return theResponse;
}

int
LumpedDamageEle2D2::getResponse(int responseID, Information &eleInfo)
{
    static Vector dd(3);
        dd(1)=d[0];
```

```
     dd(2)=d[1];
     dd(3)=d[2];
    static Vector rot_pp(3);
    rot_pp(1)=rot_p[0];
    rot_pp(2)=rot_p[1];
    rot_pp(3)=rot_p[2];
    double N, M1, M2, V;
    const Vector& disp1 = theNodes[0]->getDisp();
const Vector& disp2 = theNodes[1]->getDisp();
    const Vector& crds1 = theNodes[0]->getCrds();
    const Vector& crds2 = theNodes[1]->getCrds();

Vector diff = disp2-disp1;

    double L=sqrt(pow(crds1(0)-crds2(0)+diff(0),2)+pow(crds1(1)-crds2(1)+diff(1),2));

    this->getResistingForce();

    switch (responseID) {

    case 1: // stiffness
           return eleInfo.setMatrix(this->getTangentStiff());

    case 2: //damage

           return eleInfo.setVector(dd);

    case 3: //plastic rotation

           return eleInfo.setVector(rot_pp);

    case 4: // global forces
           return eleInfo.setVector(this->getResistingForce());

    case 5: // local forces
           // Axial
           N = q(0);
           P(3) = N;
           P(0) = -N + p0[0];
           // Moment
           M1 = Mi;
           M2 = Mj;
           P(2) = M1;
           P(5) = M2;
           // Shear
           V = (M1 + M2) / L;
           P(1) = V + p0[1];
           P(4) = -V + p0[2];
           return eleInfo.setVector(P);

    case 6: // basic forces
           return eleInfo.setVector(q);

    case 7:
           return eleInfo.setVector(theCoordTransf->getBasicTrialDisp());

    default:
           return -1;
```

```
	}
}

int
LumpedDamageEle2D2::setParameter(const char **argv, int argc, Parameter &param)
{
	if (argc < 1)
		return -1;

	// E of the beam interior
	if (strcmp(argv[0], "E") == 0)
		return param.addObject(1, this);

	// I of the beam interior
	if (strcmp(argv[0], "I") == 0)
		return param.addObject(2, this);

	// A of the beam interior
	if (strcmp(argv[0], "A") == 0)
		return param.addObject(3, this);

	// qq of the beam interior
	if (strcmp(argv[0], "qq") == 0)
		return param.addObject(4, this);

	// c of the beam interior
	if (strcmp(argv[0], "c") == 0)
		return param.addObject(5, this);

	// k0 of the beam interior
	if (strcmp(argv[0], "k0") == 0)
		return param.addObject(6, this);

	// Gcr of the beam interior
	if (strcmp(argv[0], "Gcr") == 0)
		return param.addObject(7, this);

	// Mu of the beam interior
	if (strcmp(argv[0], "Mu") == 0)
		return param.addObject(8, this);

	// du of the beam interior
	if (strcmp(argv[0], "du") == 0)
		return param.addObject(9, this);

	// qq of the beam interior
	if (strcmp(argv[0], "qq2") == 0)
		return param.addObject(10, this);

	// c of the beam interior
	if (strcmp(argv[0], "c2") == 0)
		return param.addObject(11, this);

	// k0 of the beam interior
	if (strcmp(argv[0], "k02") == 0)
		return param.addObject(12, this);

	// Gcr of the beam interior
```

```
	if (strcmp(argv[0], "Gcr2") == 0)
		return param.addObject(13, this);

	// Mu of the beam interior
	if (strcmp(argv[0], "Mu2") == 0)
		return param.addObject(14, this);

