Building the National Virtual Collaboratory for Earthquake Engineering Research

NEESgrid

Technical Report NEESgrid-2004-22
www.neesgrid.org

Last modified: August 31, 2004

FEDEASLab Getting Started Guide and Simulation Examples

Filip C. Filippou and Margarita Constantinides[1]

[1] Department of Civil and Environmental Engineering
University of California, Berkeley

Feedback on this document should be directed to filippou@ce.berkeley.edu

Acknowledgment: The documentation and validation examples for FEDEASLab were supported by NEESgrid subcontract award 04-344 from the NEESgrid SI under NSF grant CMS-0117853

This report is also published as **SEMM** report **2004-05**

Summary

The Matlab© toolbox **FEDEASLab** is a user-friendly, versatile and powerful tool for the simulation of nonlinear structural response under static and dynamic loads, which has been used successfully for the development of new elements and material models, as well as for the simulation of the response of small structural models in research and instruction. Its development started in 1998 for the support of instruction of the basic graduate courses of linear and nonlinear structural analysis at the University of California, Berkeley. To date **FEDEASLab** has evolved into a powerful framework for research in the nonlinear analysis of structures.

The toolbox consists of several functions that are grouped in categories and are, consequently, organized in separate directories. These functions operate on five basic data structures which represent the model, the loading, the element properties, the state of the structural response, and the parameters of the solution strategy. A sixth data structure is optional and carries post-processing information that can be used for response interpretation and visualization.

FEDEASLab supports path-dependent static or transient response under multiple force and displacement patterns. Transient response under multiple force patterns, displacement patterns, and acceleration patterns with uniform or multi-support excitation is also supported.

The nonlinear response analysis of structural models under static or transient conditions is decomposed into logical steps. Each step is represented by a separate function with one or more basic data objects as input and output arguments. In this way the definition of the model, the specification of element properties, the definition of applied force and displacement patterns and corresponding load histories, and the analysis of the model under the loading becomes a sequence of function calls that are organized in script files. With Matlab's scripting language it is easy to customize the analysis sequence and conduct parameter studies. The modular architecture of the toolbox and the compact organization of data allow for the easy addition of new functions for providing new capabilities. It is equally easy to access the data objects and enhance the information stored in them. A well defined interface for element, section and material models permits the user to add custom components to the available element, section and material libraries.

Post-processing is accommodated with a data object that carries all important material, element and structural information for plotting or printing. Several functions that address basic post-processing tasks are provided. The user can easily enhance and extend the current capabilities.

Berkeley, July 31, 2004

Acknowledgements

The evolution of the Matlab© toolbox **FEDEASLab** from 1998 to 2004 started with the primary objective of supporting the instruction of the basic graduate courses of linear and nonlinear structural analysis at the University of California, Berkeley, and progressed to a powerful framework for research in the nonlinear analysis of structures.

The most significant contribution to **FEDEASLab** came from program **FEAP** developed by Professor Robert L. Taylor at the University of California, Berkeley. The original version of **FEDEASLab** set out to emulate a portion of **FEAP**'s capabilities for teaching and research.

Former doctoral student Remo Magalhaes de Souza was present at the birth of the program, and was instrumental in the organization of the solution strategy data structure and the creation of the transformation functions for linear and nonlinear geometry. Recently the contributions of doctoral student Afsin Saritas have expanded the element and material library of the toolbox.

Discussions with Professor Gregory L. Fenves have stimulated the object-oriented organization of the toolbox without resorting to the use of objects. His criticism of early versions of the program has helped in its improvement.

The contribution of these colleagues over the course of **FEDEASLab**'s growth is gratefully acknowledged. Their input can be found in various functions of the toolbox. Any errors and omissions are, however, entirely the fault of its creator.

After its inclusion as simulation component of NEESgrid in late 2003, **FEDEASLab** received financial support for documentation, validation, and example preparation through NEESgrid subcontract award 04-344 from the NEESgrid System Integrator at the University of Illinois, Champaign-Urbana under NSF grant CMS-0117853. Special thanks go to Professor Bill Spencer for his vision and leadership toward the inclusion of simulation capabilities in NEESgrid and to Professor Gregory L. Fenves for championing **FEDEASLab**'s inclusion as one of the simulation components of NEESgrid.

1. Introduction

The objective of this report is to provide a brief overview of **FEDEASLab**'s data structures and functions by describing their use in typical nonlinear static and transient analysis situations for a small structural model. A thorough discussion of the toolbox' architecture and its capabilities is provided in NEES Technical Report TR-2004-50.

It is assumed that the reader is familiar with Matlab©, in particular with numeric arrays and array operations, and with data structures and cell arrays. In addition to the excellent on-line help of the program the reader is referred to chapters 5 through 7 of the book Mastering Matlab 6 by D. Hanselman and B. Littlefield published in 2001 by Prentice Hall.

The toolbox functions require Matlab© version 6.x or later.

2. Installation

FEDEASLab toolbox functions are contained in a self-extracting zip file that can be downloaded from the website: *http://fedeaslab.berkeley.edu*. Execution of the file will unzip the functions into several subdirectories under the directory containing the executable. The subdirectory organization is shown in Fig. 1.

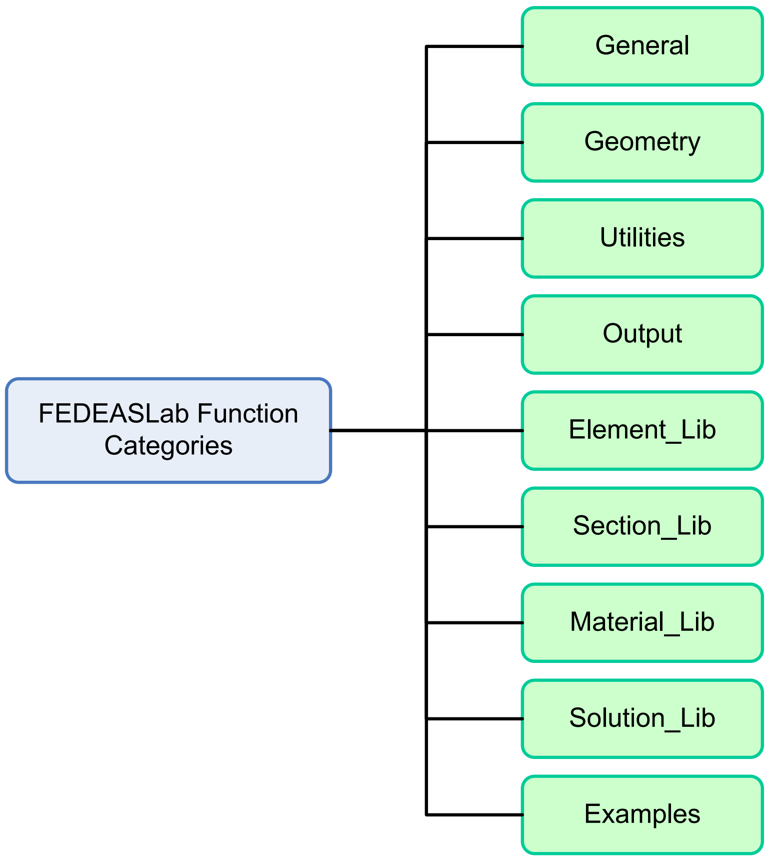


Figure 1 Directory organization of FEDEASLab functions

The execution of file `Adjust_Path.m` will add these subdirectories to Matlab's directory search path. Each subdirectory contains a `Contents` file with a brief description of each function. This can be invoked directly from Matlab's command line by typing `help directory_name`.

3. Data structures

The Matlab© toolbox **FEDEASLab** consists of several functions. These operate on five basic data objects that carry information about the structural model geometry, the element properties, the applied loading, the solution strategy parameters, and the response state. A sixth data object called **Post** is optional and carries post-processing information for response interpretation and visualization. These data structures are summarized in Fig. 2. In this report data structures are identified with a double vertical border in figures and with bold type face in text.

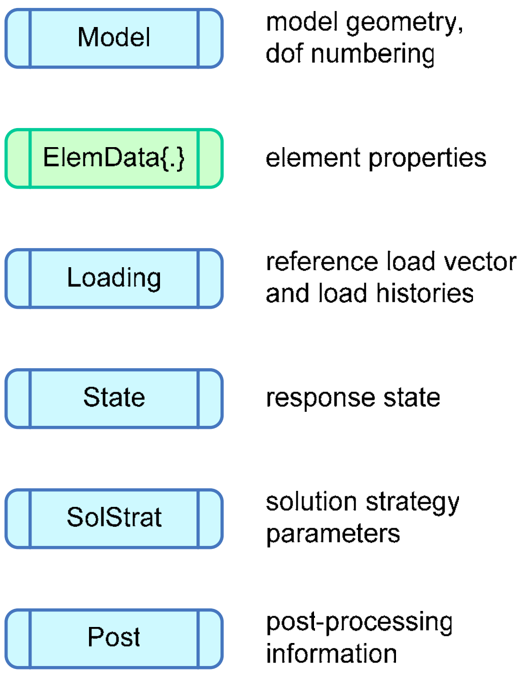

Model
model geometry, dof numbering
ElemData{.}
element properties
Loading
reference load vector and load histories
State
response state
SolStrat
solution strategy parameters
Post
post-processing information

Figure 2 Data structures in FEDEASLab

Detailed information about the basic data structures and the toolbox function architecture is provided in NEES Technical Report TR-2004-50.

4. The key role of function `Structure`

Each function in the **FEDEASLab** toolbox has a single purpose, which can often be surmised from the function name. The only exception is the function `Structure` which accomplishes several tasks by operating on all elements in the structural model, or only on a group of them, as specified in the optional input argument `ElemList`. The type of action that the function performs is specified by character variable `action` which can assume one of the four letter keywords on the leftmost column of Fig. 3. The function syntax with the order of input arguments along with a short description of the function purpose for every action keyword is shown in Fig. 3. The output of the function is either a data object, or a printing or plotting action with an empty output argument list. In the figure printing is identified by the symbol IOW and plotting by the deformed shape of a structural model.

Structure (action,Model,ElemData,State,ElemList)

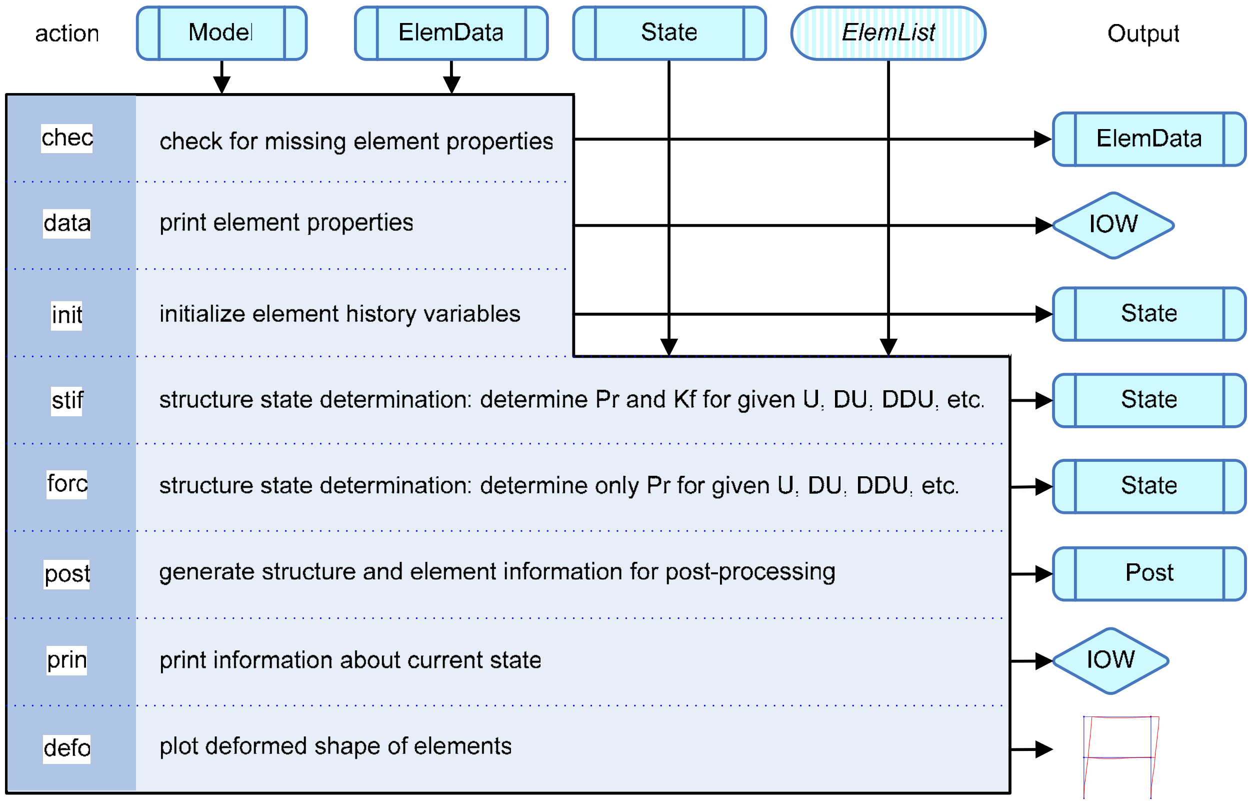

action
Model
ElemData
State
ElemList
Output
chec
check for missing element properties
ElemData
data
print element properties
IOW
init
initialize element history variables
State
stif
structure state determination: determine Pr and Kf for given U, DU, DDU, etc.
State
forc
structure state determination: determine only Pr for given U, DU, DDU, etc.
State
post
generate structure and element information for post-processing
Post
prin
print information about current state
IOW
defo
plot deformed shape of elements

Figure 3 Syntax and input/output argument list of function `Structure`

5. Organization of typical simulation script

A typical simulation script performs the following tasks:

1. Definition of model geometry and creation of data object **Model**.
2. Specification of element properties and creation of data object **ElemData**.
3. State initialization (creation of data object **State**).
4. Specification of one or more load patterns and creation of data object **Loading**.
5. Creation of data object **SolStrat** with default solution strategy parameters.
6. Initialization of solution process and application of one or more load steps with corresponding structural response determination.
7. Storage of response information for immediate or subsequent post-processing.

Because of the importance of **Model** and **ElemData** in subsequent tasks, tasks 1 and 2 must be arranged in this order at the start of the analysis script. In fact, it may be convenient to isolate the model definition and the element property specification in separate script files, as was done in the examples of this report. Fig. 4 shows the sequence of **FEDEASLab** function calls and the input and output arguments for tasks 1 and 2.

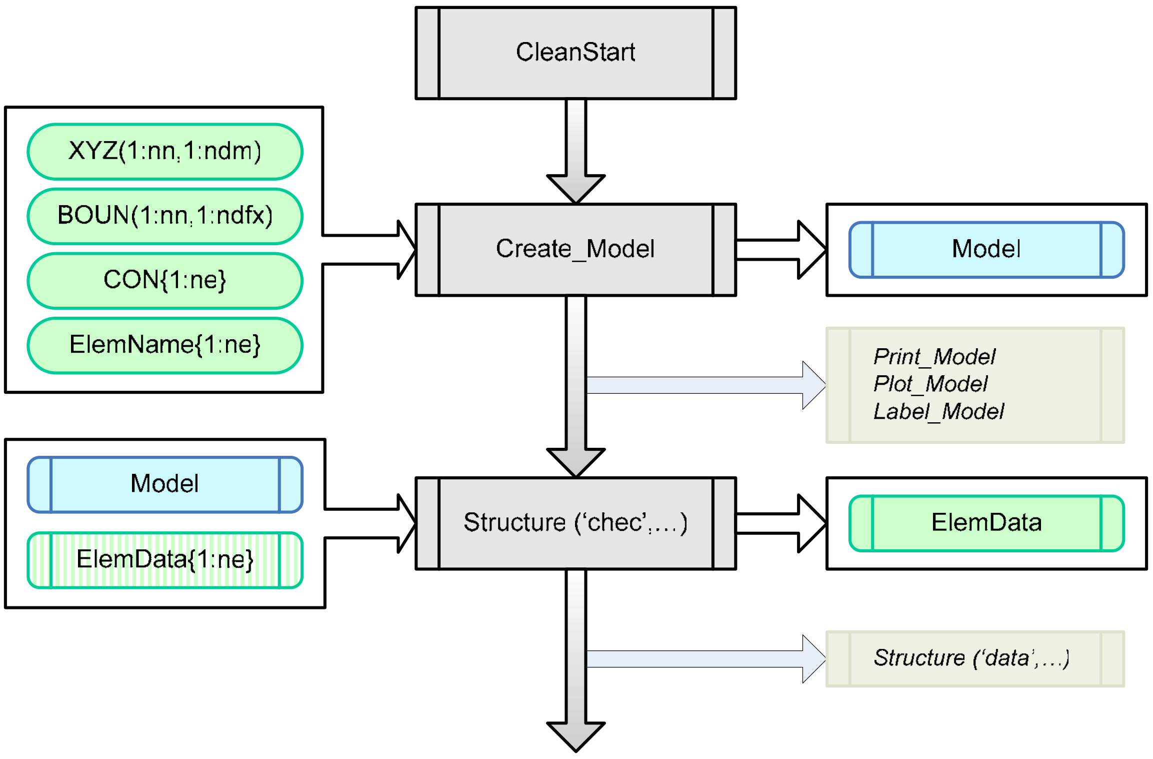


Figure 4 Model definition and ElemData specification

In Fig. 4 functions are identified by square boxes with double vertical borders and are arranged in the middle of the figure with an arrow depicting the sequence. The input arguments to every function are collected in the box pointing toward the function on the left hand side of figure. The key data objects have a double vertical border, while arrays specified by the user have a single border and a rounder outline. The output of every function is typically a single data object. It is arranged in a box on the right hand side of the figure with an arrow pointing away from the function. This graphic convention is followed consistently in this report.

