FEDEASLab
A Matlab© Toolbox for Nonlinear Structural Response Simulations

Filip C. Filippou
Department of Civil and Environmental Engineering
University of California, Berkeley

Acknowledgements

- Professor **Robert L. Taylor** for preceding FEDEASLab by some 20 years with FEAP and inspiring me to adapt it to modern tools
- Professor **Gregory L. Fenves** for continuous constructive criticism and inspiration for object-oriented programming
- former Ph.D. student **Dr. Remo Magalhaes de Souza** for being there at the beginning and contributing to the organization of nonlinear geometry and nonlinear solution strategies
- Graduate student **Margarita Constantinides** for the figures of data and function organization and for assistance with examples and documentation
- Documentation and validation examples of FEDEASLab supported since Dec. 2003 by NEESgrid system integrator with NSF grant (BF Spencer, PI)

- Ultimately, I have written or rewritten almost every line of code and I am to blame for bugs and shortcomings

Motivation

- Matlab has become a ubiquitous tool for scientific computation with extensive libraries for numerical analysis, optimization, signal processing, etc.
- Matlab interfaces with data acquisition boards for experimental testing
- Matlab is increasingly used in the instruction of scientific programming in undergraduate courses in engineering schools

- Need for toolbox for structural analysis courses (linear and nonlinear structural analysis, dynamics, finite element analysis)
- Need for toolbox for element and material model development in individual graduate research studies
- Need for toolbox for simulation and evaluation of seismic response of small structural models in course on earthquake resistant design
- Need for toolbox for teaching, concept development, simulation and experimentation within NEESgrid

Objectives of FEDEASLab

- Education: functions should illustrate principles of structural analysis and be succinct, elegant and comprehensible
- Education: start with a simple core and gradually evolve complex applications with functions that enrich basic data objects and add new capabilities
- Education: transparent access to structure of data objects, modularity

- Research: implement new solution strategies, element and material models
- Research: add new capabilities (e.g. pre- and post-processing, visualization)
- Research: facilitate concept development and validation, interface with data boards for experimental-analytical integration
- Research: capitalize on vast array of Matlab toolboxes

Objectives achieved through thorough design, data encapsulation and function modularity

FEDEASLab and OpenSees

- Concept similarities but very different objectives
  - modularity, data encapsulation, leveraging of available technologies and developments
  - share many element, section and material models; exchange of solution strategies
- FEDEASLab uses Matlab with built-in scripting; OpenSees is programmed in C++ and uses Tcl for scripting
  - FEDEASLab is very easy to learn and use
  - Element and material development and validation time is short
  - Non-specialists can develop and check models on their own
- FedeasLab is slow in execution, because Matlab is interpretive; OpenSees is fast, because it is compiled
  - FedeasLab will tax your patience for large scale simulations; OpenSees should be used for the purpose
- Bridge from FedeasLab to OpenSees
  - FedeasLab functions can be compiled with Matlab Compiler and integrated to OpenSees (benefit remains to be tested)
  - Port FEDEASLab element and material models to OpenSees after testing

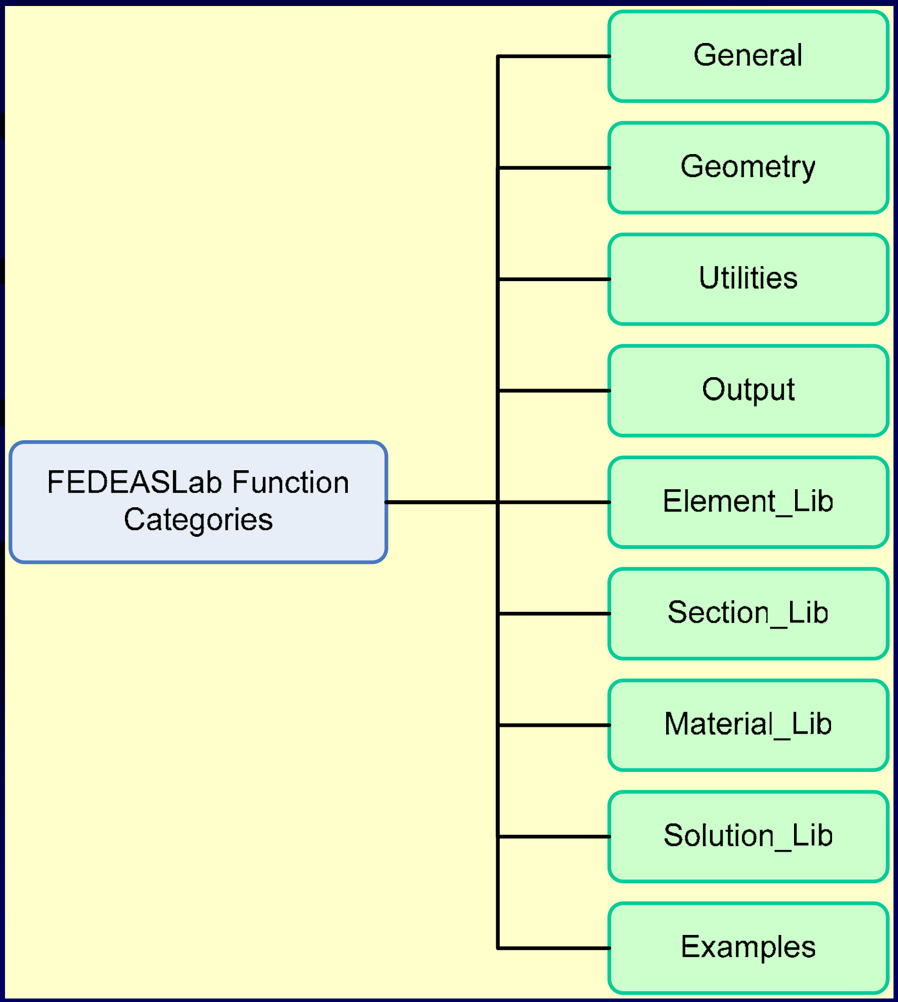
Comparison of FedeasLab to FEMLab, SDTools etc.

- FedeasLab focuses on the simulation of nonlinear structural response under static and dynamic loads, by contrast, these tools emphasize linear finite element analysis
  - Many types of structural elements: lumped plasticity, spread plasticity, nonlinear hinges, composite, prestressing, etc.
  - Many types of analyses common in structural engineering practice: second order analysis, P-Δ, corotational, push-over, many nonlinear solution strategies (several arc-length varieties, line search), several time integration strategies (Newmark, Wilson-θ, α–Method)
  - Lumped or consistent mass
  - Classical or non-classical damping
- FedeasLab is very simple in its basic architecture and is, thus, very easy to extend by the user. SDTools appears much more involved and geared for different type of applications, while FEMLab seems closed (it has the look of commercial finite element packages and its data and function organization does not seem accessible to the user). Their nonlinear capabilities are, nonetheless, very limited and their beam elements very weak.

FEDEASLab Function Organization

Go to http://fedeaslab.berkeley.edu

Download self extracting zip file; it generates following directories:



Features

- `Adjust_Path` file
- `Readme` file
- `Contents.m` in each directory
- on-line help in html format; click FEDEASLab_Help.html

FEDEASLab Data Organization

6 data structures ("objects"), 5 basic and one optional

Model	model information, geometry, element types, dof numbering, (mass)
ElemData{.}	element properties
Loading	loading information, force and displacement patterns loading histories
State	structural response: displacements, velocities, accelerations, stiffness and damping matrix, history variables
SolStrat	static or transient solution strategy parameters
Post	post-processing information (optional)

Typical task sequence for simulation

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**).

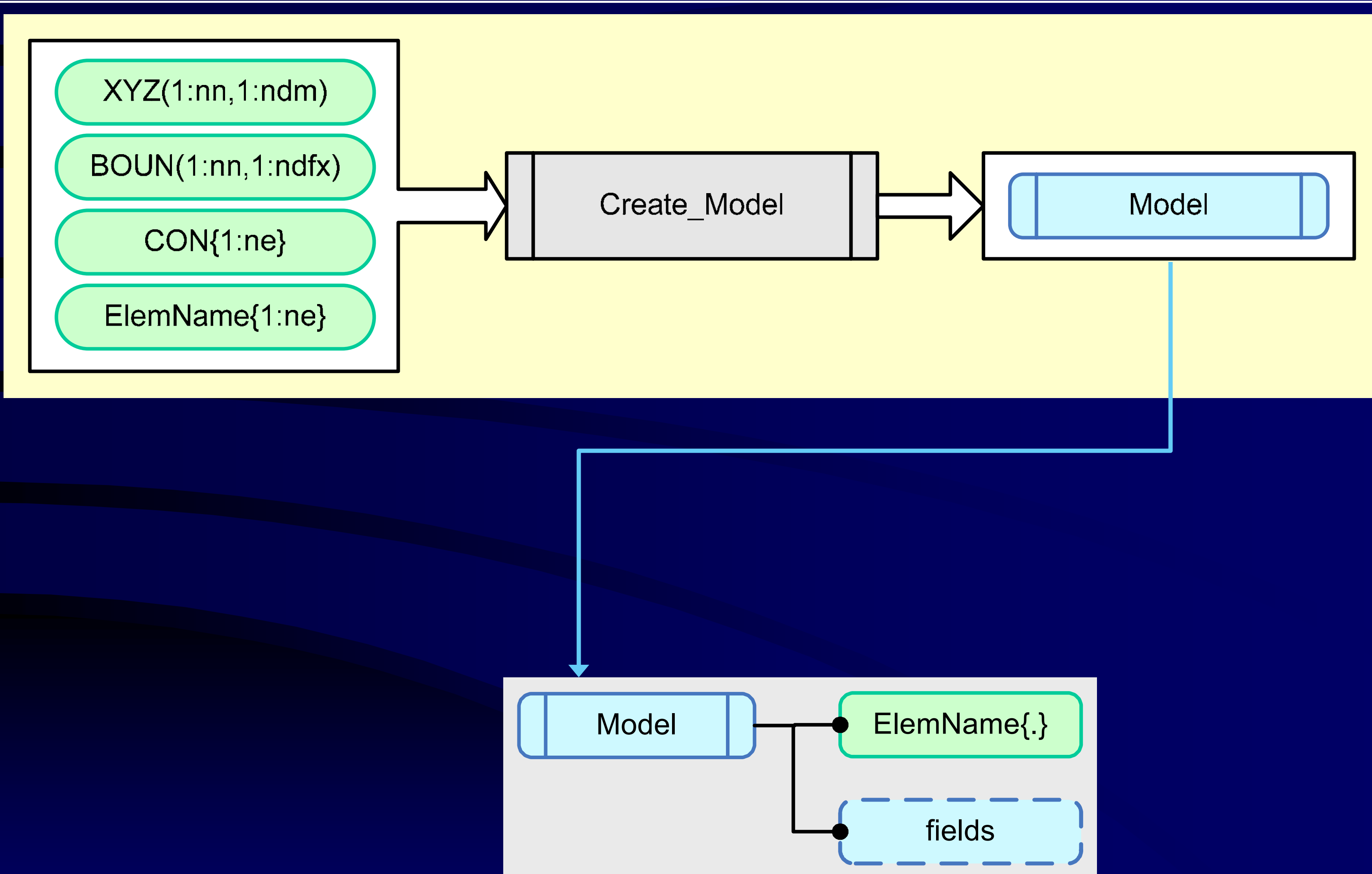
4 Specification of one or more load patterns and creation of data object **Loading**.

5 Creation of data object **SolStrat** with default solution strategy parameters.

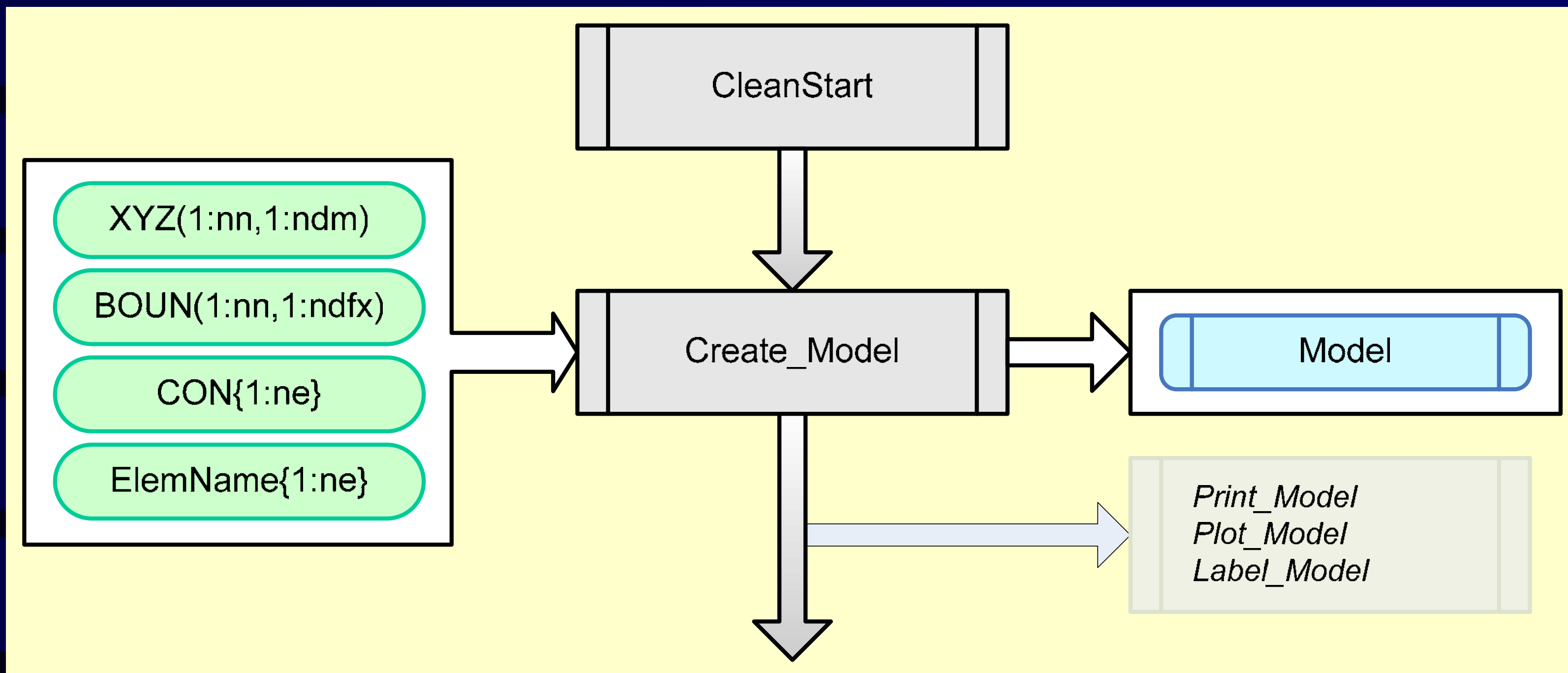
6 Initialization of solution process and application of one or more load steps with corresponding structural response determination.