	// du of the beam interior
	if (strcmp(argv[0], "du2") == 0)
		return param.addObject(15, this);

	if (strcmp(argv[0], "As1") == 0)
		return param.addObject(16, this);

	if (strcmp(argv[0], "As2") == 0)
		return param.addObject(17, this);

	if (strcmp(argv[0], "fu") == 0)
		return param.addObject(18, this);

	if (strcmp(argv[0], "strain_u") == 0)
		return param.addObject(19, this);
	if (strcmp(argv[0], "Lp") == 0)
		return param.addObject(20, this);
	return -1;
}

int
LumpedDamageEle2D2::updateParameter(int parameterID, Information &info)
{
	switch (parameterID) {
	case -1:
		return -1;
	case 1:
		E = info.theDouble;
		return 0;
	case 2:
		A = info.theDouble;
		return 0;
	case 3:
		I = info.theDouble;
		return 0;
	case 4:
		qq = info.theDouble;
		return 0;
	case 5:
		c = info.theDouble;
		return 0;
	case 6:
		k0 = info.theDouble;
		return 0;
	case 7:
		Gcr = info.theDouble;
		return 0;
	case 8:
		Mu = info.theDouble;
		return 0;
	case 9:
```

```
            du = info.theDouble;
            return 0;
        case 10:
            qq2 = info.theDouble;
            return 0;
        case 11:
            c2 = info.theDouble;
            return 0;
        case 12:
            k02 = info.theDouble;
            return 0;
        case 13:
            Gcr2 = info.theDouble;
            return 0;
        case 14:
            Mu2 = info.theDouble;
            return 0;
        case 15:
            du2 = info.theDouble;
            return 0;
        case 16:
            As1 = info.theDouble;
            return 0;
        case 17:
            As2 = info.theDouble;
            return 0;
        case 18:
            fu = info.theDouble;
            return 0;
        case 19:
            strain_u = info.theDouble;
            return 0;
        case 20:
            Lp = info.theDouble;
            return 0;
        default:
            return -1;
        }
}
```

Appendix 3. .Cpp file to connect editor

```
#include <stdlib.h>
#include <string.h>
#include <Domain.h>

#include "LumpedDamageEle2D2.h"

#include <CrdTransf.h>

#include <TclModelBuilder.h>

extern void printCommand(int argc, TCL_Char **argv);

int
TclModelBuilder_addLumpedDamageEle2D2(ClientData clientData, Tcl_Interp *interp, int
argc,
                            TCL_Char **argv, Domain *theTclDomain, TclModelBuilder
*theTclBuilder,
                            int eleArgStart)
{
  // ensure the destructor has not been called -
  if (theTclBuilder == 0) {
    opserr << "WARNING builder has been destroyed - LumpedDamageEle2D2 \n";
    return TCL_ERROR;
  }

  int ndm = theTclBuilder->getNDM();
  int ndf = theTclBuilder->getNDF();

  Element *theBeam = 0;

  if (ndm == 2) {
    // check plane frame problem has 3 dof per node
    if (ndf != 3) {
      opserr << "WARNING invalid ndf: " << ndf;
      opserr << ", for plane problem need 3 - LumpedDamageEle2D2 \n";
      return TCL_ERROR;
    }

    // check the number of arguments
    if ((argc-eleArgStart) < 25) {
      opserr << "WARNING bad command - want: LumpedDamageEle2D2 beamId iNode jNode E I A
qq c k0 Gcr Mu du transTag <-mass m> <-cMass>\n";
      printCommand(argc, argv);
      return TCL_ERROR;
    }

    // get the id, end nodes, and section properties
    int beamId, iNode, jNode, transTag;
    double A,E,I,qq,c,k0,Gcr,Mu,du,qq2,c2,k02,Gcr2,Mu2,du2,As1,As2,fu,strain_u,Lp;
    if (Tcl_GetInt(interp, argv[1+eleArgStart], &beamId) != TCL_OK) {
      opserr << "WARNING invalid beamId: " << argv[1+eleArgStart];
      opserr << " - LumpedDamageEle2D2 beamId iNode jNode E I A qq c k0 Gcr Mu du qq2 c2
k02 Gcr2 Mu2 du2 As fu strain_u\n";
      return TCL_ERROR;
```

