MASTAN2 Extension: Second Order Inelastic Structural Analysis

CEE 282: Nonlinear Structural Analysis

David Tse and Cesar Marco

March 18th, 2016

List of Figures

Figure 1: Typical force displacement behavior for 2nd order inelastic analysis 3
Figure 2: Regula Falsi loading Scenarios 4
Figure 3: Regula Falsi algorithm with notes 5
Figure 4: A) Cantilever Beam, B) Frame, C) Braced Frame (all fixed connections) 8
Figure 5: Load Norm and Energy Norm Plots of the A) cantilever, B) Frame and C) Frame with Brace 11
Figure 6: Force Displacement Plots of the A) cantilever, B) frame and C) braced frame 12
Figure 7: Effects of Tolerance Adjustments (Rows = element, Columns: ith and jth end) 13

List of Tables

Table 1: Abbreviated code design document (yellow = new function, green = updated from 2D2el code) 5
Table 2: Material Properties 8
Table 3: Member sizes 9
Table 4: Code generated hinge APRs and error from MASTAN outputs 9
Table 5: Code generated element forces for a typical column, beam and brace 9
Table 6: Code generated displacements and rotations for the node subject to both lateral and axial force 10
Table 7: Performance improvements (Our current code is the best out of the three analysis types) 12

I. Introduction and Overview

Performing a second order inelastic analysis on structures allows designers to better understand the true behavior of structures when applied loads are beyond the elastic capacity, as shown in Figure 1 below. Generally, this analysis incrementally applies loads to a structure and takes the geometric and plastic reduction stiffness matrices into account to cause a softening or stiffening effect on structures. Many structural analysis programs (e.g. ETABS) can perform the analysis, but to best understand how these software programs work it is important to understand the underlying backend algorithms used to calculate the results.

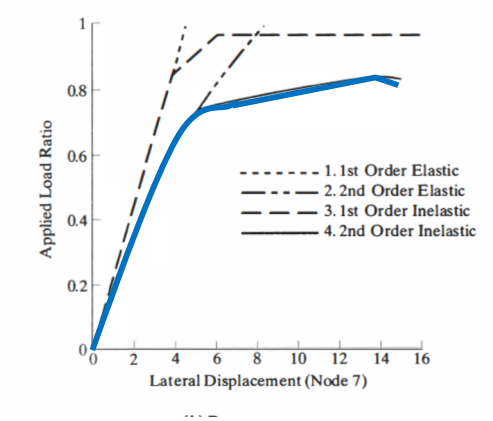


Figure 1: Typical force displacement behavior for 2nd order inelastic analysis

Our project seeks to extend the 2D 1st order elastic analysis program provided by Reagan C. to handle 2D 2nd order inelastic analysis. We implemented the Regula Falsi algorithm to return element forces below the yield surface because it generates results that are relatively more accurate than the constant P return algorithm. Also, because geometric nonlinearities are considered, we are using the natural deformation approach to calculate internal element forces. To validate our results, we will compare MASTAN2 outputs with our code results in terms of the deflection, element forces, hinge locations and applied load ratios for some simple structures. The focus of

the report will be on how to incorporate material nonlinearity given that geometric nonlinearity was investigated in the programming assignment.

II. Theoretical Background[1]

The Regula Falsi algorithm, or false position method, scales load increments so that they do not breach either the yield surface or the maximum tolerable yield surface, as shown in Figure 2. Scaling the load increment is important because otherwise the analysis would be elastic due to a lack of a yield surface bound. The focus of our project is loading, as opposed to unloading.

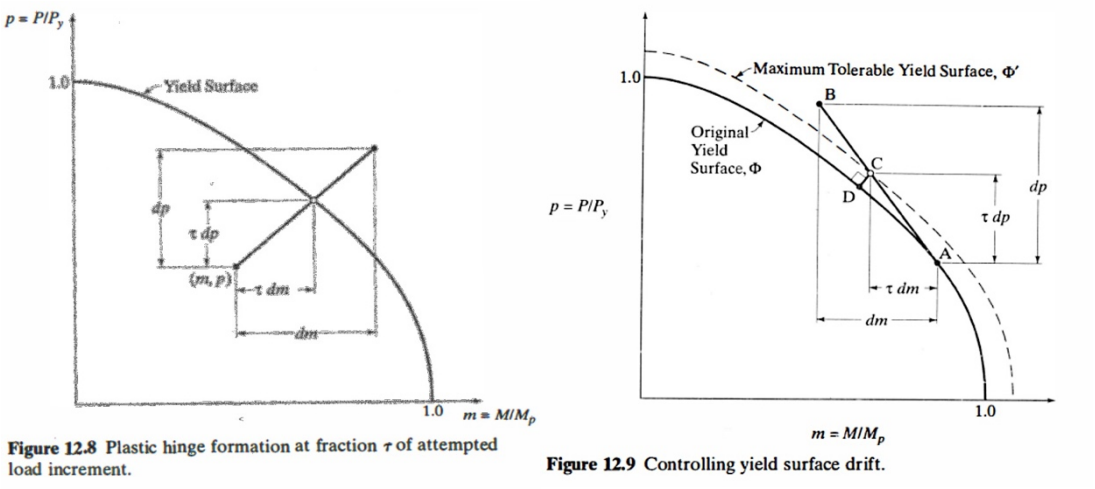


Figure 2: Regula Falsi loading Scenarios

The equations and general procedures of the Regula Falsi algorithm are detailed in Figure 3. It is important to note that the algorithm could be applied to both loading cases by simply adjusting the limit (i.e. either 1.0 or the max. tolerable yield surface) to scale τ_{regula}. Also, we are assuming hinges occur one at a time since we are performing an event-to-event analysis.

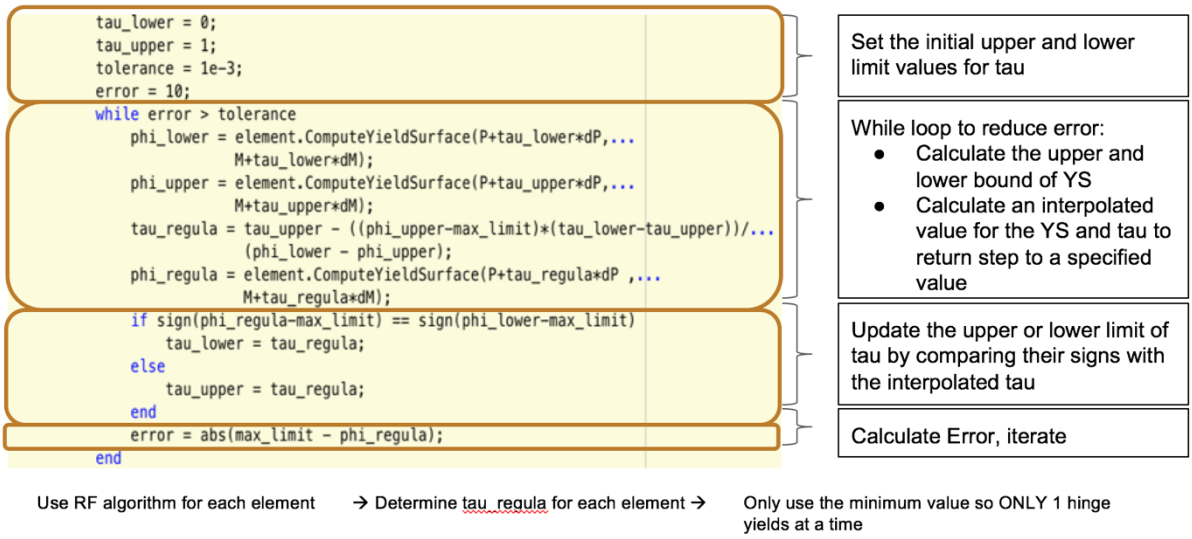


Figure 3: Regula Falsi algorithm with notes

The yield surface equation (Equation 1) we implement is specific for W-sections, which will facilitate code testing efforts because MASTAN uses the same function in its analysis.

$$1 = (P/P_y)^2 + (M/M_p)^2 + 3.5(P/P_y)^2(M/M_p)^2 \qquad \text{(Equation 1)}$$

III. Code Design

Developing an efficient 2nd order inelastic code requires inheritance from 1st order elastic analysis, overwriting functions as necessary and creating some new functions. A generalized code design document is shown below for brevity:

Table 1: Abbreviated code design document (yellow = new function, green = updated from 2D2el code)

Class	Function	Purpose
Analysis	Constructor	Creates an instance of the analysis class
	RunAnalysis	Runs a Single step incremental analysis
	Calc_Tau_min	Uses the Regula Falsi method to determine a minimum tau value for case 1 of Figure 2
	RadialReturn	Uses the regula falsi algorithm to return PM Locations that have already yielded and are straying away from the YS of 1.0. This is used for Case 2 of Figure 2