7 Storage of response information for immediate or subsequent post-processing.

Task 1: Model creation



Example: one bay, two-story frame



Matlab script

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
```

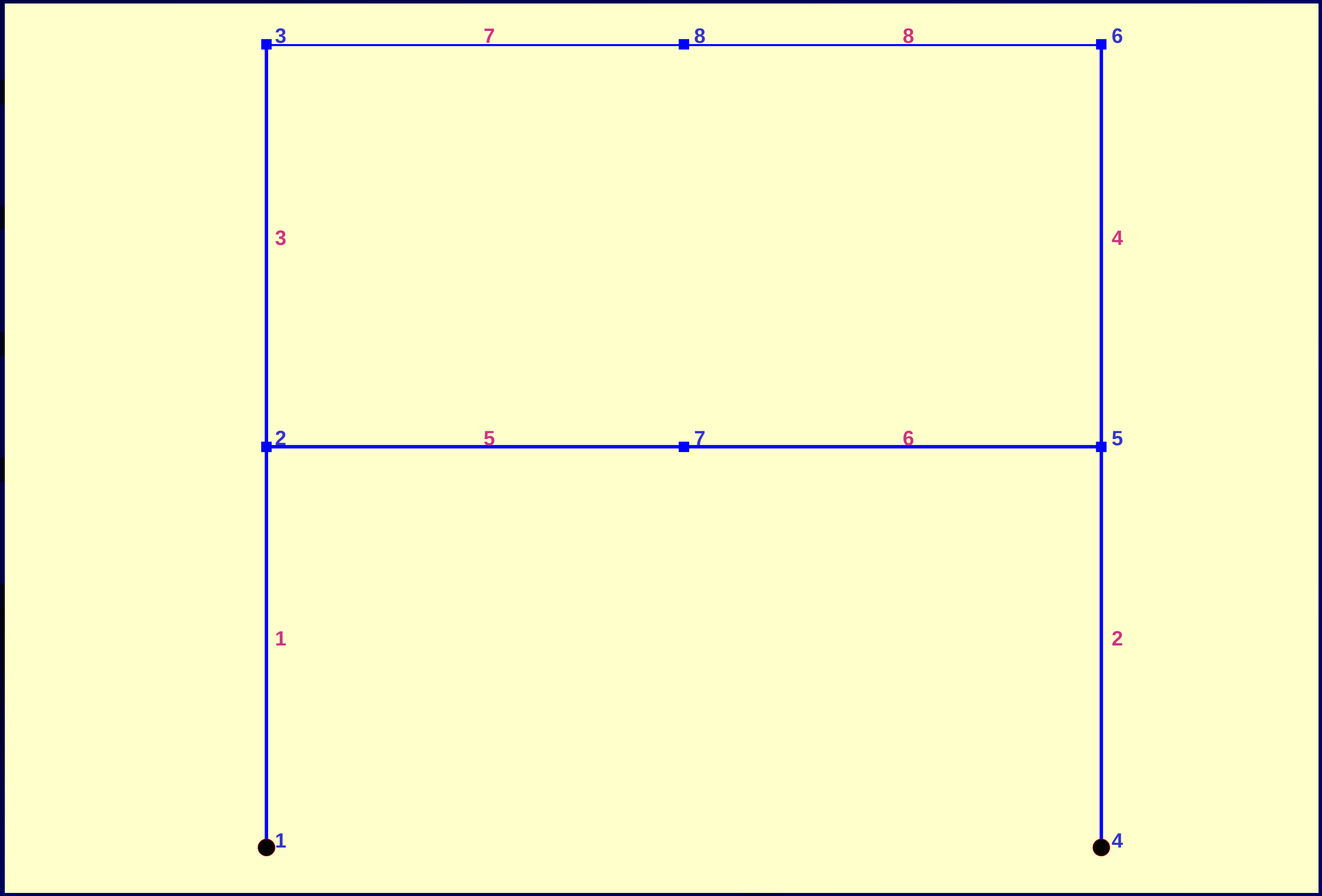
Create model data structure

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

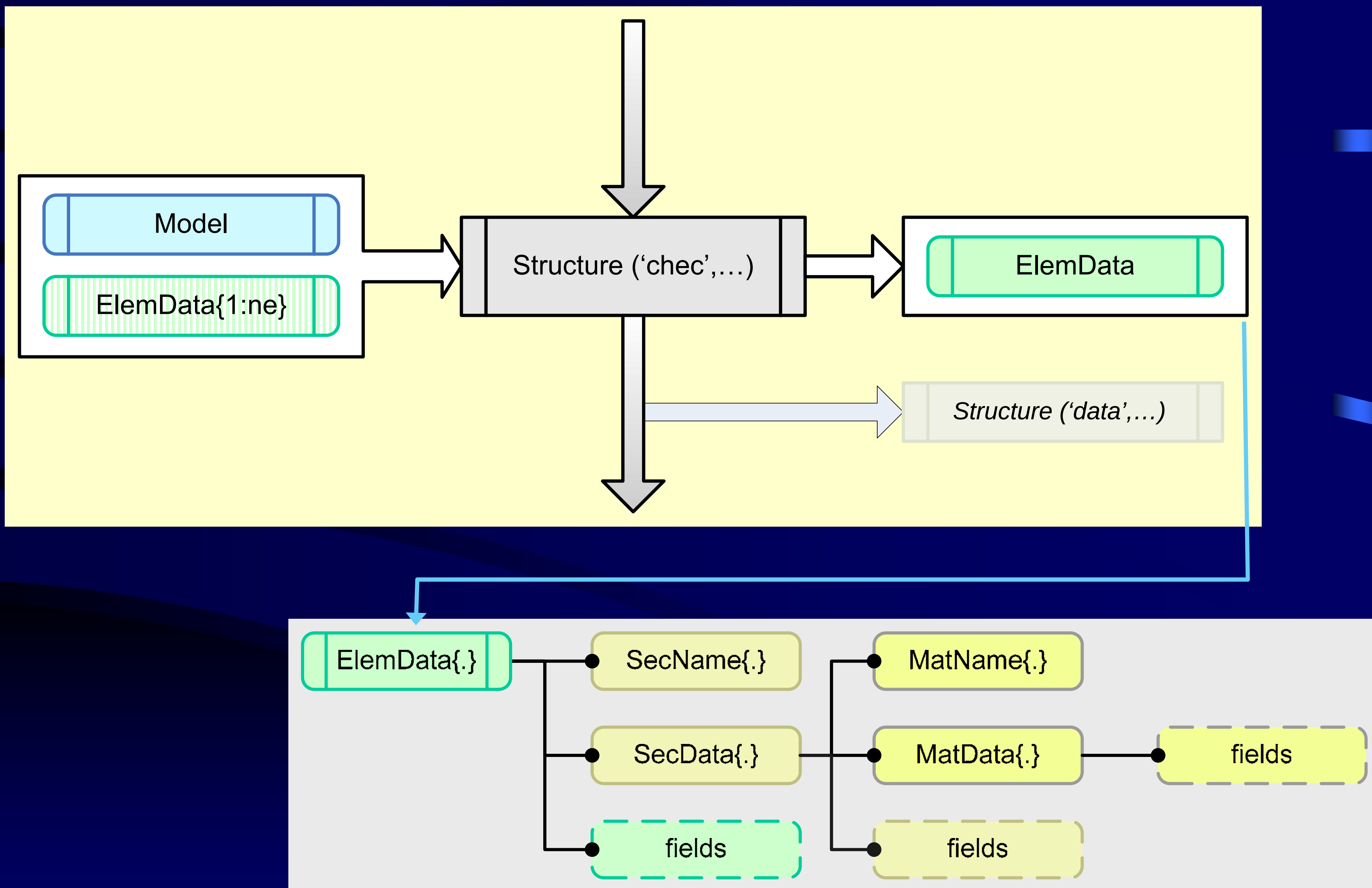
Display model and show node/element numbering (optional)

```
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)
```

Graphic output



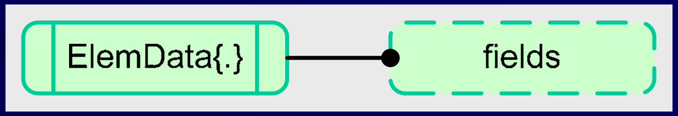
Task 2: specification and completion of element property data



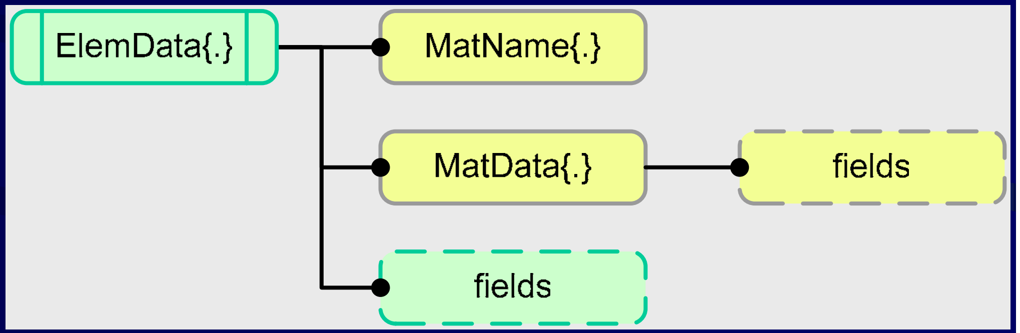
ElemData Organization

There are three possibilities depending on element type

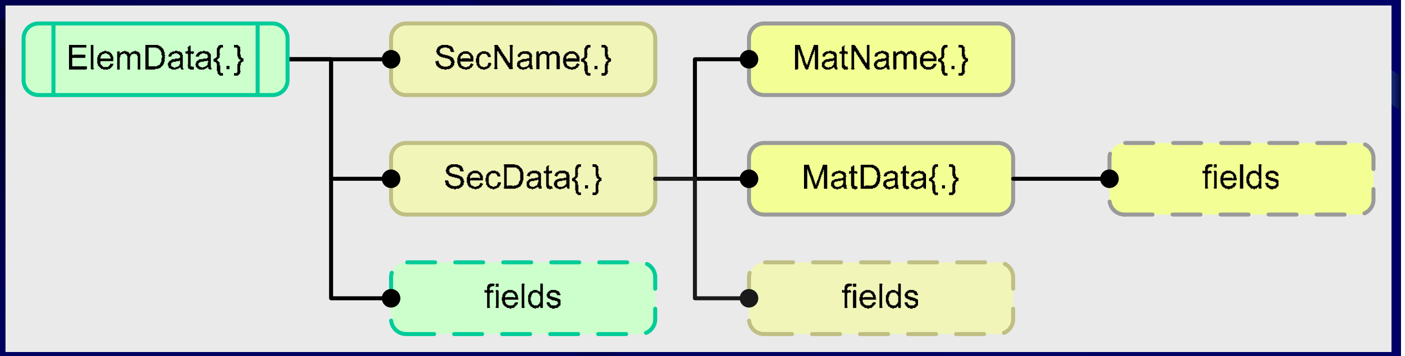
1. Simple elements,
   e.g. linear elastic material



2. Elements with nonlinear material
   e.g. 4 node plane stress



3. Beam elements with hierarchy: element →section→material



2d frame example: element properties for linear element

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);
```