```
    }
    if (Tcl_GetInt(interp, argv[2+eleArgStart], &iNode) != TCL_OK) {
      opserr << "WARNING invalid iNode - LumpedDamageEle2D2 " << beamId << " iNode jNode
E I A qq c k0 Gcr Mu du qq2 c2 k02 Gcr2 Mu2 du2 As fu strain_u\n";
      return TCL_ERROR;
    }
    if (Tcl_GetInt(interp, argv[3+eleArgStart], &jNode) != TCL_OK) {
      opserr << "WARNING invalid jNode - LumpedDamageEle2D2 " << beamId << " iNode jNode
E I A qq c k0 Gcr Mu du qq2 c2 k02 Gcr2 Mu2 du2 As fu strain_u\n";
      return TCL_ERROR;
    }

    if (Tcl_GetDouble(interp, argv[4+eleArgStart], &E) != TCL_OK) {
      opserr << "WARNING invalid E - LumpedDamageEle2D2 " << beamId << " iNode jNode E I
A qq c k0 Gcr Mu du qq2 c2 k02 Gcr2 Mu2 du2 As fu strain_u\n";
      return TCL_ERROR;
    }
    if (Tcl_GetDouble(interp, argv[5+eleArgStart], &I) != TCL_OK) {
      opserr << "WARNING invalid I - elasticBeam " << beamId << " iNode jNode E I A qq c
k0 Gcr Mu du qq2 c2 k02 Gcr2 Mu2 du2 As fu strain_u\n";
      return TCL_ERROR;
    }

    if (Tcl_GetDouble(interp, argv[6+eleArgStart], &A) != TCL_OK) {
      opserr << "WARNING invalid A - LumpedDamageEle2D2 " << beamId << " iNode jNode E I
A qq c k0 Gcr Mu du qq2 c2 k02 Gcr2 Mu2 du2 As fu strain_u\n";
      return TCL_ERROR;
    }
       if (Tcl_GetDouble(interp, argv[7+eleArgStart], &qq) != TCL_OK) {
      opserr << "WARNING invalid qq - LumpedDamageEle2D2 " << beamId << " iNode jNode E I
A qq c k0 Gcr Mu du qq2 c2 k02 Gcr2 Mu2 du2 As fu strain_u\n";
      return TCL_ERROR;
    }
       if (Tcl_GetDouble(interp, argv[8+eleArgStart], &c) != TCL_OK) {
      opserr << "WARNING invalid c - LumpedDamageEle2D2 " << beamId << " iNode jNode E I
A qq c k0 Gcr Mu du qq2 c2 k02 Gcr2 Mu2 du2 As fu strain_u\n";
      return TCL_ERROR;
    }
       if (Tcl_GetDouble(interp, argv[9+eleArgStart], &k0) != TCL_OK) {
      opserr << "WARNING invalid k0 - LumpedDamageEle2D2 " << beamId << " iNode jNode E I
A qq c k0 Gcr Mu du qq2 c2 k02 Gcr2 Mu2 du2 As fu strain_u\n";
      return TCL_ERROR;
    }
       if (Tcl_GetDouble(interp, argv[10+eleArgStart], &Gcr) != TCL_OK) {
      opserr << "WARNING invalid Gcr - LumpedDamageEle2D2 " << beamId << " iNode jNode E
I A qq c k0 Gcr Mu du qq2 c2 k02 Gcr2 Mu2 du2 As fu strain_u\n";
      return TCL_ERROR;
    }
       if (Tcl_GetDouble(interp, argv[11+eleArgStart], &Mu) != TCL_OK) {
      opserr << "WARNING invalid Mu - LumpedDamageEle2D2 " << beamId << " iNode jNode E I
A qq c k0 Gcr Mu du qq2 c2 k02 Gcr2 Mu2 du2 As fu strain_u\n";
      return TCL_ERROR;
    }
       if (Tcl_GetDouble(interp, argv[12+eleArgStart], &du) != TCL_OK) {
      opserr << "WARNING invalid du - LumpedDamageEle2D2 " << beamId << " iNode jNode E I
A qq c k0 Gcr Mu du qq2 c2 k02 Gcr2 Mu2 du2 As fu strain_u\n";
      return TCL_ERROR;
    }
```