	PlotNorms	Plots the load norm and error norm indices
	GetMastan2Returns	Returns results to MASTAN2
	CreateNodes	Creates the nodes based on the 2D2in class
	CreateElements	Creates the elements based on the 2D2in class
	ComputeDisplacementsReactions	Computes the displacements and reactions
	RecoverElementForces	Recover forces based on the natural deformation approach
	UpdateGeometry	Updates the geometry of structure
	ComputeResultant	Computes the resultant to be used in the error calculation
	ComputeError	E = P- R
	ComputeNorms	Computes the load norm and error norm indices
	CheckKffMatrix	Checks the conditioning an limit states
	InitializeOutputVariables	Initializes variables, which will be updated in the run analysis function
	ComputeReturn	Implements the regular falsi algorithm
	CreateStiffnessMatrix	Creates the stiffness matrix by considering the computation of gradients and YS
Element	Constructor	Creates an instance of the element class
	ComputeGlobalStiffnessMatrix	Computes the geometric stiffness matrix in global coordinates
	GetKGlobal	Getter functions
	GetdFlocal	
	GetTransformationMatrix	
	UpdateTransformationMatrix	Updates the transformation matrix based on new geometry
	UpdateFLocal	Updates the element forces
	ComputeForces	Computes the forces based on the natural deformation approach
	ComputeLocalGeometricStiffnessMatrix	Computes the local geometric stiffness matrix

	ComputeYieldStrength	Computes the yield strength
	ComputePlasticMoment	Computes the plastic moment
	ComputeYieldSurface	Computes the yield surface of the element
	ComputeGradient	Computes the gradient based on the hinge formations
	ComputeKPlastic	Computes the plastic reduction matrix
Nodes	Constructor	Creates an instance of the node class
	UpdateCoordinates	Updates the coordinates of the node in terms of x an y

Using the code design document, we then prepared some general procedures to call functions in the analysis and element classes as necessary. The general procedures that were carried out are shown below:

1) Asemble $[Kt]_{i-1} = [K_e] + [K_p] + [K_g]$
2) Determine the Incremental Load Vector: $\{dP\}_i = \lambda_i\{P\}$
3) Determine the Incremental Displacements: $\{d\Delta\}_i = [Kt_{i-1}]^{-1}\ \{dP\}_I$
4) Recover Member Forces based on Natural Deformation approach:

   $\{dF'\}_i = [k't]_{i-1}]^{-1}\ \{d\Delta'\}_i$

   $\{F'\}_i = \{F'\}_{i-1} + \{dF'\}_i$
5) Check Loading Events (Event-to-Event) if:
   - Elastic members reaching yield condition (YS > 0)?

   Yes – scale back load step τ for the first member to yield
   - Previously yield members loading (YS > 0)?

   Yes – scale back load step τ to maintain drift control
6) Recover Element Forces and incremental displacements if τ_{min} is <1 with τ_{min}*dP as load increment
7) Calculate Resultants and Error
   - Calculate the nodal resultants: $\{R\}_i = \sum \{F\}_{n,i}$
   - Calculate the error: $\{E\}_i = \{P\}_i - \{R\}_i$

8) Next Step

IV. Results

IV.A. Test Frames

The frames that we used for testing our code are shown in Figure 4. We started with a cantilever because it is the simplest structure to test with the formation of one hinge. Thereafter, we tested our code using a frame, where multiple hinges could form and beams and columns are present. Lastly, we tested a braced frame because these types of frames are known for having hinges that occur very close to each other, which would test our code's ability to handle tighter tolerances.

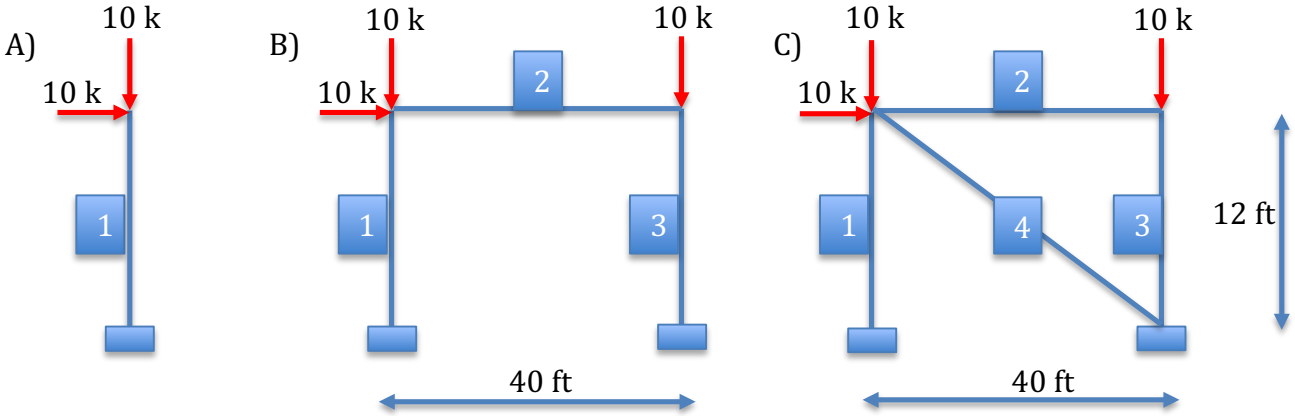


Figure 4: A) Cantilever Beam, B) Frame, C) Braced Frame (all fixed connections)

The member sizes were specifically chosen to allow for hinges in locations that would reduce the number of steps required for debugging code and cause the structure to hinge at desired locations where we could examine multiple areas of the structure. The member properties and size summary is shown below in Table 2 and 3:

Table 2: Material Properties

Member	Fy (ksi)	E (ksi)	Mp (k-in)
W30x99	50	29,000	15,600
W6x15			540

Table 3: Member sizes

Element #	Cantilever Beam	Frame	Braced Frame
1	W30x99	W30x99	W6x15
2	-		
3	-		
4	-	-	

IV.B. Hinge Applied Load Ratios (APRs)

Table 4: Code generated hinge APRs and error from MASTAN outputs

Element	Cantilever				Frame				Braced Frame			
	i-th End	% Error	j-th End	% Error	i-th End	% Error	j-th End	% Error	i-th End	% Error	j-th End	% Error
1	10.6475	0.02	0	0	28.195	0.039	37.0285	0.069	0	0	0	0
2	-	-	-	-	36.7402	0.074	29.2651	0.0345	0	0	0	0
3	-	-	-	-	0	0	0	0	0	0	20.1084	0.05
4	-	-	-	-	-	-	-	-	0	0	21.1084	0.05

IV.C. Element Forces

Table 5: Code generated element forces for a typical column, beam and brace

	Typical Column				Typical Beam				Typical Brace			
	P (kips)	V_y (kips)	M_z (k-in)	Error range (%)	P (k)	V_y (k)	M_z(k-in)	Error Range	P (k)	V_y (k)	M_z(k-in)	Error Range (%)
Cantilever	104.58	107.093	15421	0 to 1.16	-	-	-	-	-	-	-	-
Frame	298.6	198	142 17	0.47 to 3.83	177.2	-57.04	-14240	0.073 to 0.64	-	-	-	-
Braced Frame	147.37	1.23	121.147	0.02 to 0.62	-0.1724	-0.2551	-63.5509	1.53 to 7.48	-220.1518	0.0534	-34.466	0.11 to 5.88

IV.D. Displacements and Rotations

Table 6: Code generated displacements and rotations for the node subject to both lateral and axial force

	Δx (in)	% Error	Δy (in)	% Error	Θz (rad)	% Error
Cantilever	0.9216	0.03	-0.02080	0	-0.009602	0.31
Frame	1.9769	0.97	-0.1479	0.2	-0.01077	0.94
Braced Frame	0.8432	0.1	-0.167	0.1	-0.0066	0.8

IV.E. Error Plots

Error plots are based on the load norm index and the energy norm index. We implement equations 2 and 3 and concatenated the values into a vector with each iteration to develop error plots. The error plots are shown in Figure 5.

$$\text{Load Norm Index} = \frac{\text{norm}\{\text{E}\}}{\text{norm}\,\{\text{dP}\}} \qquad \text{(Equation 2)}$$

$$\text{Energy Norm Index} = \frac{|\{\text{E}\}|^T|\{d\Delta\}|}{|\{\text{P}\}|^T|\{d\Delta\}|} \qquad \text{(Equation 3)}$$