In Fig. 4 the function `CleanStart` clears the workspace memory and initializes a couple of global variables. Upon specification of the node coordinates in array `XYZ`, the boundary conditions in array `BOUN`, the element connectivity in cell array `CON` and the element type in cell array `ElemName` the function `Create_Model` generates the data object **Model** with fields carrying information about the model geometry and the degree of freedom (dof) numbering. In the next step the user specifies the element properties in cell array **ElemData** and the function `Structure` with action keyword `'chec'` checks the element property data for missing information and supplies default values, if necessary. The data object completion process is indicated in Fig. 4 by the darker background of the **ElemData** data object upon exit from the function `Structure`. Optional functions for printing model information and displaying the model geometry are shown on the right hand side of Fig. 4 enclosed in a lighter gray background. The optional invocation of function `Structure` with action keyword `'data'` prints the element properties in the output file.

With **Model** and **ElemData** defined, tasks 3 through 5 can be accomplished in any order. The necessary function calls and input arguments are shown in Fig. 5.

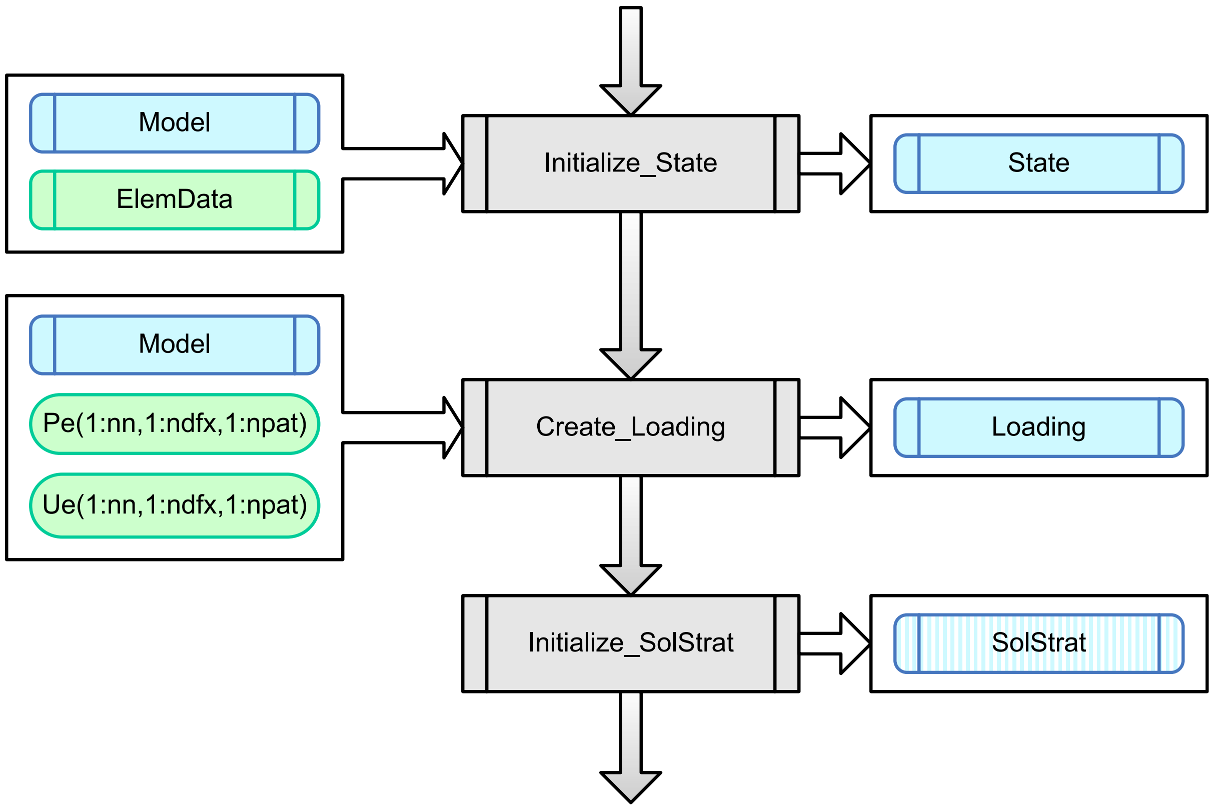


Figure 5 Loading definition and initialization of State and SolStrat

It is important to note, however, that the initialization of **State** is typically done only once in an analysis, while **Loading** and **SolStrat** can be specified several times in the script for describing load sequences and changes to solution strategy parameters. Consequently, the order of Fig. 5 should be adhered to, if possible.

It is worth noting that the data object **SolStrat** emerging from function `Initialize_SolStrat` only contains default values for the solution strategy parameters. The user should supply solution specific values for the case at hand.

Task 6 is made up of several function calls, as shown in Figs. 6 and 7 for static and transient response analysis, respectively

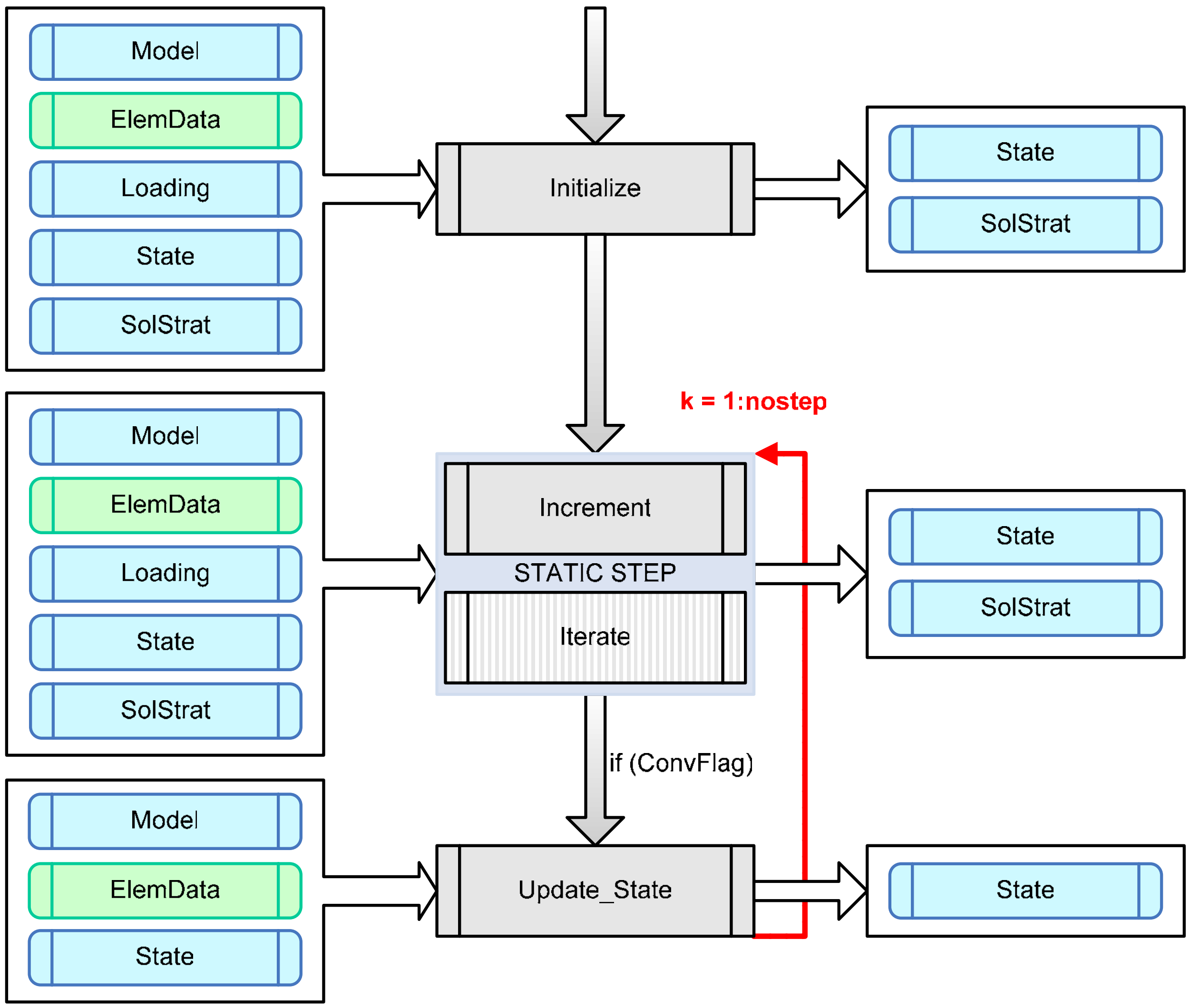


Figure 6 Nonlinear static response analysis with several load steps

Upon specification of the solution strategy parameters the call to the `Initialize` or `TransientInitialize` function sets to zero the pseudo-or real time parameter and the load factor(s) in **State** and stores any resisting forces of the structure as initial forces for the next analysis. It is assumed that these resisting forces are in equilibrium with the applied loading before the application of the new load sequence. Following the initialization, several steps of load incrementation with equilibrium iterations can be performed. Upon convergence of the iteration process in each load step the function `Update_State` or `Update_TransientState` can be used to update the response state variables in **State**. To prepare for the eventuality of lack of convergence during a particular load step it is advisable to copy the last **State** to a data object **TempState** before the start of a new step. **TempState** is then used instead of **State** as input and output argument of the incrementation and iteration functions. Upon convergence **TempState** serves as input argument to the state updating function with **State** as the output argument. This approach allows for switching to alternative solution strategies in case of convergence failure, such as time step subdivision or change of solution strategy parameters.

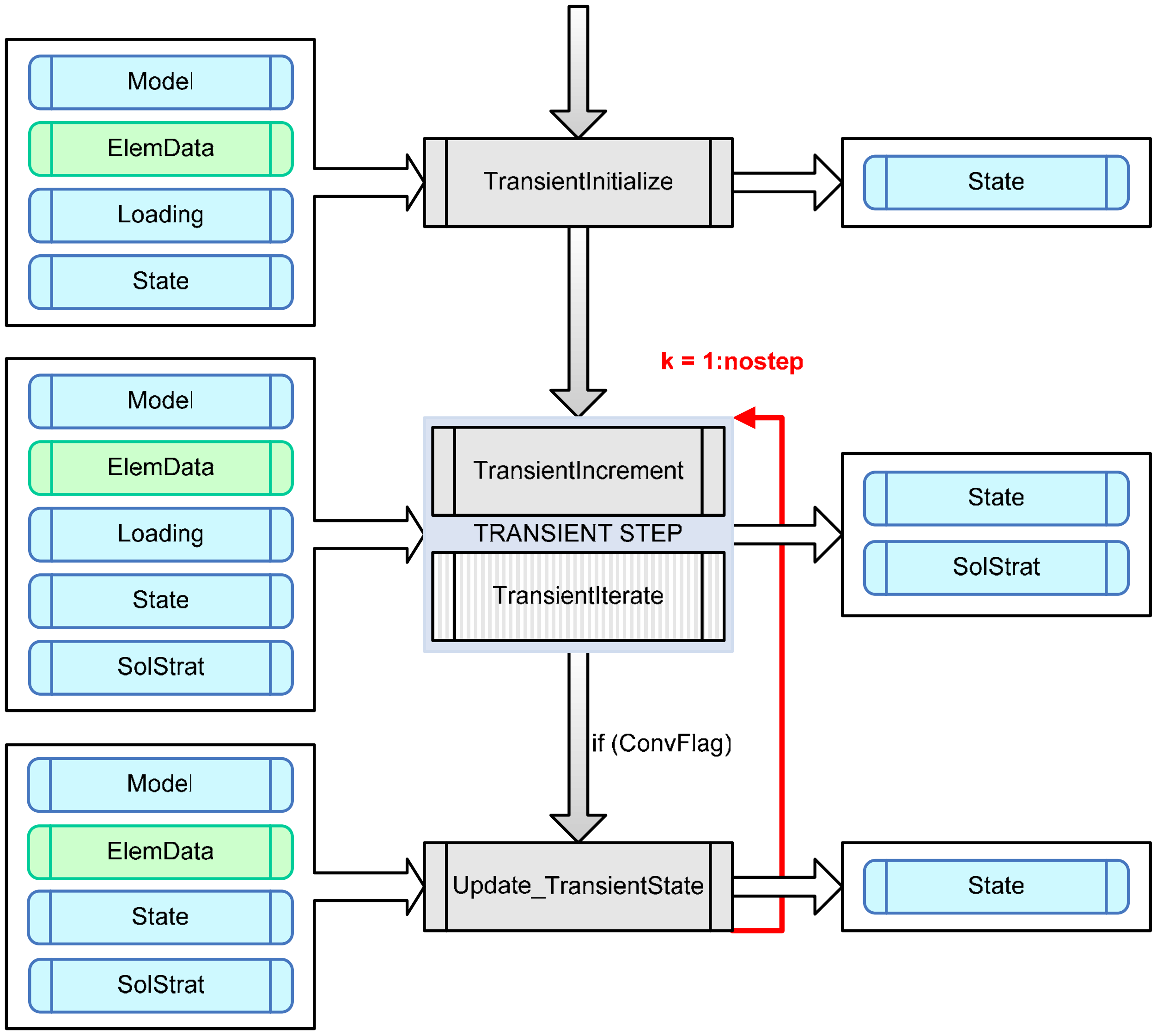


Figure 7 Nonlinear transient response analysis with several time steps

It is worth noting that **SolStrat** is required as input argument to `Update_TransientState` in Fig. 7 because it carries the time integration constants needed for updating the dof velocities and accelerations at the end of the time step.

Structural response information for post-processing can be requested with a call to function `Structure` anytime after **State** is created. This permits information to be stored for each iteration within a load step of the solution process. Such profusion of information may result in a very large data object **Post** for a model with many elements under many load steps. In such case it is advisable to limit the storage of information to a few relevant steps, or to write the data object **Post** to the hard disk and clear it from memory. The syntax for generating post-processing information is shown in Fig. 8.

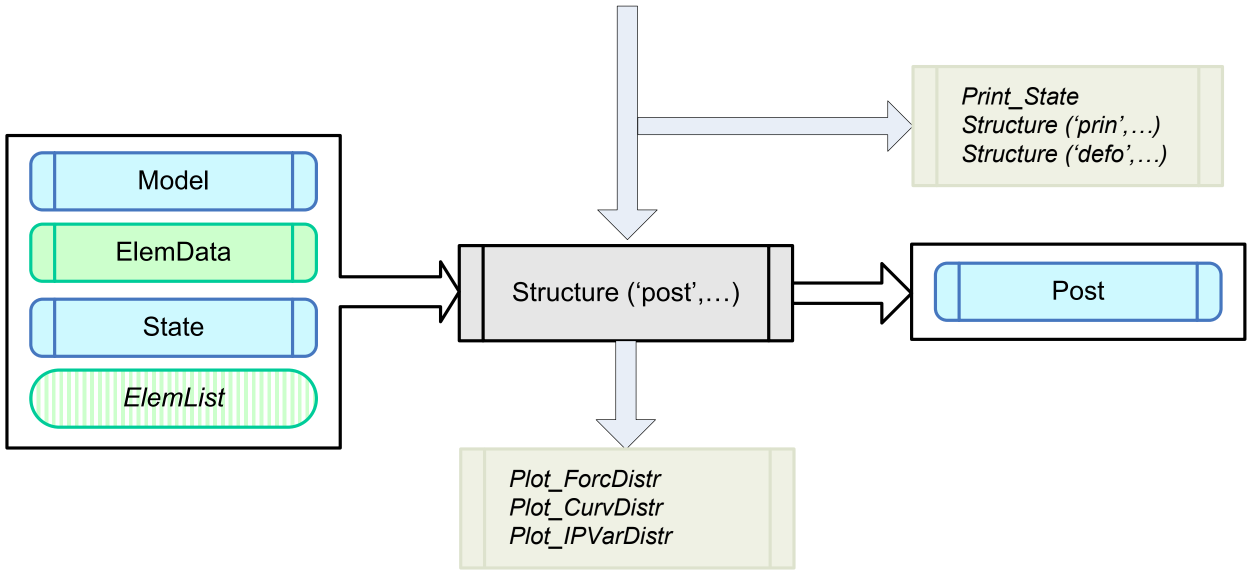


Figure 8 Generation of post-processing information in data object Post

Fig. 8 shows that it is possible to limit the generation of post-processing information to elements in an `ElemList`. The figure also shows that a few post-processing functions do not require **Post**. These functions precede the function call to `Structure` in Fig. 8. `Print_State` sends structural response information like global dof displacements and resisting forces, which is stored in data obejct **State**, to the output file. The same is true for element response information with a call to function `Structure` with action keyword `'prin'`, while the call to function `Structure` with action keyword `'defo'` plots the deformed shape of the structure under the current state. Once **Post** is available, the functions `Plot_xxDistr` can be used to plot the distribution of certain variables along the element axis.