```
        if (Tcl_GetDouble(interp, argv[13+eleArgStart], &qq2) != TCL_OK) {
       opserr << "WARNING invalid qq2 - LumpedDamageEle2D2 " << beamId << " iNode jNode E
I A qq c k0 Gcr Mu du qq2 c2 k02 Gcr2 Mu2 du2 As fu strain_u\n";
       return TCL_ERROR;
     }
        if (Tcl_GetDouble(interp, argv[14+eleArgStart], &c2) != TCL_OK) {
       opserr << "WARNING invalid c2 - LumpedDamageEle2D2 " << beamId << " iNode jNode E I
A qq c k0 Gcr Mu du qq2 c2 k02 Gcr2 Mu2 du2 As fu strain_u\n";
       return TCL_ERROR;
     }
        if (Tcl_GetDouble(interp, argv[15+eleArgStart], &k02) != TCL_OK) {
       opserr << "WARNING invalid k02 - LumpedDamageEle2D2 " << beamId << " iNode jNode E
I A qq c k0 Gcr Mu du qq2 c2 k02 Gcr2 Mu2 du2 As fu strain_u\n";
       return TCL_ERROR;
     }
        if (Tcl_GetDouble(interp, argv[16+eleArgStart], &Gcr2) != TCL_OK) {
       opserr << "WARNING invalid Gcr2 - LumpedDamageEle2D2 " << beamId << " iNode jNode E
I A qq c k0 Gcr Mu du qq2 c2 k02 Gcr2 Mu2 du2 As fu strain_u\n";
       return TCL_ERROR;
     }
        if (Tcl_GetDouble(interp, argv[17+eleArgStart], &Mu2) != TCL_OK) {
       opserr << "WARNING invalid Mu2 - LumpedDamageEle2D2 " << beamId << " iNode jNode E
I A qq c k0 Gcr Mu du qq2 c2 k02 Gcr2 Mu2 du2 As fu strain_u\n";
       return TCL_ERROR;
     }
        if (Tcl_GetDouble(interp, argv[18+eleArgStart], &du2) != TCL_OK) {
       opserr << "WARNING invalid du2 - LumpedDamageEle2D2 " << beamId << " iNode jNode E
I A qq c k0 Gcr Mu du qq2 c2 k02 Gcr2 Mu2 du2 As fu strain_u\n";
       return TCL_ERROR;
     }
        if (Tcl_GetDouble(interp, argv[19+eleArgStart], &As1) != TCL_OK) {
       opserr << "WARNING invalid du2 - LumpedDamageEle2D2 " << beamId << " iNode jNode E
I A qq c k0 Gcr Mu du qq2 c2 k02 Gcr2 Mu2 du2 As fu strain_u\n";
       return TCL_ERROR;
     }
        if (Tcl_GetDouble(interp, argv[20+eleArgStart], &As2) != TCL_OK) {
       opserr << "WARNING invalid du2 - LumpedDamageEle2D2 " << beamId << " iNode jNode E
I A qq c k0 Gcr Mu du qq2 c2 k02 Gcr2 Mu2 du2 As fu strain_u\n";
       return TCL_ERROR;
     }
        if (Tcl_GetDouble(interp, argv[21+eleArgStart], &fu) != TCL_OK) {
       opserr << "WARNING invalid du2 - LumpedDamageEle2D2 " << beamId << " iNode jNode E
I A qq c k0 Gcr Mu du qq2 c2 k02 Gcr2 Mu2 du2 As fu strain_u\n";
       return TCL_ERROR;
     }
        if (Tcl_GetDouble(interp, argv[22+eleArgStart], &strain_u) != TCL_OK) {
       opserr << "WARNING invalid du2 - LumpedDamageEle2D2 " << beamId << " iNode jNode E
I A qq c k0 Gcr Mu du qq2 c2 k02 Gcr2 Mu2 du2 As fu strain_u\n";
       return TCL_ERROR;
     }
        if (Tcl_GetDouble(interp, argv[23+eleArgStart], &Lp) != TCL_OK) {
       opserr << "WARNING invalid du2 - LumpedDamageEle2D2 " << beamId << " iNode jNode E
I A qq c k0 Gcr Mu du qq2 c2 k02 Gcr2 Mu2 du2 As fu strain_u\n";
       return TCL_ERROR;
     }
     double mass = 0.0;
     int cMass = 0;
     int argi = 0;
```