2d frame example: element properties for nonlinear distr inelasticity element

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
```

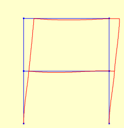
Default values for missing element properties

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

The central function Structure: operate on group of elements in Model

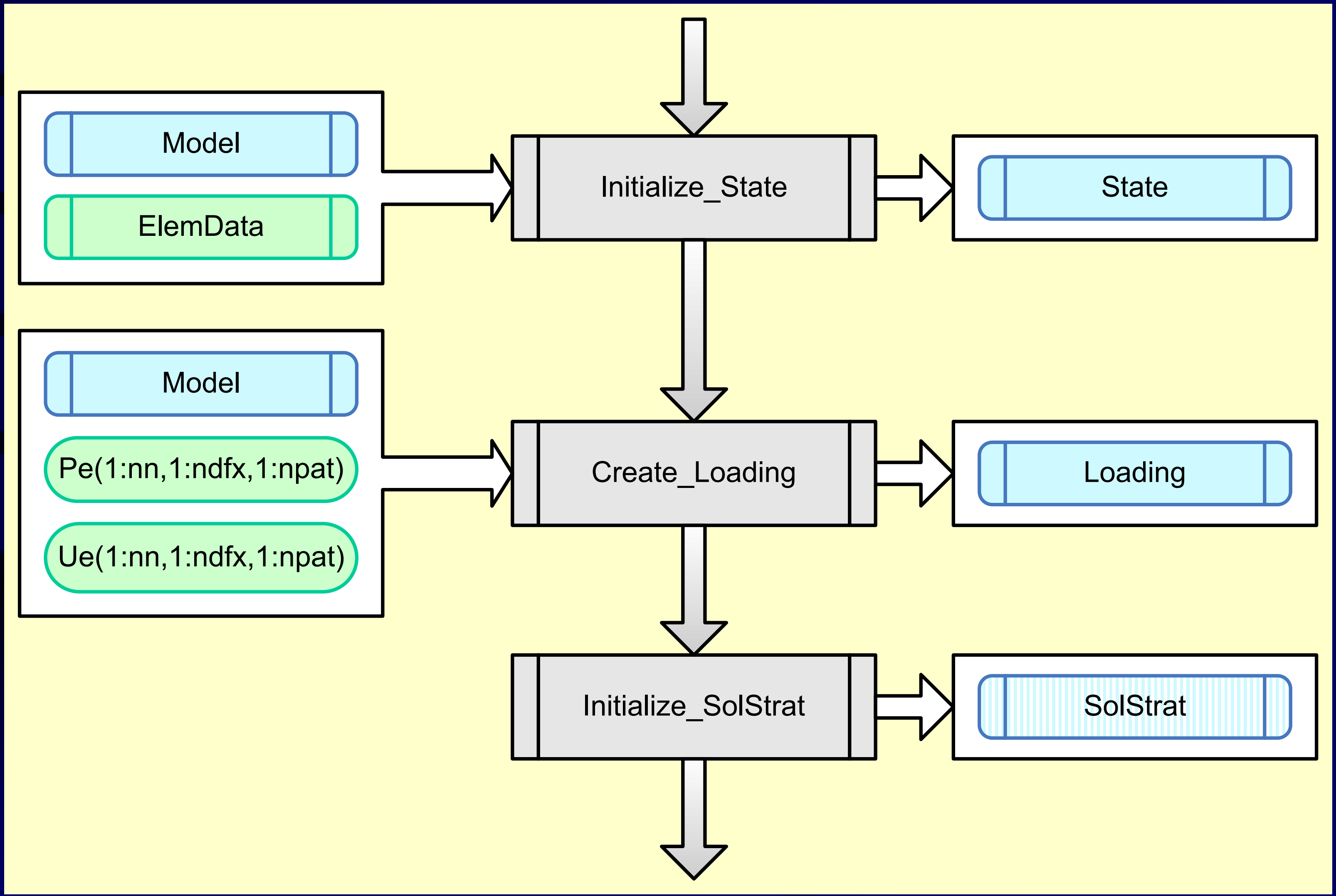
Structure (action,Model,ElemData,State,ElemList)

action | Model | ElemData | State | *ElemList* | Output

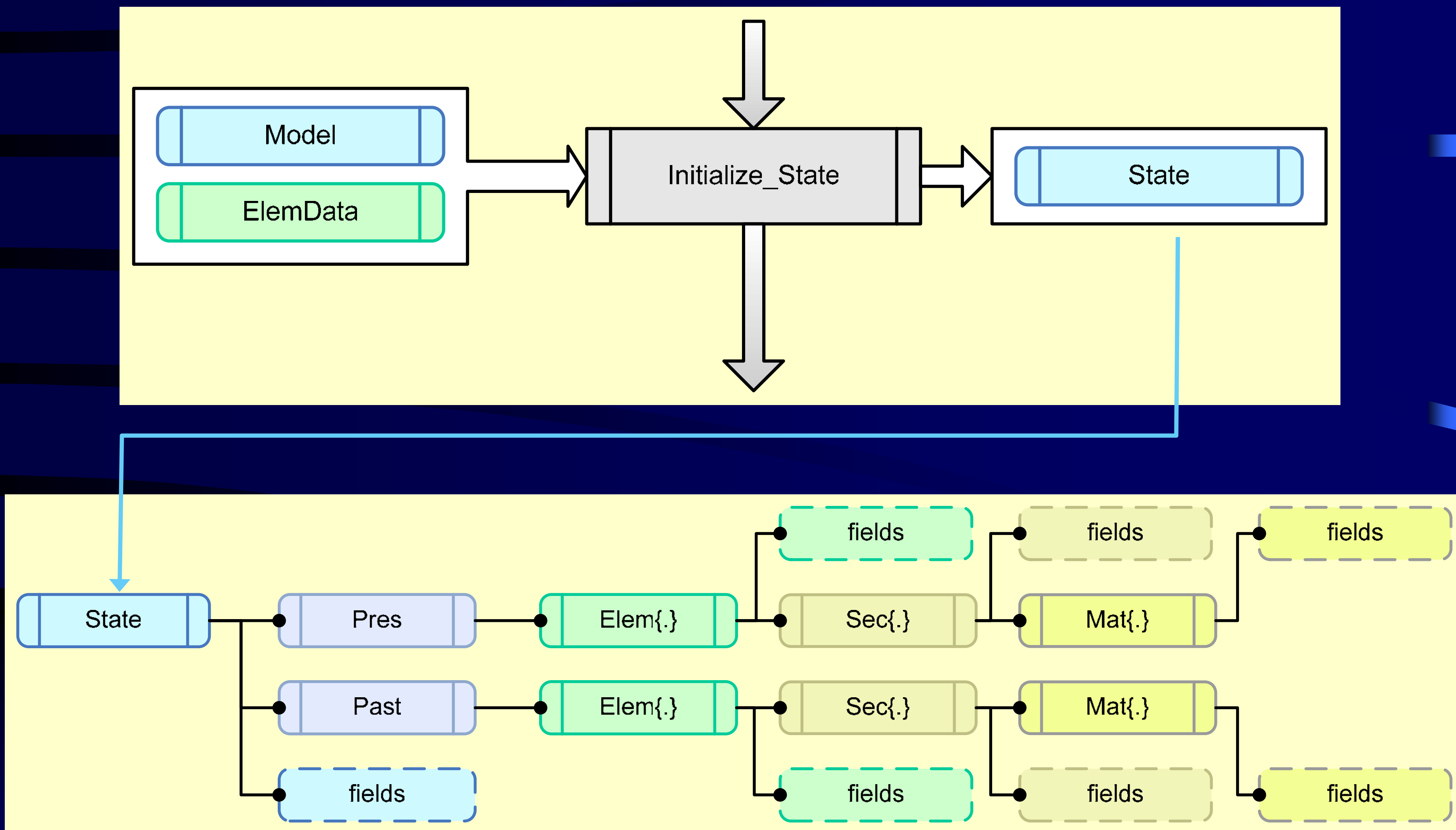
action	description	Output
chec	check for missing element properties	ElemData
data	print element properties	IOW
init	initialize element history variables	State
stif	structure state determination: determine Pr and Kf for given U, DU, DDU, etc.	State
forc	structure state determination: determine only Pr 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	

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

Tasks 3 through 5: initialize State and SolStrat, create Loading



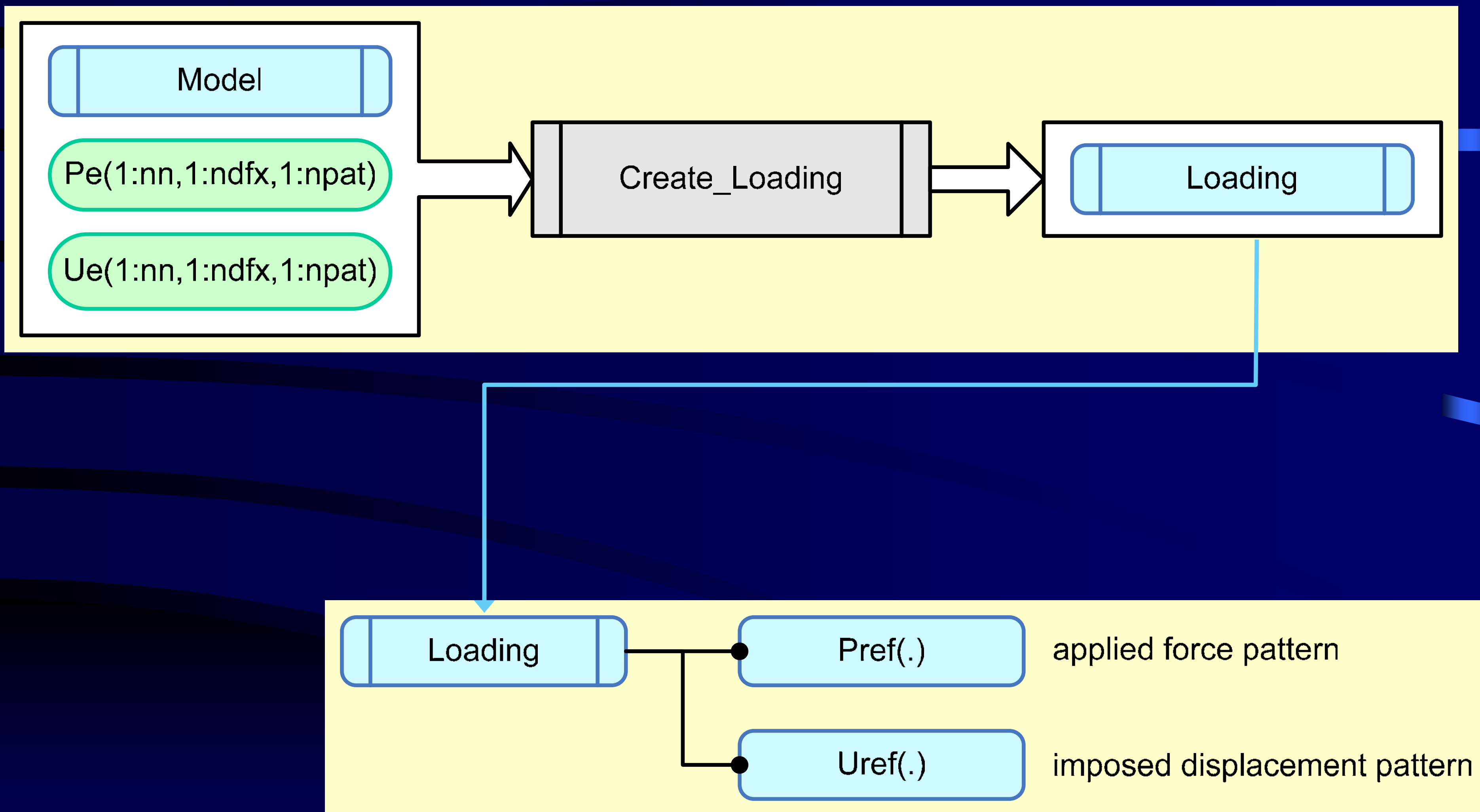
Task 3: initialization of State



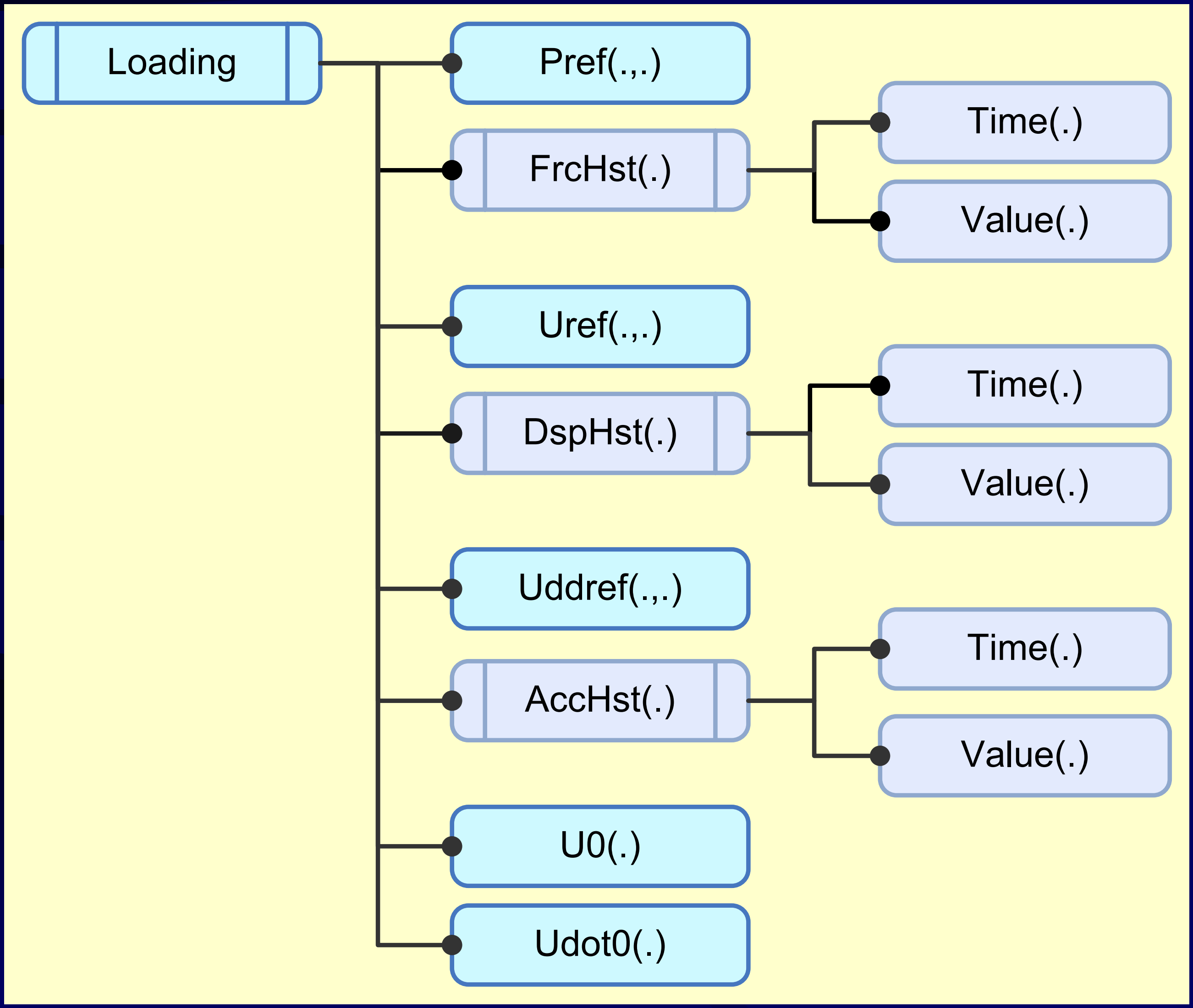
State.field

Field	Data Type	Description
U	Scalar Array	Generalized displacements at global dofs
DU	Scalar Array	Displacement increments from last converged state
DDU	Scalar Array	Displacement increment from last iteration
Udot	Scalar Array	Velocities at global dofs
Uddot	Scalar Array	Accelerations at global dofs
Kf	Array (nf x nf)	Stiffness matrix at free dofs
KL	Scalar Array	Lower diagonal stiffness matrix
KU	Scalar Array	Upper diagonal stiffness matrix
Kfd	Scalar Array	Stiffness matrix relating restrained dofs to free dofs
lamda	Scalar	Load factor
Time	Scalar Array	Pseudo-time
C	Array (nf x nf)	Damping matrix
Pr	Array	Resisting forces at the free dofs
dW	Scalar	External work increment

Task 4: Loading specification

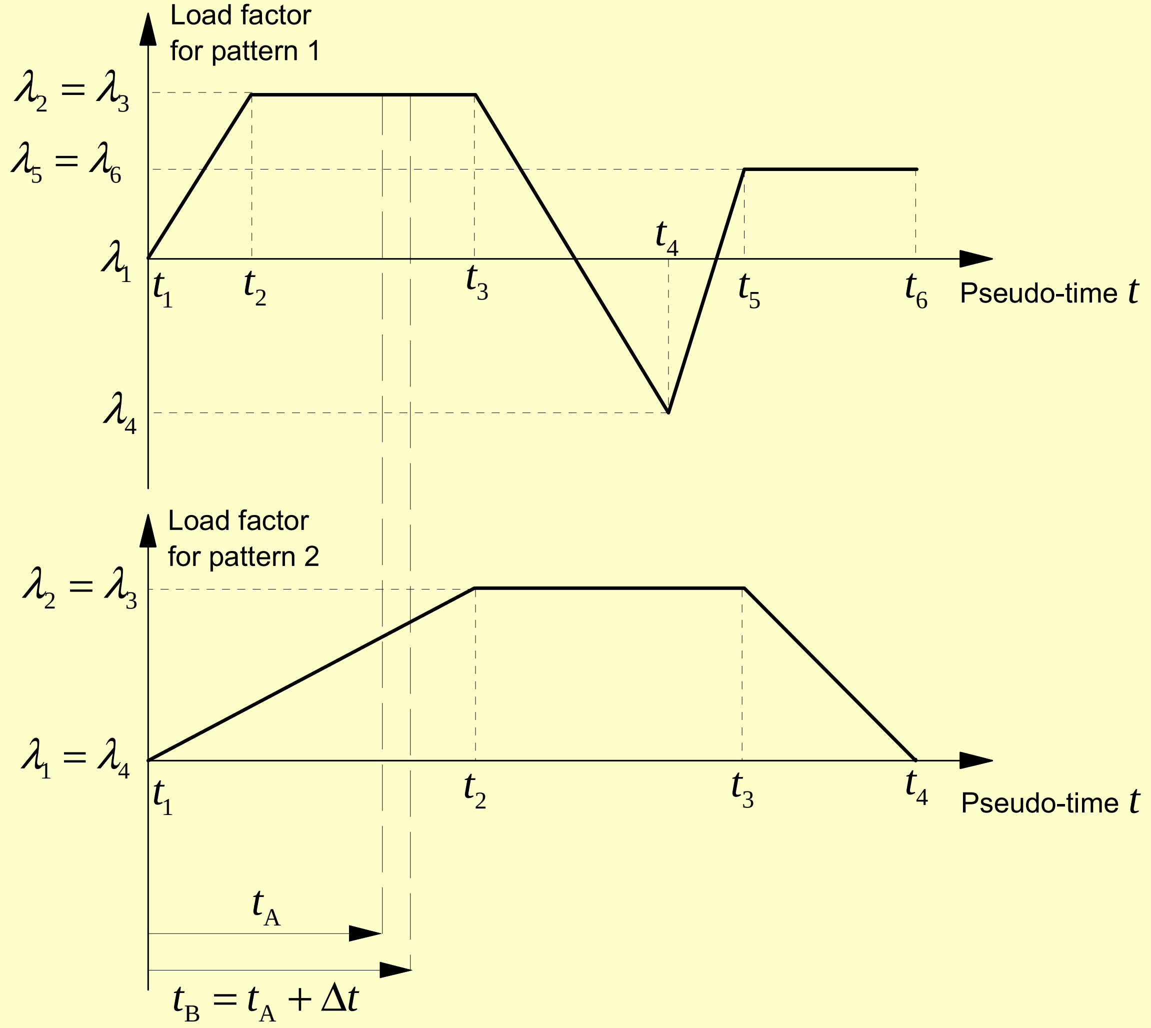


User specification of other fields for Loading

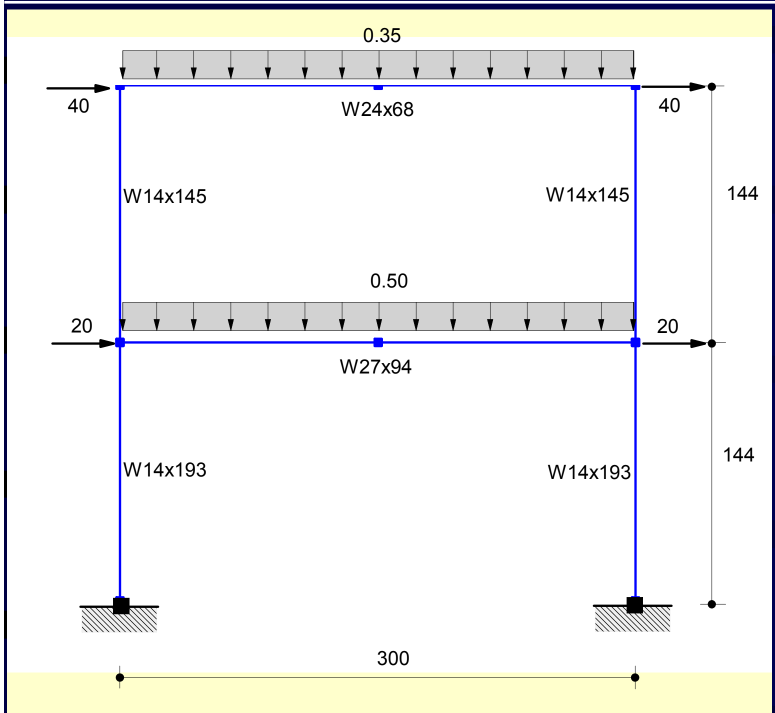


Specification of load history

```
Loading.FrcHst(1).Time   = [    t1;    t2;    t3;    t4;    t5;    t6];
Loading.FrcHst(1).Value  = [lamda1;lamda2;lamda3;lamda4;lamda5;lamda6];
Loading.FrcHst(2).Time   = [    t1;    t2;    t3;    t4];
Loading.FrcHst(2).Value  = [lamda1;lamda2;lamda3;lamda4];
```



Loading definition for 2d frame example



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);
… … … … …
```

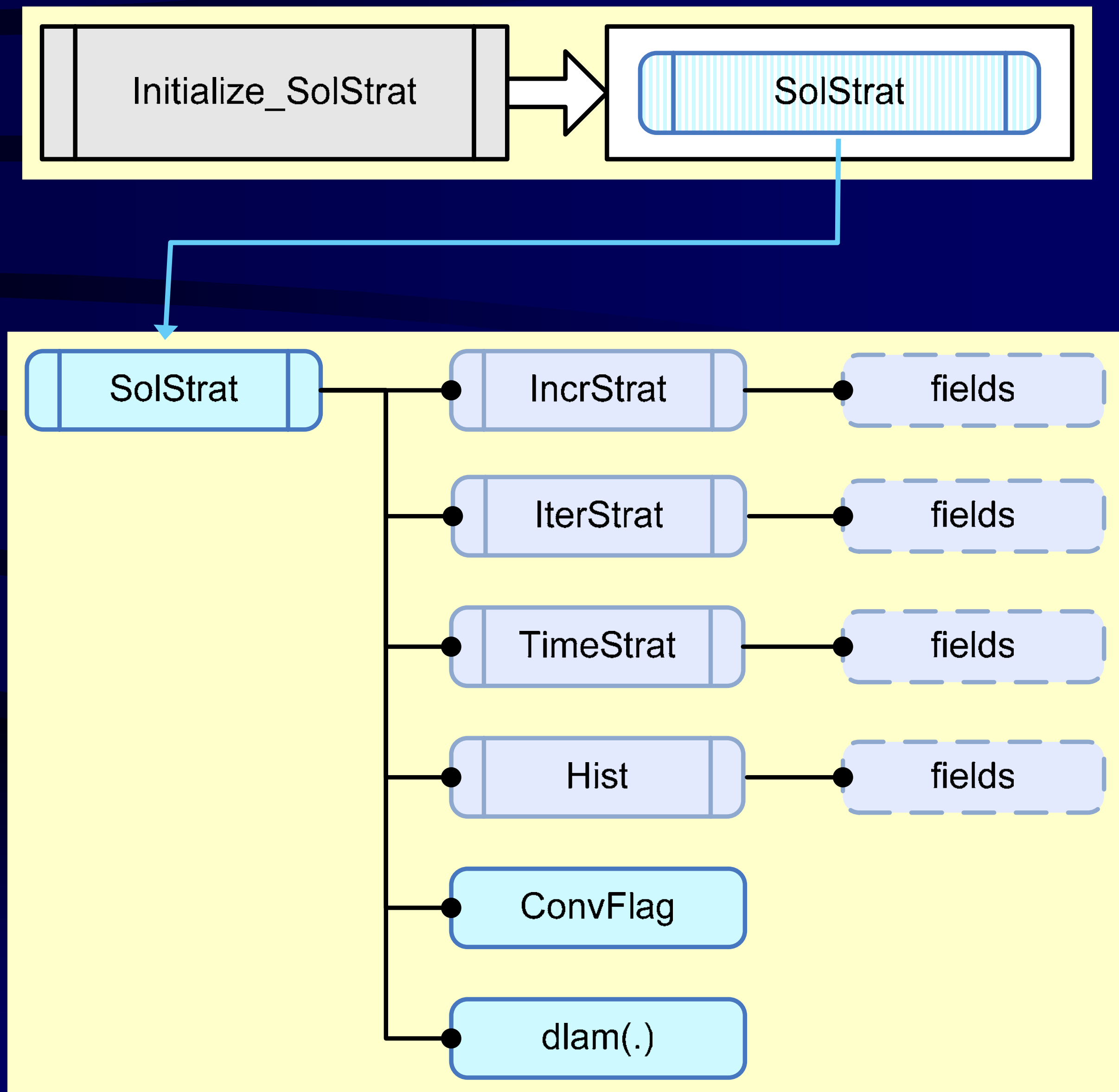
Load case 2: horizontal forces

```
% 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);
```