The syntax of the distribution plotting functions is shown in Fig. 9. Additional details about function syntax are available in NEES Technical Report TR-2004-50 and in the on-line help file that accompanies the functions.

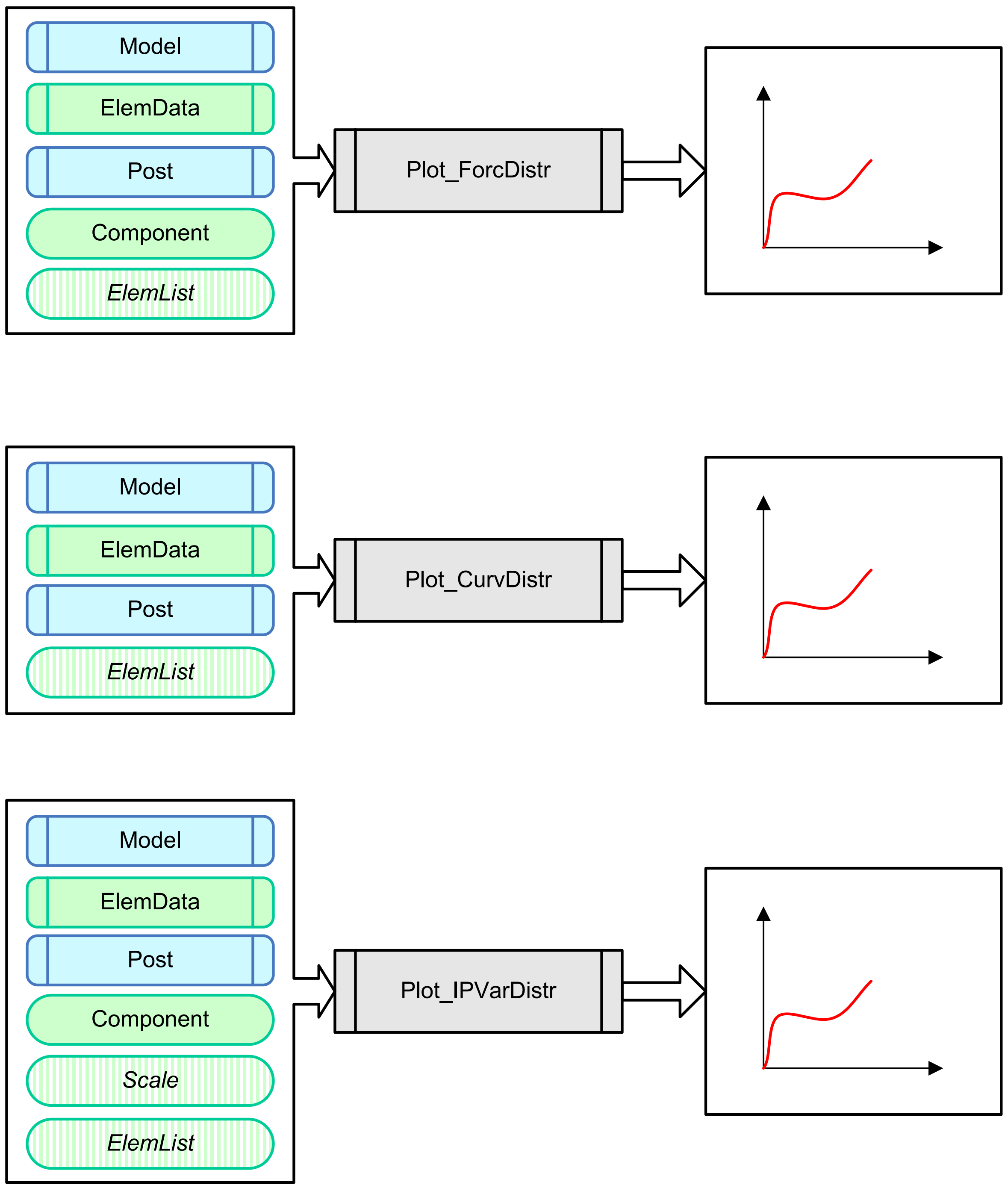


Figure 9 FEDEASLab functions for plotting element variables

6. Simulation examples

The following examples present the necessary steps for performing a linear or nonlinear static or dynamic analysis of the two-story steel frame model in Fig. 10. Some of the applied loads are also shown in Fig. 10. In addition, the following simulation studies cover the case of imposed support displacement as well as support acceleration. The model consists of eight elements: one for each column and two elements for each girder. Nodes are represented by little squares in the figure.

Because the model geometry is the same for the simulation studies it is specified in a separate script file called `Model_TwoStoryFrm`.

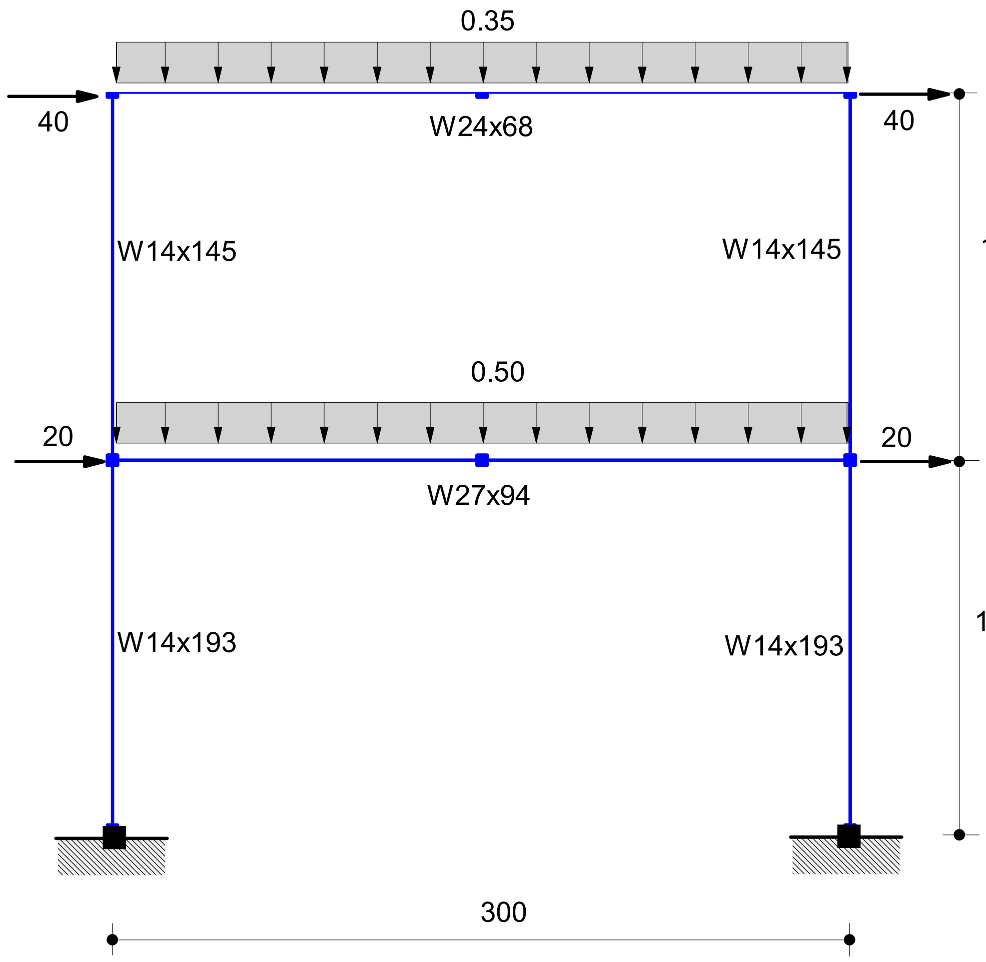
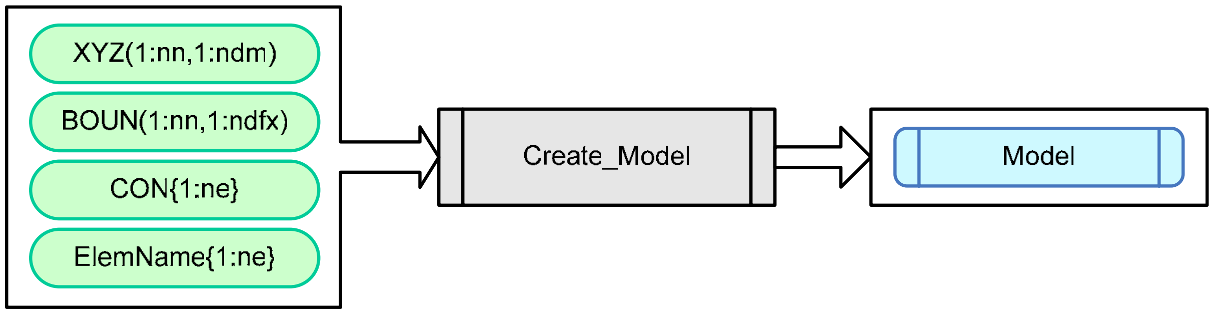


Figure 10 Two-story steel frame model used in the simulation examples

Figure 11 shows the schematic representation of the process and the actual contents of file `Model_TwoStoryFrm`. The model definition for a two-story, one-bay 3d braced frame is shown in the Appendix and in the **FEDEASLab** website.



Create Model

```
% all units in kip and inches
```

Node coordinates (in feet!)

```
XYZ(1,:) = [ 0      0];  % first node
XYZ(2,:) = [ 0     12];  % second node, etc
XYZ(3,:) = [ 0     24];  %
XYZ(4,:) = [25      0];  %
XYZ(5,:) = [25     12];  %
XYZ(6,:) = [25     24];  %
XYZ(7,:) = [12.5   12];  %
XYZ(8,:) = [12.5   24];  %
% convert coordinates to inches
XYZ = XYZ.*12;
```

Connectivity array

```
CON {1} = [   1    2];    % first story columns
CON {2} = [   4    5];
CON {3} = [   2    3];    % second story columns
CON {4} = [   5    6];
CON {5} = [   2    7];    % first floor girders
CON {6} = [   7    5];
CON {7} = [   3    8];    % second floor girders
CON {8} = [   8    6];
```

Boundary conditions

```
% (specify only restrained dof's)
BOUN(1,1:3) = [1 1 1];  % (1 = restrained,  0 = free)
BOUN(4,1:3) = [1 1 1];
```

Element type

```
% Note:  any 2 node 3dof/node element can be used at this point!
[ElemName{1:8}] = deal('Lin2dFrm_NLG');     % 2d linear elastic frame element
```

Create model data structure

```
Model = Create_Model(XYZ,CON,BOUN,ElemName);
```

Display model and show node/element numbering (optional)

```
Create_Window (0.70,0.70);          % open figure window
set(gcf,'Color',[1 1 1]);
Plot_Model  (Model);                % plot model (optional)
Label_Model (Model);                % label model (optional)
```

Figure 11 Contents of file `Model_TwoStoryFrm`

Even though `ElemName` is specified as a 2d linear elastic frame element, the actual element type is modified later with the element property information. During model creation the function `Create_Model` requires the number of nodes and the number of dofs per node for each element in the model. Because all 2d frame elements in the current library have two nodes and 3 dofs per node, it is irrelevant what 2d frame element type is specified at this stage. The output of functions `Plot_Model` and `Label_Model` is shown in Fig. 12.

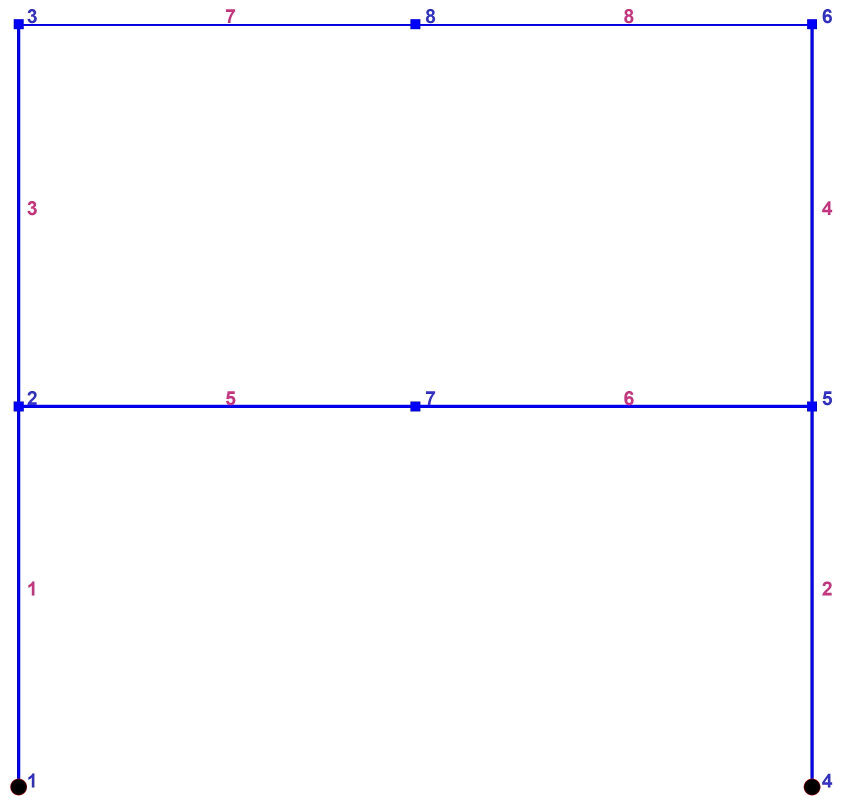
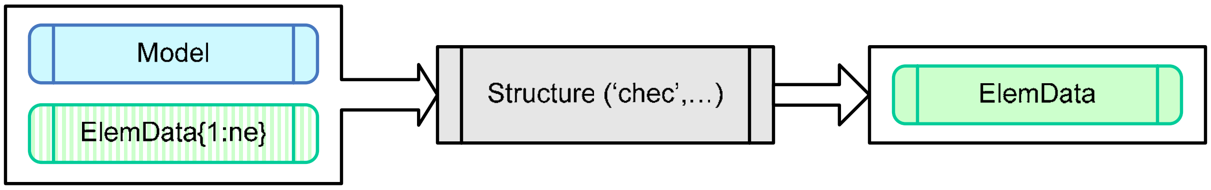


Figure 12 Output from Plot_Model and Label_Model

The next stage involves the specification of element property data. These data are also specified in a separate script file. There are three types of element in the simulation studies: a linear elastic 2d frame element, a nonlinear one-component 2d frame element with concentrated plastic hinges at the ends, and a distributed inelasticity 2d frame element with 5 integration points along the span and discretization of each section into layers with uniaxial material response. Correspondingly, three separate script files are provided, one for each case: `LinearElemData`, `SimpleNLElemData` and `DistrInelElemData`. The contents of these files are provided in Figs. 13a-c. By invoking the function `Structure` with action keyword `'chec'` the element property data are checked for missing information and default values are supplied, if necessary.