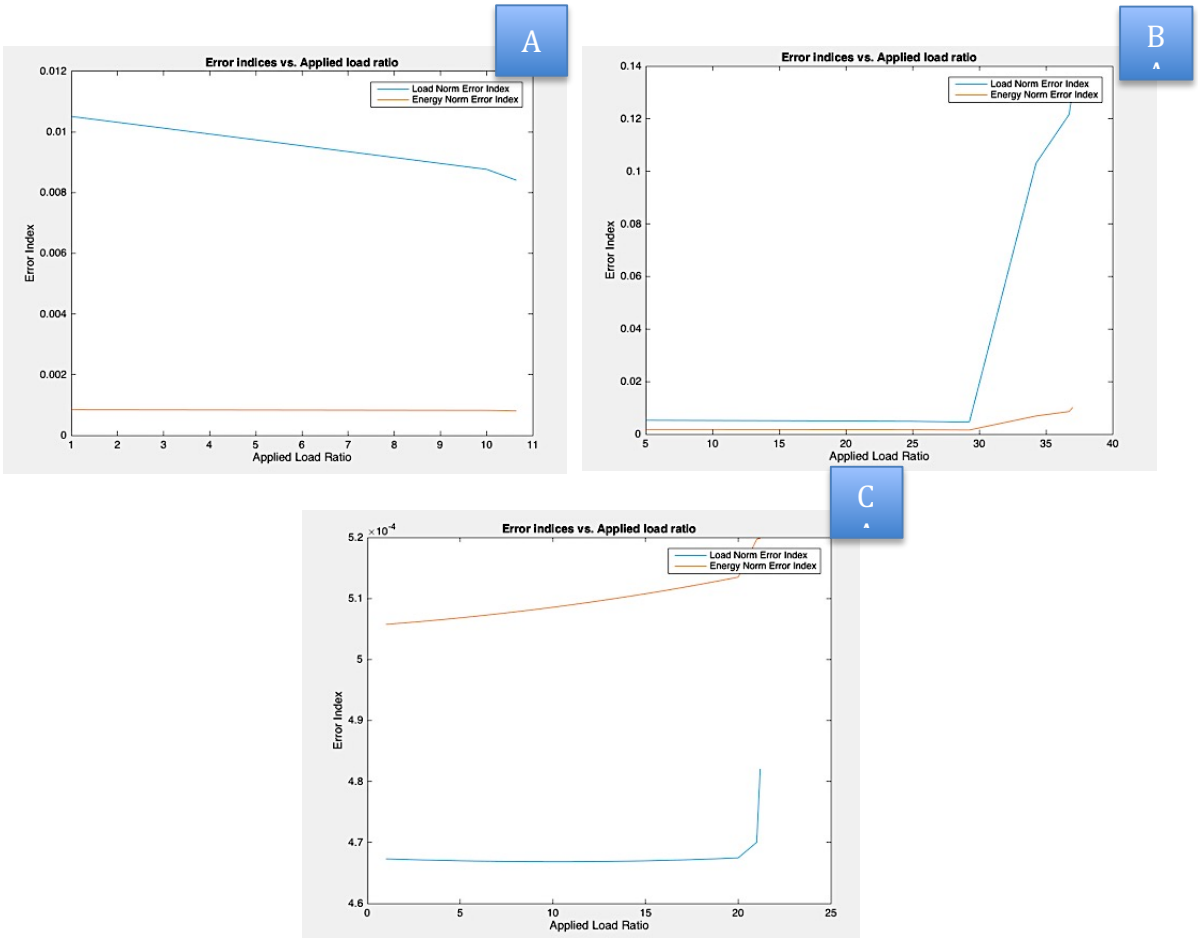


Figure 5: Load Norm and Energy Norm Plots of the A) cantilever, B) Frame and C) Frame with Brace

IV.F. Force-Displacement Plots

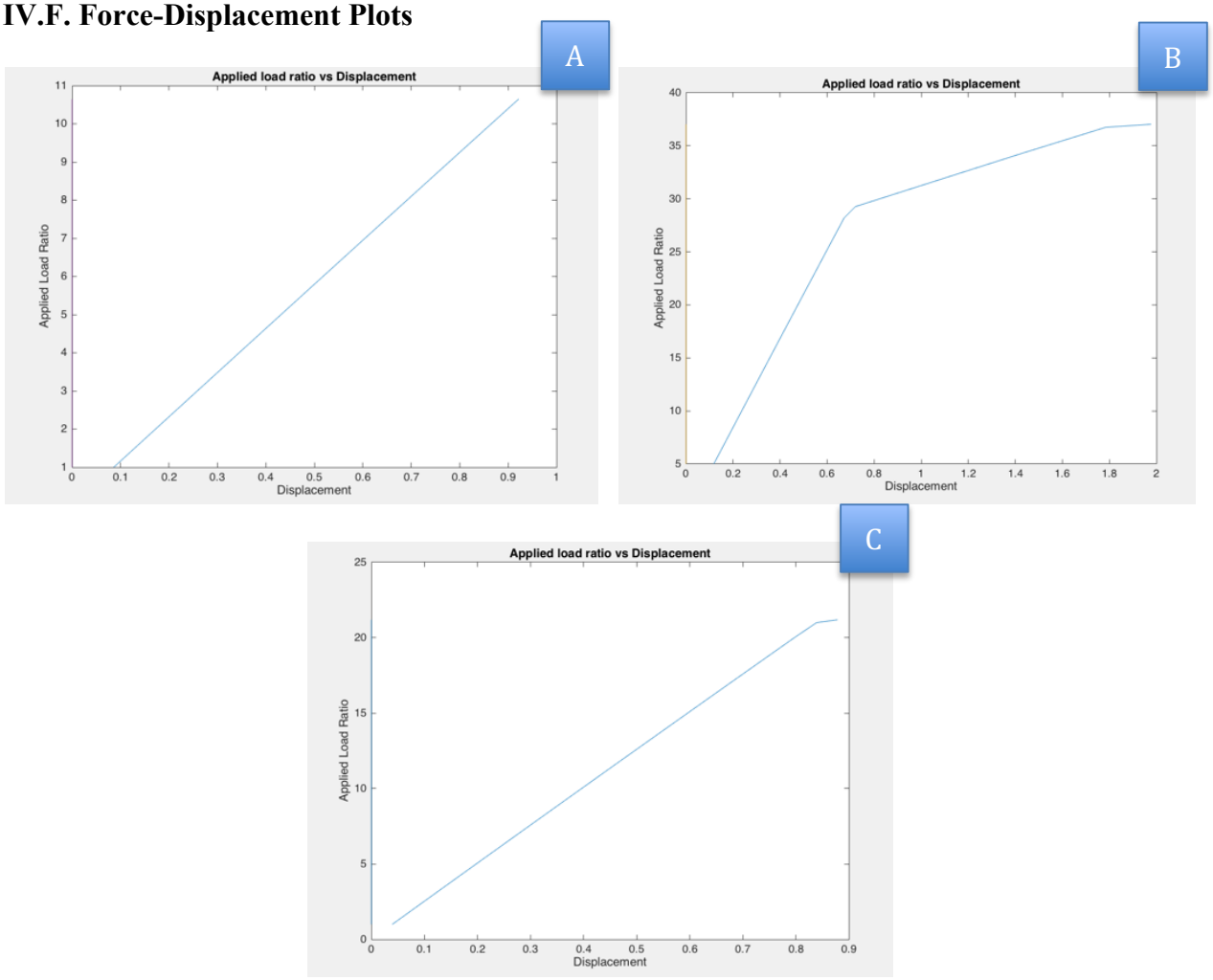


Figure 6: Force Displacement Plots of the A) cantilever, B) frame and C) braced frame

IV.G. Performance of Code

Table 7: Performance improvements (Our current code is the best out of the three analysis types)

Analysis Type	**Tic/Toc Time (s)**
2D 2nd-Order elastic analysis	3.95
2D 1st-Order inelastic analysis	19.27
2D 2nd-Order inelastic analysis	0.528

V. Discussion

V.A. Errors in element forces, displacements and hinge applied load ratios

When determining if hinges were forming, we learned that setting a reasonable tolerance is important to recognize whether the hinges were yielding or unyielding. If the tolerance was set too low, then hinges may “unyield” in a subsequent step because the result from the yield surface equation might be slightly below one minus the tolerance. For example, when decreasing the tolerance from 0.03 to 0.01 in our analysis of the frame without the diagonal brace, the program thinks that one of the hinges “unyielded” from the ninth step in the analysis to the tenth step, as seen in Figure 7. This incorrect detection of unyielding would lead to the determination of incorrect displacements and element forces for the structure in subsequent steps of the analysis because of an incorrect construction of the plastic reduction matrix for the element that is "unyielding".

```
ELE_YLD(:,:,9) =

   28.1950         0
   36.7402   29.2651
         0         0


ELE_YLD(:,:,10) =

   28.1950   37.0285
         0   29.2651
         0         0
```

Figure 7: Effects of Tolerance Adjustments (Rows = element, Columns: ith and jth end)

However, if the tolerance was set too high, then our program would not accurately determine the value at which the hinges on the structure formed. Once again, the displacements and element forces determined from our structure would be incorrect as the correct plastic reduction matrix would not be used due to the incorrect determination of the formation of hinges on the structure.