Task 5: initialization of SolStrat



SolStrat fields

SolStrat.field

Field	Data Type	Description
ConvFlag	logical	True (1) for successful completion of equilibrium iterations, false (0) otherwise
dlam	scalar	Load factor increment

SolStrat.IncrStrat.field

Field	Data Type	Description
dlam0	scalar	Initial load factor increment
Deltat	scalar	Pseudo-time increment
StifUpdt	character	Variable indicating stiffness update with ‘yes’ or ‘no’
LoadCtrl	character	Variable indicating load control with ‘yes’ or ‘no’
LCType	character	Type of load control
gamma	scalar	Exponent for current stiffness parameter formula (Bergan et al.,1978)

SolStrat fields (con'd)

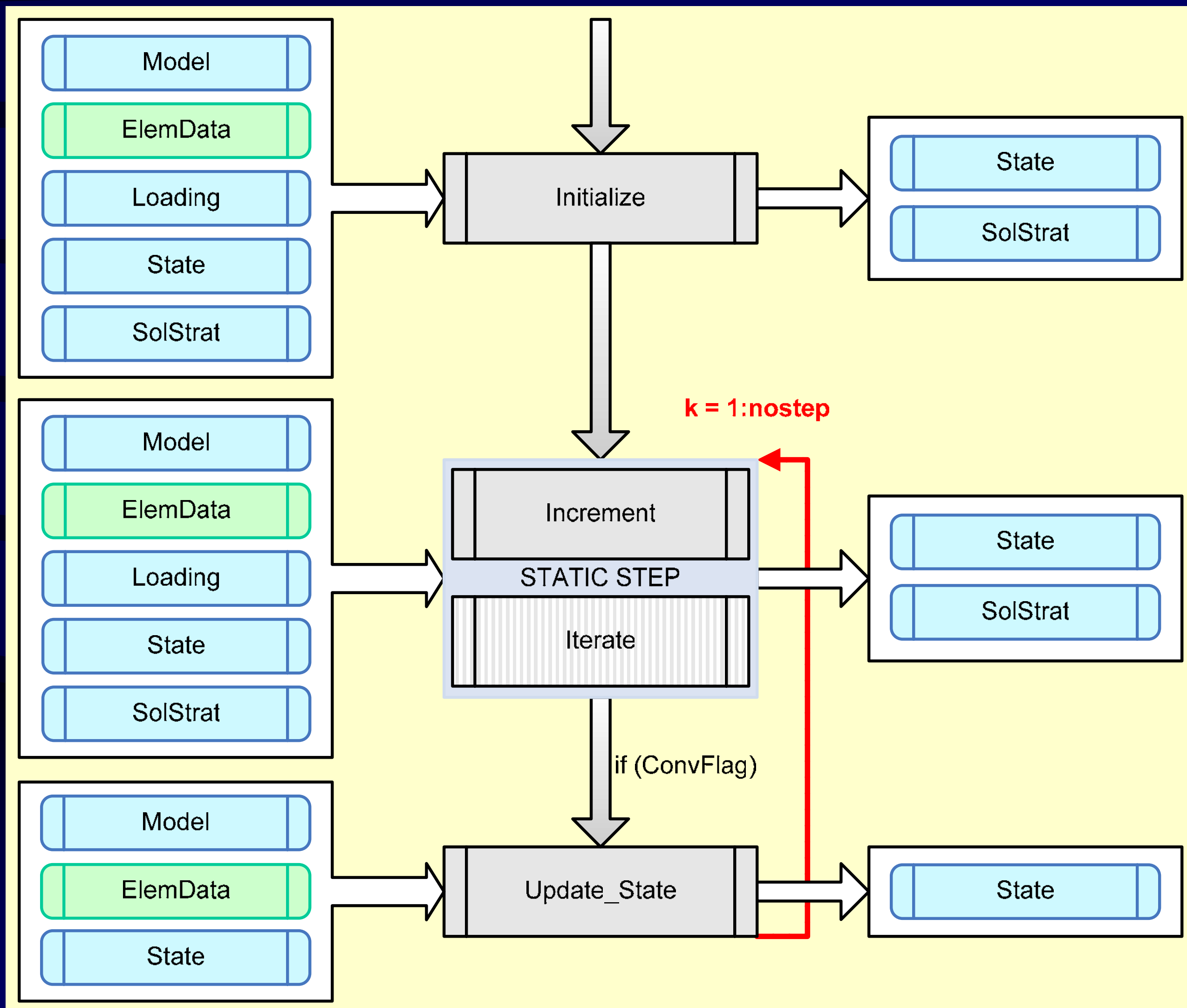
SolStrat.IterStrat.field

Field	Data Type	Description
tol	scalar	Relative work tolerance for convergence
maxiter	scalar	Maximum number of iterations in a load step
StifUpdt	character	Variable indicating stiffness update with 'yes' or 'no'
LoadCtrl	character	Variable indicating load control with 'yes' or 'no'
LCType	character	Type of load control: 'MinDispNorm', 'keyDOF'
LCParam	scalar row vector	keyDOF value

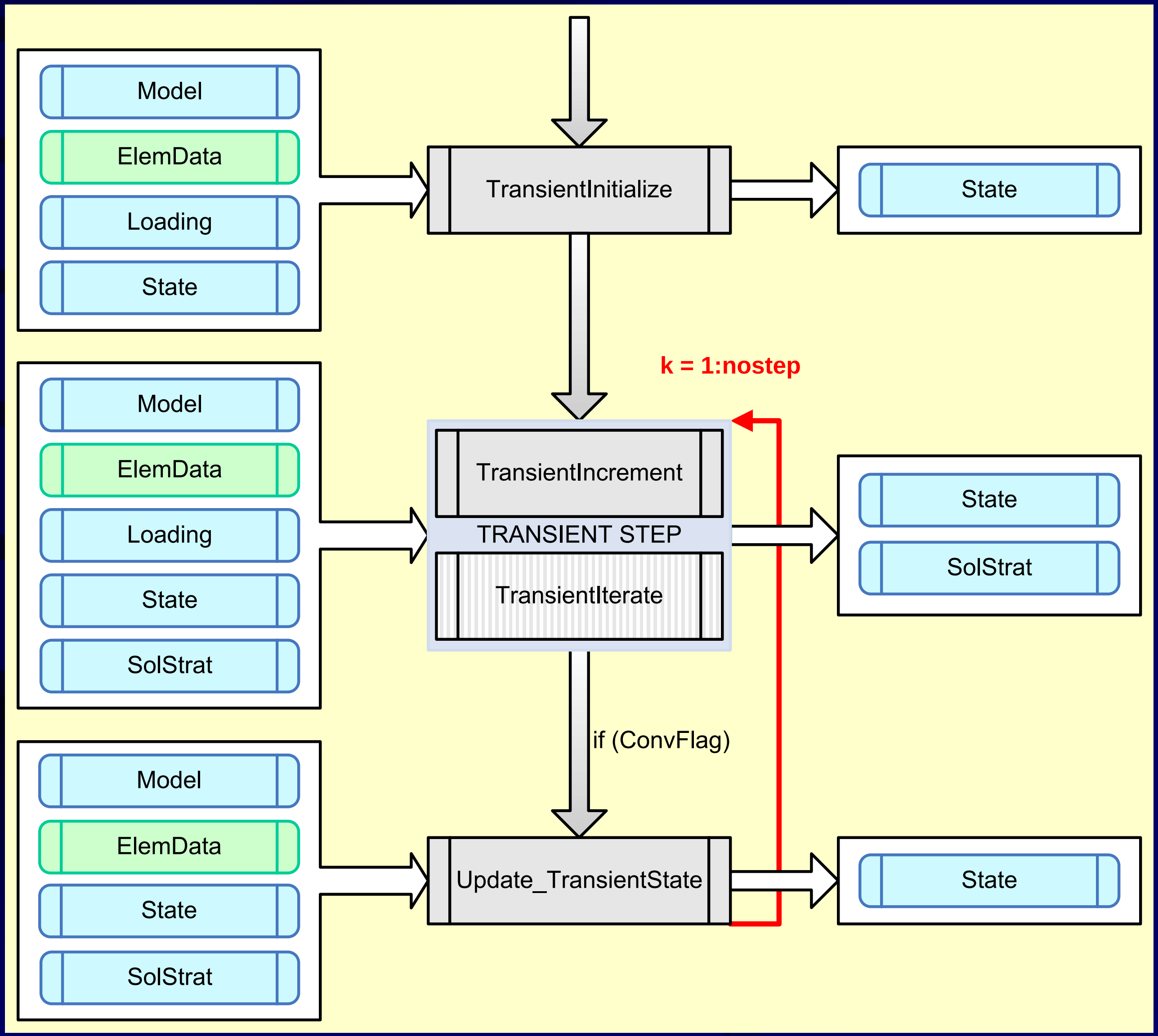
SolStrat.TimeStrat.field

Field	Data Type	Description
Deltat	scalar	Time increment for transient incremental analysis
Type	character	Type of time integration strategy (current only 'Newmark')
Param	scalar row vector	Time integration parameters (currently beta and gamma for Newmark's method)

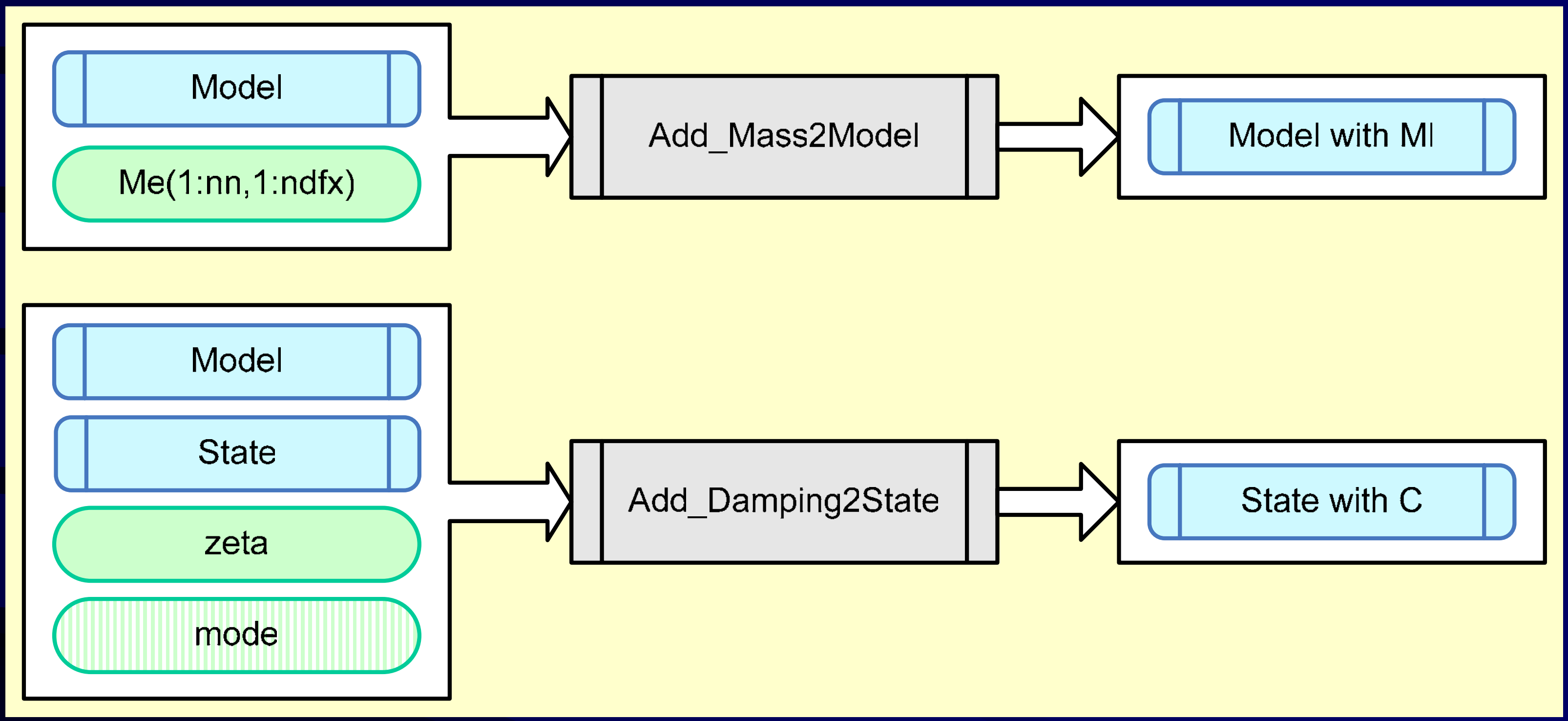
Task 6: static incremental analysis



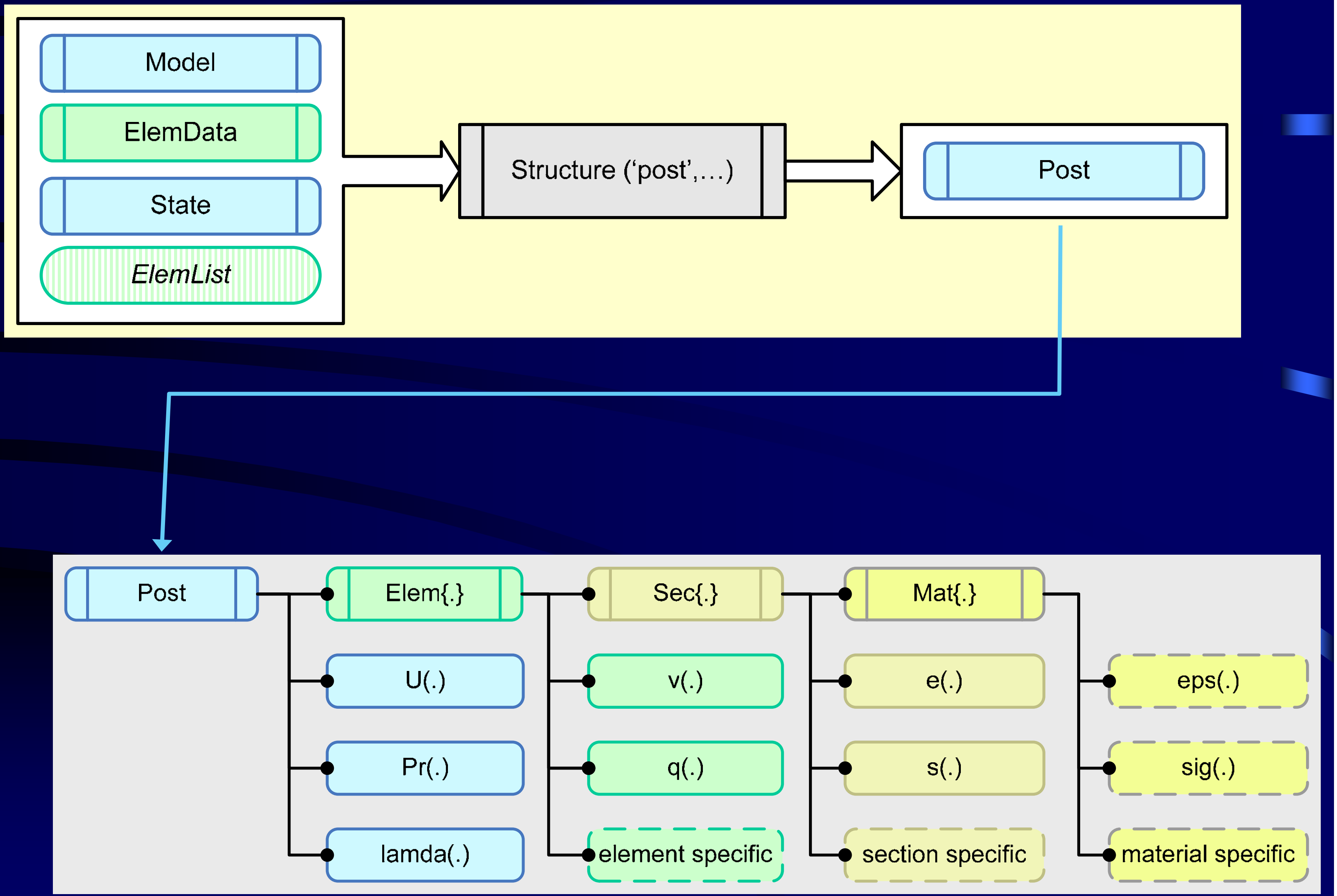
or, transient incremental analysis



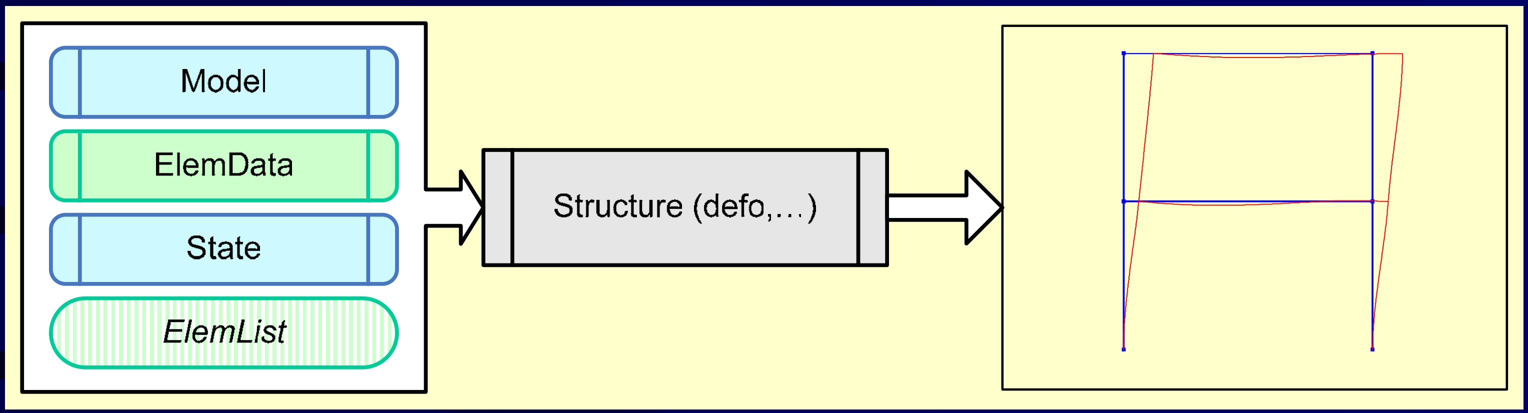
Lumped mass vector and Rayleigh damping for transient analysis