Define elements

```
% all units in kip and inches
```

Element name: 2d linear elastic frame element

```
[Model.ElemName{1:8}] = deal('Lin2dFrm_NLG');
```

Element properties

Columns of first story W14x193

```
for i=1:2;
   ElemData{i}.E = 29000;
   ElemData{i}.A = 56.8;
   ElemData{i}.I = 2400;
end
```

Columns of second story W14x145

```
for i=3:4;
   ElemData{i}.E = 29000;
   ElemData{i}.A = 42.7;
   ElemData{i}.I = 1710;
end
```

Girders on first floor W27x94

```
for i=5:6;
   ElemData{i}.E = 29000;
   ElemData{i}.A = 27.7;
   ElemData{i}.I = 3270;
end
```

Girders on second floor W24x68

```
for i=7:8;
   ElemData{i}.E = 29000;
   ElemData{i}.A = 20.1;
   ElemData{i}.I = 1830;
end
```

Default values for missing element properties

```
ElemData = Structure ('chec',Model,ElemData);
```

Figure 13a Element property specification for 2d linear frame elements

Define elements

```
% all units in kip and inches
```

Element name: 2d nonlinear frame element with concentrated inelasticity

```
[Model.ElemName{1:8}] = deal('OneCo2dFrm_NLG');     % One-component nonlinear 2d
frame element
```

Element properties

```
fy  = 50;         % yield strength
eta = 1.e-5;      % strain hardening modulus for multi-component models
```

Columns of first story W14x193

```
for i=1:2;
   ElemData{i}.E = 29000;
   ElemData{i}.A = 56.8;
   ElemData{i}.I = 2400;
   ElemData{i}.Mp  = 355*fy;
   ElemData{i}.eta = eta;
end
```

Columns of second story W14x145

```
for i=3:4;
   ElemData{i}.E = 29000;
   ElemData{i}.A = 42.7;
   ElemData{i}.I = 1710;
   ElemData{i}.Mp  = 260*fy;
   ElemData{i}.eta = eta;
end
```

Girders on first floor W27x94

```
for i=5:6;
   ElemData{i}.E = 29000;
   ElemData{i}.A = 27.7;
   ElemData{i}.I = 3270;
   ElemData{i}.Mp  = 278*fy;
   ElemData{i}.eta = eta;
end
```

Girders on second floor W24x68

```
for i=7:8;
   ElemData{i}.E = 29000;
   ElemData{i}.A = 20.1;
   ElemData{i}.I = 1830;
   ElemData{i}.Mp  = 177*fy;
   ElemData{i}.eta = eta;
end
```

Default values for missing element properties

```
ElemData = Structure ('chec',Model,ElemData);
```

Figure 13b Element property specification for 2d nonlinear one-component model

Element name: 2d nonlinear frame element with distributed inelasticity

```
[Model.ElemName{1:8}] = deal('NLdirFF2dFrm_NLG'); % NL iterative force
formulation
```

Element properties

Columns of first story W14x193

```
for i=1:2;
   ElemData{i}.nIP    = 5;            % number of integration points
   ElemData{i}.IntTyp = 'Lobatto';    % Gauss-Lobatto Integration
   ElemData{i}.SecName= 'HomoWF2dSec';     % type of section
   for j=1:ElemData{i}.nIP
      ElemData{i}.SecData{j}.d   = 15.48;  % depth
      ElemData{i}.SecData{j}.tw  =  0.89;  % web thickness
      ElemData{i}.SecData{j}.bf  = 15.71;  % flange width
      ElemData{i}.SecData{j}.tf  =  1.44;  % flange thickness
      ElemData{i}.SecData{j}.nfl =     4;  % number of flange layers
      ElemData{i}.SecData{j}.nwl =     8;  % number of web layers
      ElemData{i}.SecData{j}.IntTyp = 'Midpoint';   % midpoint integration rule
   end
end
```

Columns of second story W14x145

....

Girders on first floor W27x94

....

Girders on second floor W24x68

....

Material properties

```
for i=1:Model.ne;
   for j=1:ElemData{i}.nIP
      ElemData{i}.SecData{j}.MatName    = 'BilinearHysteretic1dMat';       %
material type
      ElemData{i}.SecData{j}.MatData.E  = 29000;  % elastic modulus
      ElemData{i}.SecData{j}.MatData.fy = 50;     % yield strength
      ElemData{i}.SecData{j}.MatData.Eh = 0.1;    % hardening modulus
   end
end
```

Default values for missing element properties

ElemData = Structure ('chec',Model,ElemData);

Figure 13c Element property specification for 2d distributed inelasticity elements

With the model geometry and element specification complete, it is now possible to apply different types of loading for linear or nonlinear, static or transient response analysis. This is discussed in the following examples.

6.1 Example 1- Linear elastic analysis with superposition of results

In this example the two story steel frame is subjected to three types of loading: (a) uniformly distributed element loads in the girders, as shown in Fig. 10, (b) horizontal forces, as shown in Fig. 10, and, (c) a horizontal displacement of 0.2 units at the left support. Three separate linear analyses are conducted and the results are then superimposed with load combination factors, as required by LRFD. The sequence of function calls is shown graphically in Fig. 14

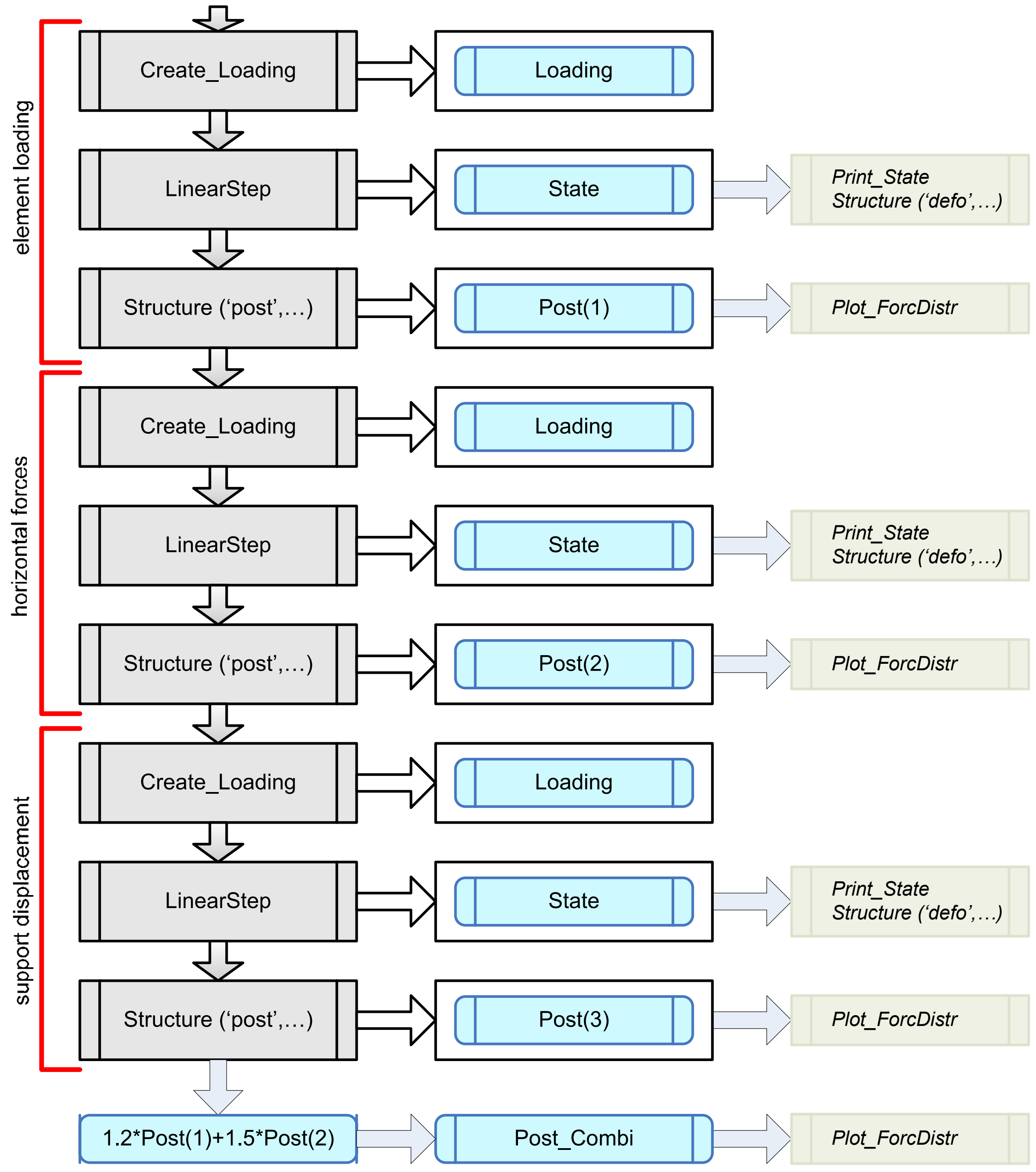


Figure 14 Function call sequence for the analysis steps of Example 1

It is noteworthy that a single analysis function called `LinearStep` is available in this case. The function syntax is shown in Fig. 15.

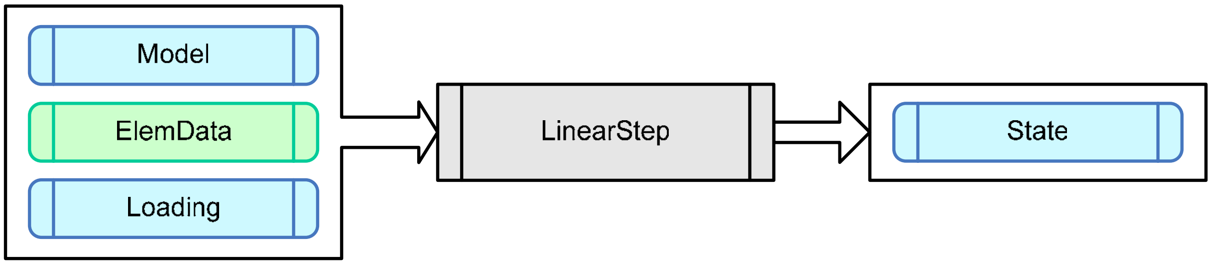


Figure 15 Input and output arguments for function LinearStep

The moment distribution under factored gravity element loads and horizontal forces is shown in Fig. 16, while Fig. 17 contains salient excerpts from the script file Example_1. It is worth noting in Fig. 16 the necessity to zero the distributed element load before defining the loading for the horizontal forces. It is equally necessary to specify an empty force vector in the third load case, so as to set to zero the horizontal forces before defining the loading for the horizontal support displacement.

Finally, the distributed element loading needs to be re-inserted with the appropriate load factor in the data object **ElemData** before invoking the last `Plot_ForcDistr` function, so that the moment diagram reflects the presence of the element loading in the factored load combination.

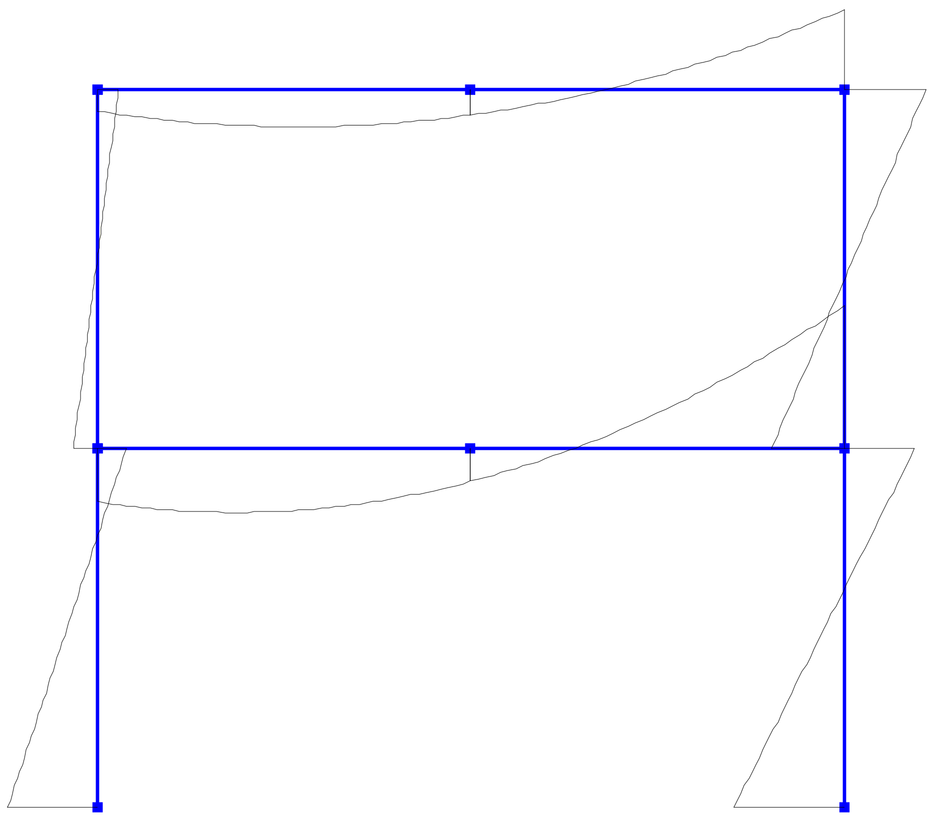


Figure 16 Moment distribution under factored gravity and horizontal forces

Load case 1 : distributed load in girders

```
% distributed load in elements 5 through 8
for el=5:6 ElemData{el}.w = [0;-0.50]; end
for el=7:8 ElemData{el}.w = [0;-0.35]; end
% there are no nodal forces for first load case
Loading = Create_Loading (Model);

% perform single linear analysis step
State = LinearStep (Model, ElemData, Loading);
… … … … …
% store element response for later post-processing
Post(1) = Structure ('post',Model,ElemData,State);
… … … … …
```

Load case 2: horizontal forces

```
% set distributed load in elements 5 through 8 from previous load case to zero
for el=5:8;  ElemData{el}.w = [0;0]; end
% specify nodal forces
Pe(2,1) = 20;
Pe(3,1) = 40;
Pe(5,1) = 20;
Pe(6,1) = 40;
Loading = Create_Loading (Model,Pe);

State = LinearStep (Model, ElemData, Loading);
… … … … …
Post(2) = Structure ('post',Model,ElemData,State);
… … … … …
```

Load case 3: support displacement

```
% zero nodal forces from previous load case and impose horizontal support
displacement
Pe = [];
Ue(1,1) = 0.2;   % horizontal support displacement
Loading = Create_Loading (Model,Pe,Ue);

State = LinearStep (Model, ElemData, Loading);
… … … … …

Post(3) = Structure ('post',Model,ElemData,State);
… … … … …
```

Load combination

```
% plot a new moment distribution for gravity and lateral force combination
% using LRFD load factors and assuming that horizontal forces are due to EQ
for el=1:Model.ne
   Post_Combi.Elem{el}.q =1.2.*Post(1).Elem{el}.q + 1.5.*Post(2).Elem{el}.q;
end

% include distributed load in elements 5 through 8 for moment diagram
for el=5:6 ElemData{el}.w = [0;-0.50]; end
for el=7:8 ElemData{el}.w = [0;-0.35]; end

% plot combined moment distribution
Create_Window(0.70,0.70);
Plot_Model(Model);
Plot_ForcDistr (Model,ElemData,Post_Combi,'Mz');
```

Figure 17 Salient analysis steps for Example 1

6.2 Example 2 - Modal analysis of linear transient response

The second example deals with the modal analysis of linear transient response of the two-story steel frame under support acceleration from a recorded motion from the Erzincan, Turkey, earthquake of 1992. The function call sequence is presented graphically in Fig. 18 and salient analysis steps from the script file Example_2 are shown in Fig. 19.

It is worth noting that a linear elastic analysis under an imposed unit support displacement is used in determining the reference acceleration vector at the free dofs of the model. A direct specification of this vector is more straightforward in this case, but the method used is more general, since it is equally valid under multi-support excitation.

For the case of support acceleration the data object **Loading** consists of fields `Uddref` and `AccHst` with fields `Time` and `Value`, as shown in Fig. 18. These fields are specified by the user, as the input file shows. There is no need for a reference force or displacement vector in this case.