We use the natural deformation approach for element force recovery and doing so leads to an accumulation of error. MASTAN2 makes use of the transformation approach which means that there would be differences in the forces that were determined in our program from those determined in MASTAN2. Our results seem to indicate that this is especially prevalent in the braced frame, where the error is the highest. As the analysis continues, we believe that the errors accumulate and grow larger in each subsequent step.

These differences in the forces recovered, due to different force recovery methods, do not seem to have much of an impact on the determination of the displacements of the structures as the largest error in the displacement between MASTAN2's results and our own results was only 0.97%, or less than 1%.

V.B. Error Plots

The error plots seem to increase in error if hinges occur in the structure. If multiple hinges occur at an APR that is somewhat distant from one another, then there will generally be an exponential trend as seen in the frame without a diagonal brace because four hinges form. However, for the cantilever and the braced frame, the hinges occur later, so there is a large delta in error only towards the end of the analysis. These trends make sense because the hinging is changing the resultant, which is more error-prone based on the reasons explained in section V.A.

The magnitude of the error is the greatest in the frame without a diagonal brace, while the smallest errors is for the braced frame. This may once again be attributable to the number of hinges that form and the relative closeness of the hinge APRs. The braced frame only formed two hinges with an APR difference of about 1.0 between the two hinges, while the unbraced frame experienced hinging four times with an APR difference of approximately 8.8. To reduce these errors, a Newton-Raphson algorithm could probably be used to iterate out the error.

V.C. Force-displacement Plots

In the force-displacement plots for the cantilever and the frame *with* a diagonal brace, we observe a mostly linear behavior between Applied Load and Displacement since the hinges in both of these structures form close to the end of the analysis and thus inelastic analysis is observed only after these hinges form. We suspect that the braced frame would be a higher performing structure if hinges were to form distributed throughout the brace and beam. Adjusting the section properties to comply with the strong-column/weak beam principle would likely lead to better performance, thus avoiding sudden collapse.

As for the Force-Displacement plot for the frame *without* a diagonal brace, the plot exhibits more inelastic behavior and mimics the graph presented in the introduction for 2^{nd}-order inelastic behavior since the hinges form throughout the analysis.

VI. Conclusion

The original motivation for this project was to investigate the underlying algorithms that structural analysis programs use to implement a 2^{nd} order inelastic analysis. We aimed to not only implement this analysis, but also to obtain results that are low in error for a variety of simple structures to demonstrate a high level of robustness in our code and reduce run-time via more organization in our code.

The Regula Falsi algorithm and natural deformation method provided the basis for our analysis program to capture the effects of material and geometric nonlinearities. Following procedures for a simple step incremental approach resulted in outputs that match closely to those obtained from MASTAN. Along the way, the main issues include: 1) tolerance optimization and 2) code compartmentalizing, given the high number of functions we dealt with. Overall, the project

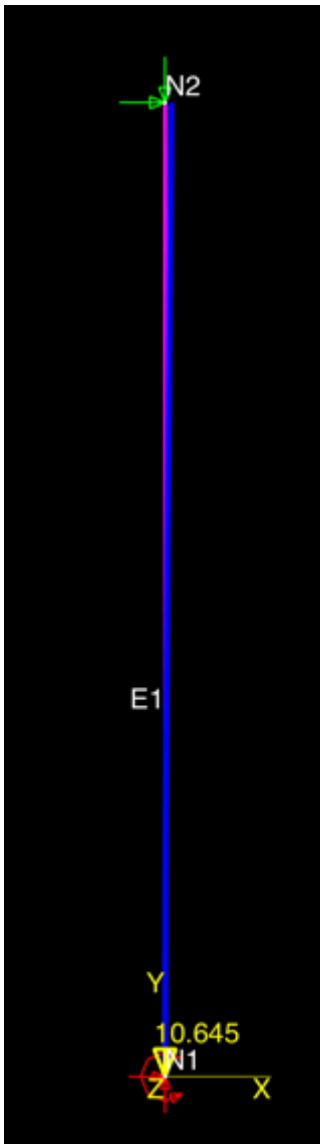
has given us a deeper appreciation into the trials and tribulations of debugging and algorithmic thinking as it pertains to 2nd order inelastic analysis.

VII. Acknowledgements

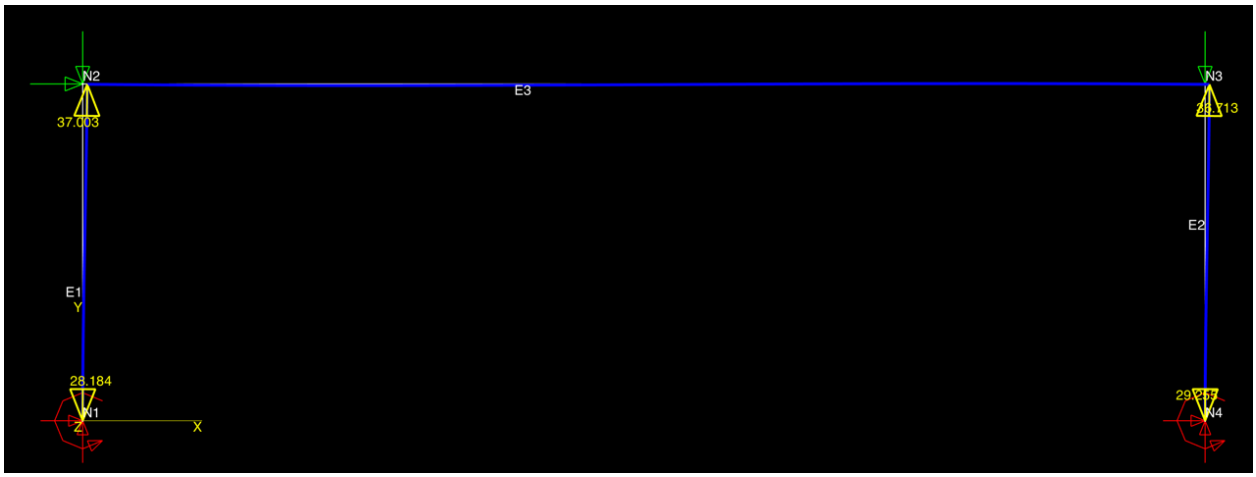
The team would like to thank Professor Deierlein for providing direction through the latter half of the quarter when extensive issues arose.

VIII. Appendix

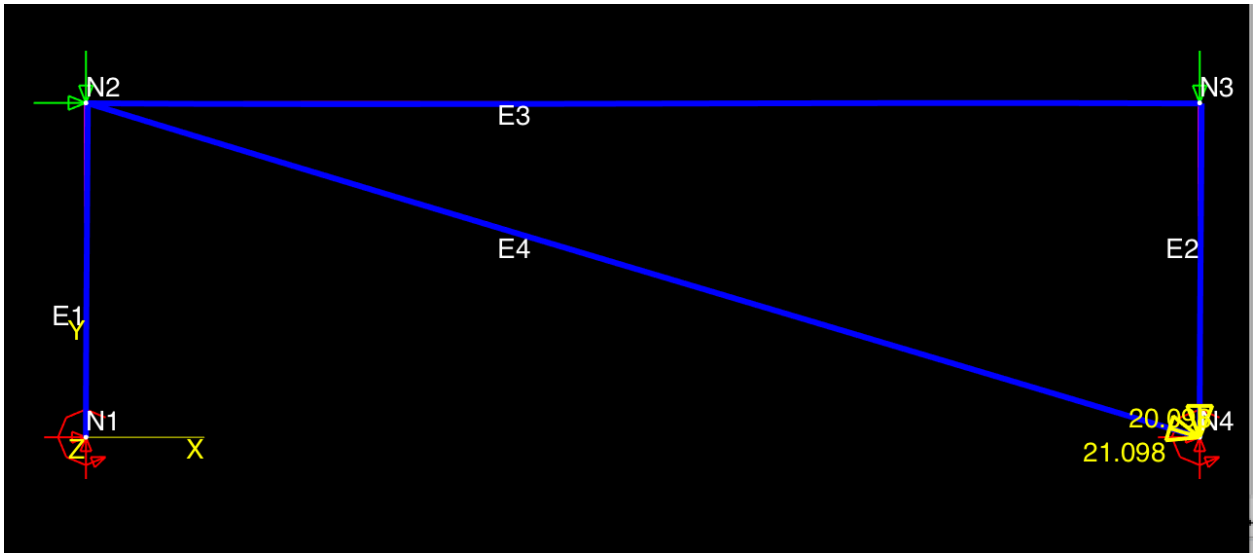
- MASTAN2 2nd order analysis outputs:
    - Cantilevered beam