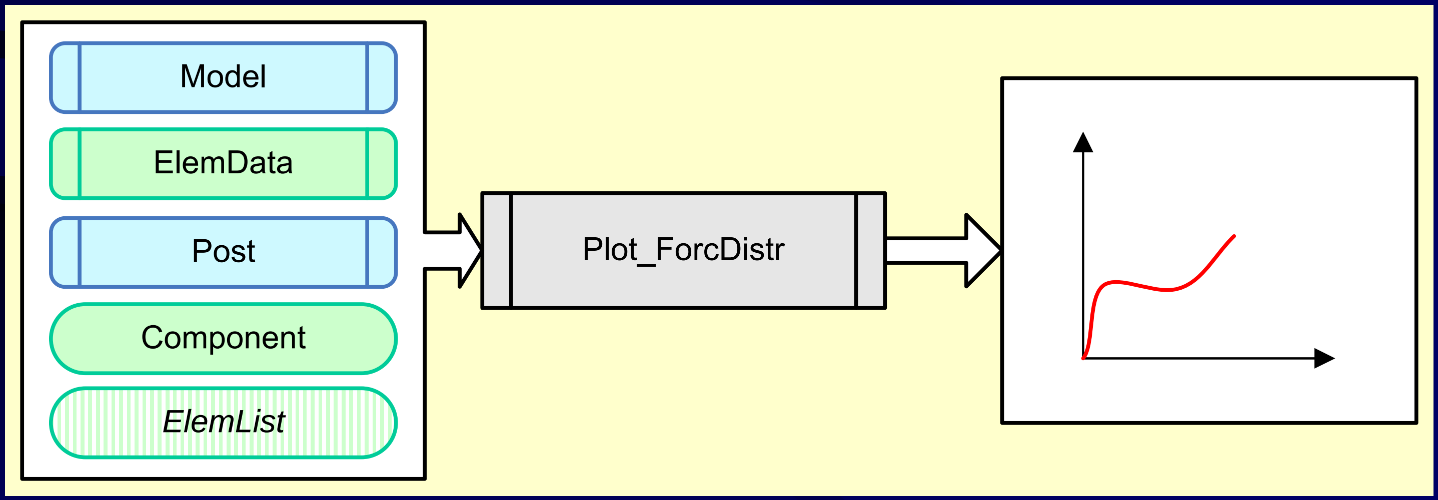


Task 7: generation of Post for immediate or later post-processing

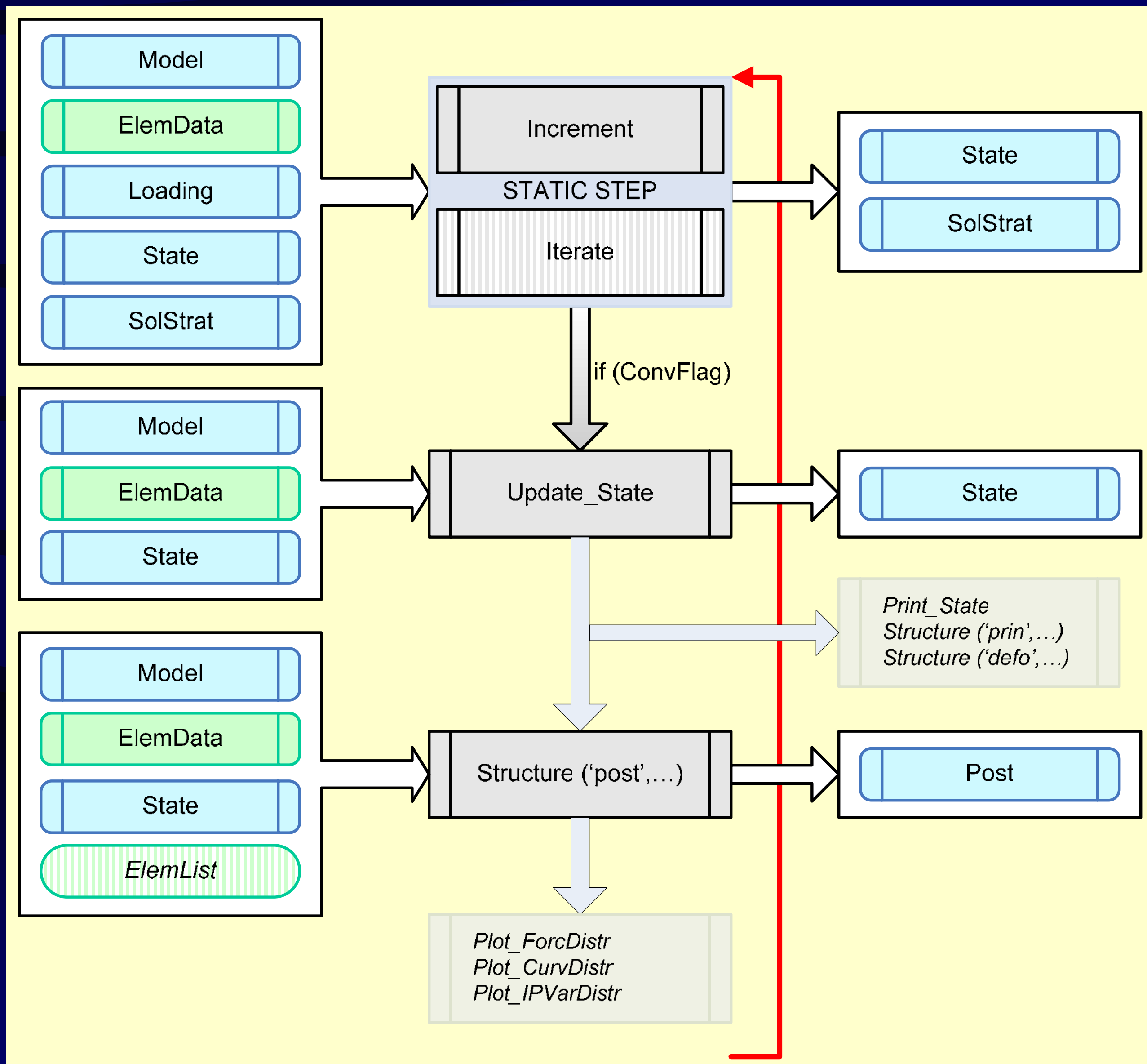


Post-processing examples





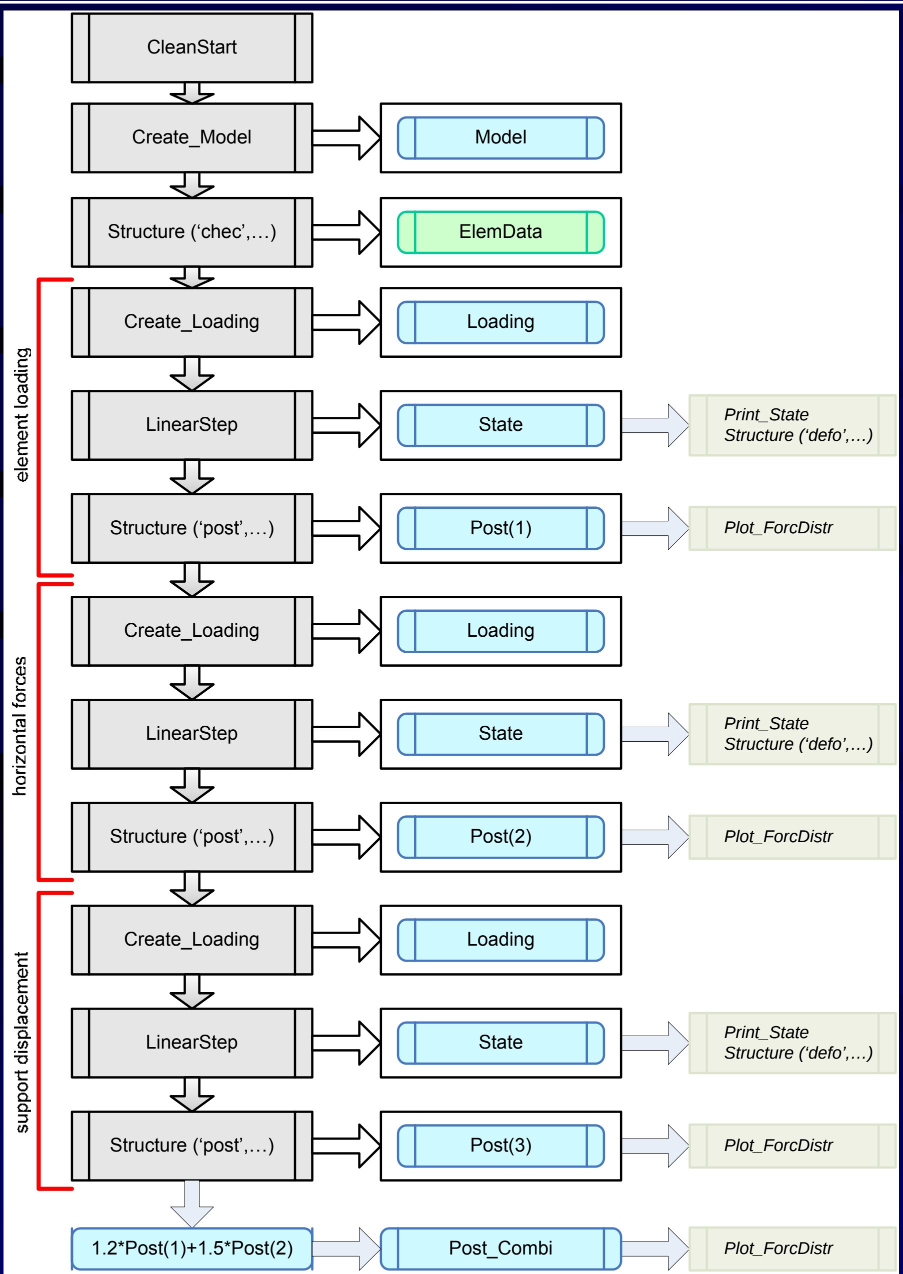
Post-processing at every load step



Getting Started Guide with Simulation Examples

- Ex1: Linear elastic analysis with superposition of results from 3 load cases
- Ex2: Linear transient response to support acceleration by modal analysis
- Ex3: Linear transient response to support acceleration by time integration
- Ex4: Push-over with constant gravity loads and incremental lateral forces with force histories
- Ex5: Push-over with constant gravity loads, followed in sequence by incremental lateral forces with load control during incrementation
- Ex6: same as 5 with P-Δ effect, load control during iteration
- Ex7: same as 6 with distributed inelasticity element (examples 4-6 use the one-component model)
- Ex8: nonlinear transient response with constant gravity loads and imposed support acceleration with distributed inelasticity frame elements for the model

Example 1



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);
… … … … …
```

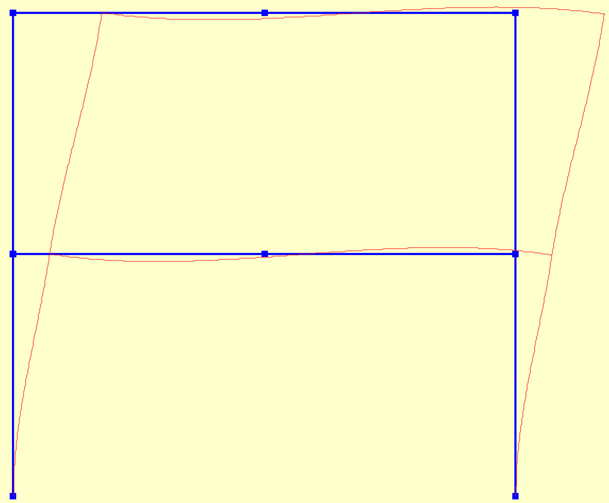
Load case 3: support displacement

```
% 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);
… … … … …
```

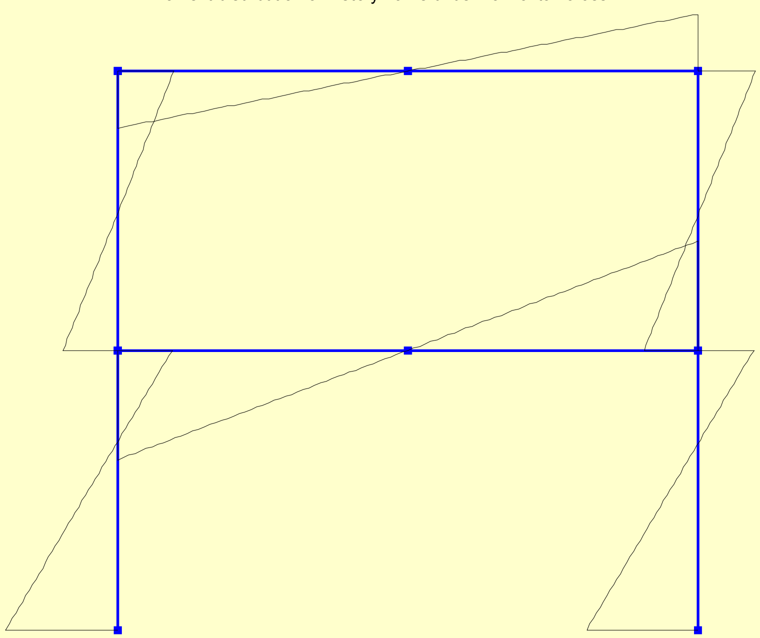
Load combination