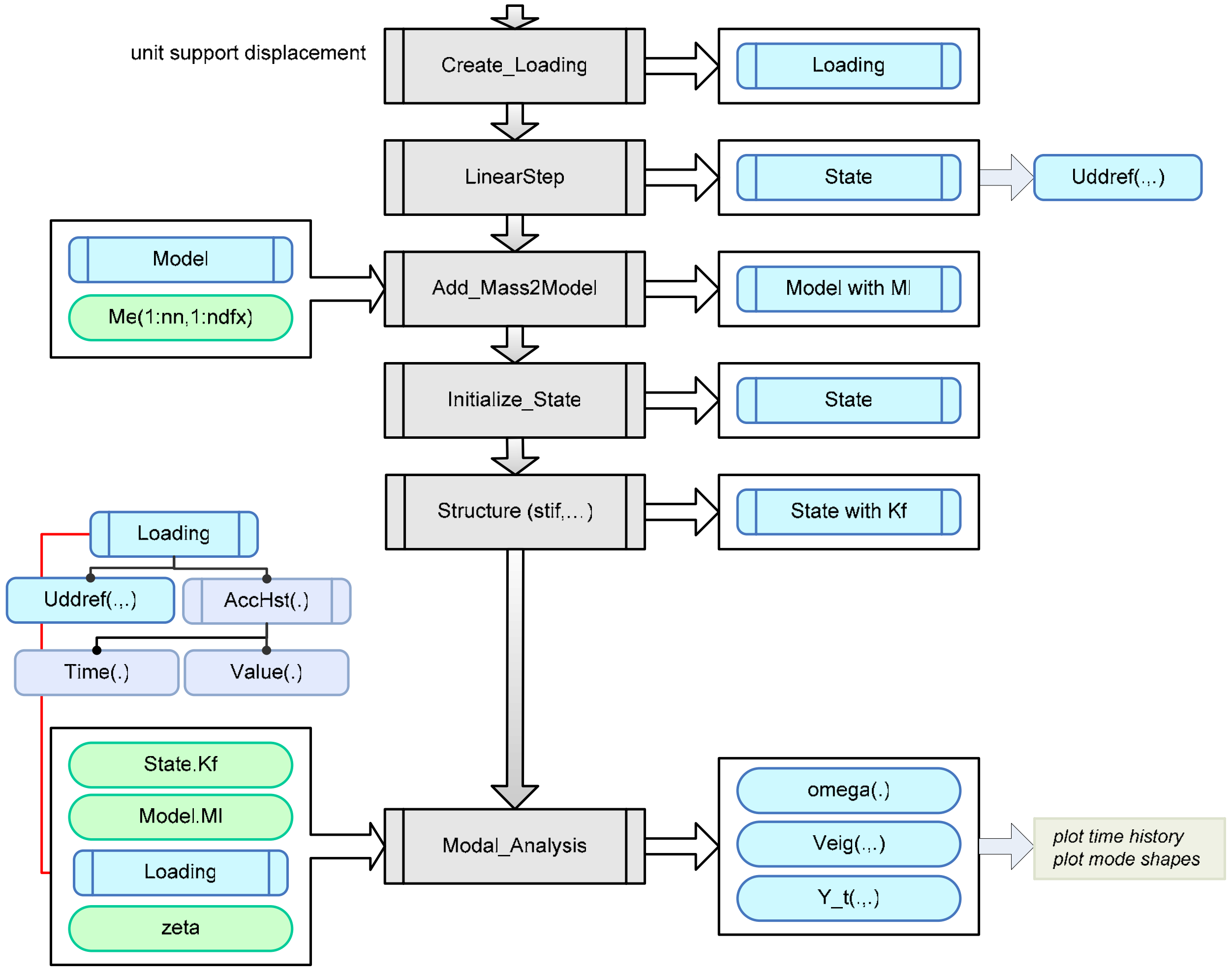


Figure 18 Function call sequence for the analysis steps of Example 2

Specify ground acceleration

```
% Reference acceleration vector by linear analysis under unit support
displacement
Ue([1,4],1)  = ones(2,1);
SupLoading = Create_Loading (Model,[],Ue); % need to include an empty array for
Pe

State = LinearStep(Model,ElemData,SupLoading);
% create actual loading vector with reference acceleration vector
Loading.Uddref = State.U(1:Model.nf);      % reference acceleration vector in
Loading
% NOTE: the above reference acceleration vector could also be specified
directly for this
% simple case of rigid body motion due to support displacement

% load ground motion history into Loading: 2% in 50 years motion from Erzincan,
Turkey
load EZ02;
Loading.AccHst(1).Time  = EZ02(1:500,1);        % Load time values into field
Time
Loading.AccHst(1).Value = EZ02(1:500,2)/2.54;   % Load acceleration values and
convert to in/sec^2
```

```
 Norm of equilibrium error = 2.066638e-012
```

Lumped mass vector

```
% define distributed mass m
m = 0.6;
Me([2 3 5:8],1) = m.*ones(6,1);
% create nodal mass vector and stored it in Model
Model = Add_Mass2Model(Model,Me);
```

Modal analysis

```
% determine stiffness matrix at initial State
State = Initialize_State(Model,ElemData);
State = Structure('stif',Model,ElemData,State);

% % number of modes to include in modal analysis
no_mod = 2;
% % modal damping ratios
zeta = 0.02.*ones(1,no_mod);

% Integration time step
Dt = 0.03;

% modal analysis
[omega, Veig, Y_t] = ModalAnalysis(State.Kf,Model.Ml,Loading,Dt,zeta,no_mod);

% global dof response history
U_t = Y_t*Veig';
```

Figure 19 Salient analysis steps for Example 2

In this example the deformed shape of the structure yields the mode shapes in Fig. 20 and the response time history of the horizontal roof displacement in shown in Fig. 21.

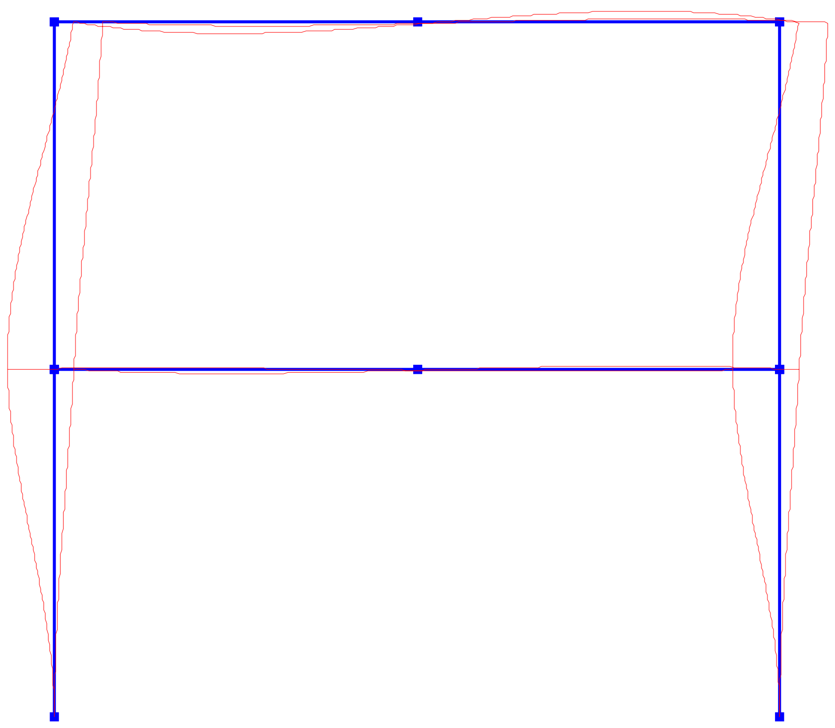


Figure 20 Mode shapes for two lowest eigenfrequencies of two-story steel frame

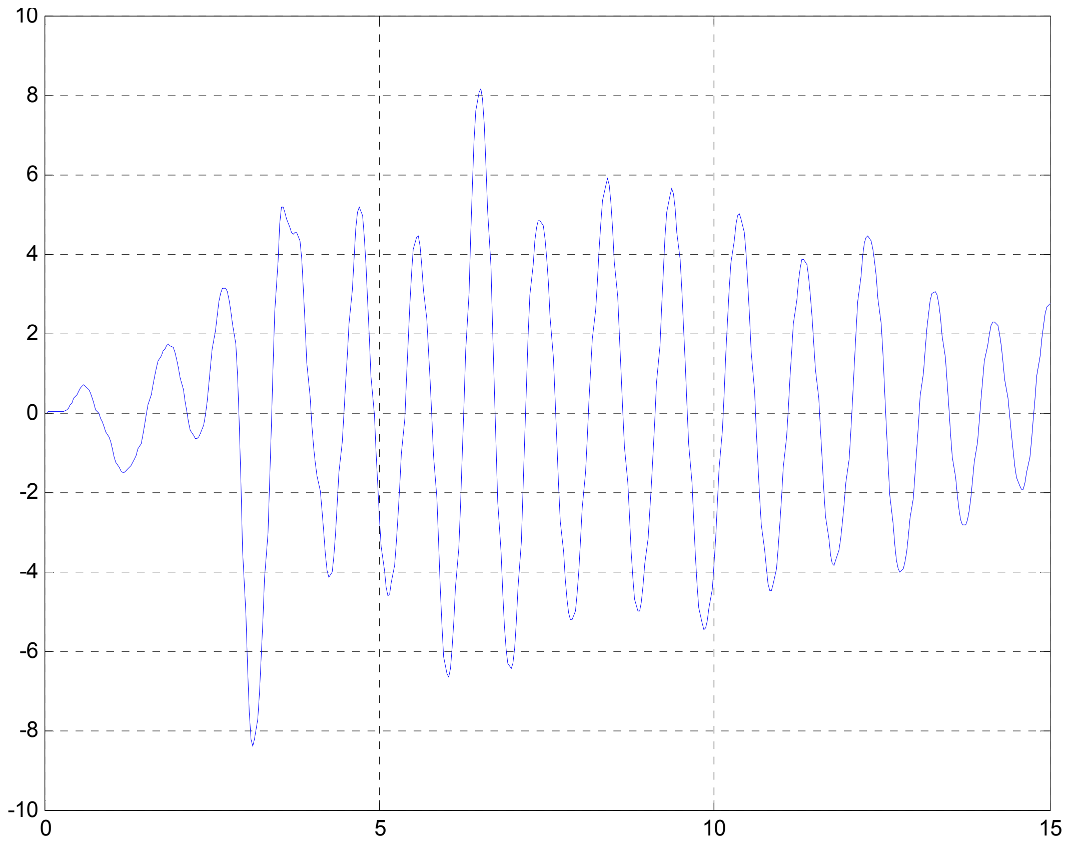


Figure 21 Time history of horizontal roof displacement of two-story steel frame

6.3 Example 3 - Time history analysis of linear transient response

The third example deals with the time history analysis of linear response of the two-story steel frame under support acceleration from a recorded motion at Erzincan, Turkey. The function call sequence is presented graphically in Fig. 22 and salient analysis steps from the script file Example_3 are shown in Fig. 23. The time history response results in Fig. 24 can be compared directly with those from Example 2. Very small discrepancies in value are due to the approximate nature of the time integration strategy for the given time step.

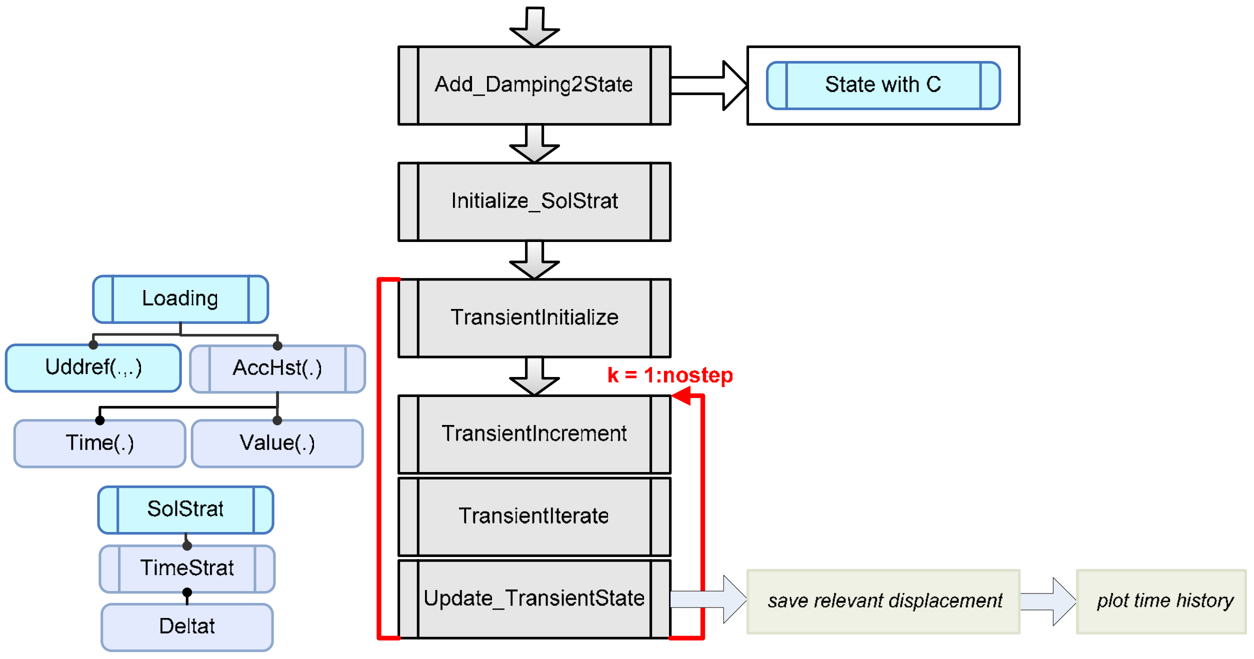


Figure 22 Function call sequence for time history analysis

The steps preceding the call to function `Add_Damping2State` in Fig. 22 are identical to those of Example 2 in Fig. 18 before the call to function `Modal_Analysis`. In addition to data object **Loading**, which has the same fields as in Example 2, the time history analysis requires specification of the time step for the numerical integration of transient response in data object **SolStrat**. This information is specified in field `Deltat` of field `TimeStrat` of **SolStrat**, as shown in Fig. 22. The function `Initialize_SolStrat` uses by default Newmark's constant acceleration method for the time integration. The user can modify the parameters of the method in field `Param` of field `TimeStrat` of **SolStrat.**

Specify ground acceleration

```
% Reference acceleration vector by linear analysis under unit support
displacement
Ue([1,4],1)   = ones(2,1);
SupLoading = Create_Loading (Model,[],Ue); % need to include an empty array for
Pe

State = LinearStep(Model,ElemData,SupLoading);
% create Loading data object
Loading = Create_Loading(Model);
% insert reference acceleration vector into Loading
Loading.Uddref = State.U(1:Model.nf);      % reference acceleration vector in
Loading
… … … … …
```

Lumped mass vector

```
… … … … …
```

Rayleigh damping matrix

```
% re-initialize State (zero support displacement this time)
State = Initialize_State(Model,ElemData);
% determine initial stiffness matrix
State = Structure('stif',Model,ElemData,State);
% specify modal damping ratios
zeta = [0.02 0.02];
% specify modes
mode  = [1 2];
State = Add_Damping2State ('Caughey',Model,State,zeta,mode);
```

Transient time history analysis

```
% initialize solution strategy parameters
SolStrat = Initialize_SolStrat;
% specify time interval for numerical integration
SolStrat.TimeStrat.Deltat = 0.01;

… … … … …
Tmax = Loading.AccHst(1).Time(end);

% initialize analysis parameters for transient response
State = TransientInitialize(Model,ElemData,Loading,State);
```

Time stepping until maximum specified time Tmax

```
while (State.Time < Tmax)
   [State,SolStrat] = TransientIncrement
(Model,ElemData,Loading,State,SolStrat);
   [State,SolStrat] = TransientIterate
(Model,ElemData,Loading,State,SolStrat);
   if (SolStrat.ConvFlag)
      State = Update_TransientState (Model,ElemData,State,SolStrat);
      % store displacement value at pltDOF
      pc = pc+1;
      Disp (pc) = State.U(pltDOF);
   else
      break
   end
end
```

Figure 23 Salient analysis steps for Example 3

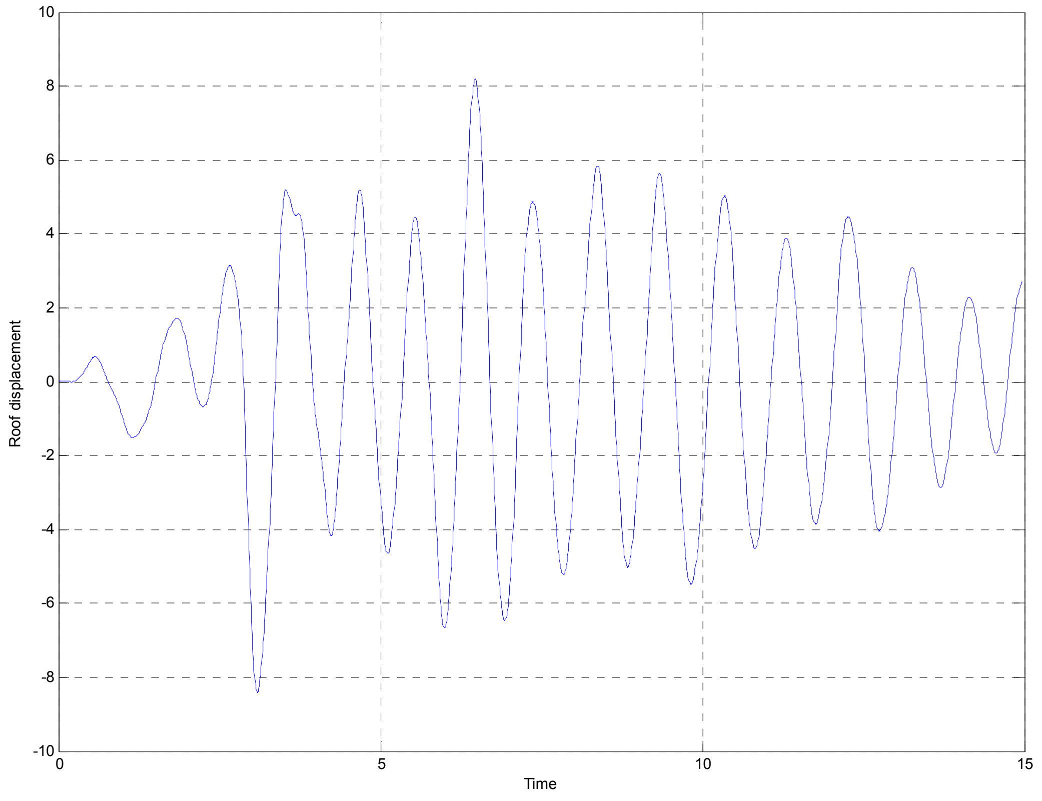


Figure 24 Time history of horizontal roof displacement of two-story steel frame

6.4 Example 4 – Nonlinear static analysis with different load histories

The fourth example is the first of four examples for the simulation of the nonlinear static (push-over) analysis of the two story frame under constant gravity loads and a horizontal force pattern that is incremented until the ultimate lateral shear force is reached. In this example the nonlinear response of the elements is described by the one-component model with properties specified in script file `SimpleNLElemData`.