```
    if (Tcl_GetInt(interp, argv[24+eleArgStart], &transTag) != TCL_OK) {
         opserr << "WARNING invalid transTag - LumpedDamageEle2D2 " << beamId << " iNode
jNode E I A qq c k0 Gcr Mu du qq2 c2 k02 Gcr2 Mu2 du2 transTag\n";
         return TCL_ERROR;
      }


    CrdTransf *theTrans = OPS_GetCrdTransf(transTag);

    if (theTrans == 0) {
       opserr << "WARNING transformation object not found - LumpedDamageEle2D2 " <<
beamId;
       return TCL_ERROR;
    }

    while (argi < argc) {
      if (strcmp(argv[argi],"-mass") == 0) {
       if (argc < argi+2) {
         opserr << "WARNING not enough -mass args need -mass mass??\n";
         opserr << argv[1] << " element: " << beamId << endln;
         return TCL_ERROR;
       }
       if (Tcl_GetDouble(interp, argv[argi+1], &mass) != TCL_OK) {
         opserr << "WARNING invalid mass\n";
         opserr << argv[1] << " element: " << beamId << endln;
         return TCL_ERROR;
       }
       argi += 2;
      } else if ((strcmp(argv[argi],"-lMass") == 0) || (strcmp(argv[argi],"lMass") == 0))
{
       cMass = 0;  // lumped mass matrix (default)
    argi++;
      } else if ((strcmp(argv[argi],"-cMass") == 0) || (strcmp(argv[argi],"cMass") == 0))
{
       cMass = 1;  // consistent mass matrix
    argi++;
      } else
       argi++;
    }


    // now create the beam and add it to the Domain
    theBeam = new LumpedDamageEle2D2
(beamId,E,I,A,qq,c,k0,Gcr,Mu,du,qq2,c2,k02,Gcr2,Mu2,du2,As1,As2,fu,strain_u,Lp,iNode,jNod
e, *theTrans, mass, cMass);

    if (theBeam == 0) {
      opserr << "WARNING ran out of memory creating beam - LumpedDamageEle2D2 ";
      opserr << beamId << " iNode jNode E I A qq c k0 Gcr Mu du qq2 c2 k02 Gcr2 Mu2 du2
As fu strain_u\n";
      return TCL_ERROR;
    }
  }

  else {
    opserr << "WARNING LumpedDamageEle2D2 command only works when ndm is 2, ndm: ";
    opserr << ndm << endln;
```

```
    return TCL_ERROR;
  }

  // now add the beam to the domain
  if (theTclDomain->addElement(theBeam) == false) {
    opserr << "WARNING TclModelBuilder - addBeam - could not add beam to domain ";
    opserr << *theBeam;
    delete theBeam; // clean up the memory to avoid leaks
    return TCL_ERROR;
  }

  // if get here we have sucessfully created the node and added it to the domain
  return TCL_OK;
}
```