```
% 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');
```

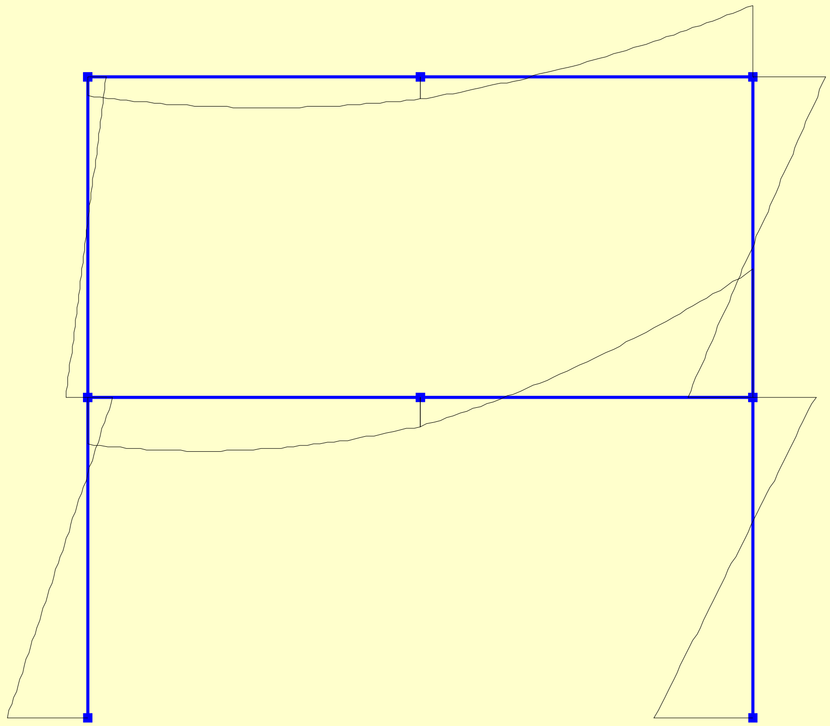
Results for example 1



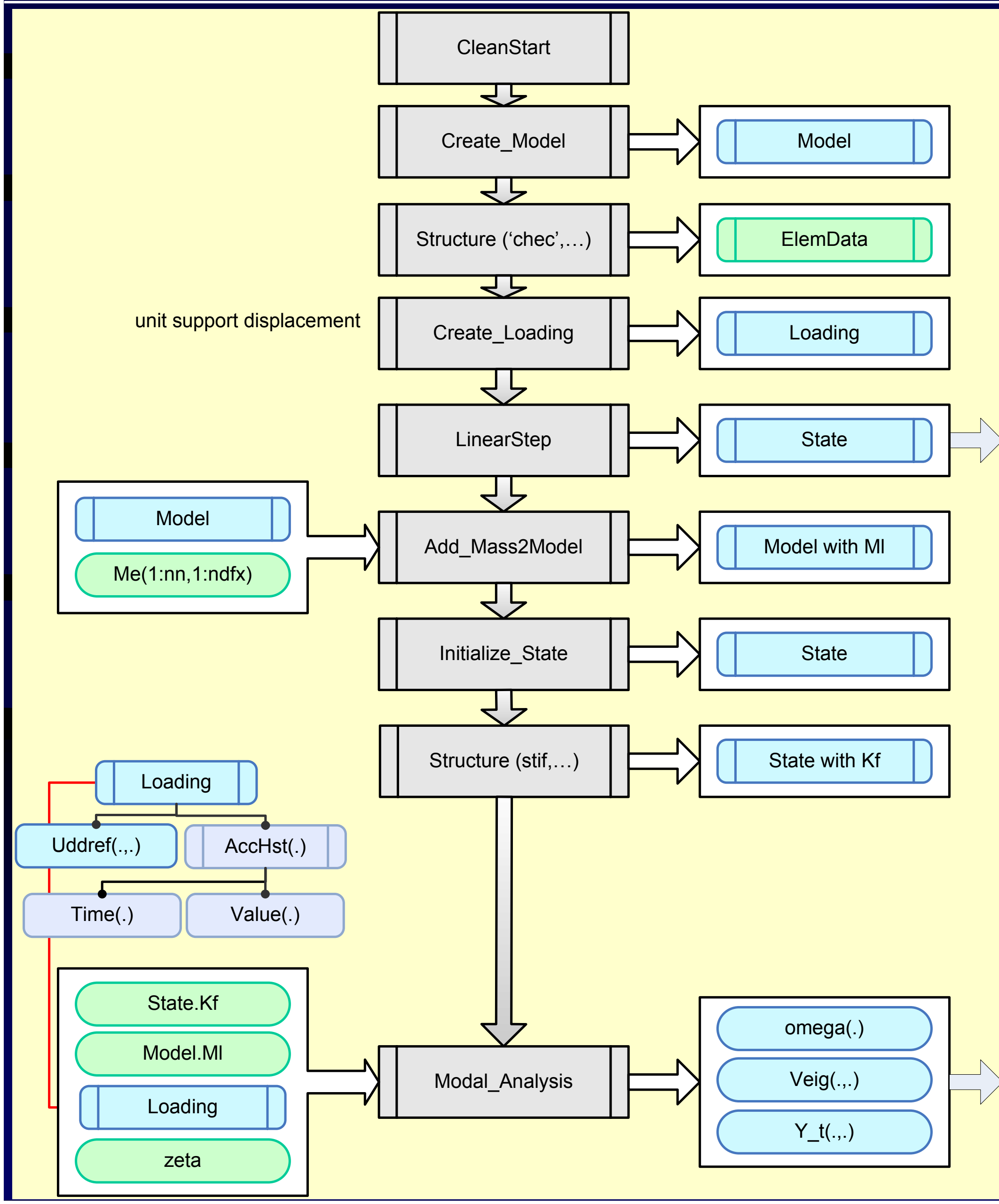
Moment distribution of 2-story frame under horizontal forces



Moment distribution of 2-story frame under load combination of 1.2DL+1.5EQ



Example 2



Specify ground acceleration

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

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

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

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

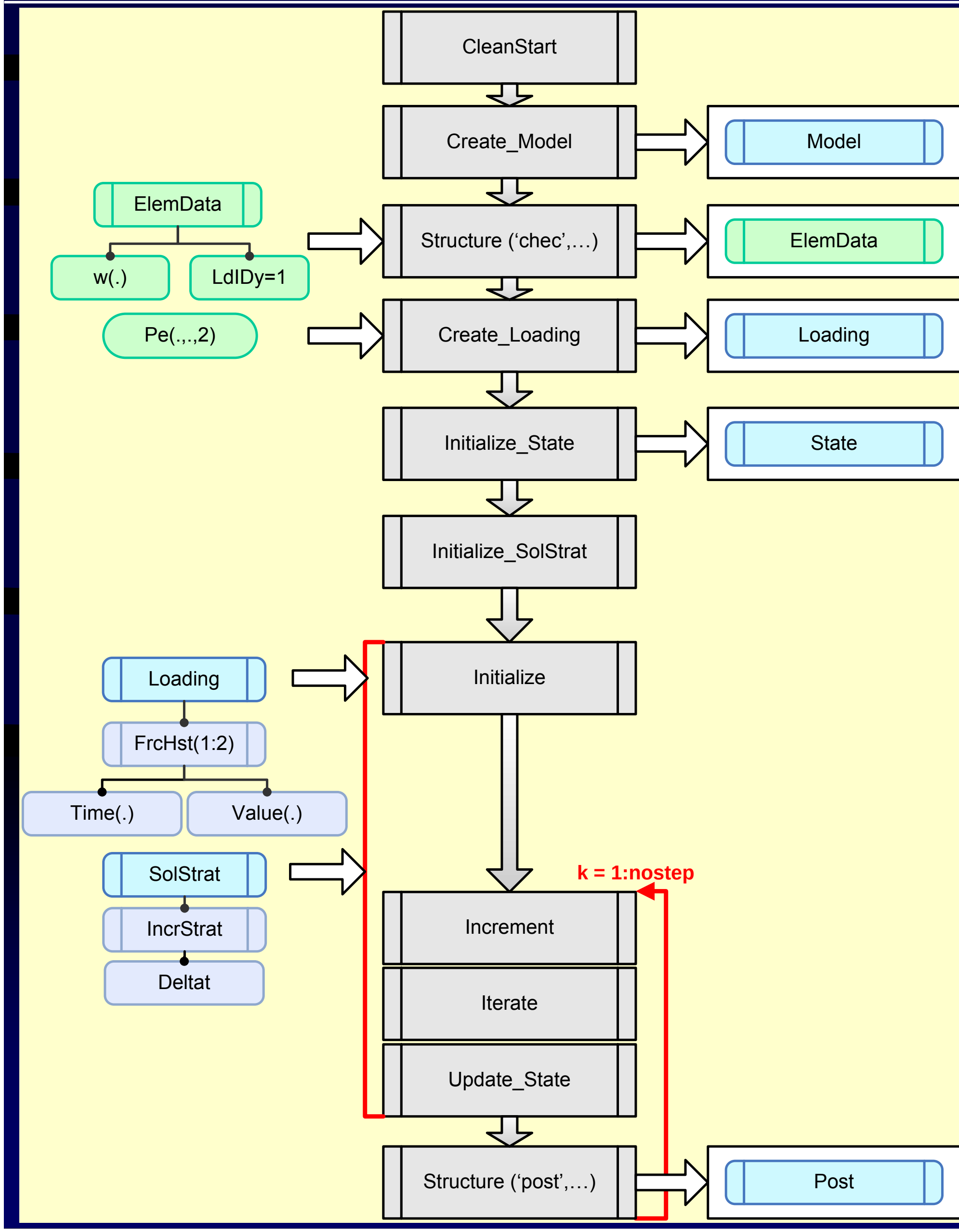
Lumped mass vector

```
% 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

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

Example 4



Distributed element loads with load pattern number 1

```matlab
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

```matlab
% 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

```matlab
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];
```

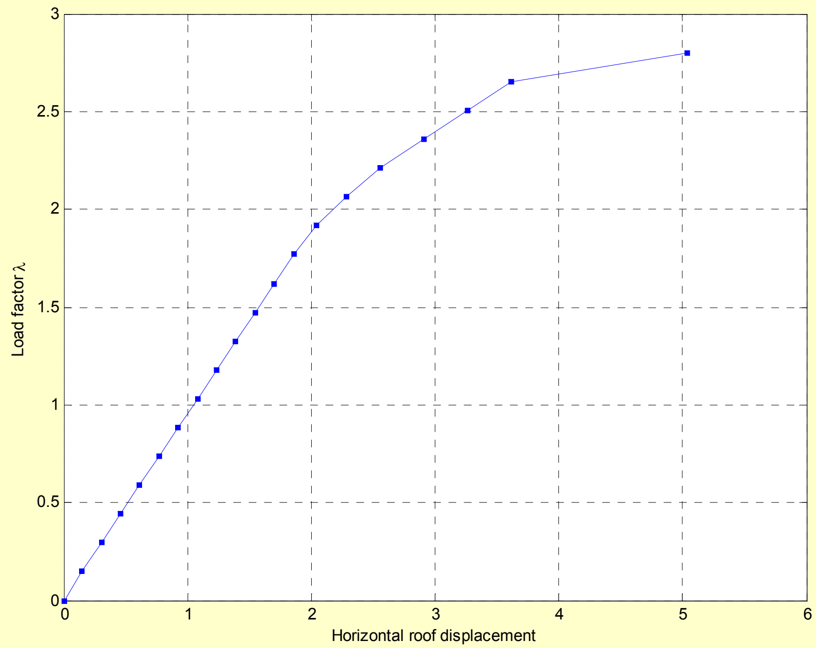
Incremental analysis by pseudo-time incrementation

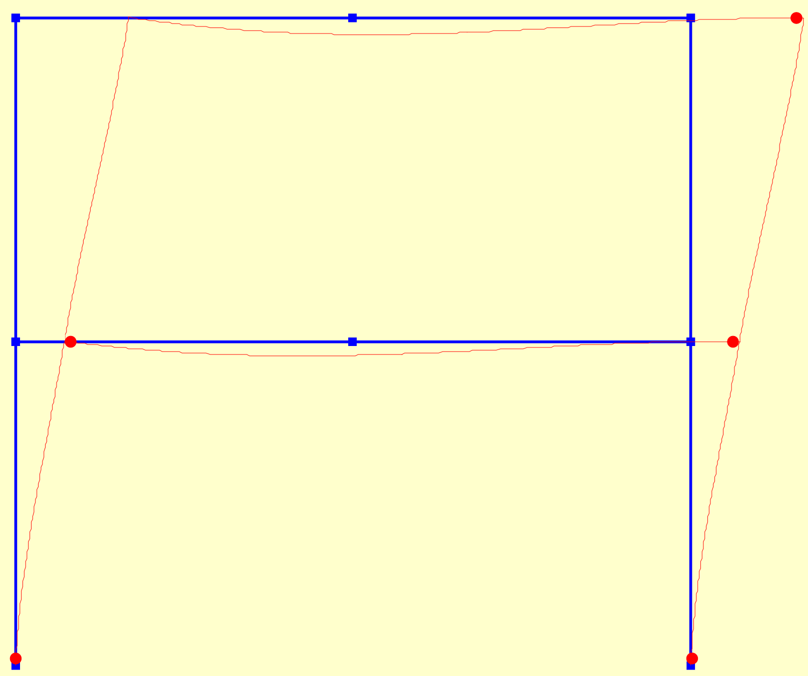
```matlab
% initialize State
State = Initialize_State(Model,ElemData);
% initialize default SolStrat parameters
SolStrat = Initialize_SolStrat;
% specify pseudo-time step increment (does not have to be the same as Deltat,
smaller value
% results in more steps to reach end of analysis)
SolStrat.IncrStrat.Deltat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

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

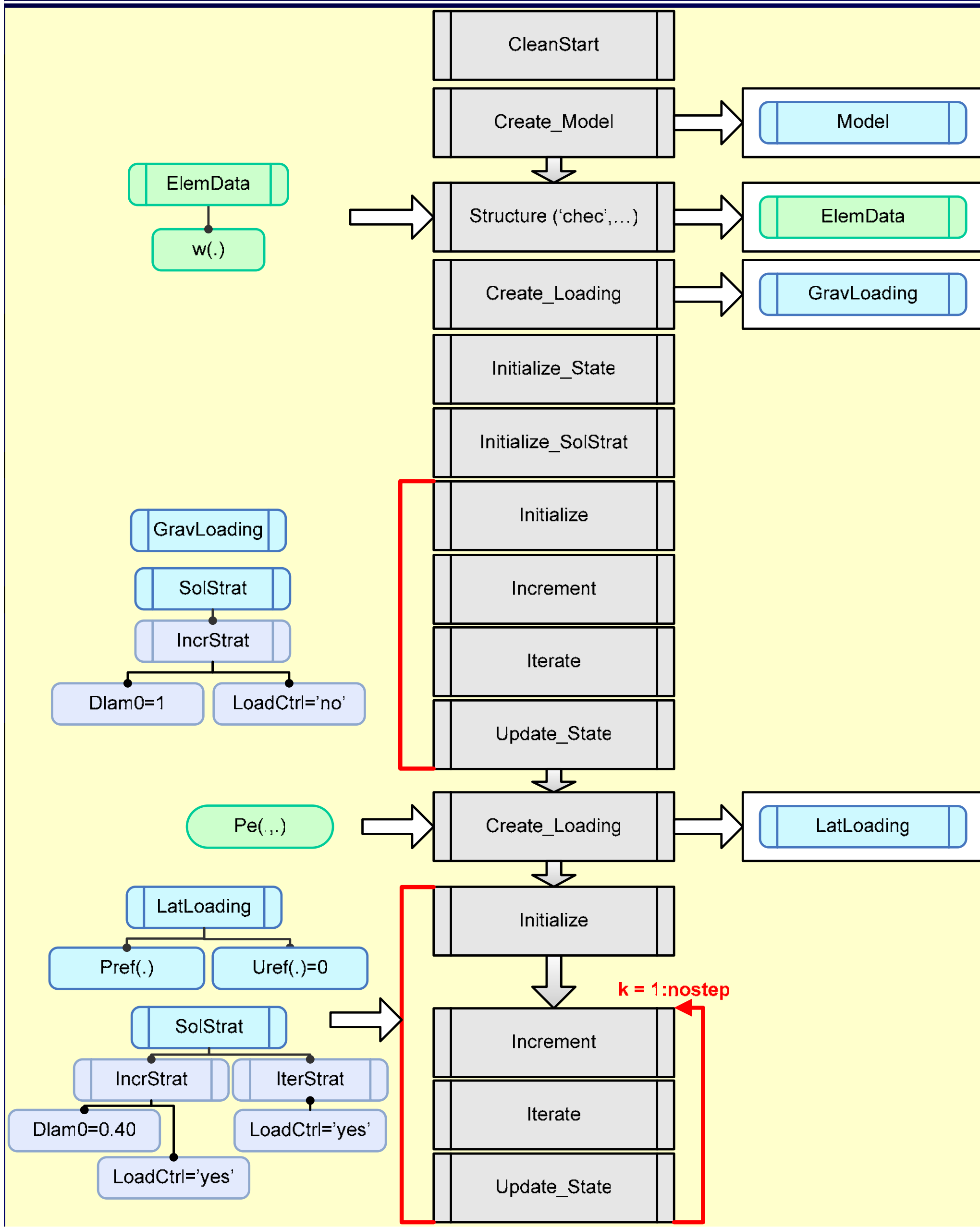
```matlab
while (State.Time < Tmax-10^3*eps)
   [State SolStrat] = Increment(Model,ElemData,Loading,State,SolStrat);
   [State SolStrat] = Iterate  (Model,ElemData,Loading,State,SolStrat);
   if (SolStrat.ConvFlag)
      State = Update_State(Model, ElemData, State);
   else
      break
   end
   pc =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

Results for example 4





Example 5



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
```

Example 6

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
```