To simulate the loading history two separate load histories are specified in this example: one for the gravity loads, which consist of uniformly distributed element loads and are assigned load history number 1, and one for the horizontal forces, which are assigned load history number 2. This approach is certainly more straightforward, but has the limitation that an automatic load control strategy for the lateral force pattern is not possible.

The function call sequence for the nonlinear push-over analysis is shown in Fig. 25, where the salient data specification steps are highlighted: specification of element loading and load history number in **ElemData,** load pattern number specification for horizontal forces in Loading that matches the load history number in `FrcHst`, specification of load histories as time-value pairs of field `FrcHst` in data object **Loading,** and specification of pseudo-time increment `Deltat` under field `IncrStrat` of **SolStrat** data object.

It is worth noting that the creation of a new **Post** object in every load step in Fig. 25 permits the subsequent post-processing of the entire incremental analysis at the expense of storage.

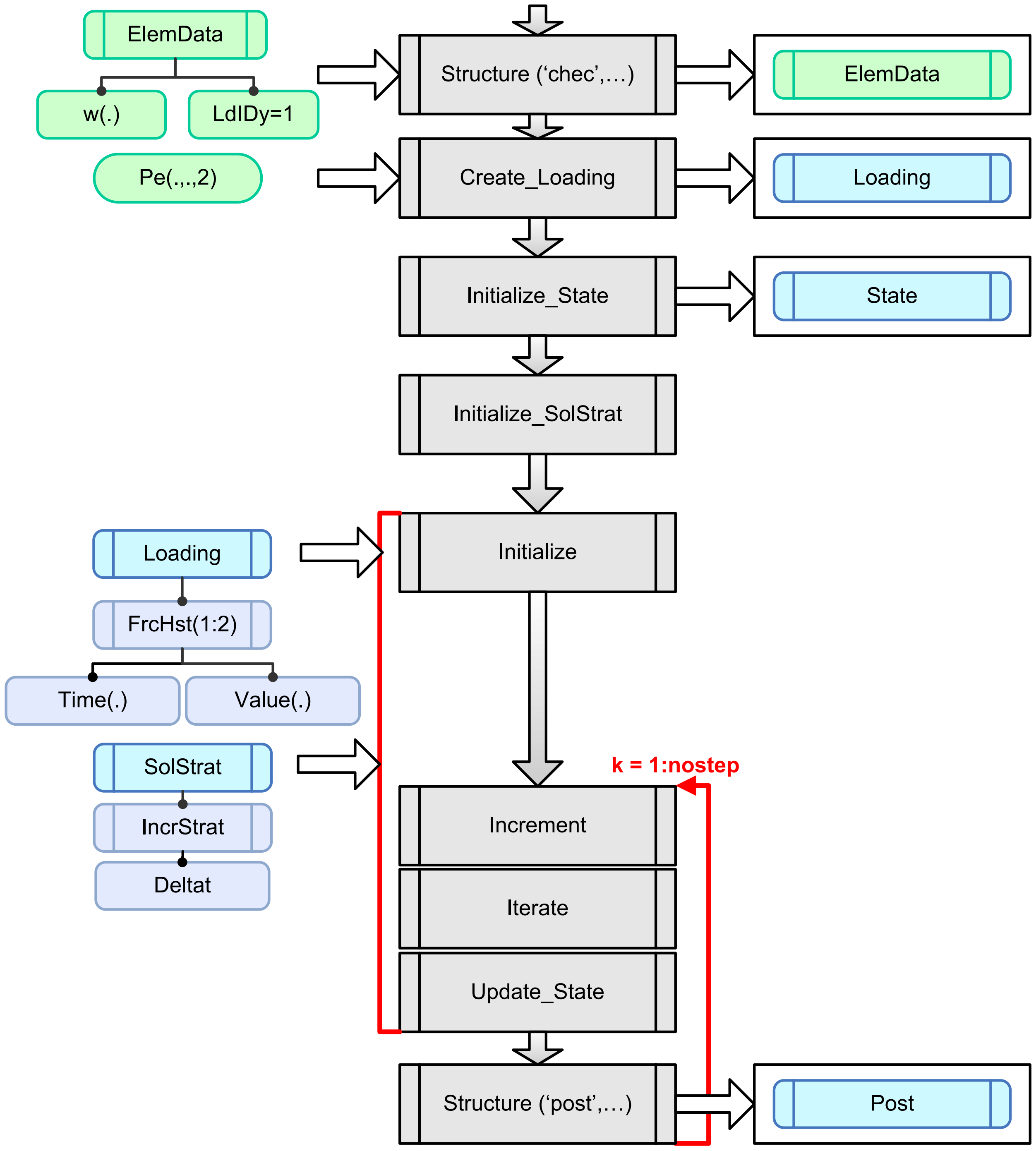


Figure 25 Function call sequence for load specification and analysis steps in Example 4

Fig. 27 shows the relation between lateral load factor and horizontal roof displacement (also known as push-over curve) of the 2-story steel frame. Fig. 28 shows the deformed shape with a magnification factor of 10 and the plastic hinge locations of the 2-story frame at the last load step (near lateral strength).

Distributed element loads with load pattern number 1

```
for el=5:6
   ElemData{el}.w     = [0;-0.50];
   ElemData{el}.LdIdy = 1;
end
for el=7:8
   ElemData{el}.w = [0;-0.35];
   ElemData{el}.LdIdy = 1;
end
```

Horizontal forces with laod pattern number 2

```
% specify nodal forces values in first two columns, pattern number in third
Pe(2,1,2) =  20;     % force at node 2 in dof 1 (force in global X) for load
pattern 2
Pe(3,1,2) =  40;
Pe(5,1,2) =  20;
Pe(6,1,2) =  40;     % force at node 6 in dof 1 (force in global X) for load
pattern 2
Loading = Create_Loading (Model,Pe);
```

Applied force time histories

```
Deltat = 0.10;
Tmax   = 2.00;

Loading.FrcHst(1).Time  = [0;Deltat;Tmax];
% force pattern 1 is applied over Deltat and then kept constant
Loading.FrcHst(1).Value = [0;1;1];
Loading.FrcHst(2).Time  = [0;Deltat;Tmax];
% force pattern 2 is linearly rising between Deltat and Tmax up to value of 2.8
Loading.FrcHst(2).Value = [0;0;2.8];
```

Incremental analysis by pseudo-time incrementation

```
% initialize State
State = Initialize_State(Model,ElemData);
% initialize default SolStrat parameters
SolStrat = Initialize_SolStrat;
% specify pseudo-time step increment (does not have to be the same as Deltat,
smaller value
% results in more steps to reach end of analysis)
SolStrat.IncrStrat.Deltat = 0.10;
% initialize analysis parameters
[State SolStrat] = Initialize(Model,ElemData,Loading,State,SolStrat);
… … … … … … …
```

Load incrementation until maximum specified time Tmax (pseudo-time stepping)

```
while (State.Time < Tmax-10^3*eps)
   [State SolStrat] = Increment(Model,ElemData,Loading,State,SolStrat);
   [State SolStrat] = Iterate  (Model,ElemData,Loading,State,SolStrat);
   if (SolStrat.ConvFlag)
      State = Update_State(Model, ElemData, State);
   else
      break
   end
   pc = pc+1;
   Post(pc) = Structure ('post',Model,ElemData,State);
end
```

Figure 26 Salient steps for load specification and analysis for Example 4

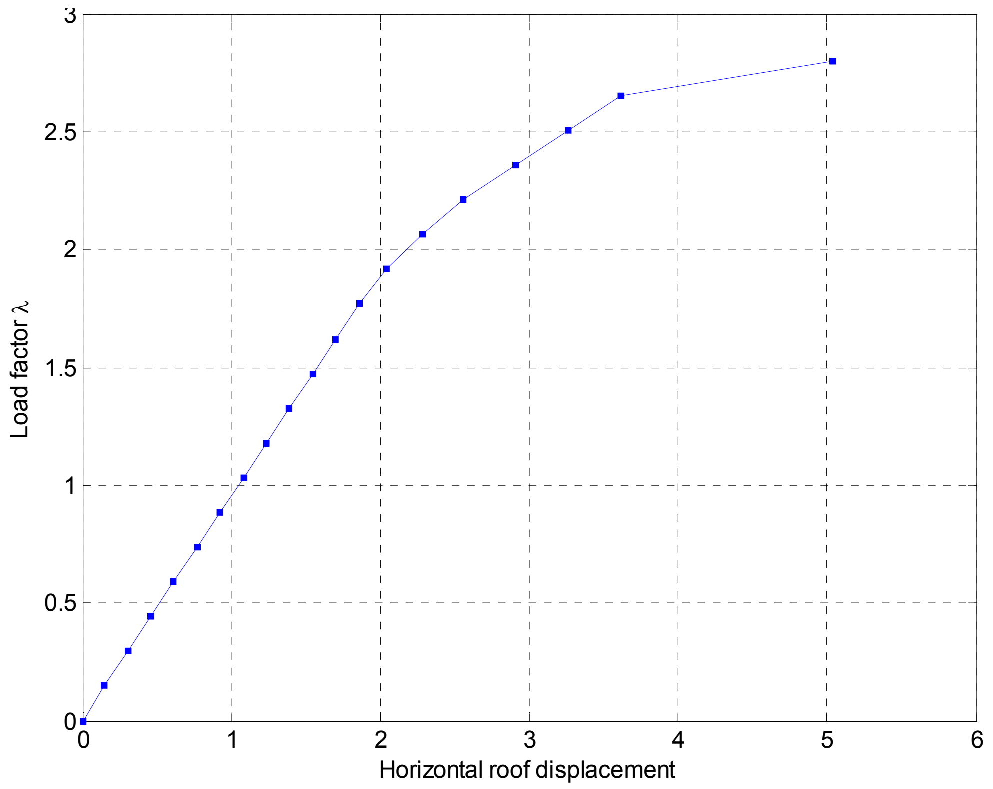


Figure 27 Load factor vs. horizontal roof displacement for 2-story steel frame

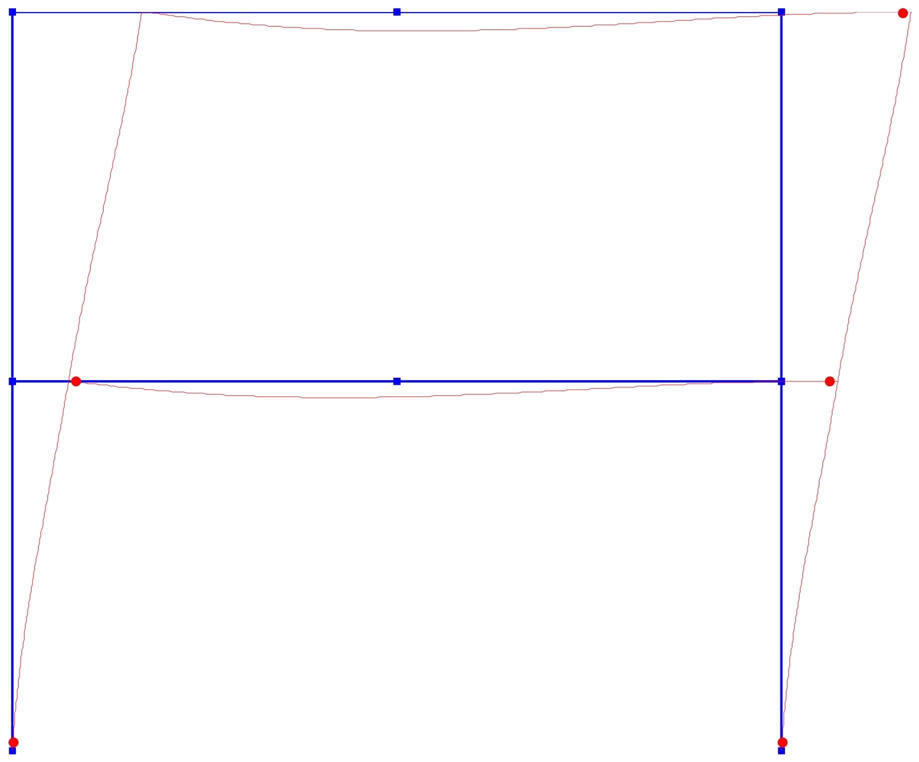


Figure 28 Deformed shape and plastic hinge locations of 2-story frame at last load step

6.5 Example 5 – Nonlinear static analysis with loading sequence and control

The specification of two separate load histories for gravity and horizontal forces in the preceding example does not allow for automatic load control strategy, which is essential for the determination of the lateral strength of the frame and the post-peak response. Since load control is only possible with the application of a single force pattern, the gravity loads are applied first in this example, followed by the analysis for the horizontal forces. The function call sequence for the loading sequence is shown in Fig. 29.

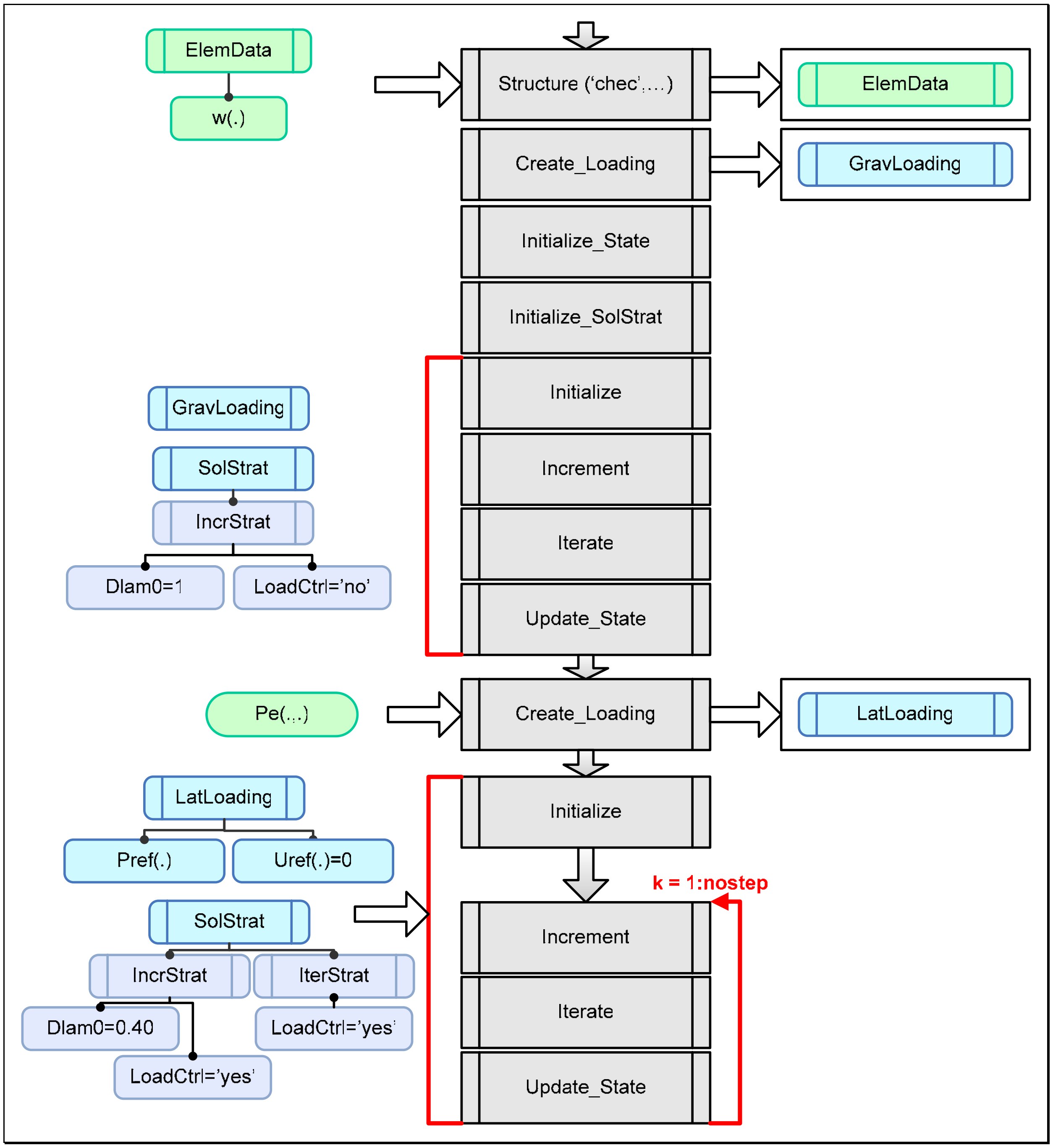


Figure 29 Function call sequence for gravity load and horizontal force loading sequence

The gravity loads consist only of distributed element loads specified in **ElemData**. The function call to function `Create_Loading` without input arguments generates a **Loading** data object with zero nodal reference force and displacement vectors. For easier identification this data object is called **GravLoading** in this example. After initializing the **State** and **SolStrat** data object a single load step is applied with the sequence of function calls `Initialize`, `Increment`, `Iterate` and `Update_State` (under the assumption that convergence is achieved during equilibrium iterations). Following the completion of the load step it is possible to specify the lateral forces and call function `Create_Loading` to generate a new loading data object, called **LatLoading** in this example. It contains the lateral forces in reference vector `Pref`. The call to function `Initialize` resets the load factor and stores the resisting force vector at the time of the call as initial force vector. This allows the application of load sequences in separate sequential analyses. During the second analysis it is possible to modify **SolStrat** parameters and include the load control option, as Fig. 29 shows. With this option there is automatic load control for the horizontal force pattern, so that the push-over analysis can be continued past the displacement value at attainment of lateral strength, as Fig. 30 shows.