    - Frame without Brace



- Frame with Brace



ANALYSIS CLASS

```
% Cesar Y. Marco & David Tse (CMDT)
% CEE 282: Programming Project
% 2d2in Analysis Class
% 03/18/16

classdef CMDT_Analysis_2d2in < RC_Analysis_2d1el
% This is the child class of RC_Analysis_2d1el.m.
% All the protected & public data properties of parent class will be
% inherited to this class

% Analysis class for 2nd order analysis of a 2-dimensional structure
        properties (Access = public)
%         Miscellaneous
        Apratio
        tau_min
```

```
%         Tolerances
        tau_min_tol
        radtolerance
        tolerance
%         % Resultant: Free DOF
        R_free
%
%         % New MASTAN2 returns
        APRATIOS
        LIMIT_STATE
%
%         % Incremental version of MASTAN outputs
        DEFLstep
        REACTstep
        ELE_FORstep
        Kffstep
        ELE_YLD
        ELE_YLDstep
%         % Used for calculating error or Norm indices
        Error
        Load_Norm
        Energy_Norm

%         % Ratio required is often used, so create a protected property
        ratio_req

    end
    %% Public methods
    methods (Access = public)
        % Constructor inherits properties from the 2d1el analysis class
        % Add 2d2el specific object parameters, including:
        % 1) numsteps
        % 2) ratio_req
        % 3) stop_ratio
        % 4) restart
        function self = CMDT_Analysis_2d2in(nnodes, coord, fixity, concen,
...
                        nele, ends, A, Ayy, Izz, E, v, truss, numsteps, ...
                        ratio_req, stop_ratio,Fy,Zzz)
            self = self@RC_Analysis_2d1el(nnodes, coord, fixity, concen, ...
                nele, ends, A, Ayy, Izz, E, v, truss,Fy,Zzz);

            % Considers whether yielding has occurred
            self.tolerance = 0.01;
            % Consider with error in the regula falsi
            self.tau_min_tol = 1e-10;
            % Consider for radial return
            self.radtolerance = 0.03;

            self.ratio_req = ratio_req;
            self.Apratio = 0;
            % Initialize output variables and other necessary variables
            self.InitializeOutputVariables(); % CMDT_Analysis_2d2el
            self.CreateLoadVectors(); % RC_Analysis_2d1el

            %Computes initial stiffness matrix with kg = 0 and ke = #
            self.CreateStiffnessMatrix(1) % RC_Analysis_2d1el
```

```
            % Computes DEFLstep and REACTstep
            self.ComputeDisplacementsReactions(self.Pf*ratio_req);%
CMDT_Analysis_2d2el

            % Run the incremental single step analysis with no error
            % iterations
            self.RunAnalysis(numsteps,stop_ratio,ratio_req); %
CMDT_Analysis_2d2el
        end

        %% Run the single step incremental analysis with no error iterations
        function RunAnalysis(self,numsteps,stop_ratio,ratio_req)
            % Run a for loop to start at 1 & end at a user-specified point
            % This for loop the minimum of the number of steps and stop
            % ratio divided by the ratio req.
%             endpoint = (min(numsteps,stop_ratio/ratio_req));
            i = 1;
            while i <= numsteps && self.Apratio <= stop_ratio
                % for each element calculate the local geometric stiffness
                % matrix to be added with ke during each load step
                for j = 1:self.nele
                    self.elements(j).ComputeLocalGeometricStiffnessMatrix()
                    % CMDT_Element_2d2el
                end
%
                % Recreate the stiffness matrix based on the new
                % geometry, which would affect only the kg and the
                % transformation matrix. The local ke would not change
                % because this value is only calculated once
                self.CreateStiffnessMatrix(i)

                % Run the analysis only if the Kff matri is
                % well-conditioned and K is positive definite according to
                % the results from cholesky decomposition
                if (self.AFLAG == 1) && (self.LIMIT_STATE == 0)
                    % Calculate the global DEFL and REACT to be used for
                    % obtaining the element forces via natural deformation
                    self.ComputeDisplacementsReactions(self.Pf*ratio_req); %
RC_Analysis_2d1el
                    % Calculate the theta_an,theta_bn and un used to
                    % calculate the incremental element forces
                    self.RecoverElementForces(i); % CMDT_Analysis_2d1in
                    %
                    %
                    %
                    %% RETURN METHOD
                    self.tau_min = 1;
                    for elenum = 1:self.nele

                        % Obtain the incremental element force to determine
                        % the P-M values as well as the incremental dP and
                        % dM values
                        dF = self.elements(elenum).GetdFlocal();
                        End_i_dP = dF(1);
                        End_i_dM = dF(3);
                        End_j_dP = dF(4);
```

```
                        End_j_dM = dF(6);
                        if i > 1
                            F = self.ELE_FOR(elenum,:,i-1);
                            End_i_P = F(1);
                            End_i_M = F(3);
                            End_j_P = F(4);
                            End_j_M = F(6);
                        else
                            End_i_P = 0;
                            End_i_M = 0;
                            End_j_P = 0;
                            End_j_M = 0;
                        end
%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%
% i-th end calculations when the PM location is below the YS and is going to
% breach the YS
                        % Compute the prior step and post step to determine
                        % which case the step falls in
                        Yield_Surface_End_i_prior_step = ...

self.elements(elenum).ComputeYieldSurface(End_i_P,End_i_M);
                        Yield_Surface_End_i_post_step = ...
                            self.elements(elenum).ComputeYieldSurface(...
                            End_i_P + End_i_dP,...
                            End_i_M + End_i_dM);
                        self.Calc_Tau_min
(1,Yield_Surface_End_i_prior_step,...
                            Yield_Surface_End_i_post_step, ...
                            self.tolerance, 1.0 + self.tolerance,...
                            End_i_P, End_i_M, End_i_dP, End_i_dM, elenum, i);
%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%
% j-th end calculations when the PM location is below the YS and is going to
% breach the YS
                        Yield_Surface_End_j_prior_step = ...

self.elements(elenum).ComputeYieldSurface(End_j_P,End_j_M);
                        Yield_Surface_End_j_post_step = ...
                            self.elements(elenum).ComputeYieldSurface(...
                            End_j_P + End_j_dP,...
                            End_j_M + End_j_dM);
                        self.Calc_Tau_min
(2,Yield_Surface_End_j_prior_step,...
                            Yield_Surface_End_j_post_step,...
                            self.tolerance, 1.0 + self.tolerance, End_j_P,
...
                                End_j_M, End_j_dP, End_j_dM, elenum, i);
                    end
                    %Update the Applied load ratio
                    if i > 1
                        if self.APRATIOS(i-1)+self.ratio_req > ...
                                self.APRATIOS(i-
1)+self.tau_min*self.ratio_req
                           self.APRATIOS(i)=self.APRATIOS(i-
1)+self.tau_min*self.ratio_req;
```

```
                        else
                            self.APRATIOS(i)=self.APRATIOS(i-
1)+self.ratio_req;
                        end
                    else
                        self.APRATIOS(i)=self.ratio_req;
                    end
                    self.Apratio = self.APRATIOS(i);
                    % algorithm to format what hinges yield
                     ele_yield_step = find(self.ELE_YLDstep >
self.APRATIOS(i));
                    self.ELE_YLDstep(ele_yield_step) = 0;
                    % Recalculate the displacement, reactions, and element
                    % forces by including the tau_min coefficient
                    % calculated earlier
                    if self.tau_min < 1
                        self.ComputeDisplacementsReactions(self.Pf *
ratio_req * self.tau_min);
                        self.RecoverElementForces(i);
                    end
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% LOADING AN ELEMENT ALREADY ON THE YS TO ANOTHER POINT ON THE YS
                    for elenum = 1:self.nele
                        dF = self.elements(elenum).GetdFlocal();
                        End_i_dP = dF(1);
                        End_i_dM = dF(3);
                        End_j_dP = dF(4);
                        End_j_dM = dF(6);

                        if i > 1
                            F = self.ELE_FOR(elenum,:,i-1);
                            End_i_P = F(1);
                            End_i_M = F(3);
                            End_j_P = F(4);
                            End_j_M = F(6);
                        else
                            End_i_P = 0;
                            End_i_M = 0;
                            End_j_P = 0;