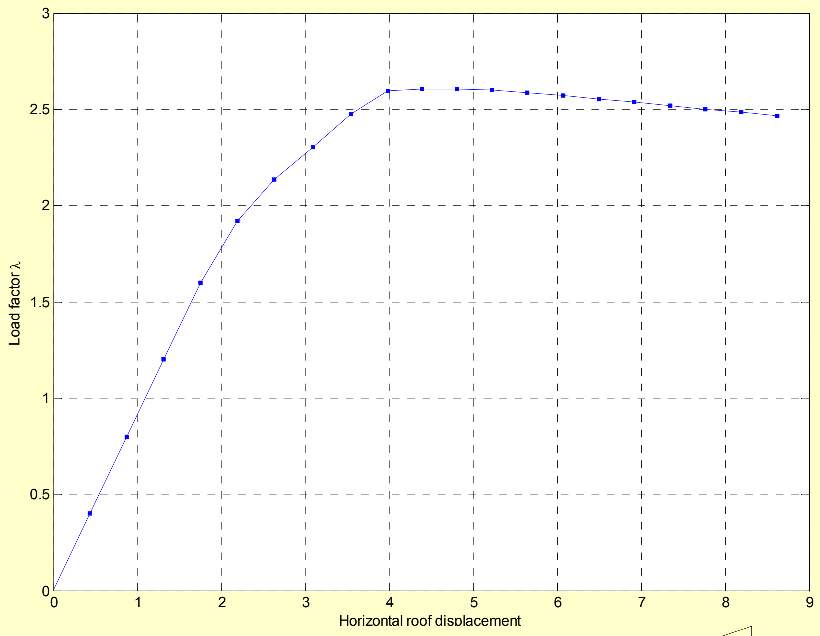
Incremental analysis for distributed element loading (single load step)