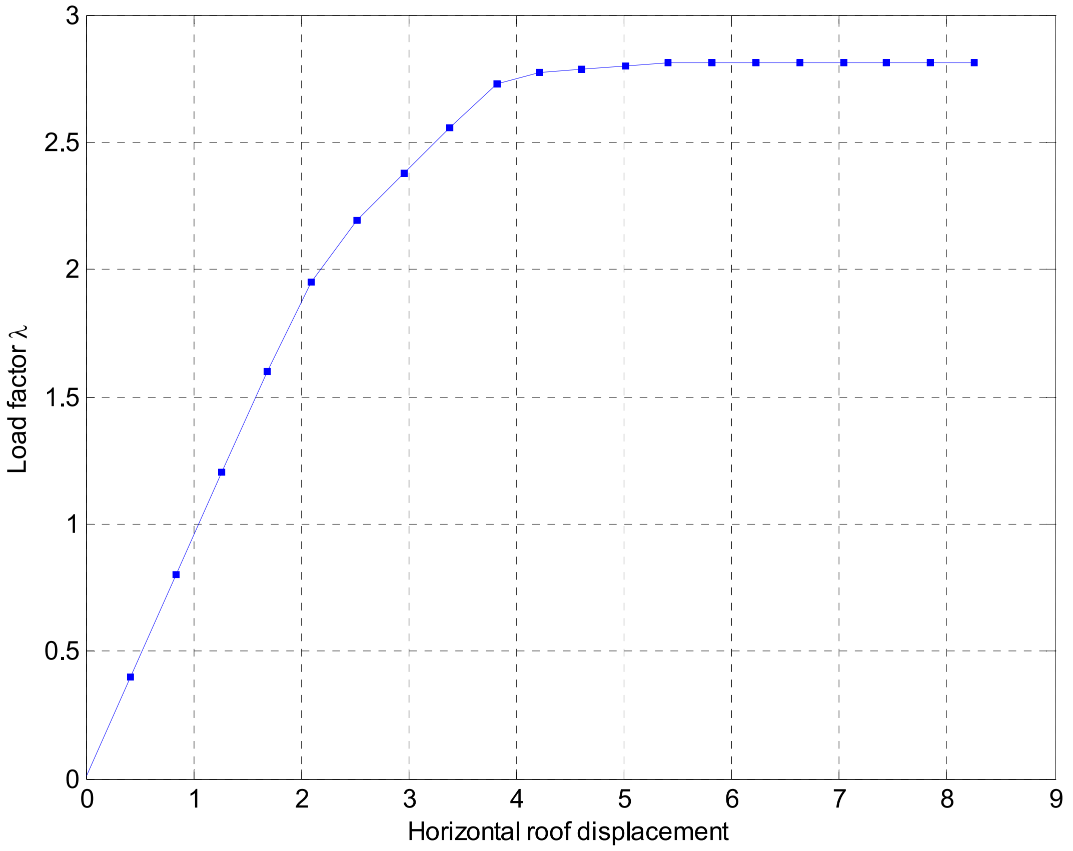


Figure 30 Load factor vs. horizontal roof displacement for 2-story steel frame

Element properties

```
SimpleNLElemData
% print element properties (optional)
Structure ('data',Model,ElemData);
```

Loading (distributed loads only)

```
% define loading
for el=5:6 ElemData{el}.w = [0;-0.50]; end
for el=7:8 ElemData{el}.w = [0;-0.35]; end
GravLoading = Create_Loading (Model);
```

Incremental analysis for distributed element loading (single load step)

```
% initialize state
State = Initialize_State(Model,ElemData);
% initialize solution strategy parameters
SolStrat = Initialize_SolStrat;
% specify initial load increment (even though it is the same as the default
value and could be omitted)
SolStrat.IncrStrat.Dlam0 = 1;
% initialize analysis sequence
[State SolStrat] = Initialize(Model,ElemData,GravLoading,State,SolStrat);
% apply load in one increment
[State SolStrat] = Increment(Model,ElemData,GravLoading,State,SolStrat);
% perform equilibrium iterations (we assume that convergence will occur!)
[State SolStrat] = Iterate  (Model,ElemData,GravLoading,State,SolStrat);
% update State
State = Update_State(Model,ElemData,State);
… … … … …
```

2. Loading in sequence: horizontal forces

```
% specify nodal forces
Pe(2,1) =  20;
Pe(3,1) =  40;
Pe(5,1) =  20;
Pe(6,1) =  40;
LatLoading = Create_Loading (Model,Pe);
```

Incremental analysis for horizontal force pattern (load control is on)

```
% (gravity forces are left on by not initializing State!)
% specify initial load increment and turn load control on
SolStrat.IncrStrat.Dlam0 = 0.40;
SolStrat.IncrStrat.LoadCtrl = 'yes';
SolStrat.IterStrat.LoadCtrl = 'yes';
% specify number of load steps
nostep = 20;
% initialize analysis sequence
[State SolStrat] = Initialize(Model,ElemData,LatLoading,State,SolStrat);

% for specified number of steps, Increment, Iterate and Update_State (we assume
again convergence!)
for j=1:nostep
   [State SolStrat] = Increment(Model,ElemData,LatLoading,State,SolStrat);
   [State SolStrat] = Iterate  (Model,ElemData,LatLoading,State,SolStrat);
   State = Update_State(Model,ElemData,State);
   k = k+1;
   Post(k) = Structure ('post',Model,ElemData,State);
   % print results to output file
   Structure('prin',Model,ElemData,State);
end
```

Figure 31 Salient steps for load specification and analysis for Example 5

6.6 Example 6 – Nonlinear static analysis with nonlinear geometry

This example is the same as example 5, except for the fact that nonlinear geometry effects are included for the frame columns. To accentuate the nonlinear geometry effect additional vertical forces are applied at the top of each column element of the model. In this case it is very important to clear the applied force vector before specifying the lateral forces for the second loading case. The salient steps of this process are shown in Fig. 32. The subsequent steps are identical to the steps following the lateral load definition in Fig. 31.

1. Loading (distributed loads and vertical forces on columns)

```
% define loading
for el=5:6 ElemData{el}.w = [0;-0.50]; end
for el=7:8 ElemData{el}.w = [0;-0.35]; end

Pe(2,2) =  -200;
Pe(3,2) =  -400;
Pe(5,2) =  -200;
Pe(6,2) =  -400;
GravLoading = Create_Loading (Model,Pe);
```

Specify nonlinear geometry option for columns

```
for el=1:4 ElemData{el}.Geom = 'PDelta'; end
```

Incremental analysis for distributed element loading (single load step)

```
% initialize state
State = Initialize_State(Model,ElemData);
% initialize solution strategy parameters
SolStrat = Initialize_SolStrat;
% specify initial load increment (even though it is the same as the default
value and could be omitted)
SolStrat.IncrStrat.Dlam0 = 1;
% initialize analysis sequence
[State SolStrat] = Initialize(Model,ElemData,GravLoading,State,SolStrat);
% apply load in one increment
[State SolStrat] = Increment(Model,ElemData,GravLoading,State,SolStrat);
% perform equilibrium iterations (we assume that convergence will occur!)
[State SolStrat] = Iterate  (Model,ElemData,GravLoading,State,SolStrat);
% update State
State = Update_State(Model,ElemData,State);
% determine resisting force vector
State   = Structure ('forc',Model,ElemData,State);
… … … … …
```

2. Loading in sequence: horizontal forces

```
% specify nodal forces
% !!!! IMPORTANT!!!! CLEAR PREVIOUS PE
clear Pe;
Pe(2,1) =  20;
Pe(3,1) =  40;
Pe(5,1) =  20;
Pe(6,1) =  40;
LatLoading = Create_Loading (Model,Pe);
```

Incremental analysis for horizontal force pattern (load control is switched on)

Figure 32 Salient steps for load specification and gravity load analysis for Example 6

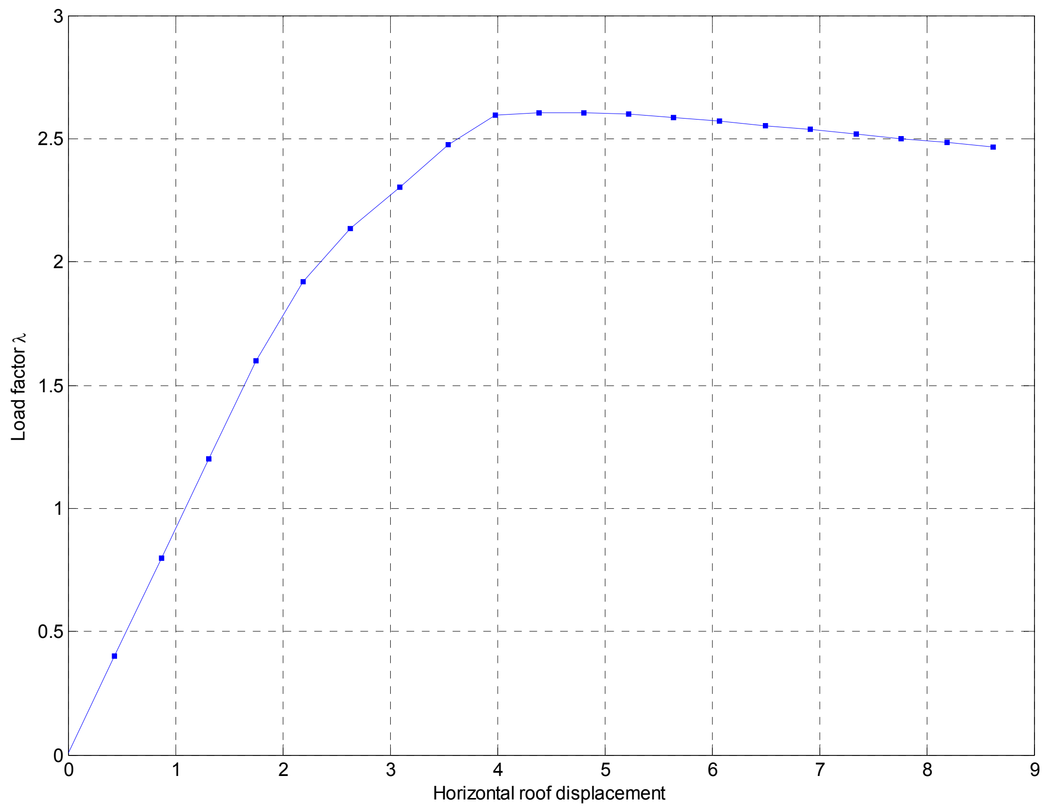


Figure 33 Load factor vs. horizontal roof displacement for P-Δ geometry

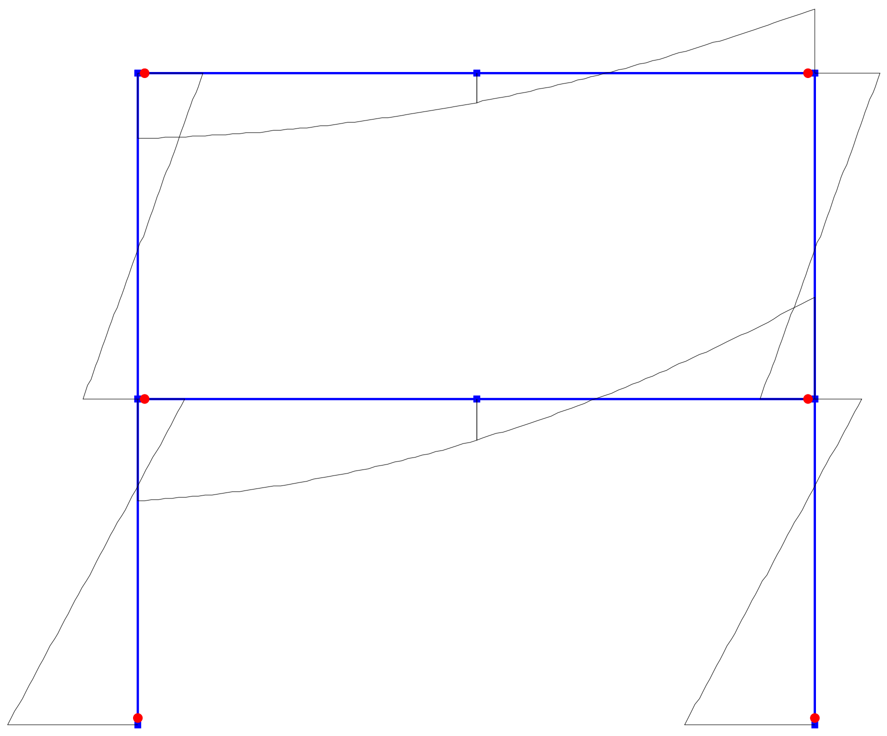


Figure 34 Moment distribution and plastic hinge location at last load step

6.7 Example 7 – Nonlinear static analysis with distributed inelasticity element

Example 7 is the same as example 6 except for the use of the distributed inelasticity frame element with force formulation for modeling the girders and columns of the two-story steel frame. Thus, the element properties are defined by the script file `DistrInelElemData` instead of `SimpleNLElemData`. Figs. 35 and 36 show the push-over curve under nonlinear geometry and the curvature distribution which confirms the plastic hinge locations of Fig. 34.