                            End_j_M = 0;
                        end
                        Yield_Surface_End_i_post_step = ...
                            self.elements(elenum).ComputeYieldSurface(...
                            End_i_P + End_i_dP,...
                            End_i_M + End_i_dM);
                        Yield_Surface_End_j_post_step = ...
                            self.elements(elenum).ComputeYieldSurface(...
                            End_j_P + End_j_dP,...
                            End_j_M + End_j_dM);
                        if Yield_Surface_End_i_post_step > (1 +
self.radtolerance)
                            self.RadialReturn(End_i_P, End_i_M, End_i_dP, ...
                                End_i_dM, elenum, [1;3]);
                        end
                        if Yield_Surface_End_j_post_step > (1 +
self.radtolerance)
                            self.RadialReturn(End_j_P, End_j_M, End_j_dP, ...
                                End_j_dM, elenum, [4;6]);
```

```
                    end
                end
            % for steps 2 onwards, take a cumulative sum approach
            % for calculating the global deflections and reactions
            if(i > 1)
                self.DEFL(:,:,i) = cat(3,self.DEFLstep+self.DEFL(:,:,i-
1));
                self.REACT(:,:,i) =
cat(3,self.REACTstep+self.REACT(:,:,i-1));
                self.ELE_FOR(:,:,i) = cat(3,self.ELE_FORstep);
                self.ELE_YLD(:,:,i) = cat(3,self.ELE_YLDstep);
                % for step 1 just add the increment to the initial zero
                % 3D matrix
            else
                self.ELE_YLD(:,:,1) = self.ELE_YLD(:,:,1) +
self.ELE_YLDstep;
                self.DEFL(:,:,1) =  self.DEFL(:,:,1) + self.DEFLstep;
                self.REACT(:,:,1) = self.REACT(:,:,1) + self.REACTstep;
                self.ELE_FOR(:,:,1) = self.ELE_FOR(:,:,1) +
self.ELE_FORstep;
            end

            % Update the node coordinates
            self.UpdateGeometry(); % CMDT_Analysis_2d2el

            % Computations used for error and norm
            self.ComputeResultant(); % CMDT_Analysis_2d2el
            self.ComputeError(i); % CMDT_Analysis_2d2el
            self.ComputeNorms(i); % CMDT_Analysis_2d2el
            i = i + 1;

                end
            if self.LIMIT_STATE == 1
                % break the for loop if the above if condition is not
                % met, this means that either a limit is reached or the
                % stiffness matrix is ill-conditioned
                break
            end
            end
        end
        %% Calulate the tau_min used to scale increments that breach the
yield surface
        function Calc_Tau_min (self,hinge,YSprestep,YSpoststep, YStolerance,
...
                max_tolerance, P, M, dP, dM, elementnum, stepnum)
% hinge:
    % 1 = ith-end
    % 2 = jth-end
% YSprestep: Prior step PM location of the Yield Surface
% YSpoststep: Post step PM location of the Yield Surface
% YStolerance: tolerance for the YS of 1.0
% max_tolerance: max tolerable YS
% P: axial force
% M: Moment
% dP: incremental axial force
% dM: incremental moment
% elenum: element number
% stepnum: step number
```

```
% Case A: When the YS post step has not breached the YS
                if YSpoststep < (1 - YStolerance)
                    self.ELE_YLDstep(elementnum,hinge) = 0;
% Case B: When the YS poststep is between 1.0-YS_tolerance and 1.0
                elseif YSpoststep >= (1 - YStolerance) && YSpoststep <= 1
                    if self.ELE_YLDstep(elementnum,hinge) == 0;
                      self.ELE_YLDstep(elementnum,hinge) =
self.APRATIOS(stepnum-1)...
                          + self.ratio_req;
                    end
% Case C: When the YS poststep is between 1.0 and 1.0+YS_tolerance
                elseif YSpoststep > 1  && YSpoststep <= (1 + YStolerance)
                    if YSprestep < (1 - YStolerance) &&
self.ELE_YLDstep(elementnum,hinge)==0;
                       tau_update = self.ComputeReturn(dP,dM,...
                        P,M,self.elements(elementnum),1.0);
                      self.ELE_YLDstep(elementnum,hinge) =
self.APRATIOS(stepnum-1)...
                          + tau_update*self.ratio_req;
                       if tau_update < self.tau_min
                        self.tau_min = tau_update;
                       end
                    end
% Case D: When the YS poststep is between 1.0+YS_tolerance and the max
% specified tolerance
                elseif YSpoststep > (1 + YStolerance) && YSpoststep <=
max_tolerance
                    if YSprestep < (1 - YStolerance) &&
self.ELE_YLDstep(elementnum,hinge)==0;
                       tau_update = self.ComputeReturn(dP,dM,...
                        P,M,self.elements(elementnum),1.0);
                       if self.ELE_YLDstep(elementnum,hinge) == 0
                           self.ELE_YLDstep(elementnum,hinge) =
self.APRATIOS(stepnum-1)...
                                + tau_update*self.ratio_req;
                       end
                       if tau_update < self.tau_min
                       self.tau_min = tau_update;
                       end
                    end

% Case E: When the YS poststep is lower than the max_tolerance specified
                elseif YSpoststep >= max_tolerance
                    if YSprestep >= (1 - YStolerance)
                    tau_update = self.ComputeReturn(dP,dM,...
                        P,M,self.elements(elementnum),max_tolerance);
                    elseif YSprestep < (1 - YStolerance)
                        tau_update = self.ComputeReturn(dP,dM,...
                         P,M,self.elements(elementnum),1.0);
                        if self.ELE_YLDstep(elementnum,hinge) == 0
                            self.ELE_YLDstep(elementnum,hinge) =
self.APRATIOS(stepnum-1)...
                            + tau_update*self.ratio_req;
                        end
                    end
                    if tau_update < self.tau_min
```

```
                        self.tau_min = tau_update;
                    end
                end
            end
            %% Radial return method for when a hinge is already yielding and
strays away from the YS
            function RadialReturn(self, P, M, dP, dM, elementnum, DOF)
% P: axial force
% M: moment
% dP: incremental axial force
% dM: incremental moment
% elenum: element number
% DOF: Degrees of freedom
                tau_update = self.ComputeReturn(0 - (P+dP), 0-(M+dM), P+dP , M+dM
, ...
                    self.elements(elementnum) , 1.0);
                self.ELE_FORstep(elementnum , DOF(1)) = P+tau_update * (0-
(P+dP));
                self.ELE_FORstep(elementnum , DOF(2)) = M+tau_update * (0-
(M+dM));
            end

            %% Plots the load norm indices & energy norm indices with the Applied
Load Ratio
            function PlotNorms(self)
                % x-axis: Applied load ratio

                % y-axis: energy or load norm index values

plot(self.APRATIOS,self.Load_Norm,self.APRATIOS,self.Energy_Norm);

                % Plotting modifications
                xlabel('Applied Load Ratio');
                ylabel('Error Index');
                title('Error indices vs. Applied load ratio');
                legend('Load Norm Error Index','Energy Norm Error Index');
            end
            %% Get Mastan2 Returns
            %  Returns the matrices that need to be returned to Mastan2
            function [DEFL, REACT, ELE_FOR, AFLAG, APRATIOS, LIMIT_STATE,ELE_YLD]
= ...
                      GetMastan2Returns(self)
                DEFL = self.DEFL;
                REACT = self.REACT;
                ELE_FOR = self.ELE_FOR;
                AFLAG = self.AFLAG;
                APRATIOS = self.APRATIOS';
                LIMIT_STATE = self.LIMIT_STATE;
                ELE_YLD = self.ELE_YLD;
            end
        end
        %% Protected Methods
        methods (Access = protected)
            %% Create Nodes
            % Create the nnodes x 1 vector of 2d2el node objects representing all
            % the nodes in the structure
            function CreateNodes(self)
                for i = 1:self.nnodes
```

```
                self.nodes = [self.nodes; CMDT_Node_2d2in(i,
self.coord_t(:,i))];
            end
        end
        %% Create Elements
        % Create the nele x 1 vector of 2d2el element objects representing
all
        % the elements in the structure
        % Arguments are all matrices received from Mastan2. Refer to
        % comments in ud_2d1el.m for details.
        function CreateElements(self, A, Ayy, Izz, E, v, Fy, Zzz)
            for i = 1:self.nele
                % Create an Element object and append it to the "elements"
vector
                self.elements = [self.elements; CMDT_Element_2d2in(...
                    self.nodes(self.ends(i, 1:2)), A(i), Ayy(i), Izz(i),...
                    E(i), v(i), self.truss,Fy(i),Zzz(i))];
            end
        end
        %% Compute Displacements Reactions
        %  Compute the displacements and reactions and format them to return
to Mastan2
        function ComputeDisplacementsReactions(self,dP) % RC_Analysis_2d1el
            % Initialize
            self.DEFLstep = zeros(3,self.nnodes);
            self.REACTstep = zeros(3,self.nnodes);

            % Compute the displacements, dD
            self.delf = self.Kff \ (dP - self.Kfn*self.deln);

            % Format the computed displacements using linear indexing of