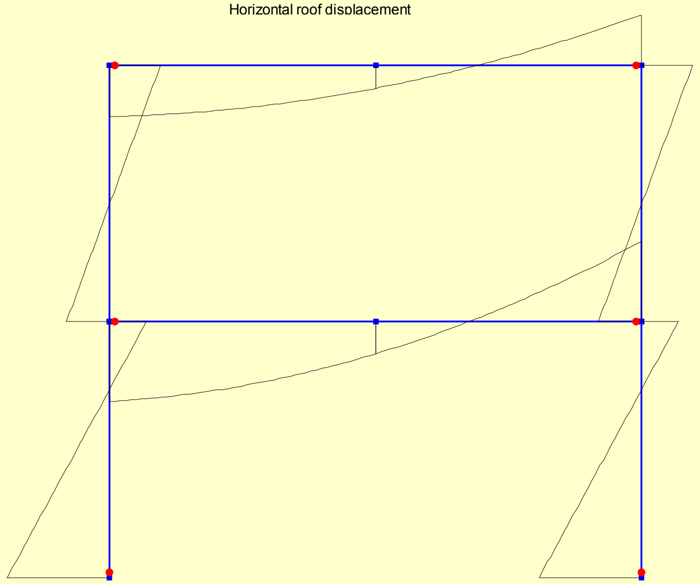
```
% initialize state
State = Initialize_State(Model,ElemData);
% initialize solution strategy parameters
SolStrat = Initialize_SolStrat;
% specify initial load increment (even though it is the same as the default
value and could be omitted)
SolStrat.IncrStrat.Dlam0 = 1;
% initialize analysis sequence
[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

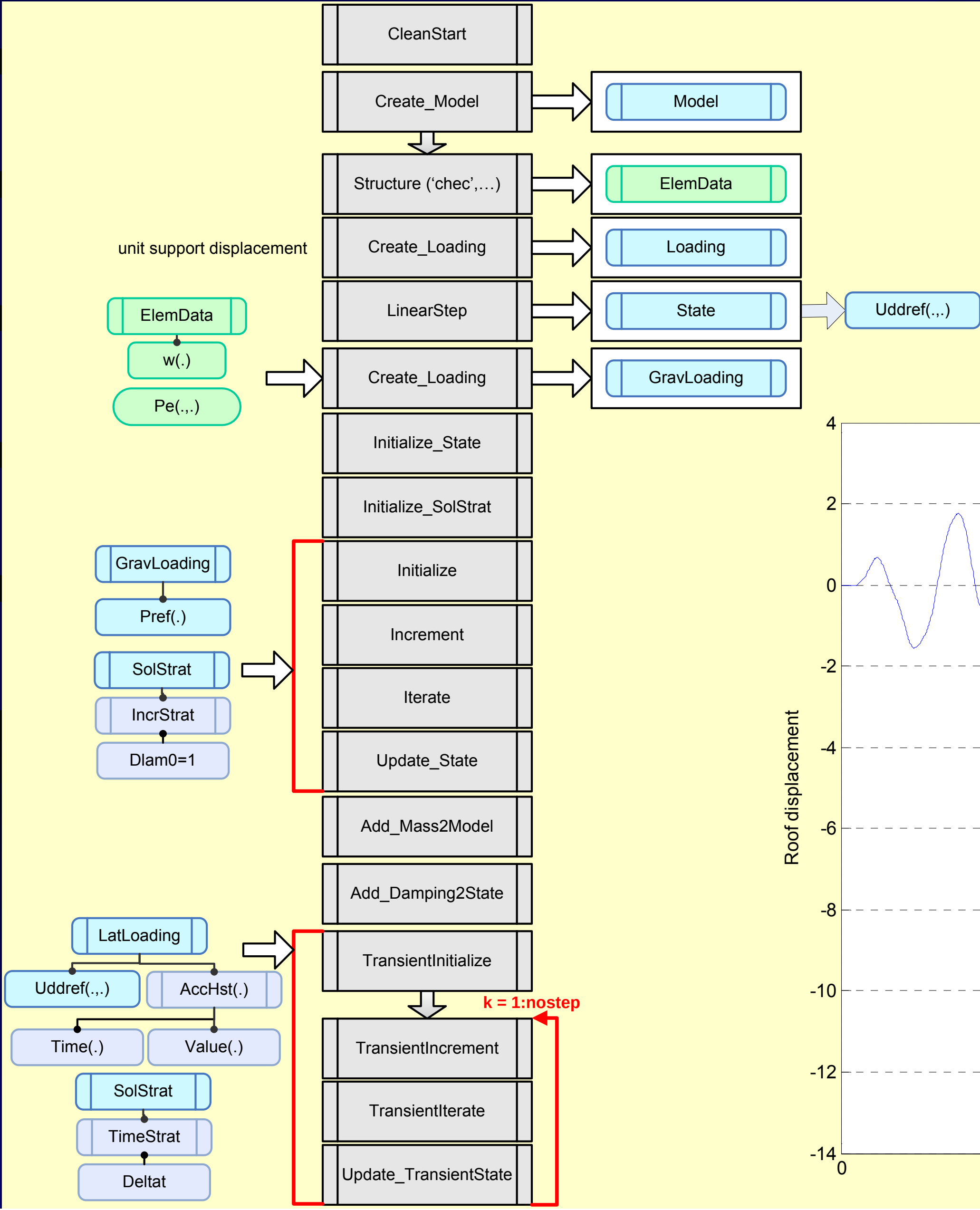
```
% specify nodal forces
% !!!! IMPORTANT!!!! CLEAR PREVIOUS PE
clear Pe;
Pe(2,1) =  20;
Pe(3,1) =  40;
Pe(5,1) =  20;
Pe(6,1) =  40;
LatLoading = Create_Loading (Model,Pe);
```

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

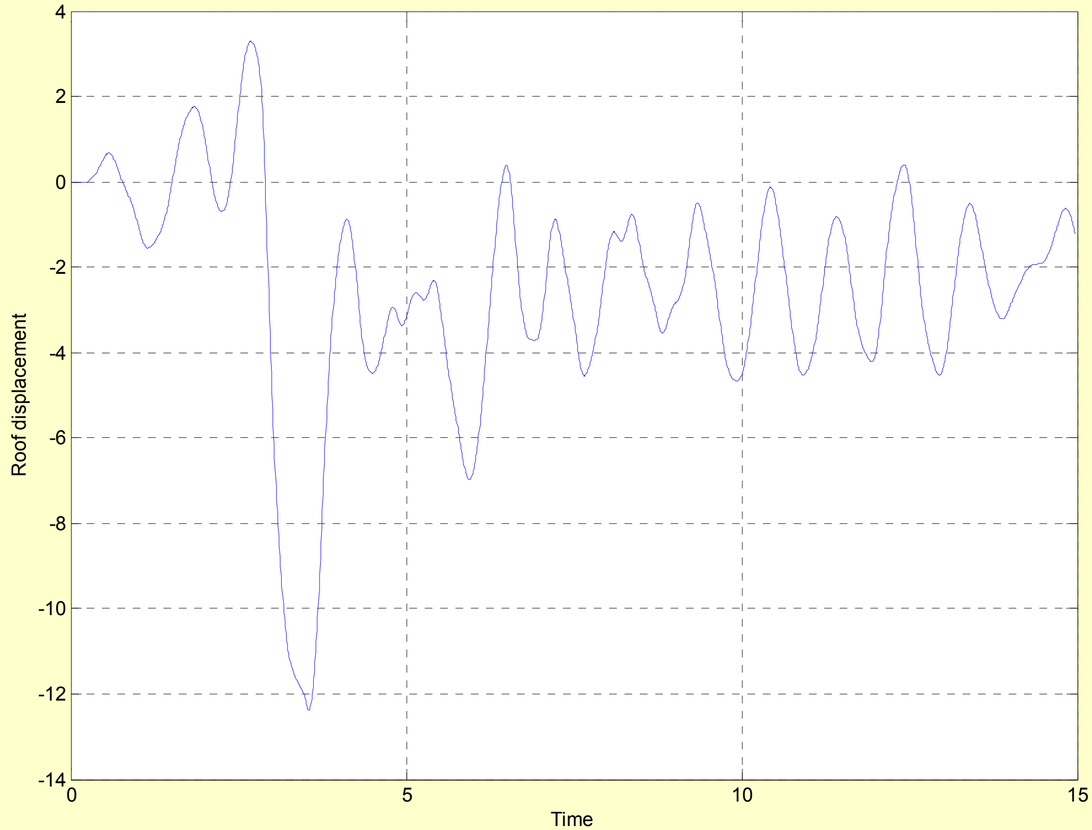




Example 8

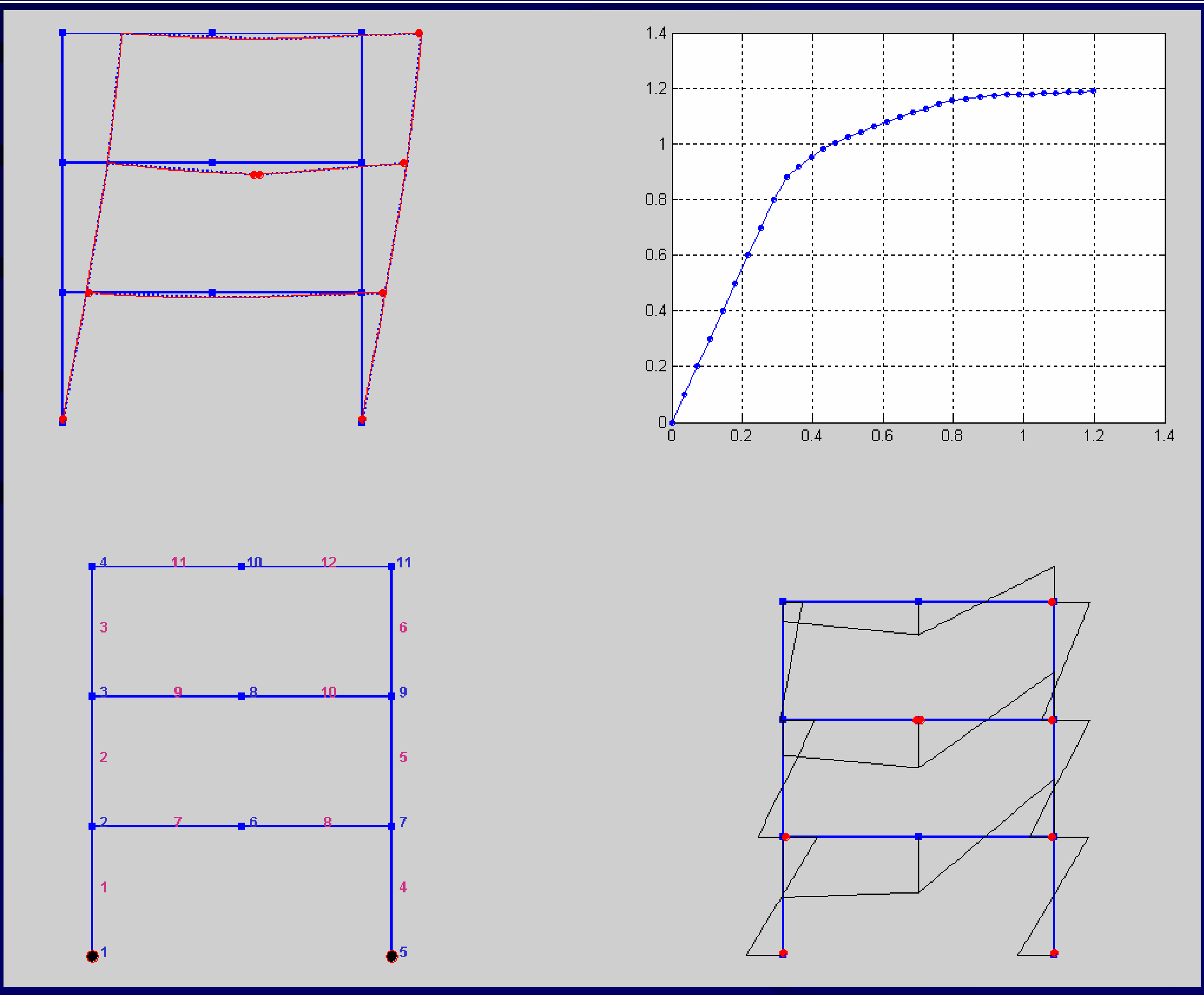


CleanStart
Create_Model
Model
Structure ('chec',...)
ElemData
unit support displacement
Create_Loading
Loading
LinearStep
State
Uddref(..,)
ElemData
w(.)
Pe(..,)
Create_Loading
GravLoading
Initialize_State
Initialize_SolStrat
GravLoading
Pref(.)
SolStrat
IncrStrat
Dlam0=1
Initialize
Increment
Iterate
Update_State
Add_Mass2Model
Add_Damping2State
LatLoading
Uddref(..,)
AccHst(.)
Time(.)
Value(.)
SolStrat
TimeStrat
Deltat
TransientInitialize
k = 1:nostep
TransientIncrement
TransientIterate
Update_TransientState

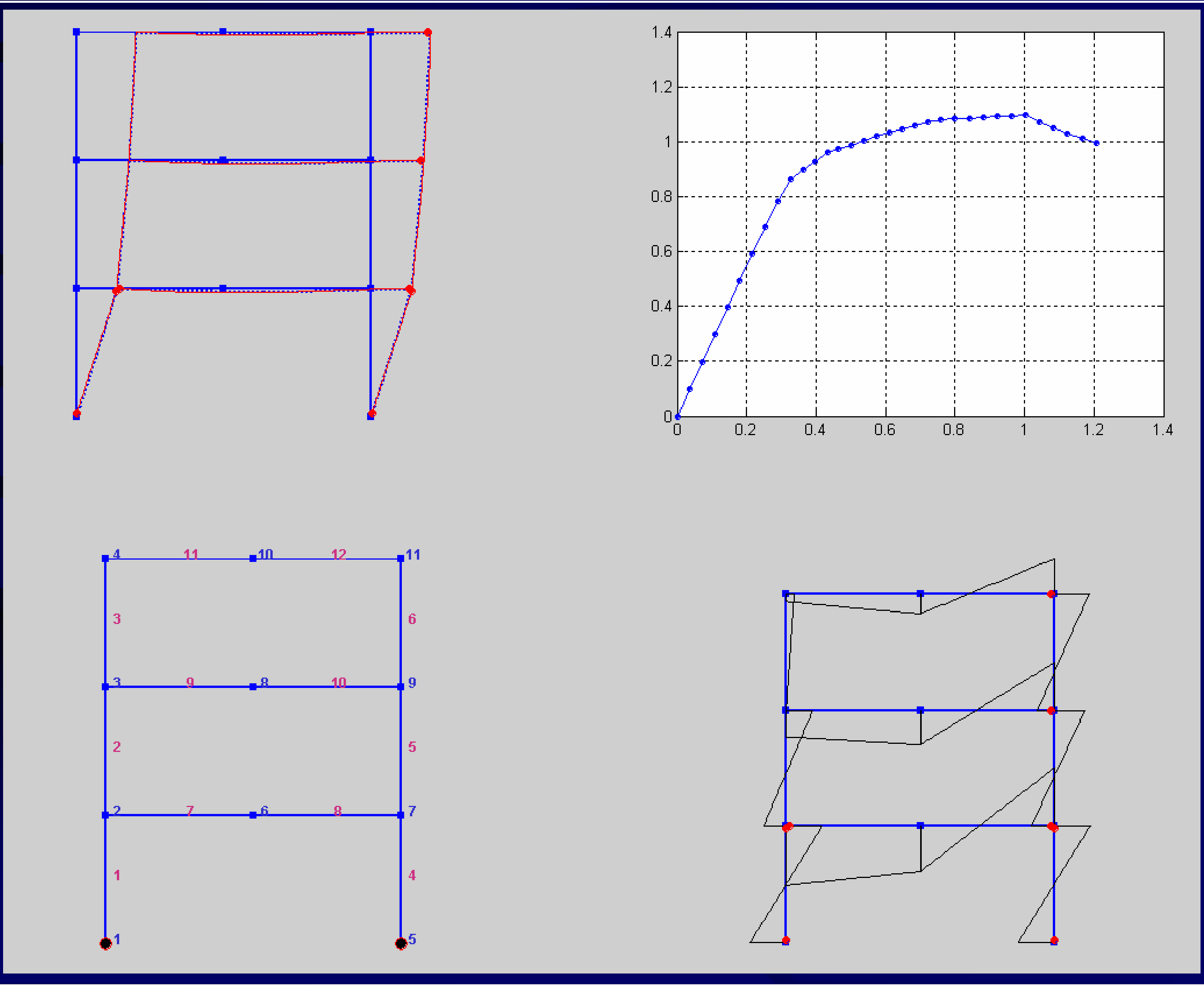


Roof displacement
Time
4
2
0
-2
-4
-6
-8
-10
-12
-14
0
5
10
15

Example: push-over analysis of 3-story steel frame



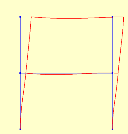
Push-over analysis of 3-story steel frame with nonlinear geometry



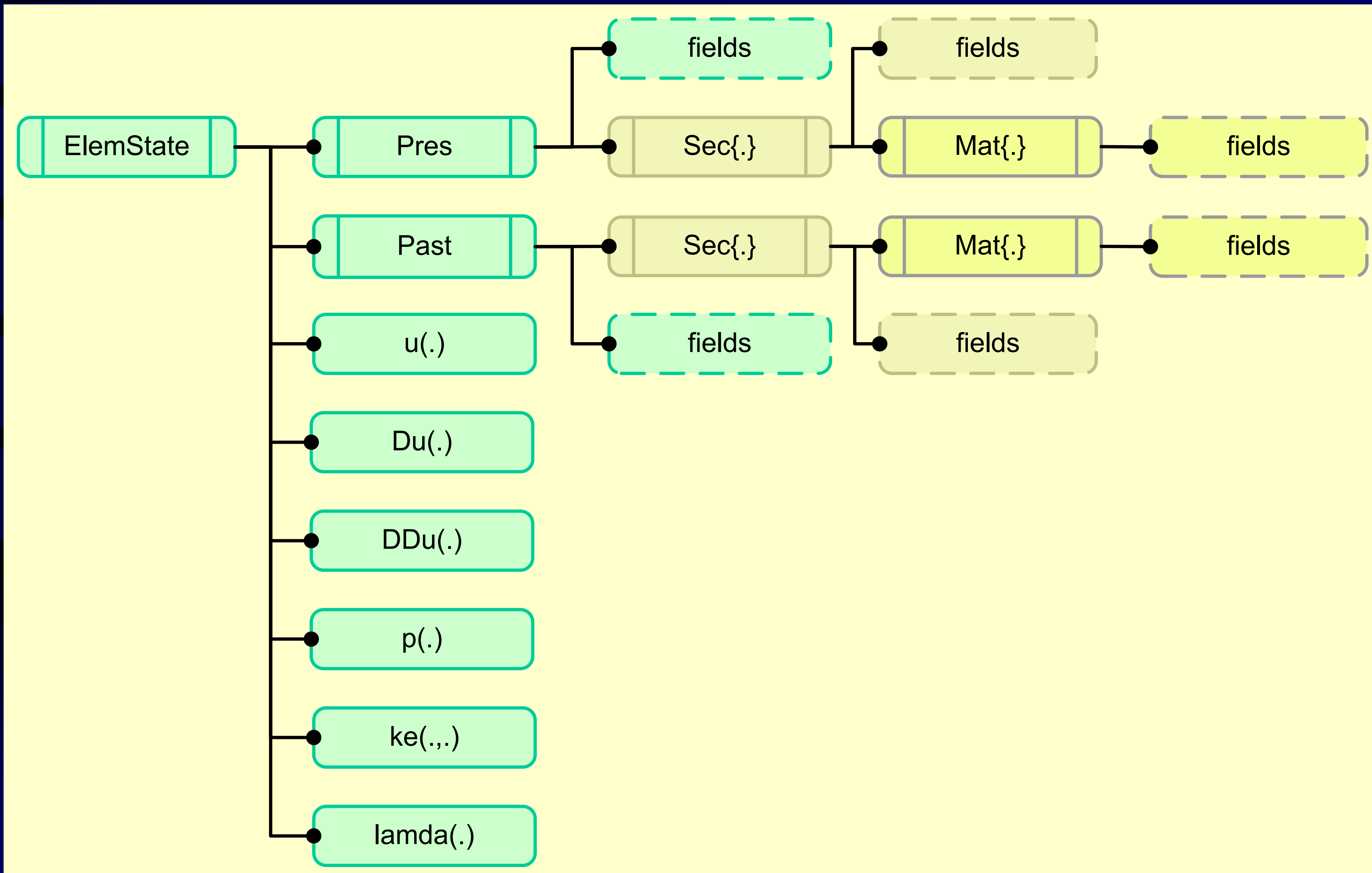
Adding element models

ElementName (action,el_no,ndm,ElemData,ElemState)

action | el_no | ndm | ElemData | ElemState

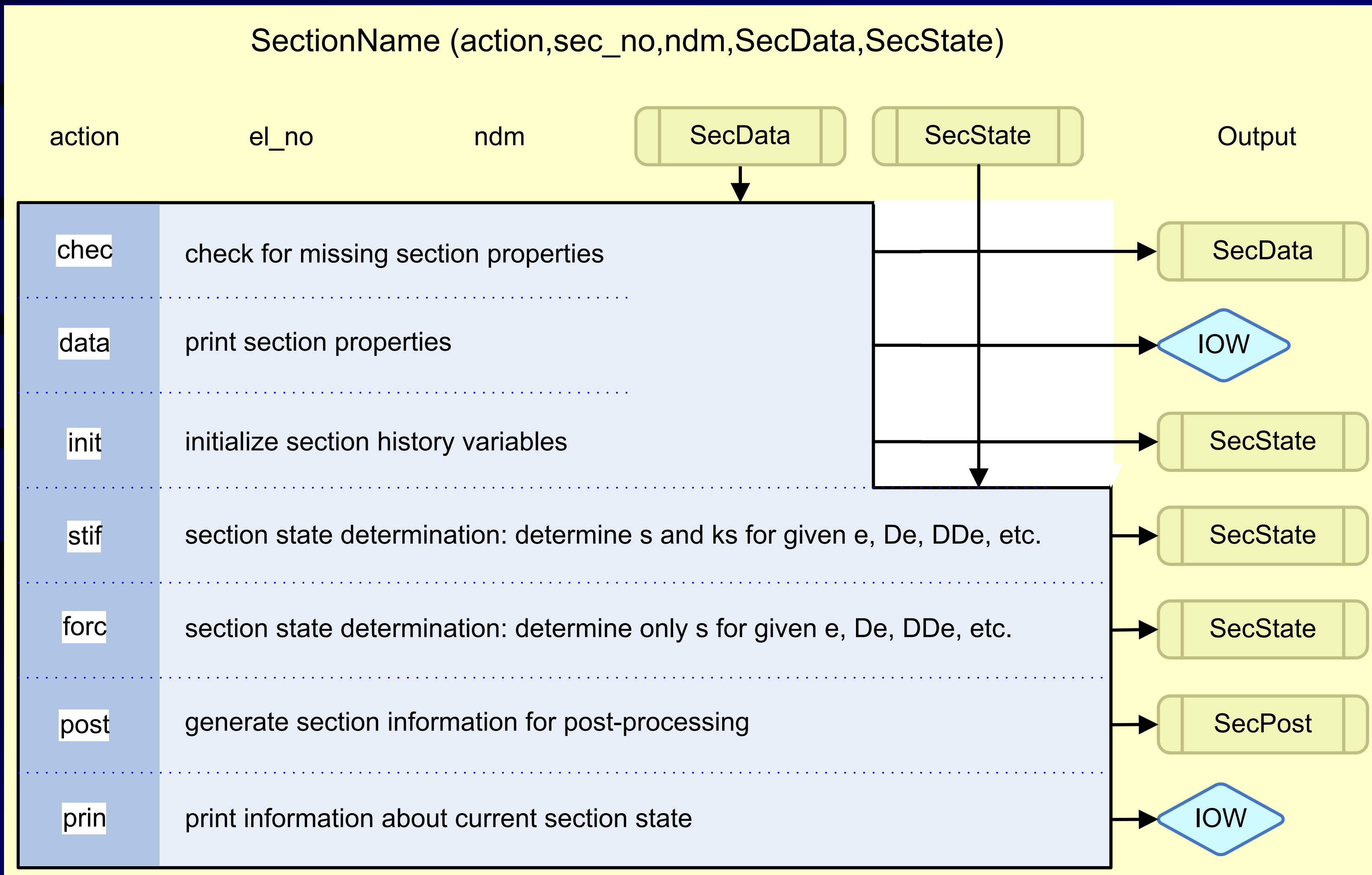
action		Output
chec	check for missing element properties	ElemData
data	print element properties	IOW
init	initialize element history variables	ElemState
stif	element state determination: determine p and ke for given u, Du, DDu, etc.	ElemState
forc	element state determination: determine only p for given u, Du, DDu, etc.	ElemState
post	generate element information for post-processing	ElemPost
prin	print information about current element state	IOW
defo	plot deformed shape of element	

ElemState

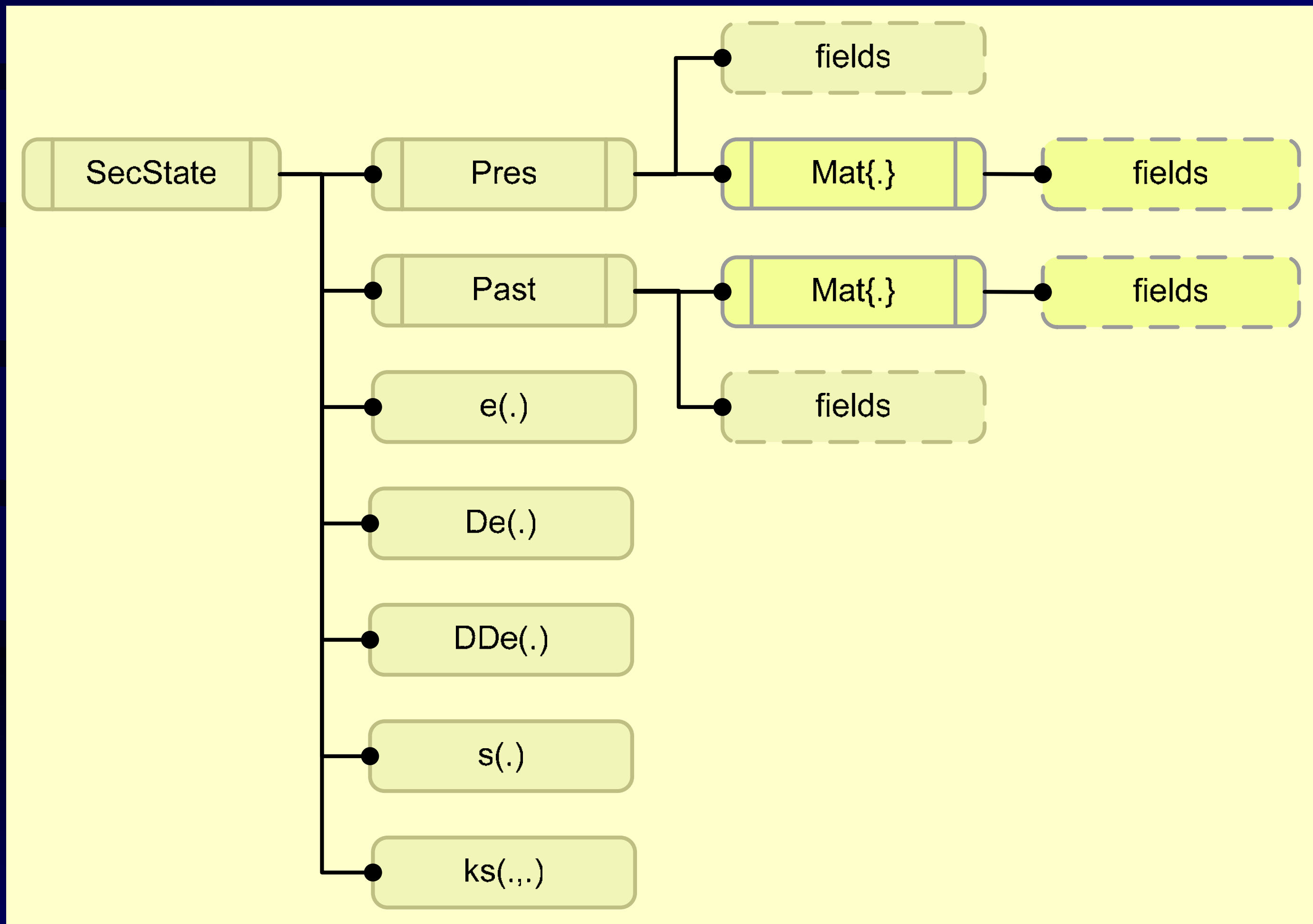


ElemState
Pres
fields
Sec{.}
fields
Mat{.}
fields
Past
Sec{.}
Mat{.}
fields
fields
fields
u(.)
Du(.)
DDu(.)
p(.)
ke(.,.)
lamda(.)

Adding section models

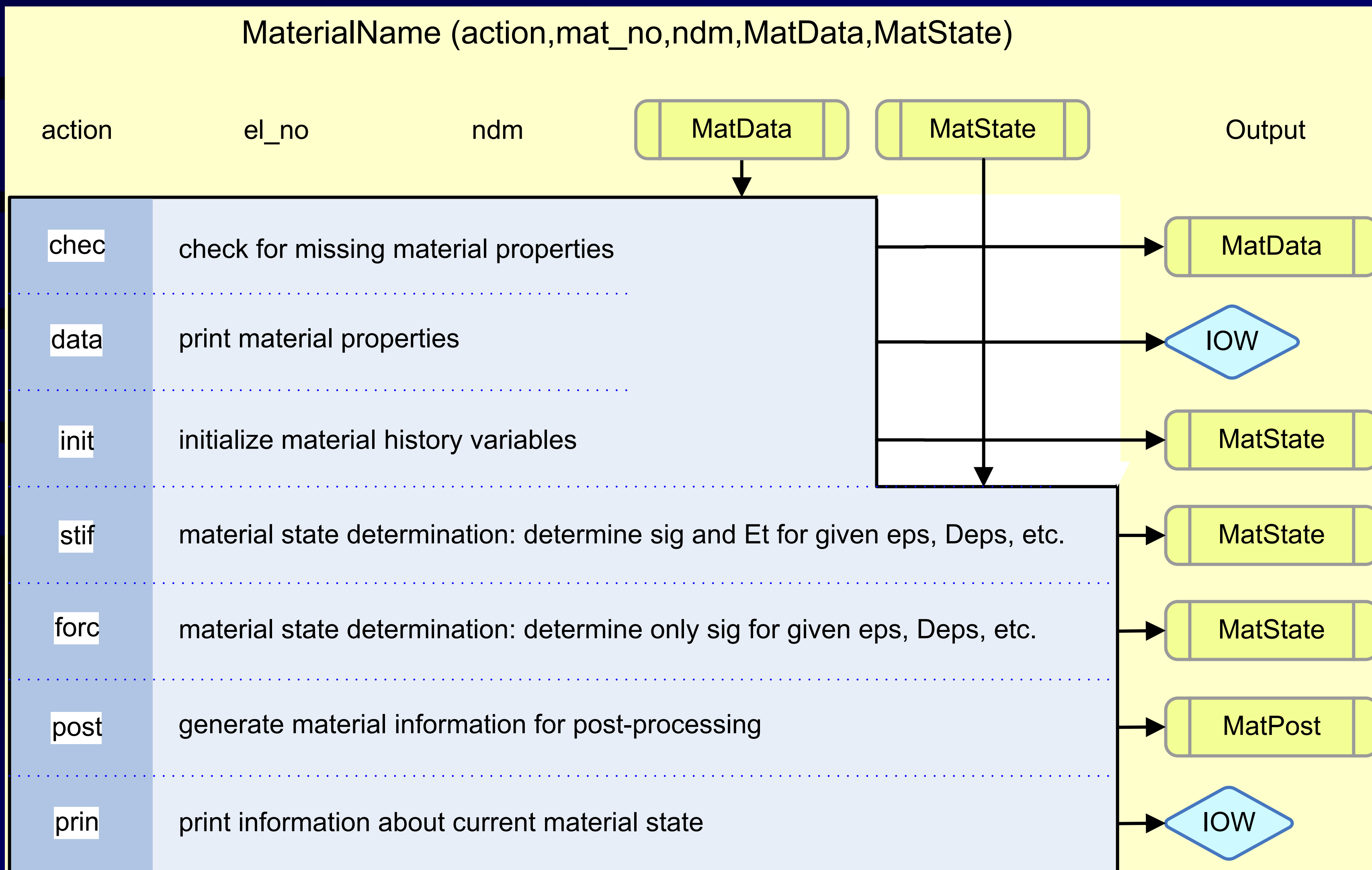


SecState



SecState
Pres
Past
e(.)
De(.)
DDe(.)
s(.)
ks(.,.)
fields
Mat{.}
fields
Mat{.}
fields
fields

Adding material models



MatState