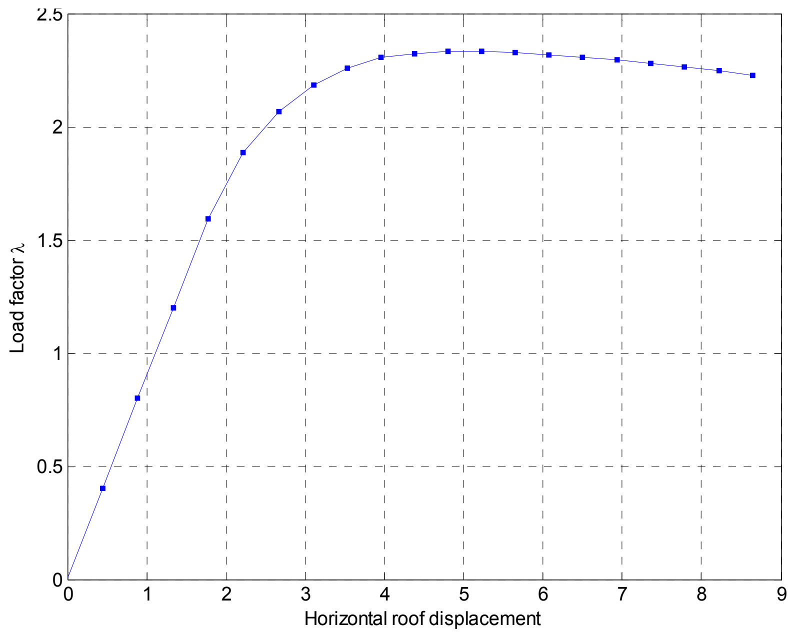


Figure 35 Load factor vs. horizontal roof displacement for distributed inelasticity frame element under P-Δ geometry

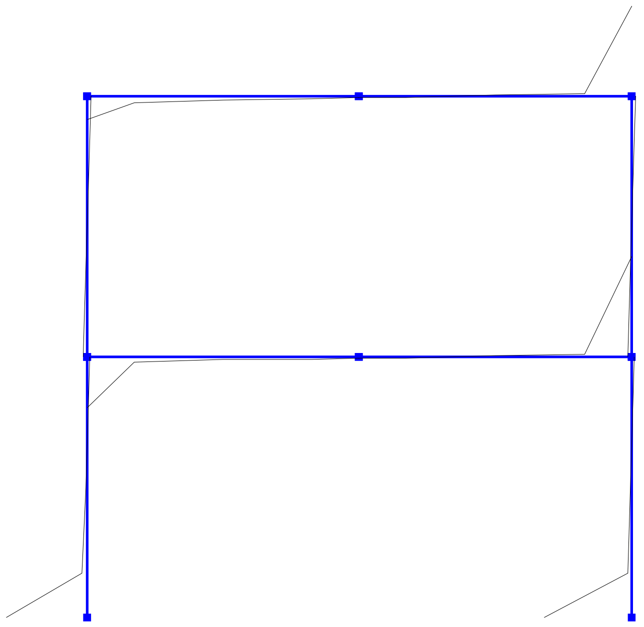


Figure 36 Curvature distribution at last load step of push-over curve of Fig. 35

6.8 Example 8 – Nonlinear transient response analysis with distributed inelasticity element

The last example covers the nonlinear transient response analysis of the 2-story steel frame with distributed inelasticity frame elements under constant gravity load and support acceleration. Because the analysis steps are a combination of steps from similar examples, the comments are brief. Fig. 37 shows the sequence of function calls for loading and analysis definition.

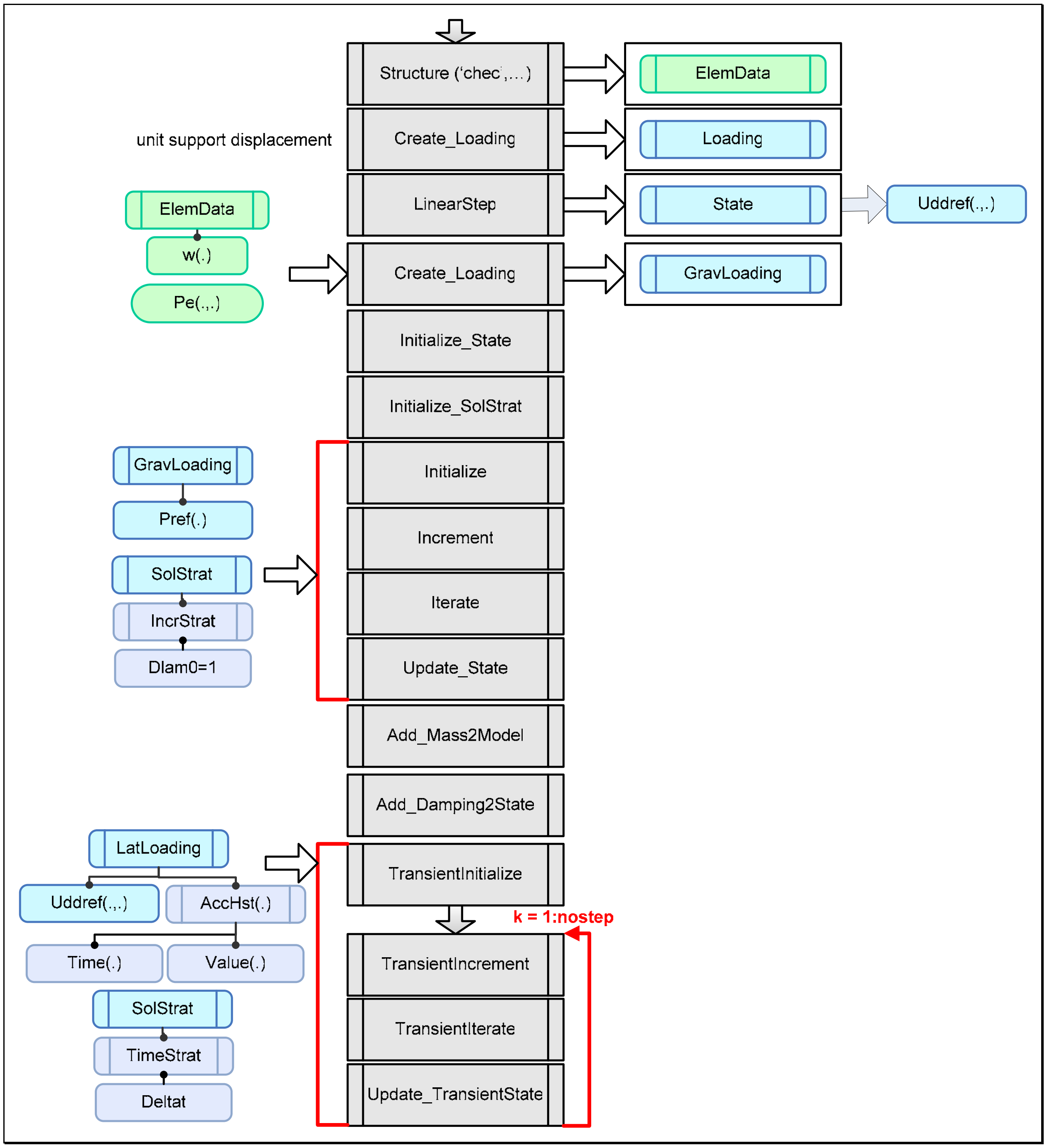


Figure 37 Function call sequence for load specification and analysis steps in Example 8

At first, a linear analysis under imposed support displacement is used to generate the vector of reference accelerations at the free dofs for support excitation (similar to Example 2). Then, the gravity loading consisting of uniformly distributed element loads and additional vertical forces at the top of column elements is specified to generate the loading data object **GravLoading**. After initializing **State** and **SolStrat** this loading is applied on the structural model in a single load step (similar to example 5). After adding a lumped mass vector and a Rayleigh damping matrix, a sequence of transient steps is performed for the lateral loading consisting of the imposed acceleration history and the reference acceleration vector at the free dofs. During transient response the gravity loads are maintained constant. The time step of numerical integration is specified in field `Deltat` of field `TimeStrat` of the **SolStrat** data object. This sequence of steps is identical to those for a linear transient response analysis in example 3 (consult also Fig. 22). The roof displacement time history is shown in Fig. 38. It is clear that a large excursion into the inelastic range takes place approximately 4 seconds into the response.

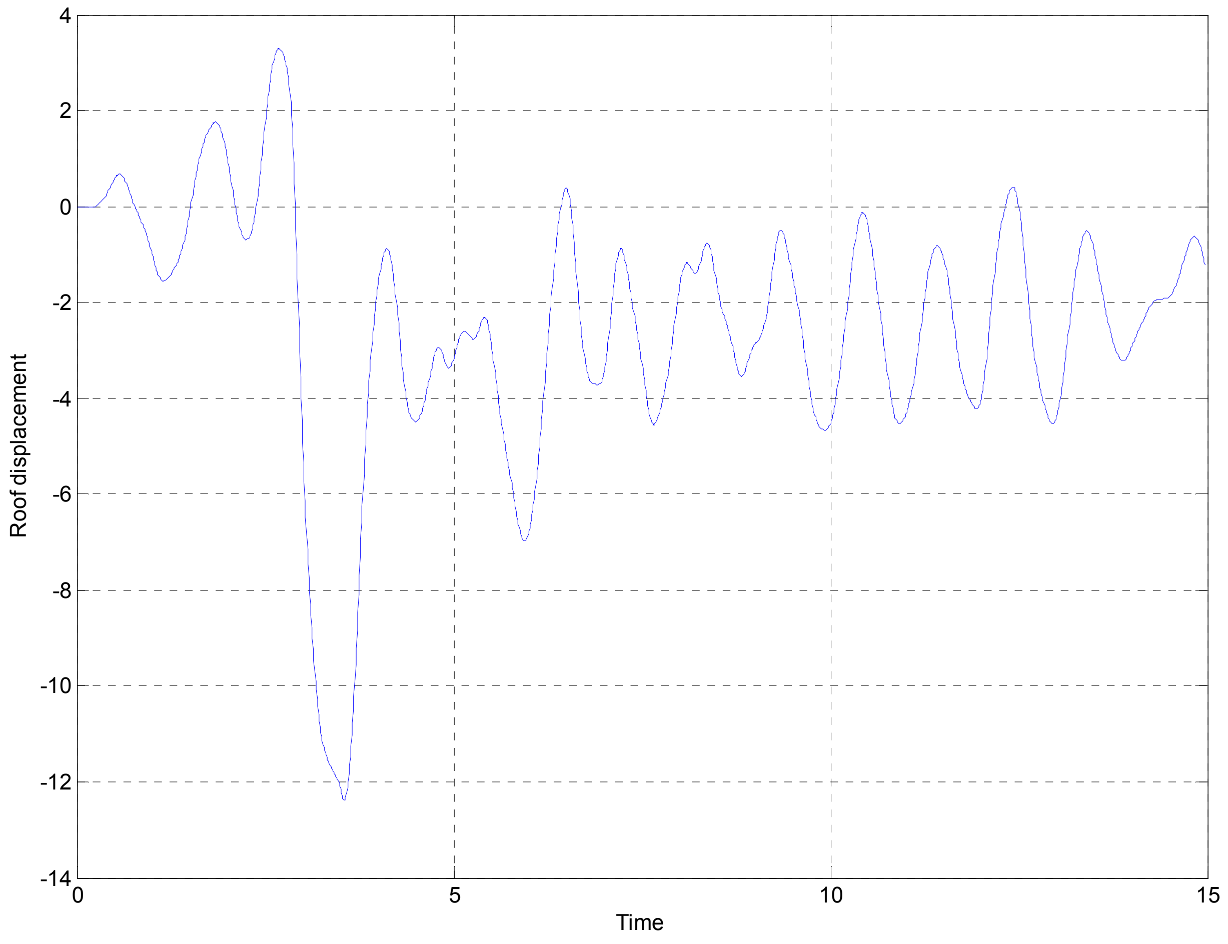


Figure 38 Time history of horizontal roof displacement of two-story steel frame under nonlinear transient response with distributed inelasticity element

Appendix

Model definition of 2-story, one bay 3d braced frame

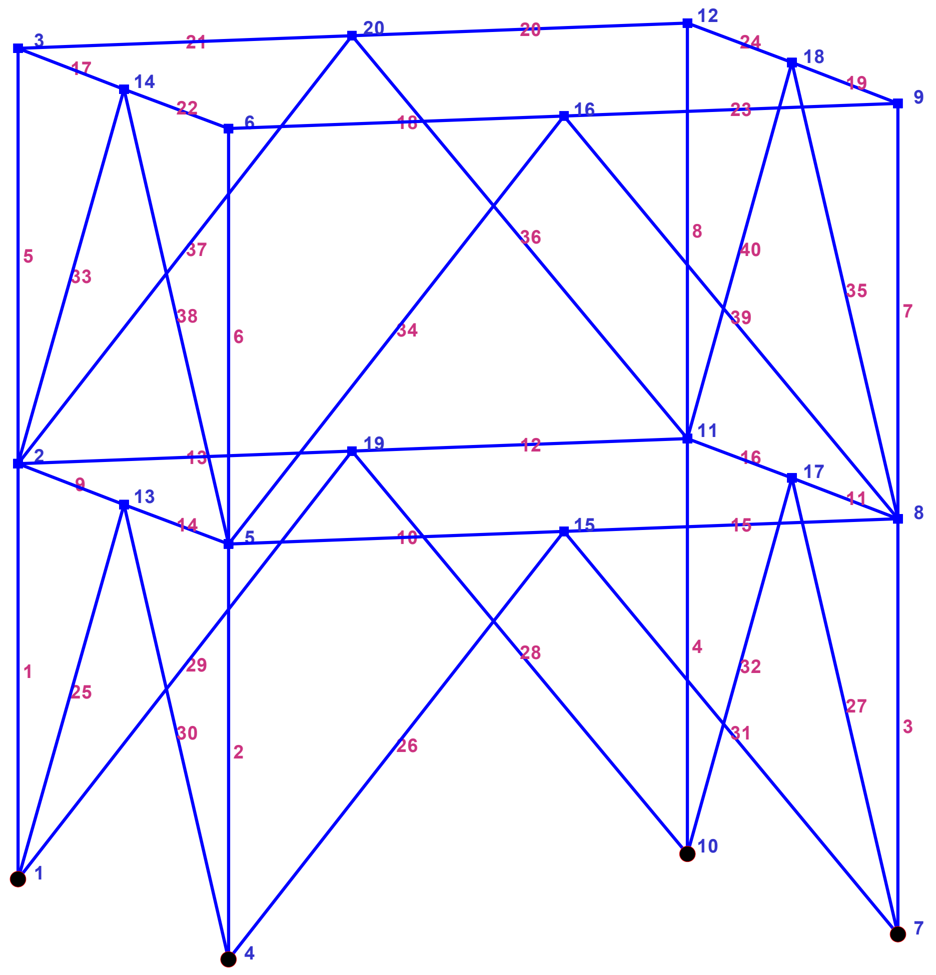


Figure A1 – 2-story, one bay 3d braced frame model

Create Model

```
% all units in kip and inches
```

Node coordinates (in feet!)

```
height = 12;
width  = 20;
XYZ([ 1: 3],:) =  [zeros(3,1)         zeros(3,1)        linspace(0,2*height,3)'];
XYZ([ 4: 6],:) =  [width*ones(3,1)    zeros(3,1)        linspace(0,2*height,3)'];
XYZ([ 7: 9],:) =  [width*ones(3,1)    width*ones(3,1)   linspace(0,2*height,3)'];
XYZ([10:12],:) =  [zeros(3,1)         width*ones(3,1)   linspace(0,2*height,3)'];
XYZ([13:14],:) =  [width/2*ones(2,1)  zeros(2,1)        linspace(height,2*height,2)'];
XYZ([15:16],:) =  [width*ones(2,1)    width/2*ones(2,1) linspace(height,2*height,2)'];
XYZ([17:18],:) =  [width/2*ones(2,1)  width*ones(2,1)   linspace(height,2*height,2)'];
XYZ([19:20],:) =  [zeros(2,1)         width/2*ones(2,1) linspace(height,2*height,2)'];
% convert coordinates to inches
XYZ = XYZ.*12;
```

Connectivity array

```
no_columns = 4;
for i = 0:3
    n = i+1;
    % first story columns
    CON(n,:)                = num2cell([3*i+1  3*i+2],2);
    % second story columns
    CON(n + no_columns,:)   = num2cell([3*i+2  3*i+3],2);
    % first story beams
    CON(n + 2*no_columns,:) = num2cell([3*i+2  2*i+13],2);
    CON(n + 3*no_columns,:) = num2cell([3*i+2  2*i+11],2);
    CON{13} = [2 19];
    % second story beams
    CON(n + 4*no_columns,:) = num2cell([3*i+3  2*i+14],2);
    CON(n + 5*no_columns,:) = num2cell([3*i+3  2*i+12],2);
    CON{21} = [3 20];
    % first floor braces
    CON(n + 6*no_columns,:) = num2cell([3*i+1  2*i+13],2);
    CON(n + 7*no_columns,:) = num2cell([3*i+1  2*i+11],2);
    CON{29} = [1 19];
    % second floor braces
    CON(n + 8*no_columns,:) = num2cell([3*i+2  2*i+14],2);
    CON(n + 9*no_columns,:) = num2cell([3*i+2  2*i+12],2);
    CON{37} = [2 20];
end
```

Boundary conditions

```
% (specify only restrained dof's)
BOUN( 1,:) = ones(1,3);     % (1 = restrained,  0 = free)
BOUN( 4,:) = ones(1,3);
BOUN( 7,:) = ones(1,3);
BOUN(10,:) = ones(1,3);
```

Element type

```
[ElemName{ 1:24}] = deal('Lin3dFrm_NLG');     % 3d linear frame element
[ElemName{25:40}] = deal('LinTruss_NLG');     %    linear truss element
```

Create model data structure

```
Model = Create_Model(XYZ,CON,BOUN,ElemName);
```

Display model and show node/element numbering (optional)

```
figA = Create_Window (0.70,0.70);  % open figure window named figA
Plot_Model   (Model);              % plot model (optional)
Label_Model  (Model);              % label model (optional)
```

Figure A2 – Model definition of 3d braced frame in FEDEASLab