            % the "DEFL" matrix
            self.DEFLstep(self.dof_free) = self.delf;
            self.DEFLstep(self.dof_disp) = self.deln;

            % Columns: DOF, Rows: Nodes
            self.DEFLstep = self.DEFLstep';

            % Compute the reactions, accounting for loads applied directly
            % on the supports
            self.Ps = self.Ksf*self.delf + self.Ksn*self.deln - ...
                      self.Psupp * self.ratio_req ;
            self.Pn = self.Knf*self.delf + self.Knn*self.deln;

            % Format the computed reactions using linear indexing of the
            % "REACT" matrix
            self.REACTstep(self.dof_supp) = self.Ps;
            self.REACTstep(self.dof_disp) = self.Pn;
            self.REACTstep = self.REACTstep';
        end

        %% Recover element forces based on the natural deformation approach
        function RecoverElementForces(self,step)
            % Columns: Nodes, Rows: DOF
            DEFL_t = self.DEFLstep';
            for i = 1:self.nele
                % Obtain the displacements at the degrees of freedom
```

```
                % corresponding to element i using linear indexing of the
                % "DEFL_t" matrix
                self.elements(i).ComputeForces(...
                                DEFL_t(self.elements(i).GetElementDOF()));
                if step >1
                    self.ELE_FORstep(i,:) = self.ELE_FOR(i,:,step-1) + ...
                                        self.elements(i).GetdFlocal()';
                else
                    self.ELE_FORstep(i,:) = self.ELE_FOR(i,:,1) + ...
                                        self.elements(i).GetdFlocal()';
                end
                self.elements(i).UpdateFLocal(self.ELE_FORstep,i);
            end
        end
        %% Global Coordinates, updates after the load step
        % Purpose: Update the geometry to calculate the incremental output
        % variables, such as the incremental internal force
        function UpdateGeometry(self)
            % Columns: DOF, Rows: Nodes
            defl_t = self.DEFLstep';
            for i = 1:self.nnodes
                nodeDOF = self.nodes(i).GetNodeDOF();
                % Obtain only the displacements in the x and y direction to
                % update the geometry
                nodeDOF = nodeDOF(1:2);


self.nodes(i).UpdateCoordinates(self.nodes(i).GetNodeCoord()...
                    + defl_t(nodeDOF));
            end
        end
        %% Computes the Resultant
        % Purpose: The resultant is used to calculate the error where
        % {E}={P}-{R}
        function ComputeResultant(self)
            R=zeros(self.num_dof_total,1);

            %loop over all elements, pulling each elements DOF and updating
their
            %transformation matrix. Use DOF to ensure that the
            %corresponding global force of each element is added to the
            %corresponding entry in the resultant vector.
            for i = 1:self.nele
               %pull current elements dof
               element_dof = self.elements(i).GetElementDOF();
              self.elements(i).UpdateTransformationMatrix();
               gamma = self.elements(i).GetTransformationMatrix();
               %update element forces to global coordinates
               F_global = gamma'*self.ELE_FORstep(i,:)';

               for j=1:length(element_dof)
                   % add the global forces of the element to Resultant
                   % based on the corresping dof
                   R(element_dof(j))=R(element_dof(j))+F_global(j);
               end
            end
            self.R_free=R(self.dof_free);
        end
```

```
        %% Compute Error
        % Computes the incremental error based on the applied load and
        % resultant
        function ComputeError(self,step)
            self.Error = self.Pf*self.APRATIOS(step) - self.R_free;
        end
        %% Compute load norm indices and the error indices
        % Purpose: Compute norms to be plotted against the applied load
        % ratio
        function ComputeNorms(self,i)
            self.Load_Norm = [self.Load_Norm ; norm(self.Error) / ...
                              norm(self.Pf*self.APRATIOS(i))];
            self.Energy_Norm = [self.Energy_Norm;(abs(self.Error') * ...

abs(self.delf))/(abs(self.Pf'*self.APRATIOS(i))...
                                 *abs(self.delf))];
        end
        %% Check Kff Matrix
        % Purpose: Determines whether the incremental SS analysis should stop
        % prematurely or reaches the user-specified end points based on
        % matrix conditioning and cholesky decomposition

        % AFLAG = 1     Successful
        % AFLAG = 0     Unstable Structure
        % AFLAG = -1    Analysis Halted: Limit Load Reached
        % LIMIT_STATE = 0  limit state not reached, system is loading
        % LIMIT_STATE = 1  limit state reached or exceeded, system is
        %                  unloading
        function CheckKffMatrix(self)
            [~,p] = chol(self.Kff);
            if self.Kffstep == 1 && ((condest(self.Kff) ...
               > self.Kff_condition_threshold) || (p > 0))
                self.LIMIT_STATE = 0;
                self.AFLAG = 0;
            elseif (p > 0) % Limit Point based on Cholesky decomposition
                self.LIMIT_STATE = 1;
                self.AFLAG = -1;
            else
                self.LIMIT_STATE = 0;
                self.AFLAG = 1;
            end
            self.Kffstep = self.Kffstep + 1;
        end
        %% Initialize variables to be able to run the analysis
        % 2D initialization does not include the number of steps
        function InitializeOutputVariables(self)
           self.Kffstep = 1;
           self.DEFL = zeros(self.nnodes,self.num_dof_node,1);
           self.REACT = zeros(self.nnodes,self.num_dof_node,1);
           self.ELE_FOR = zeros(self.nele,self.num_dof_node*2,1);
           self.ELE_FORstep = zeros(self.nele, self.num_dof_node*2);
           self.ELE_YLD = zeros(self.nele,2,1);
           self.ELE_YLDstep = zeros(self.nele, 2);
        end
        %% Reguli falsi method
        % Return an attempted load increment to below the yield surface
        function [tau_regula] =
```

```
ComputeReturn(self,dP,dM,P,M,element,max_limit)
            % Pass in YS_current_step and YS_next_step for one end at a time,
            % i.e. a 2x1 vector.
            % one element and one hinge at a time
            % Run if statement to determine the state of the element, it
            % could be 1 out of 3 cases, set tau = 0, and only update it
            % with tau_new if tau_new is lower in value than tau

            % Initialize
            tau_lower = 0;
            tau_upper = 1;
            error = 10;
            % Error reduction based on while loop
            while error > self.tau_min_tol
                phi_lower = element.ComputeYieldSurface(P+tau_lower*dP,...
                            M+tau_lower*dM);
                phi_upper = element.ComputeYieldSurface(P+tau_upper*dP,...
                            M+tau_upper*dM);
                tau_regula = tau_upper - ((phi_upper-max_limit)*(tau_lower-
tau_upper))/...
                             (phi_lower - phi_upper);
                phi_regula = element.ComputeYieldSurface(P+tau_regula*dP ,...
                             M+tau_regula*dM);
                if sign(phi_regula - max_limit) == sign(phi_lower -
max_limit)
                    tau_lower = tau_regula;
                else
                    tau_upper = tau_regula;
                end
                error = abs(max_limit - phi_regula);
            end
        end
        %% Create Stiffness Matrix
        %  Create the global stiffness matrix for the structure and store
        %  it in sparse format modified to include the gradient and yield
        % surface calculations
        function CreateStiffnessMatrix(self,step)

            % Initialize the vectors that will be store the coordinates and
            % values of the non-zero elements in K
            K_row = [];
            K_col = [];
            K_data = [];

            % Loop over all elements and append the contribution of each
            % element's global stiffness matrix to the "K_row", "K_col",
            % and "K_data" vectors
            for i = 1:self.nele
                if step > 1
                    F = self.ELE_FOR(i,:,step-1);
                else
                    F = self.ELE_FOR(i,:,1);
                end
                YS_i = self.elements(i).ComputeYieldSurface(F(1),F(3));
                YS_j = self.elements(i).ComputeYieldSurface(F(4),F(6));
                self.elements(i).ComputeGradient([F(1);F(4)],[F(3);F(6)],...
                    [YS_i;YS_j],self.tolerance);
                self.elements(i).ComputeGlobalStiffnessMatrix();
```

```
                [row, col, data] = find(self.elements(i).GetKGlobal());

                element_dof = self.elements(i).GetElementDOF();
                K_row = [K_row; element_dof(row)];
                K_col = [K_col; element_dof(col)];
                K_data = [K_data; data];
            end
            % Convert the "K_row", "K_col", and "K_data" vectors to a sparse
matrix
            self.K = sparse(K_row, K_col, K_data, self.num_dof_total,
self.num_dof_total);
            self.ComputeStiffnessSubMatrices();
            self.CheckKffMatrix();
        end
    end

end
```

ELEMENT CLASS

```
% Cesar Y. Marco & David Tse (CMDT)
% CEE 282: Programming Project
% 2d2in Element Class
% 03/18/16

classdef CMDT_Element_2d2in < RC_Element_2d1el
% Replace XYZ by your initials and rename the file accordingly before
proceeding
% This is the child class of RC_Element_2d1el.m.
% All the protected & public data properties of parent class will be
inherited to this class


% Element class for 2nd order analysis of a 2-dimensional structure

    % Protected properties go here
    properties (Access = public)
        % Use this space to define all the additional dataproperties if
required.
        kg
        K
        dFLocal
        Fy
        Mp
        G
        YS
        kp
    end

    % Public methods go here
    methods (Access = public)
        % Define the constructor here and use it to call the parent class.
        function self = CMDT_Element_2d2in(element_nodes, A, Ayy, Izz, E, v,
truss,Fy,Zzz)
            self = self@RC_Element_2d1el(element_nodes, A, Ayy, Izz, E, v,
```

```
truss);
            self.dFLocal = zeros(6,1);
            self.f_local = zeros(6,1);
            self.kp = 0;
            self.ComputeLocalGeometricStiffnessMatrix();
            syms P M
            self.ComputeYieldStrength(Fy,A);
            self.ComputePlasticMoment(Zzz,Fy);
            self.YS = self.ComputeYieldSurface(P,M)
        end
        %% Compute the geometric stiffness matrix in global coordinates
        function ComputeGlobalStiffnessMatrix(self)
            self.K = self.gamma' * (self.ke_local + self.kp +
self.kg)*self.gamma;
        end
        %% Getter Functions
        % Allows superclasses (i.e. 2d2el Analysis) to obtain protected
        % variables from this class
        function K = GetKGlobal(self)
            K = self.K;
        end

        function dF = GetdFlocal(self)
            dF = self.dFLocal;
        end

        function gamma = GetTransformationMatrix(self)
            gamma = self.gamma;
        end
        %% Updates the transformation matrix based on the new geometry
        % Used in ComputeResultant() in the 2d2el analysis class
        function UpdateTransformationMatrix(self)
            axis = self.element_nodes(2).GetNodeCoord() -
self.element_nodes(1).GetNodeCoord();
            % Calculate terms
            theta = cart2pol(axis(1), axis(2));
            c = cos(theta);
            s = sin(theta);
            % Assemble 3x3 gamma terms
            gamma3 = [ c,  s,  0;
                      -s,  c,  0;
                       0,  0,  1];
            zeros3 = zeros(3);
            % Assemble big gamma
            self.gamma = sparse([gamma3, zeros3;
                                 zeros3, gamma3]);
        end
        %%  Updates the element force
        % This method is used within the the RecoverElementForces() method
        % in the 2d2el analysis class
        function UpdateFLocal(self,ELE_FOR,ele_num)
            self.f_local = ELE_FOR(ele_num,:)';
        end
        %% Element forces based on the natural deformation approach
         function ComputeForces(self, del_global)
            self.del_global = del_global;

            % Compute the element displacement vector in local coordinates
```

```
            self.del_local = self.gamma * self.del_global;

            % Compute theta_r
            theta_r = atan((self.del_local(5) - self.del_local(2))/...
                      (self.L + self.del_local(4) - self.del_local(1))); %rad

            % Compute theta_an
            theta_an = self.del_local(3) - theta_r;

            % Compute theta_bn
            theta_bn = self.del_local(6) - theta_r;

            % Compute un
            un = (self.del_local(4)-self.del_local(1)) ...
                + ((self.del_local(4)-self.del_local(1))^2 + ...
                (self.del_local(5)-self.del_local(2))^2)/(2*self.L);

            % Assemble del_delta_n in local coordinates
            dDn = [0;0;theta_an;un;0;theta_bn];

%             Compute the element force vector in local coordinates
            self.dFLocal = (self.ke_local + self.kp + self.kg) * dDn;
%             self.dFLocal = (self.ke_local + self.kp) * self.del_local;

         end
        %% Compute Local Geometric Stiffness Matrix
        %  Check whether the element is part of a truss or not, and compute
its local elastic stiffness matrix
        %  accordingly. Store the computed matrix in sparse format.
        function ComputeLocalGeometricStiffnessMatrix(self)
            self.kg = sparse(6,6);
            self.kg(1,1) = self.f_local(4)/self.L;
            self.kg(1,4) = -self.kg(1,1);
            self.kg(2,2) = 1.2*self.f_local(4)/self.L;
            self.kg(2,3) = self.f_local(4)/10;
            self.kg(2,5) = -self.kg(2,2);
            self.kg(2,6) = self.kg(2,3);
            self.kg(3,3) = 2*self.L*self.f_local(4)/15;
            self.kg(3,5) = -self.kg(2,3);
            self.kg(3,6) = -self.f_local(4)*self.L/30;
            self.kg(4,4) = self.kg(1,1);
            self.kg(5,5) = self.kg(2,2);
            self.kg(5,6) = -self.kg(2,3);
            self.kg(6,6) = self.kg(3,3);
            self.kg(3,2) = self.kg(2,3);
            self.kg(4,1) = self.kg(1,4);
            self.kg(5,2) = self.kg(2,5);
            self.kg(5,3) = self.kg(3,5);
            self.kg(6,2) = self.kg(2,6);
            self.kg(6,3) = self.kg(3,6);
            self.kg(6,5) = self.kg(5,6);
        end
        %% Compute the Yield Strength
        function ComputeYieldStrength(self,fy,A)
            % Convert stress to force
            self.Fy = fy*A;
        end
```

```
        %% Compute the Plastic Moment
        function ComputePlasticMoment(self,Zx,fy)
            self.Mp = Zx*fy;
        end
        %% Compute the Yield Surface
        function YieldS = ComputeYieldSurface(self,P,M)
            YieldS = (P/self.Fy)^2 + (M/self.Mp)^2 + 3.5*(P/self.Fy)^2 *
(M/self.Mp)^2;
        end
        %% Compute the gradient matrix of the yield surface
        function ComputeGradient(self,P,M,Yield,tolerance)
%             Force and Moment are 2x1 vectors, where the 1st row is the i
%             end and the 2nd row is the j end
            % Yield is a 2x1 vector where i is the 1st row and j is the 2nd
            % row, check using if statments to determine if both or one end
            % has yielded
            if Yield(1) >= (1 - tolerance) && Yield(2) >= (1-tolerance)
                % both i and j end yield
                self.G = zeros(6,2);
                 self.G = [
(2*P(1))/self.Fy^2 + (7*M(1)^2*P(1))/(self.Fy^2*self.Mp^2),
0;
                                                        0,
0;
(2*M(1))/self.Mp^2 + (7*M(1)*P(1)^2)/(self.Fy^2*self.Mp^2),
0;
                                                        0, (2*P(2))/self.Fy^2 +
(7*M(2)^2*P(2))/(self.Fy^2*self.Mp^2);
                                                        0,
0;
                                                        0, (2*M(2))/self.Mp^2 +
(7*M(2)*P(2)^2)/(self.Fy^2*self.Mp^2)]  ;
                self.ComputeKPlastic();
            elseif Yield(1) >= (1 - tolerance) % i end yields
                self.G = zeros(6,1);
self.G = [
(2*P(1))/self.Fy^2 + (7*M(1)^2*P(1))/(self.Fy^2*self.Mp^2);
                                                              0;
(2*M(1))/self.Mp^2 + (7*M(1)*P(1)^2)/(self.Fy^2*self.Mp^2);
                                                              0;
                                                              0;
                                                              0];
                self.ComputeKPlastic();
             elseif Yield(2) >= (1 - tolerance) % j end yields
                self.G = zeros(6,1);
                self.G  = [0;
                           0;
                           0;
                           (2*P(2))/self.Fy^2 +
(7*M(2)^2*P(2))/(self.Fy^2*self.Mp^2);
                           0;
                           (2*M(2))/self.Mp^2 +
(7*M(2)*P(2)^2)/(self.Fy^2*self.Mp^2)];
                self.ComputeKPlastic();
            else
                self.kp = 0;
            end
        end
```

```
        %% Calculate the plastic reduction matrix, kp
        function ComputeKPlastic(self)
            self.kp = -self.ke_local * self.G * (self.G' * self.ke_local *...
                      self.G)^-1 * self.G' * self.ke_local;
        end
    end
end
```

NODE CLASS

```
% Cesar Y. Marco & David Tse (CMDT)
% CEE 282: Programming Project
% 2d2in Node Class
% 03/18/16

classdef CMDT_Node_2d2in < RC_Node_2d1el
% This is the child class of RC_Node_2d1el.m.
% All the protected & public data properties of parent class will be
% inherited to this class

% Node class for 2nd order analysis of a 2-dimensional structure

    % Protected properties go here
    properties (Access = protected)
    end

    % Public methods go here
    methods (Access = public)
        % Constructor function inherits 2d1el parameters
        function self = CMDT_Node_2d2in(node_number, node_coord)
            self = self@RC_Node_2d1el(node_number, node_coord);
        end
        % Updates the x and y coordinates only
        function UpdateCoordinates(self,new_node_coord)
            self.node_coord = new_node_coord;
        end
    end
end
```

Works Cited

[1] McGuire, William, Richard H. Gallagher, and Ronald D. Ziemian. Matrix Structural Analysis. John Wiley & Sons, 2000. Print.